



PeopleTools 8.12 PeopleCode
Reference PeopleBook

SKU MTPRr8SP1B 1200

PeopleBooks Contributors: Teams from PeopleSoft Product Documentation and Development.

Copyright © 2001 by PeopleSoft, Inc. All rights reserved.

Printed in the United States of America.

All material contained in this documentation is proprietary and confidential to PeopleSoft, Inc. and is protected by copyright laws. No part of this documentation may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, including, but not limited to, electronic, graphic, mechanical, photocopying, recording, or otherwise without the prior written permission of PeopleSoft, Inc.

This documentation is subject to change without notice, and PeopleSoft, Inc. does not warrant that the material contained in this documentation is free of errors. Any errors found in this document should be reported to PeopleSoft, Inc. in writing.

The copyrighted software that accompanies this documentation is licensed for use only in strict accordance with the applicable license agreement which should be read carefully as it governs the terms of use of the software and this documentation, including the disclosure thereof.

PeopleSoft, the PeopleSoft logo, PeopleTools, PS/nVision, PeopleCode, PeopleBooks, Vantive, and Vantive Enterprise are registered trademarks, and *PeopleTalk* and "People power the internet." are trademarks of PeopleSoft, Inc. All other company and product names may be trademarks of their respective owners.

Contents

About This PeopleBook

Audience	lxv
Before You Begin	lxvi
Related Documentation	lxvi
Documentation on the Internet	lxvi
Documentation on CD-ROM	lxvii
Hardcopy Documentation	lxvii
Typographical Conventions and Visual Cues	lxvii
Comments and Suggestions	lxix

Chapter 1

PeopleCode Built-in Functions

Functions by Category	1-1
ActiveX Controls	1-1
APIs	1-1
Application Engine	1-1
Application Messages	1-1
Arrays	1-2
Attachment	1-2
Business Interlink	1-2
Component Buffer	1-2
Component Interface	1-3
Conversion	1-3
Currency and Financial	1-3
Current Date and Time	1-4
Custom Display Formats	1-4
Database and Platform	1-4
Data Buffer Access	1-4
Date and Time	1-4
Defaults, Setting	1-5
DOS	1-6
Double-Byte Characters	1-6
Dynamic Tree Controls	1-6

Effective Date and Effective Sequence	1-6
Executable Files, Running.....	1-6
Files	1-7
Financial	1-7
Grids	1-7
Images	1-7
Import Manager.....	1-7
Internet	1-8
Java.....	1-8
Language Preference and Locale	1-8
Language Constructs	1-8
Logical (Tests for Blank Values)	1-8
Mail	1-9
Math	1-9
Menu Appearance	1-9
Message Agent.....	1-9
Message Catalog and Message Display	1-10
Message, Application	1-10
Modal Transfers	1-10
Object-Based	1-10
OLE	1-11
Page	1-11
Page Control Appearance.....	1-11
Process Scheduler.....	1-12
Remote Call.....	1-12
Saving and Canceling.....	1-12
Scroll Select	1-12
Search Dialog.....	1-12
Secondary Pages.....	1-12
SQL	1-13
SQL Date and Time.....	1-13
SQL Shortcuts	1-14
String	1-14
Subrecords.....	1-14
TimeZone	1-14
Trace Control	1-15
Transfers.....	1-15
User Information	1-15
User Security	1-15
Validation.....	1-16

Workflow	1-16
PeopleCode Built-in Functions and Language Constructs A-D	1-16
Abs	1-16
AccruableDays	1-17
AccrualFactor	1-18
Acos.....	1-19
ActiveRowCount.....	1-20
AddAttachment	1-21
AddKeyListItem.....	1-27
AddSystemPauseTimes.....	1-27
AddToDate	1-30
AddToDateTime.....	1-31
AddToTime	1-32
All.....	1-33
AllOrNone.....	1-34
AllowEmplIdChg	1-35
Amortize.....	1-36
Asin	1-37
Atan.....	1-37
BlackScholesCall	1-38
BlackScholesPut.....	1-39
BootstrapYTM.....	1-39
Break	1-40
CallAppEngine	1-41
Char	1-44
CharType.....	1-44
ChDir.....	1-47
ChDrive	1-48
CheckMenuItem	1-49
Clean	1-50
ClearKeyList	1-50
ClearSearchDefault	1-51
ClearSearchEdit.....	1-52
Code	1-52
Codeb	1-53
CommitWork.....	1-53
CompareLikeFields	1-56
Component	1-57
ComponentChanged.....	1-58
ContainsCharType.....	1-59

ContainsOnlyCharType.....	1-61
ConvertChar	1-63
ConvertCurrency	1-68
ConvertDatetimeToBase	1-70
ConvertRate.....	1-71
ConvertTimeToBase	1-73
CopyFields	1-74
CopyRow.....	1-75
Cos.....	1-76
Cot.....	1-76
CreateArray	1-77
CreateArrayRept	1-78
CreateJavaArray.....	1-80
CreateJavaObject.....	1-80
CreateMessage	1-82
CreateObject.....	1-83
CreateProcessRequest	1-84
CreateRecord.....	1-85
CreateRowset	1-87
CreateSQL.....	1-90
CubicSpline	1-91
CurrEffDt	1-93
CurrEffRowNum.....	1-94
CurrEffSeq	1-95
CurrentLevelNumber	1-95
CurrentRowNumber.....	1-96
Date	1-97
Date3	1-97
DatePart.....	1-98
DateTime6.....	1-98
DateTimeToLocalizedString.....	1-99
DateTimeToTimeZone.....	1-101
DateTimeValue	1-103
DateValue.....	1-104
Day	1-105
Days.....	1-105
Days360.....	1-106
Days365.....	1-106
DBCSTrim	1-107
Declare Function	1-107

Decrypt.....	1-111
Degrees.....	1-112
DeleteAttachment.....	1-112
DeleteImage	1-116
DeleteRecord.....	1-117
DeleteRow.....	1-117
DeleteSQL.....	1-119
DeleteSystemPauseTimes	1-121
DisableMenuItem	1-123
DiscardRow	1-124
DoCancel.....	1-125
DoModal	1-126
DoModalComponent.....	1-128
DoModalPanelGroup	1-132
DoSave	1-132
DoSaveNow	1-133

Chapter 2

PeopleCode Built-in Functions and Language Constructs E-M

EnableMenuItem	2-1
EncodeURL.....	2-2
EncodeURLForQueryString.....	2-3
Encrypt.....	2-4
EndMessage	2-4
EndModal.....	2-6
Error	2-7
EscapeHTML	2-9
EscapeJavascriptString.....	2-10
EscapeWML.....	2-10
Evaluate.....	2-11
Exact.....	2-12
Exec.....	2-12
ExecuteRolePeopleCode	2-14
ExecuteRoleQuery	2-15
ExecuteRoleWorkflowQuery	2-16
Exit	2-17
Exp	2-18
ExpandBindVar.....	2-18
ExpandEnvVar	2-19
ExpandSqlBinds	2-20

Fact.....	2-21
FetchSQL	2-22
FetchValue	2-23
FieldChanged	2-24
FileExists.....	2-26
Find	2-28
Findb	2-29
FindFiles.....	2-30
FlushBulkInserts	2-32
For	2-34
FormatDateTime	2-34
Function.....	2-36
GenerateTree	2-38
GetAESection.....	2-39
GetAttachment	2-40
GetBiDoc.....	2-43
GetCalendarDate	2-45
GetControl.....	2-47
GetControlOccurrence	2-49
GetCwd	2-51
GetEnv.....	2-52
GetField.....	2-52
GetFile.....	2-53
GetGrid.....	2-57
GetHTMLText	2-60
GetImageExtents	2-61
GetInterlink	2-62
GetJavaClass	2-65
GetLevel0.....	2-66
GetMethodNames.....	2-68
GetMessage	2-68
GetMessageInstance.....	2-69
GetNextNumber	2-70
GetNextNumberWithGaps	2-72
GetPage	2-73
GetPubContractInstance.....	2-75
GetRecord	2-75
GetRelField	2-77
GetRow	2-78
GetRowset.....	2-79

GetSelectedTreeNode.....	2-80
GetSession.....	2-83
GetSetId.....	2-83
GetSQL	2-85
GetStoredFormat	2-87
GetSubContractInstance.....	2-88
GetTreeNodeParent.....	2-91
GetTreeNodeRecordName	2-92
GetTreeNodeValue	2-93
GetURL	2-95
GetWLFieldValue	2-96
Global.....	2-97
Gray.....	2-98
GrayMenuItem	2-100
Hash.....	2-100
HermiteCubic	2-101
Hide.....	2-101
HideMenuItem	2-103
HideRow	2-104
HideScroll	2-106
HistVolatility.....	2-107
Hour.....	2-108
If	2-108
InsertImage.....	2-109
InsertRow	2-111
Int	2-113
IsDaylightSavings	2-113
IsHidden	2-115
IsMenuItemAuthorized	2-116
IsModal	2-117
IsModalComponent.....	2-118
IsModalPanelGroup	2-119
IsOperatorInClass.....	2-120
IsSearchDialog	2-120
IsUserInPermissionList.....	2-121
IsUserInRole	2-121
Left	2-122
Len.....	2-122
Lenb.....	2-123
LinearInterp.....	2-123

Ln	2-124
Local.....	2-124
Log10	2-125
Lower	2-126
LTrim	2-126
MarkWLItemWorked.....	2-127
MessageBox	2-127
Minute	2-134
Mod	2-134
Month	2-135
MsgGet.....	2-135
MsgGetExplainText	2-136
MsgGetText.....	2-138

Chapter 3

PeopleCode Built-in Functions and Language Constructs N-Z

NextEffDt.....	3-1
NextRelEffDt	3-1
None	3-2
ObjectDoMethod.....	3-3
ObjectGetProperty.....	3-4
ObjectSetProperty	3-6
OnlyOne	3-8
OnlyOneOrNone	3-9
PanelGroupChanged.....	3-10
PingNode.....	3-11
PriorEffDt.....	3-11
PriorRelEffDt	3-12
PriorValue	3-13
Product	3-13
Prompt.....	3-14
Proper	3-16
PutAttachment.....	3-16
Radians.....	3-18
Rand	3-19
RecordChanged	3-19
RecordDeleted.....	3-21
RecordNew.....	3-24
RefreshTree	3-26
RemoteCall.....	3-27

Repeat.....	3-30
Replace.....	3-31
Rept.....	3-32
Return.....	3-32
ReturnToServer.....	3-33
RevalidatePassword.....	3-35
Right.....	3-36
Round.....	3-37
RoundCurrency.....	3-37
RowFlush.....	3-38
RowScrollSelect.....	3-39
RowScrollSelectNew.....	3-42
RTrim.....	3-44
ScheduleProcess.....	3-44
ScrollFlush.....	3-48
ScrollSelect.....	3-49
ScrollSelectNew.....	3-53
Second.....	3-55
SendMail.....	3-55
SetAuthenticationResult.....	3-59
SetChannelStatus.....	3-60
SetControlValue.....	3-61
SetCursorPos.....	3-64
SetDefault.....	3-66
SetDefaultAll.....	3-67
SetDefaultNext.....	3-68
SetDefaultNextRel.....	3-68
SetDefaultPrior.....	3-69
SetDefaultPriorRel.....	3-69
SetDisplayFormat.....	3-70
SetLabel.....	3-71
SetLanguage.....	3-72
SetNextPanel.....	3-73
SetNextPage.....	3-74
SetPasswordExpired.....	3-75
SetReEdit.....	3-75
SetSearchDefault.....	3-76
SetSearchDialogBehavior.....	3-77
SetSearchEdit.....	3-78
SetTempTableInstance.....	3-79

SetTracePC.....	3-80
SetTraceSQL	3-83
Sign	3-85
Sin	3-86
SinglePaymentPV	3-86
SortScroll.....	3-87
Split	3-88
SQLExec	3-89
Sqrt	3-95
StopFetching	3-96
StoreSQL.....	3-97
String	3-98
Substitute.....	3-99
Substring	3-100
Substringb	3-100
SwitchUser	3-101
Tan	3-103
Time	3-104
Time3	3-104
TimePart.....	3-105
TimeToTimeZone	3-106
TimeValue.....	3-107
TimeZoneOffset	3-108
TotalRowCount.....	3-109
Transfer	3-110
TransferPanel	3-114
TransferPage	3-115
TreeDetailInNode.....	3-116
TriggerBusinessEvent	3-118
Truncate.....	3-119
UnCheckMenuItem	3-120
Unencode.....	3-121
Ungray.....	3-122
Unhide	3-124
UnhideRow	3-125
UnhideScroll	3-127
UniformSeriesPV	3-128
UpdateSysVersion.....	3-129
UpdateValue.....	3-129
Upper.....	3-131

Value	3-131
ViewAttachment	3-132
ViewURL	3-135
Warning.....	3-137
Weekday.....	3-139
While	3-139
WinEscape.....	3-140
WinExec	3-141
WinMessage.....	3-143
Year	3-147

Chapter 4

PeopleCode Classes

Classes by Category	4-1
API Classes	4-1
Application Engine Class	4-1
Data Buffer Access Classes.....	4-2
Internet Class.....	4-2
Page Display Classes.....	4-2
Integration Classes	4-2
Miscellaneous Classes.....	4-2
AESession Class	4-2
How an AESession is Accessed	4-4
AESession Example	4-5
Declaring an AESession Object	4-7
Scope of an AESession Object.....	4-7
AESession Class Built-in Function	4-7
AESession Class Methods.....	4-7
AddStep.....	4-7
Close.....	4-8
Open	4-9
Save	4-10
SetSQL	4-10
SetTemplate	4-12
AESession Class Property	4-12
IsOpen	4-12
Array Class	4-13
Creating Arrays	4-13
Populating an Array	4-15
Removing Items from an Array	4-16

Creating Empty Arrays.....	4-17
Creating and Populating Multi-Dimensional Arrays.....	4-18
Flattening and Promotion	4-21
Declaring Array Objects.....	4-21
Scope of an Array Object.....	4-22
Array Class Built-in Functions.....	4-22
Array Class Methods.....	4-22
Clone	4-22
Find	4-23
Get.....	4-24
Join.....	4-25
Next.....	4-26
Pop	4-27
Push.....	4-29
Replace.....	4-30
Reverse.....	4-32
Set.....	4-33
Shift.....	4-34
Sort.....	4-34
Subarray	4-36
Substitute.....	4-37
Unshift.....	4-38
Array Class Properties.....	4-39
Dimension	4-39
Len	4-39
Business Interlink Class.....	4-40
Using the Interlink Object.....	4-40
Deciding Which Methods to Use	4-41
Executing the Business Interlink Object	4-41
Supporting Batch Input and Output	4-41
Supporting Rowsets	4-41
Using the Flat Table Methods	4-42
Supporting Dynamic Output	4-42
Using Hierarchical Data (BIDocs)	4-42
Using the Incoming Business Interlink Methods and Properties	4-46
State of an Interlink Object	4-47
Using Business Interlink with Application Engine	4-47
Declaring a Business Interlink Object.....	4-48
Scope of a Business Interlink Object	4-48
Business Interlink Class Built-in Function.....	4-48

Business Interlink Class Methods	4-49
AddDoc	4-49
AddInputRow	4-50
AddNextDoc	4-52
AddValue	4-55
BulkExecute	4-57
Clear	4-62
Execute	4-63
FetchIntoRecord	4-64
FetchIntoRowset	4-67
FetchNextRow	4-70
GetCount	4-71
GetDoc	4-73
GetFieldCount	4-76
GetFieldType	4-77
GetFieldValue	4-78
GetInputDocs	4-80
GetNextDoc	4-80
GetOutputDocs	4-83
GetPreviousDoc	4-84
GetStatus	4-86
GetValue	4-87
InputRowset	4-89
MoveFirst	4-91
MoveNext	4-93
MoveToDoc	4-94
ResetCursor	4-96
Business Interlink BIDocs Methods	4-97
AddAttribute	4-97
AddComment	4-98
AddProcessInstruction	4-100
AddText	4-101
CreateElement	4-102
GenXMLString	4-104
GetAttributeName	4-104
GetAttributeValue	4-106
GetNode	4-107
ParseXMLString	4-108
Business Interlink BIDocs Properties	4-109
AttributeCount	4-109

ChildNodeCount	4-110
NodeName.....	4-111
NodeType.....	4-112
NodeValue	4-114
Business Interlink Class Property	4-114
StopAtError	4-114
Configuration Parameters.....	4-115
Interlink Plug-in Configuration Parameters	4-115
URL Configuration Parameter	4-116
BIDocValidate Configuration Parameter.....	4-116
Component Interface Classes.....	4-116
Life Cycle of a Component Interface	4-118
Setting Component Interface Keys	4-119
Standard and User Defined Component Interface Methods.....	4-120
Accessing Component Interface Standard Properties	4-122
Accessing User Defined Component Interface Properties.....	4-123
Declaring a Component Interface Object.....	4-123
Scope of a Component Interface Object.....	4-124
Implementing a Component Interface.....	4-124
Component Interface Methods and Timeouts	4-135
Traversing a Component Interface and Using Data Collections	4-135
Working with Effective Dated Data.....	4-148
How a Developer would Use GetEffectiveItem.....	4-148
Reusing Existing Code	4-149
Differences in Search Dialog Processing.....	4-149
Differences in PeopleCode Event and Function Behavior.....	4-149
Component Interface Reference.....	4-150
Session Object Methods	4-150
FindCompIntfc	4-150
GetCompIntfc.....	4-152
Component Interface Collection Methods	4-152
First	4-152
Item	4-153
Next.....	4-153
Component Interface Collection Property.....	4-154
Count.....	4-154
Component Interface Class Methods	4-154
Cancel.....	4-154
CopyRowset	4-154
CopyRowsetDelta	4-158

CopySetupRowset.....	4-160
CopySetupRowsetDelta	4-164
Create	4-167
Find	4-168
Get.....	4-169
Save.....	4-170
Component Interface Class Properties	4-171
CreateKeyInfoCollection	4-171
FindKeyInfoCollection	4-171
GetHistoryItems	4-172
GetKeyInfoCollection	4-172
InteractiveMode	4-172
PropertyInfoCollection.....	4-173
StopOnFirstError.....	4-173
Data Collection Methods.....	4-173
CurrentItem	4-173
DeleteItem	4-174
GetEffectiveItem	4-175
GetEffectiveItemNum	4-176
InsertItem	4-177
Item	4-178
ItemByKeys.....	4-178
Data Collection Properties	4-181
Count.....	4-181
CurrentItemNumber	4-182
Data Item Class Property.....	4-182
ItemNum	4-182
Accessing the Structure of a Component Interface	4-183
CompIntfPropInfoCollection Collection.....	4-183
CompIntfPropInfoCollection Collection Methods.....	4-186
First	4-186
Item	4-186
Next.....	4-187
CompIntfPropInfoCollection Collection Properties.....	4-187
Count.....	4-187
CompIntfPropInfoCollection Object Properties.....	4-187
Format	4-187
IsCollection	4-188
Key	4-188
LabelLong	4-188

LabelShort.....	4-189
Name	4-189
Prompt.....	4-189
PropertyInfoCollection.....	4-189
Required	4-190
Type	4-190
Xlat.....	4-190
YesNo.....	4-190

Chapter 5

Field Class

Declaring a Field Object	5-1
Scope of a Field Object	5-2
Field Class Built-in Function	5-2
Field Class Methods	5-2
GetLongLabel	5-2
GetRelated.....	5-3
GetShortLabel	5-4
SetCursorPos	5-5
SetDefault.....	5-6
Field Class Properties.....	5-7
DisplayFormat.....	5-7
DisplayOnly	5-9
EditError.....	5-9
Enabled.....	5-9
FormattedValue.....	5-10
IsAltKey	5-10
IsAuditFieldAdd.....	5-11
IsAuditFieldChg.....	5-11
IsAuditFieldDel.....	5-11
IsAutoUpdate	5-11
IsChanged.....	5-11
IsDateRangeEdit	5-12
IsDescKey	5-12
IsDuplKey	5-12
IsEditTable	5-12
IsEditXlat	5-12
IsFromSearchField	5-12
IsInBuf	5-12
IsKey	5-13

IsListItem	5-13
IsRequired	5-14
IsSearchItem.....	5-14
IsSystem.....	5-14
IsThroughSearchField	5-14
IsUseDefaultLabel.....	5-14
IsYesNo.....	5-14
Label.....	5-14
LongTranslateValue.....	5-15
MessageNumber.....	5-16
MessageSetNumber.....	5-17
Name	5-17
OriginalValue.....	5-17
ParentRecord	5-18
SearchDefault.....	5-18
SearchEdit	5-18
ShortTranslateValue.....	5-19
ShowRequiredFieldCue	5-20
StoredFormat.....	5-20
Style	5-21
Type	5-22
Value	5-22
Visible	5-23
File Class	5-24
Declaring a File Object	5-24
Scope of a File Object	5-24
File Layout	5-24
File Security Considerations	5-25
Recovering from File Access Interruptions.....	5-25
Using Plain Text Files.....	5-26
Using File Layouts	5-28
WriteRecord Example.....	5-29
ReadRecord Example.....	5-32
WriteRowset Example	5-34
ReadRowset Example	5-39
File Rowset Considerations.....	5-40
Handling Multiple File Layouts	5-40
Application Engine Example	5-46
Handling File Layout Errors	5-48
File Class Built-In Functions.....	5-50

File Class Methods	5-50
Close	5-50
CreateRowset	5-51
GetPosition	5-52
Open	5-53
ReadLine	5-57
ReadRowset	5-58
SetFileId	5-60
SetFileLayout	5-62
SetPosition	5-63
WriteLine	5-65
WriteRaw	5-66
WriteRecord	5-67
WriteRowset	5-70
WriteString	5-71
File Class Properties	5-72
CurrentRecord	5-72
IgnoreInvalidId	5-72
IsError	5-73
IsNewFileId	5-73
IsOpen	5-74
Name	5-75
RecTerminator	5-75
Grid Class	5-75
Using the Grid Class in PeopleCode	5-76
Declaring a Grid or Grid Column Object	5-77
Scope of a Grid or Grid Column Object	5-77
Grid Class Built-In Function	5-78
Grid Class Method	5-78
GetColumn	5-78
Grid Class Property	5-81
<i>gridcolumn</i>	5-81
Label	5-81
GridColumn Class	5-82
GridColumn Class Properties	5-82
Enabled	5-82
Label	5-82
Name	5-83
Visible	5-83
Internet Script Classes	5-85

The Big Picture	5-85
URL vs. URI	5-86
Creating Web Libraries	5-87
Internet Script Security.....	5-89
When to use an Internet Script	5-91
Style Sheets and Styles.....	5-92
Other Considerations.....	5-92
Accessing an Internet Script.....	5-92
Error Handling	5-93
Scope of the Internet Script Classes.....	5-93
Internet Script Reference.....	5-94
Request Class	5-94
Response	5-95
Cookies.....	5-95
Request Class Methods	5-95
GetContentBody.....	5-95
GetCookieNames	5-96
GetCookieValue.....	5-96
GetHeader	5-97
GetHeaderNames	5-97
GetHelpURL	5-97
GetParameter.....	5-97
GetParameterNames.....	5-98
GetParameterValues.....	5-98
Request Class Properties	5-98
BrowserPlatform	5-98
BrowserType.....	5-98
BrowserVersion	5-98
ExpireMeta.....	5-99
FullURI	5-99
HTTPMethod	5-99
LogoutURL	5-100
PathInfo.....	5-100
Protocol	5-100
QueryString.....	5-100
RequestURI.....	5-100
Scheme	5-101
ServerName.....	5-101
ServerPort.....	5-102
Timeout	5-102

Response Class.....	5-102
Response Class Methods.....	5-102
Clear.....	5-102
CreateCookie.....	5-103
GetCookie.....	5-103
GetHeaderNames.....	5-103
GetImageURL.....	5-104
GetJavaScriptURL.....	5-104
GetStyleSheetURL.....	5-105
RedirectURL.....	5-105
SetContentType.....	5-106
SetHeader.....	5-106
Write.....	5-106
WriteLine.....	5-107
Cookie Class.....	5-107
Cookie Class Properties.....	5-107
Domain.....	5-107
MaxAge.....	5-108
Name.....	5-108
Path.....	5-108
Secure.....	5-108
Value.....	5-108
Java Class.....	5-109
Supported Versions of Java.....	5-109
Naming Classes and Packages.....	5-109
Setting up your System to use your own Classes.....	5-110
From PeopleCode to Java.....	5-111
State Management Concerns.....	5-111
CreateJavaObject Example.....	5-112
CreateJavaArray Example.....	5-113
GetJavaClass Example.....	5-114
From Java to PeopleCode.....	5-114
SysVar Java Class.....	5-114
SysCon Java Class.....	5-115
Func Java Class.....	5-115
Name Java Class.....	5-115
Using PeopleCode Objects.....	5-115
Mapping PeopleCode and Java Data Types.....	5-117
Considerations when using the PeopleCode Java Functions.....	5-118
Error Handling and the PeopleCode Java Functions.....	5-119

Using the Java Debugging Environment.....	5-119
Declaring a Java Object.....	5-120
Scope of a Java Object	5-120
PeopleCode Java Functions.....	5-120
Message Class.....	5-121
Application Messaging PeopleCode Events.....	5-121
Declaring a Message Object.....	5-122
Scope of a Message Object	5-122
Populating a Message Object	5-122
Considerations when Populating a Rowset from a Message.....	5-124
Considerations for Publishing and Subscribing to Partial Records.....	5-126
Considerations when Subscribing to Character Fields.....	5-127
Working with XML.....	5-127
Error Handling	5-127
Message Class Built-In Functions.....	5-128
Message Class Methods	5-129
Cancel.....	5-129
Clone	5-130
CopyRowset	5-131
CopyRowsetDelta	5-133
ExecuteEdits.....	5-135
GenXMLString	5-137
GenFormattedXMLString	5-138
GetFormattedRawXML	5-139
GetMessageVersion	5-140
GetRawXML.....	5-141
GetRowset.....	5-142
LoadXMLString.....	5-143
Publish.....	5-144
Resubmit	5-145
SetEditTable.....	5-147
Update	5-148
Message Class Properties	5-149
ChannelName	5-149
DoNotPubToNodeName	5-149
IsActive	5-149
IsDelta	5-150
IsEditError.....	5-151
IsLocal.....	5-152
Name	5-152

PubID	5-152
PubNodeName	5-153
Size	5-153
SubName	5-154
SubscriptionProcessId	5-154

Chapter 6

Page Class

Declaring a Page Object	6-1
Scope of a Page Object	6-1
Page Class Built-in Function	6-2
Page Class Properties	6-2
DisplayOnly	6-2
Name	6-2
Visible	6-2
PortalRegistry Classes	6-3
Portal Registry Overview	6-3
Folders	6-4
Content References	6-4
Content Providers	6-5
Security	6-5
Attributes	6-6
Using the PortalRegistry API	6-6
Security Considerations	6-6
Using Object Properties	6-6
Accessing Folders and Content References	6-7
Using PermissionValue and Cascading Permissions	6-7
Working with ValidFrom and ValidTo	6-9
Error Handling	6-10
Life-Cycle of a PortalRegistry Object	6-12
PortalRegistry Class Hierarchy	6-13
Using Content References	6-14
UsageType	6-14
StorageType	6-16
URLType	6-17
ContentProvider and URL	6-17
Summary	6-19
Naming Conventions	6-21
Considerations when Deleting Content	6-21
Considerations when Saving Content	6-22

Declaring a PortalRegistry Object.....	6-22
Scope of a PortalRegistry Object	6-23
PortalRegistry Reference.....	6-23
Session Object Methods.....	6-23
FindPortalRegistries	6-24
GetPortalRegistry	6-24
PortalRegistry Class	6-25
PortalRegistry Class Methods	6-25
Close.....	6-25
Create	6-26
Delete	6-27
FindCRefByName.....	6-28
FindCRefByURL	6-29
FindFolderByName.....	6-31
GetQualifiedURL.....	6-32
Open.....	6-33
Save.....	6-33
PortalRegistry Class Properties	6-34
ContentProviders.....	6-34
DefaultTemplate.....	6-34
Description	6-35
Name	6-35
RootFolder	6-35
TemplateObject.....	6-35
PortalRegistry Collection	6-35
PortalRegistry Collection Methods	6-35
First	6-35
Item	6-36
Next.....	6-36
PortalRegistry Collection Property	6-37
Count.....	6-37
ContentProvider Class.....	6-37
ContentProvider Class Properties.....	6-37
DefaultTemplate.....	6-37
Description	6-37
Name	6-38
TemplateObject.....	6-38
URI.....	6-38
ContentProvider Collection.....	6-39
ContentProvider Collection Methods.....	6-39

DeleteItem	6-39
First	6-40
InsertItem	6-40
ItemByName	6-41
ItemByURI.....	6-41
Next.....	6-42
ContentProvider Collection Properties.....	6-42
Count.....	6-42
Folder Class.....	6-42
Folder Class Method	6-43
Save.....	6-43
Folder Class Properties	6-43
Attributes.....	6-43
Author	6-44
AuthorAccess	6-44
Authorized.....	6-44
CascadedPermissions	6-44
ContentRefs.....	6-45
CreationDate	6-45
Description	6-45
Folders.....	6-45
Label.....	6-45
Name	6-45
ParentName.....	6-46
Path.....	6-46
Permissions	6-46
Product	6-46
SequenceNumber	6-46
PublicAccess	6-47
ValidFrom	6-47
ValidTo	6-47
Folder Collection.....	6-47
Folder Collection Methods.....	6-47
DeleteItem.....	6-47
First	6-48
InsertItem	6-49
ItemByName	6-49
Next.....	6-50
Folder Collection Property	6-51
Count.....	6-51

Content Reference Class	6-51
Content Reference Class Methods	6-51
Save	6-51
Content Reference Class Properties	6-52
Attributes	6-52
Author	6-52
AuthorAccess	6-52
Authorized	6-52
CascadedPermissions	6-53
ContentProvider	6-53
CreationDate	6-53
Data	6-53
Description	6-54
Label	6-54
Name	6-54
ParentName	6-54
Path	6-54
Permissions	6-55
Product	6-55
PublicAccess	6-55
QualifiedURL	6-55
SequenceNumber	6-55
StorageType	6-56
Template	6-56
TemplateObject	6-57
TemplateType	6-57
URL	6-57
URLType	6-58
UsageType	6-58
ValidFrom	6-59
ValidTo	6-59
Content Reference Collection	6-59
Content Reference Collection Methods	6-60
DeleteItem	6-60
First	6-60
InsertItem	6-61
ItemByName	6-62
Next	6-63
Content Reference Collection Property	6-63
Count	6-63

AttributeValue Class	6-63
AttributeValue Class Properties	6-64
Label.....	6-64
Name	6-64
Translatable	6-65
Value	6-65
Attribute Collection.....	6-65
Attribute Collection Methods.....	6-65
DeleteItem.....	6-65
First	6-66
InsertItem	6-66
ItemByName	6-67
Next.....	6-68
Attribute Collection Property	6-68
Count.....	6-68
PermissionValue Class.....	6-68
PermissionValue Class Properties.....	6-69
Cascade	6-69
Name	6-69
PermissionValue Collection.....	6-69
PermissionValue Collection Methods.....	6-70
DeleteItem.....	6-70
First	6-70
InsertItem	6-71
ItemByName	6-72
Next.....	6-72
PermissionValue Collection Property	6-73
Count.....	6-73
PortalRegistry Examples	6-73
Changing PortalRegistry Properties	6-73
Adding a ContentProvider.....	6-75
Changing ContentProvider Properties.....	6-77
Adding a Folder	6-81
Adding a Content Reference	6-85
Setting Permissions using the PermissionValue Object.....	6-88
Using Attributes	6-91
Visual Basic Example	6-92
C/C++ Example.....	6-93
ProcessRequest Class.....	6-99

Restrictions on Use in PeopleSoft Internet Architecture, Three-Tier Mode and Windows Client.....	6-100
Example.....	6-100
Declaring a ProcessRequest Object.....	6-102
Scope of a ProcessRequest Object	6-102
ProcessRequest Class built-in Function	6-102
ProcessRequest Class Method.....	6-103
Schedule	6-103
ProcessRequest Class Properties	6-104
EmailAttachLog	6-104
EmailSubject	6-104
EmailText.....	6-104
JobName.....	6-104
LanguageCd	6-105
OutDest	6-105
OutDestFormat.....	6-106
OutDestType	6-108
ProcessInstance	6-109
ProcessName	6-109
ProcessSeq	6-109
ProcessType	6-110
RunControlID.....	6-111
RunDateTime	6-111
RunLocation.....	6-111
RunRecurrence.....	6-112
SubmitJobItem	6-112
Status	6-119
TimeZone	6-120
Query Classes	6-120
Query Classes Overview	6-121
Query.....	6-121
QueryRecords.....	6-121
QueryOutputFields.....	6-122
QuerySelectedFields	6-122
QueryCriteria	6-123
QueryDBRecords and QueryDBRecordFields.....	6-124
Understanding QueryOutputFields and QuerySelectedFields.....	6-125
Collections in the Query Classes.....	6-125
Life-Cycle of a Query	6-126
Query Class Hierarchy	6-127

Working with Criteria and Expressions	6-128
Setting the Type	6-129
Considerations Adding New Expressions	6-129
Operator and Expression 2 Dependencies.....	6-129
Using Metadata	6-130
Running a Query	6-132
Query Security	6-132
Error Handling with Query Classes	6-133
Declaring Query Objects.....	6-134
Scope of the Query Objects.....	6-134
Session Object Methods	6-135
FindQueryDBRecords.....	6-135
FindQueries.....	6-136
FindQueriesByDateRange.....	6-136
GetQuery	6-137
Query Collection	6-138
Query Collection Methods	6-138
First	6-138
Item	6-139
ItemByName	6-139
Next.....	6-140
Query Collection Property	6-140
Count.....	6-140
Query Class	6-141
Query Class Methods	6-141
AddAllFields	6-141
AddCriteria.....	6-141
AddExpression	6-142
AddQueryOutputField.....	6-142
AddQueryRecord	6-143
AddQuerySelectedField	6-143
Close.....	6-144
Create	6-145
Delete	6-147
DeleteCriteria	6-147
DeleteExpression	6-148
DeleteField.....	6-148
Open	6-149
QueryOutputFields.....	6-149
QueryRecords.....	6-150

QuerySelectedFields	6-150
Rename.....	6-150
Save.....	6-151
Query Class Properties	6-152
Criteria	6-152
Description	6-152
Distinct	6-152
Expressions	6-152
LongDescription.....	6-152
Metadata.....	6-152
Name	6-153
PermissionList.....	6-153
Public	6-153
SQL	6-153
Type	6-154
QueryRecord Collection.....	6-154
QueryRecord Collection Methods.....	6-154
First	6-154
Item	6-155
ItemByName	6-155
Next.....	6-156
QueryRecord Collection Property	6-156
Count.....	6-156
QueryRecord Class.....	6-156
QueryRecord Class Properties	6-157
Alias	6-157
Name	6-157
QueryFields.....	6-157
RecordHierachy.....	6-157
QueryField Collection	6-157
QueryField Collection Methods	6-158
First	6-158
Item	6-158
ItemByName	6-159
Next.....	6-159
QueryField Collection Property	6-160
Count.....	6-160
QueryField Class.....	6-160
QueryField Class Properties.....	6-160
Aggregate	6-160

ColumnNumber.....	6-161
Format	6-161
HeadingText.....	6-161
HeadingType.....	6-161
HeadingUniqueFieldName.....	6-162
Name	6-162
OrderByDirection.....	6-162
OrderByNumber.....	6-162
QueryDBRecordField	6-162
QueryRecord	6-163
QueryRelatedRecords	6-163
TranslateEffDtLogic	6-163
TranslateOption.....	6-163
Type	6-163
QueryCriteria Collection	6-164
QueryCriteria Collection Methods	6-164
First	6-164
Item	6-164
Next.....	6-165
QueryCriteria Collection Property	6-166
Count.....	6-166
QueryCriteria Class	6-166
QueryCriteria Class Methods	6-166
AddExpr1Expression	6-166
AddExpr1Field.....	6-167
AddExpr2Expression	6-167
AddExpr2Field.....	6-168
AddExpr2Subquery.....	6-168
QueryCriteria Class Properties.....	6-169
Expr1Expression	6-169
Expr1Field.....	6-169
Expr1Type.....	6-169
Expr2Constant.....	6-171
Expr2Expression	6-171
Expr2Field.....	6-171
Expr2Subquery.....	6-171
Expr2Type.....	6-171
Logical	6-174
Operator	6-174
QueryExpression Collection	6-175

QueryExpression Collection Methods	6-175
First	6-175
Item	6-176
Next.....	6-176
QueryExpression Collection Property.....	6-177
Count.....	6-177
QueryExpression Class	6-177
QueryExpression Class Properties	6-178
Aggregate	6-178
Decimal	6-178
Length	6-178
Text	6-178
Type	6-178
QueryRecordHierarchy Collection.....	6-179
QueryRecordHierarchy Collection Methods.....	6-179
First	6-179
Item	6-179
ItemByName	6-180
Next.....	6-181
QueryRecordHierarchy Collection Property	6-181
Count.....	6-181
QueryRecordHierarchy Class.....	6-181
QueryRecordHierarchy Class Properties.....	6-182
Description	6-182
Level.....	6-182
Name	6-182
ParentFlag	6-182
Metadata Collection	6-182
Metadata Collection Methods	6-183
First	6-183
Item	6-183
ItemByName	6-184
Next.....	6-184
Metadata Collection Property.....	6-185
Count.....	6-185
Metadata Class	6-185
Metadata Class Properties	6-185
Name	6-185
Value	6-186
PermissionList Collection	6-186

PermissionList Collection Methods	6-186
First	6-186
Item	6-187
ItemByName	6-187
Next.....	6-188
PermissionList Collection Property	6-188
Count.....	6-188
PermissionList Class	6-188
PermissionList Class Property	6-189
Name	6-189
QueryDBRecord Collection	6-189
QueryDBRecord Collection Methods	6-189
First	6-189
Item	6-190
ItemByName	6-190
Next.....	6-191
QueryDBRecord Collection Property	6-191
Count.....	6-191
QueryDBRecord Class	6-191
QueryDBRecord Class Methods	6-192
QueryDBRecordFieldByIndex.....	6-192
QueryDBRecordFieldByName	6-192
QueryDBRecord Class Properties	6-193
Description	6-193
Name	6-193
QueryDBRecordFields.....	6-193
QueryDBRecordField Collection	6-193
QueryDBRecordField Collection Methods	6-193
First	6-193
Item	6-194
ItemByName	6-194
Next.....	6-195
QueryDBRecordField Collection Property	6-195
Count.....	6-195
QueryDBRecordField Class.....	6-195
QueryDBRecordField Class Properties.....	6-196
Decimal	6-196
Format	6-196
Length	6-196
LongName.....	6-197

Name	6-197
Shortname	6-197
Type	6-197
Record Class	6-197
Declaring a Record Object	6-199
Scope of a Record Object	6-199
Record Class Built-in Functions	6-199
Record Class Methods	6-199
CompareFields	6-199
CopyChangedFieldsTo	6-200
CopyFieldsTo	6-201
Delete	6-203
ExecuteEdits	6-204
GetField	6-207
Insert	6-209
SelectByKey	6-211
SetDefault	6-212
SetEditTable	6-213
Update	6-215
Record Class Properties	6-218
FieldCount	6-218
<i>fieldname</i> property	6-218
IsChanged	6-218
IsDeleted	6-219
IsEditError	6-219
Name	6-220
ParentRow	6-220
RelLangRecName	6-220
Row Class	6-221
Declaring a Row Object	6-221
Scope of a Row Object	6-222
Row Class Built-in Function	6-222
Row Class Methods	6-222
CopyTo	6-222
GetNextEffRow	6-223
GetPriorEffRow	6-224
GetRecord	6-224
GetRowset	6-226
<i>scrollname</i>	6-227
Row Class Properties	6-228

ChildCount	6-228
DeleteEnabled	6-228
IsChanged.....	6-229
IsDeleted	6-230
IsEditError.....	6-230
IsNew	6-231
ParentRowset	6-231
<i>recname</i> property	6-232
RecordCount	6-232
RowNumber	6-232
Selected	6-233
Style	6-233
Visible	6-234
Rowset Class.....	6-235
Declaring a Rowset Object.....	6-236
Scope of a Rowset Object	6-236
Rowset Class Built-in Functions	6-236
Rowset Class Methods	6-237
CopyTo.....	6-237
DeleteRow	6-239
Fill	6-241
FillAppend	6-243
Flush.....	6-245
FlushRow	6-246
GetCurrEffRow	6-247
GetRow	6-247
HideAllRows.....	6-248
InsertRow	6-249
Refresh	6-250
Select.....	6-251
SelectNew	6-254
SetDefault.....	6-255
ShowAllRows	6-256
Sort.....	6-257
Rowset Class Properties	6-259
ActiveRowCount.....	6-259
DBRecordName	6-259
DeleteEnabled	6-259
EffDt.....	6-260
EffSeq.....	6-260

InsertEnabled	6-261
IsEditError	6-262
Level	6-262
Name	6-263
ParentRow	6-263
ParentRowset	6-263
RowCount	6-264
TopRowNumber	6-264

Chapter 7

Search Classes

Search Overview	7-1
Using the SearchResults Collection	7-2
Understanding SearchResults	7-2
Search Classes Hierarchy	7-3
Error Handling	7-4
Declaring Search Objects	7-6
Scope of the Search Objects	7-6
Session Class Method	7-6
GetSearchQuery	7-7
PortalRegistry Class Methods	7-7
BuildSearchIndex	7-7
GetSearchQuery	7-8
SearchQuery Class	7-8
SearchQuery Class Method	7-9
Execute	7-9
SearchQuery Class Properties	7-9
HitCount	7-9
IndexName	7-10
Language	7-10
ProcessedCount	7-10
QueryText	7-10
SearchResult Collection	7-10
SearchResult Collection Methods	7-11
First	7-11
Item	7-11
Next	7-12
SearchResult Collection Property	7-12
Count	7-12
SearchResult Class	7-12

SearchResult Class Properties	7-12
Key	7-12
Score	7-13
SearchFields	7-13
SearchField Collection	7-13
SearchField Collection Methods	7-13
First	7-13
ItemByName	7-13
Next	7-14
SearchField Collection Property	7-14
Count	7-14
SearchField Class	7-14
SearchField Class Properties	7-14
Name	7-14
Value	7-15
Search API Examples	7-15
General Purpose Search API Example	7-15
Portal Search API Example	7-17
Session Class	7-19
Security and Access to the PeopleSoft System	7-20
About Error Handling	7-20
About Regional Settings	7-22
About Trace Settings	7-22
Session Class Reference	7-23
Declaring a Session Object	7-24
Scope of a Session Object	7-25
Session Class Built-in Function	7-25
Session Class Methods	7-25
Connect	7-25
Disconnect	7-27
API Instantiation Methods	7-27
Session Class Properties	7-28
ErrorPending	7-28
PSMessages	7-29
RegionalSettings	7-29
Repository	7-29
SuspendFormatting	7-29
TraceSettings	7-30
WarningPending	7-30
Accessing a PSMessages Collection	7-30

PSMessages Collection Methods	7-31
DeleteAll	7-31
DeleteItem	7-31
First	7-32
Item	7-32
Next.....	7-33
PSMessages Collection Property	7-33
Count.....	7-33
PSMessage Class Properties.....	7-33
ExplainText.....	7-33
MessageNumber.....	7-34
MessageSetNumber.....	7-34
Source.....	7-34
Text	7-36
Regional Settings Class Properties.....	7-37
ClientTimeZone	7-37
CurrencyFormat	7-37
CurrencySymbol	7-37
DateFormat.....	7-38
DateSeparator.....	7-38
DecimalSymbol.....	7-38
DigitGroupingSymbol.....	7-38
LanguageCD	7-39
UseLocalTime	7-40
1159Separator	7-40
2359 Separator	7-40
Trace Setting Class Properties.....	7-40
API	7-41
COBOLStmtTimings	7-41
ConnDisRollbackCommit	7-41
DBSpecificCalls.....	7-41
ManagerInfo.....	7-41
NetworkServices	7-41
NonSSBs	7-42
OutputUNICODE.....	7-42
PCExtFcnCalls	7-42
PCFcnReturnValues.....	7-42
PCFetchedValues	7-42
PCIntFcnCalls	7-42
PCListProgram.....	7-43

PCParameterValues.....	7-43
PCProgramStatements.....	7-43
PCStack.....	7-43
PCStartOfPrograms.....	7-43
PCTraceProgram.....	7-43
PCVariableAssignments	7-44
RowFetch	7-44
SQLStatement	7-44
SQLStatementVariables.....	7-44
SSBs.....	7-44
SybBindInfo	7-44
SybFetchInfo.....	7-45
TraceFile	7-45
SQL Class	7-45
Record Class SQL	7-46
Creating a SQL Definition	7-46
Binding and Executing of SQL Statements.....	7-46
Fetching from a Select.....	7-48
Different "Styles" of SELECT	7-48
Reusing a Cursor	7-49
Global SQL Objects and Application Engine Programs	7-51
Declaring a SQL Object	7-51
Scope of an SQL Object.....	7-51
SQL Class Built-in Functions	7-52
SQL Class Methods.....	7-52
Close.....	7-52
Execute.....	7-53
Fetch.....	7-55
Open	7-57
SQL Class Properties	7-58
BulkMode.....	7-58
IsOpen	7-60
RowsAffected.....	7-60
Status	7-60
TraceName	7-61
Value	7-63
Tree Classes.....	7-63
Relationships between Different Tree Classes.....	7-65
Collections in the Tree Classes	7-66
Error Handling with Trees.....	7-67

Verifying Leaves and Nodes being Inserted	7-69
Declaring Tree Objects.....	7-69
Scope of the Tree Objects	7-70
Implementing Tree Classes	7-70
Session Object Methods	7-75
FindTree	7-75
FindTreeStructure	7-77
GetTree.....	7-78
GetTreeStructure	7-79
Branch Collections	7-79
Branch Collection Method	7-80
Item	7-80
Branch Collection Properties	7-81
Count.....	7-81
First	7-81
Last.....	7-81
Next.....	7-81
Leaf Class.....	7-81
Leaf Class Methods.....	7-81
Cut.....	7-81
Delete	7-82
DeleteByRange	7-82
InsertDynSib	7-83
InsertSib	7-84
MoveAsChild	7-84
MoveAsChildByName.....	7-85
MoveAsSib.....	7-86
MoveAsSibByRange.....	7-86
PasteSib	7-87
Leaf Class Properties.....	7-88
Dynamic	7-88
HasNextSib	7-88
HasPrevSib.....	7-88
IsChanged.....	7-88
IsDeleted	7-88
IsInserted.....	7-89
NextSib.....	7-89
Parent	7-89
PrevSib	7-89
RangeFrom.....	7-90

RangeTo	7-90
TreeBranchName	7-90
TreeEffDt	7-90
TreeName	7-90
TreeSetId	7-90
TreeUserKeyValue	7-91
Level Collection	7-91
LevelCollection Methods	7-91
Add	7-91
Item	7-91
Remove	7-92
Save	7-92
Level Collection Properties	7-93
Count	7-93
First	7-93
Last	7-93
Next	7-93
Level Class	7-93
Level Class Methods	7-93
Create	7-93
Delete	7-94
Level Class Properties	7-94
AllValuesAudit	7-94
Description	7-95
Name	7-95
Number	7-95
TreeBranchName	7-95
TreeEffDt	7-95
TreeName	7-95
TreeSetId	7-95
TreeUserKeyValue	7-96
Node Class	7-96
Node Class Methods	7-96
Branch	7-96
Cut	7-97
Delete	7-97
DeleteByName	7-97
Expand	7-98
InsertChildLeaf	7-98
InsertChildNode	7-99

InsertChildRecord	7-100
InsertDynChildLeaf.....	7-101
InsertSib	7-101
InsertSibRecord.....	7-102
MoveAsChild	7-102
MoveAsChildByName	7-103
MoveAsSib.....	7-104
MoveAsSibByName	7-105
PasteChild	7-105
PasteSib.....	7-106
Rename.....	7-106
SwitchLevel	7-107
Unbranch.....	7-107
Node Class Properties	7-108
AllChildCount	7-108
AllChildNodeCount	7-108
ChildLeafCount.....	7-108
ChildNodeCount	7-108
Description.....	7-108
FirstChildLeaf	7-108
FirstChildNode.....	7-109
HasChildLeaves	7-109
HasChildNodes	7-109
HasChildren	7-109
HasNextSib	7-109
HasPrevSib.....	7-110
IsBranched	7-110
IsChanged.....	7-110
IsDeleted	7-110
IsInserted.....	7-110
IsRoot	7-110
LastChildLeaf.....	7-111
LastChildNode	7-111
LevelNumber.....	7-111
Name	7-111
NextSib.....	7-111
Parent	7-112
PrevSib.....	7-112
State.....	7-112
TreeBranchName	7-113

TreeEffDt	7-113
TreeName	7-113
TreeSetId	7-113
TreeUserKeyValue	7-113
Type	7-114
Tree Class	7-114
Tree Class Methods	7-115
Audit	7-115
AuditByName	7-115
Close	7-116
Copy	7-117
Create	7-119
Delete	7-120
Exists	7-121
FindLeaf	7-122
FindNode	7-123
FindRoot	7-124
InsertRoot	7-125
LeafExists	7-125
Open	7-126
NodeExists	7-128
Rename	7-128
Save	7-130
SaveAs	7-130
SaveAsDraft	7-132
SaveDraft	7-133
Tree Class Properties	7-134
AllValues	7-134
AuditDetails	7-134
Branches	7-134
BranchLevel	7-135
BranchName	7-135
Category	7-135
CheckLeafUserData	7-135
Description	7-136
DuplicateLeaves	7-136
EffDt	7-136
HasDetailRanges	7-136
IsBranched	7-137
IsChanged	7-137

IsOpen	7-137
IsQueryTree	7-138
IsValid.....	7-138
KeyBranchName.....	7-138
KeyEffDt.....	7-138
KeyName.....	7-139
KeySetId.....	7-139
KeyUserKeyValue	7-139
LeafCount.....	7-139
LevelCount.....	7-139
Levels	7-140
LevelUse	7-140
Name	7-140
NodeCount	7-141
ParentLevel	7-141
ParentName	7-141
PerformanceMethod	7-141
PerformanceSelector	7-142
PerformanceSelectorOption	7-142
SetID	7-143
SilentMode.....	7-143
Status.....	7-143
Structure	7-143
StructureName.....	7-144
TotalNodeCount.....	7-144
UserKeyValue	7-144
Tree Collection.....	7-145
Tree Collection Method	7-146
Item	7-146
Tree Collection Properties.....	7-147
Count.....	7-147
First	7-148
Last.....	7-148
Next.....	7-148
Tree Structure Class	7-148
Tree Structure Class Methods	7-148
Close.....	7-148
Copy	7-149
Create	7-149
Delete	7-150

Open	7-150
Rename.....	7-151
Save	7-151
Tree Structure Class Properties	7-151
Description	7-151
DetailComponent	7-152
DetailField.....	7-152
DetailMenu.....	7-152
DetailMenuBar	7-153
DetailMenuItem	7-153
DetailPage	7-153
DetailRecord	7-154
IndirectionMethod.....	7-154
KeyName.....	7-155
LevelComponent	7-156
LevelMenu	7-156
LevelMenuBar	7-156
LevelMenuItem	7-157
LevelPage.....	7-157
LevelRecord	7-157
Name	7-157
NodeComponent	7-158
NodeField.....	7-158
NodeMenu.....	7-158
NodeMenuBar	7-158
NodeMenuItem	7-159
NodePage	7-159
NodeRecord	7-159
NodeUserKeyField.....	7-160
SummarySetId.....	7-160
SummaryLevelNumber	7-160
SummaryTreeName	7-160
SummaryUserKeyValue	7-161
Type	7-161
Tree Structure Collection	7-161
Tree Structure Collection Method.....	7-162
Item	7-162
Tree Structure Collection Properties	7-162
Count.....	7-162
First	7-162

Last.....	7-162
Next.....	7-163
Summary of Class Methods and Properties	7-163
Summary of Methods for Tree Classes	7-163
Summary of Properties for Tree Classes.....	7-164

Chapter 8

ActiveX Controls in PeopleTools

Chart Control	8-1
Chart Objects.....	8-2
Setting Indexed Properties	8-4
Additional Properties	8-5
Setting or Returning a Data Point	8-5
Using the DataGrid	8-6
Manipulating the Chart Appearance	8-7
Formatting Fonts.....	8-8
Change the Scale Using the Type Property.....	8-8
Three-Dimensional Features of the Chart Control	8-9
Rotate the Chart	8-9
Seeing the Light	8-10
Light Basics.....	8-10
Ambient Light	8-10
Using the Add Method to Add a LightSource.....	8-11
Using the OnPointActivated Event to Change a Data Point	8-12
Definitions of Constants.....	8-12
VtBorderStyle	8-12
VtChAxisId	8-13
VtChChartType.....	8-13
VtChLocationType.....	8-14
VtChMouseFlag	8-15
VtChPartType	8-15
VtChSeriesType	8-16
VtChStats	8-17
VtChUpdateFlags.....	8-17
VtMousePointer	8-18
VtPenStyle	8-19
VtTextLengthType.....	8-19
PeopleCode Events.....	8-20
Declaring a Chart Object.....	8-30
Scope of a Chart Object	8-30

Chart Properties.....	8-31
ActiveSeriesCount	8-31
AllowDithering	8-31
AllowDynamicRotation	8-31
AllowSelections	8-31
AllowSeriesSelection	8-32
AutoIncrement.....	8-32
Backdrop	8-32
BorderStyle	8-32
Chart3d.....	8-33
ChartType.....	8-33
Column.....	8-33
ColumnCount	8-34
ColumnLabel.....	8-34
ColumnLabelCount	8-34
ColumnLabelIndex.....	8-34
Data	8-35
DataGrid.....	8-35
DoSetCursor.....	8-35
DrawMode	8-36
Enabled.....	8-36
Footnote	8-36
FootnoteText	8-37
Legend.....	8-37
MousePointer	8-37
OLEDragMode.....	8-37
OLEDropMode	8-38
Plot	8-38
RandomFill.....	8-38
Repaint	8-39
Row	8-39
RowCount	8-40
RowLabel.....	8-40
RowLabelCount	8-40
RowLabelIndex.....	8-40
SeriesColumn	8-40
SeriesType.....	8-40
ShowLegend.....	8-41
Stacking.....	8-41
TextLengthType.....	8-41

Title	8-41
TitleText.....	8-41
Chart Methods	8-42
EditCopy	8-42
EditPaste.....	8-42
Layout	8-42
OLEDrag.....	8-43
Refresh	8-43
SelectPart	8-43
ToDefaults.....	8-44
Axis Object.....	8-45
Axis Properties	8-45
AxisGrid.....	8-45
AxisScale	8-45
AxisTitle.....	8-45
CategoryScale	8-45
Intersection.....	8-45
LabelLevelCount.....	8-45
Labels	8-46
Pen.....	8-46
Tick	8-46
ValueScale	8-46
AxisGrid Object	8-46
AxisGrid Properties.....	8-46
MinorPen.....	8-46
MajorPen.....	8-47
AxisScale Object.....	8-47
AxisScale Properties	8-47
Hide.....	8-47
LogBase	8-47
PercentBasis	8-47
Type	8-48
AxisTitle Object	8-49
AxisTitle Properties	8-49
Backdrop	8-49
Font	8-49
Text	8-49
TextLayout.....	8-50
TextLength.....	8-50
Visible	8-50

VtFont	8-50
Backdrop Object.....	8-50
Backdrop Properties	8-50
Fill	8-50
Frame	8-51
Shadow.....	8-51
Brush Object.....	8-51
Brush Properties	8-51
FillColor.....	8-51
Index.....	8-51
PatternColor	8-52
Style	8-52
CategoryScale Object.....	8-52
CategoryScale Properties	8-52
Auto.....	8-52
DivisionsPerLabel.....	8-53
DivisionsPerTick.....	8-53
LabelTick	8-53
Coor Object	8-53
Coor Properties.....	8-53
X.....	8-53
Y	8-54
Coor Method	8-54
Set.....	8-54
DataGrid Object	8-54
DataGrid Properties.....	8-54
ColumnCount	8-54
ColumnLabelCount.....	8-55
RowCount	8-55
RowLabelCount	8-55
DataGrid Methods.....	8-55
ColumnLabel.....	8-55
CompositeColumnLabel	8-56
CompositeRowLabel.....	8-57
DeleteColumnLabels.....	8-57
DeleteColumns.....	8-58
DeleteRowLabels	8-59
DeleteRows	8-59
GetData	8-60
InitializeLabels.....	8-61

MoveData.....	8-61
RandomDataFill	8-62
RandomFillColumns	8-62
RandomFillRows	8-63
SetData	8-64
SetSize.....	8-65
DataGrid Special Property	8-66
RowLabel	8-66
DataPoint Object	8-67
DataPoint Properties.....	8-67
Brush	8-67
DataPointLabel.....	8-67
EdgePen	8-67
Marker.....	8-68
Offset.....	8-68
DataPoint Methods.....	8-68
ResetCustom	8-68
Select.....	8-68
DataPointLabel Object	8-69
DataPointLabel Properties.....	8-69
Backdrop	8-69
Component.....	8-69
Custom	8-70
Font	8-70
LineStyle	8-70
LocationType	8-71
Offset.....	8-71
PercentFormat	8-71
Text	8-71
TextLayout	8-71
TextLength.....	8-72
ValueFormat.....	8-72
VtFont	8-72
DataPointLabel Methods.....	8-72
ResetCustomLabel	8-72
Select.....	8-72
DataPoints Object.....	8-73
DataPoints Properties	8-73
Item	8-73
Count.....	8-73

Fill Object.....	8-73
Fill Properties.....	8-74
Brush.....	8-74
Style.....	8-74
Font Object.....	8-74
Font Object Properties.....	8-75
Bold.....	8-75
Charset.....	8-75
Italic.....	8-75
Name.....	8-75
Size.....	8-76
Strikethrough.....	8-76
Underline.....	8-76
Weight.....	8-76
Footnote Object.....	8-76
Footnote Properties.....	8-76
Backdrop.....	8-76
Font.....	8-76
Location.....	8-77
Text.....	8-77
TextLayout.....	8-77
TextLength.....	8-77
VtFont.....	8-77
Footnote Method.....	8-77
Select.....	8-77
Frame Object.....	8-78
Frame Properties.....	8-78
FrameColor.....	8-78
SpaceColor.....	8-78
Style.....	8-78
Width.....	8-79
Intersection Object.....	8-79
Intersection Properties.....	8-79
Auto.....	8-79
AxisID.....	8-79
Index.....	8-79
LabelsInsidePlot.....	8-80
Point.....	8-80
Label Object.....	8-80
Label Properties.....	8-80

Auto.....	8-80
Backdrop	8-80
Font	8-81
Format	8-81
FormatLength.....	8-81
Standing	8-81
TextLayout.....	8-81
VtFont	8-81
Labels Collection	8-82
Labels Collection Properties	8-82
Count.....	8-82
Item	8-82
LCoor Object.....	8-82
LCoor Properties	8-82
X.....	8-82
Y	8-82
LCoor Method.....	8-83
Set.....	8-83
Legend Object	8-83
Legend Properties.....	8-83
Backdrop	8-83
Font	8-83
Location	8-83
TextLayout.....	8-84
VtFont	8-84
Legend Method	8-84
Select.....	8-84
Light Object	8-84
Light Properties	8-84
AmbientIntensity.....	8-84
EdgeIntensity.....	8-85
EdgeVisible	8-85
LightSources	8-85
Lightsource Object	8-85
Lightsource Properties.....	8-85
Intensity.....	8-85
X.....	8-86
Y	8-86
Z	8-86
Lightsource Method	8-86

Set.....	8-86
Lightsources Collection.....	8-87
Lightsources Properties	8-87
Count.....	8-87
Item	8-87
Lightsources Methods	8-87
Add.....	8-87
Remove	8-88
Location Object.....	8-88
Location Properties	8-88
LocationType	8-88
Rect	8-89
Visible	8-89
Marker Object	8-89
Marker Properties.....	8-89
FillColor.....	8-89
Pen.....	8-89
Size.....	8-89
Style	8-90
Visible	8-90
Pen Object.....	8-91
Pen Properties.....	8-91
Cap	8-91
Join.....	8-92
Limit.....	8-92
Style	8-92
VtColor.....	8-93
Width.....	8-93
Plot Object.....	8-93
Plot Properties	8-93
AngleUnit.....	8-93
AutoLayout	8-94
Backdrop	8-94
BarGap	8-94
Clockwise.....	8-95
DataSeriesInRow.....	8-95
DefaultPercentBasis	8-95
DepthToHeightRatio	8-95
Light	8-95
LocationRect	8-96

PlotBase	8-96
Projection	8-96
SeriesCollection	8-97
Sort	8-97
StartingAngle	8-97
SubPlotLabelPosition	8-98
UniformAxis	8-98
View3d	8-99
Wall	8-99
Weighting	8-99
WidthToHeightRatio	8-99
XGap	8-99
ZGap	8-99
Additional Plot Property	8-100
Axis	8-100
Plotbase Object	8-100
Plotbase Properties	8-100
BaseHeight	8-100
Brush	8-101
Pen	8-101
Rect Object	8-101
Rect Properties	8-101
Min	8-101
Max	8-101
Series Object	8-101
Series Properties	8-102
DataPoints	8-102
GuideLinePen	8-102
LegendText	8-102
Pen	8-102
Position	8-102
SecondaryAxis	8-102
SeriesMarker	8-103
SeriesType	8-103
ShowLine	8-103
StatLine	8-103
Series Special Properties	8-103
ShowGuideLine	8-104
TypeByChartType	8-104
Series Method	8-105

Select.....	8-105
SeriesCollection Collection.....	8-105
SeriesCollection Property.....	8-105
Item	8-105
SeriesCollection Method.....	8-105
Count.....	8-105
SeriesMarker Object.....	8-105
SeriesMarker Properties	8-106
Auto.....	8-106
Show.....	8-106
SeriesPosition Object	8-106
SeriesPosition Properties.....	8-106
Excluded.....	8-106
Hidden.....	8-107
Order	8-107
StackOrder	8-107
Shadow Object	8-107
Shadow Properties.....	8-108
Brush	8-108
Offset.....	8-108
Style	8-108
StatLine Object.....	8-108
StatLine Properties	8-109
Flag.....	8-109
VtColor.....	8-109
Width.....	8-109
StatLine Special Property.....	8-109
Style	8-109
TextLayout Object	8-110
TextLayout Properties	8-110
HorzAlignment.....	8-110
Orientation	8-111
VertAlignment.....	8-111
WordWrap.....	8-112
Tick Object.....	8-112
Tick Properties	8-112
Length	8-112
Style	8-112
Title Object.....	8-113
Title Properties	8-113

Backdrop	8-113
Font	8-113
Location	8-113
Text	8-113
TextLayout	8-113
TextLength	8-114
VtFont	8-114
Title Method.....	8-114
Select.....	8-114
ValueScale Object.....	8-114
ValueScale Properties	8-114
Auto.....	8-114
MajorDivision	8-115
Maximum	8-115
Minimum.....	8-116
MinorDivision	8-117
View3D Object.....	8-117
View3D Properties	8-117
Elevation	8-117
Rotation.....	8-117
View3D Method.....	8-118
Set.....	8-118
VtColor Object.....	8-118
VtColor Properties	8-119
Automatic.....	8-119
Blue	8-119
Green.....	8-119
Red	8-119
VtColor Method	8-120
Set.....	8-120
VtFont Object.....	8-120
VtFont Properties	8-121
Effect.....	8-121
Name	8-121
Size.....	8-121
Style	8-121
VtColor.....	8-122
Wall Object	8-122
Wall Properties.....	8-122
Brush	8-122

Pen.....	8-122
Width.....	8-122
Weighting Object	8-123
Weighting Properties.....	8-123
Basis	8-123
Style	8-124
Weighting Method	8-124
Set.....	8-124

Chapter 9

TreeView

Associating a TreeView with an ImageList	9-3
Setting Indexed Properties	9-3
TreeView Events	9-4
Declaring a TreeView Object.....	9-9
Scope of a TreeView Object	9-9
TreeView Properties.....	9-10
Appearance.....	9-10
BorderStyle	9-10
Checkboxes	9-10
DropHighlight	9-11
Enabled.....	9-11
Font	9-11
FullRowSelect	9-11
HideSelection	9-12
HotTracking	9-12
ImageList	9-12
Indentation	9-12
LabelEdit.....	9-13
LineStyle	9-13
MousePointer	9-13
Nodes	9-14
OLEDragMode.....	9-14
OLEDropMode	9-15
PathSeparator	9-15
Scroll	9-16
SelectedItem.....	9-16
SingleSel	9-16
Sorted	9-16
Style	9-16

TreeView Methods	9-17
GetVisibleCount.....	9-17
HitTest.....	9-17
StartLabelEdit	9-17
Font Object.....	9-18
Font Properties	9-18
Bold.....	9-18
Charset	9-18
Italic	9-18
Name	9-19
Size.....	9-19
Strikethrough.....	9-19
Underline.....	9-19
Weight.....	9-19
Node Object	9-19
Using the Index Property	9-20
Using the Key Property	9-21
Node Properties.....	9-21
Bold.....	9-21
Checked.....	9-21
Child.....	9-21
Children.....	9-22
Expanded.....	9-22
ExpandedImage.....	9-22
FirstSibling.....	9-22
FullPath	9-22
Image.....	9-22
Index.....	9-23
Key	9-23
LastSibling	9-23
Next.....	9-23
Parent	9-23
Previous.....	9-24
Root.....	9-24
Selected	9-24
SelectedImage	9-24
Sorted	9-24
Tag	9-24
Text	9-24
Visible	9-25

Node Methods	9-25
CreateDragImage	9-25
EnsureVisible	9-25
Nodes Collection.....	9-25
Nodes Property.....	9-26
Count.....	9-26
Nodes Methods.....	9-26
Add.....	9-26
Clear	9-28
Item	9-28
Remove	9-28
Picture Object.....	9-28
Picture Properties	9-29
Height.....	9-29
Type	9-29
Width.....	9-29
ImageList	9-29
Declaring ImageList Objects.....	9-30
Scope of ImageList Objects	9-30
ImageList Properties	9-30
ImageHeight.....	9-30
ImageWidth.....	9-31
ListImages	9-31
MaskColor.....	9-31
UseMaskColor	9-31
ImageList Method	9-31
Overlay.....	9-31
Images Collection.....	9-32
Images Properties	9-32
Count.....	9-32
Images Methods	9-32
Add.....	9-32
Clear	9-33
Item	9-33
Remove	9-33
Image Object	9-34
Image Properties.....	9-34
Index.....	9-34
Key	9-34
Picture	9-34

Tag	9-35
Image Methods.....	9-35
ExtractIcon	9-35
Index.....	9-35
TreeView Examples.....	9-35
TreeView and Tree API	9-35
TreeView and ImageList.....	9-39
TreeView and Grid.....	9-41
Chart, TreeView, and ImageList Cheat Sheets	9-42
Chart Control Objects, Properties and Methods.....	9-42
TreeView Objects, Properties and Methods.....	9-54
ImageList Objects, Properties and Methods.....	9-56

Chapter 10

Meta-SQL

Date Considerations.....	10-1
Use of Date, Datetime, or Time Wrappers with an Application Engine Program ..	10-1
Date and Time Out Wrappers for Static and Dynamic Views	10-2
Using {DateTimein-prefix} in SQR.....	10-2
Meta-SQL Placement Considerations.....	10-2
Meta-SQL Reference	10-5
%Abs	10-5
%Concat	10-6
%CurrentDateIn	10-6
%CurrentDateOut.....	10-6
%CurrentDateTimeIn	10-6
%CurrentDateTimeOut	10-7
%CurrentTimeIn.....	10-7
%CurrentTimeOut	10-7
%DateAdd	10-7
%DateDiff	10-7
%DateIn.....	10-8
%DateOut.....	10-8
%DateTimeDiff.....	10-8
%DateTimeIn	10-9
%DateTimeOut	10-9
%DecDiv	10-9
%DecMult	10-10
%DTTM	10-11
%EffDtCheck	10-11

%FirstRows	10-13
%InsertSelect.....	10-14
%InsertValues	10-17
%Join.....	10-18
%KeyEqual	10-20
%KeyEqualNoEffDt	10-21
%Like	10-22
%LikeExact.....	10-24
%List	10-27
%ListBind	10-31
%ListEqual.....	10-32
%OldKeyEqual	10-33
%OPRCLAUSE	10-33
%Round.....	10-34
%SQL.....	10-34
%Substring	10-36
%SUBREC	10-36
%Table	10-37
%TextIn.....	10-38
%TimeAdd	10-39
%TimeIn.....	10-40
%TimeOut	10-40
%TrimSubstr	10-40
%Truncate	10-40
%TruncateTable	10-41
%UpdatePairs.....	10-42
%Upper	10-43
Meta-SQL Shortcuts	10-44
%Delete(:num)	10-44
%Insert(:num).....	10-44
%SelectAll(:num [correlation_id])	10-44
%SelectDistinct(:num [prefix])	10-45
%SelectByKey(:num [correlation_id])	10-45
%SelectByKeyEffDt(:num1, :num2).....	10-45
%Update(:num [, :num2]).....	10-45

Chapter 11

System Variables

%AsOfDate	11-1
%AuthenticationToken.....	11-1

%BPName	11-1
%ClientDate	11-1
%ClientTimeZone	11-2
%Component.....	11-2
%CompIntfcName.....	11-2
%Currency.....	11-2
%Date	11-3
%DateTime	11-3
%DbName	11-3
%DbType	11-3
%EmailAddress.....	11-3
%EmployeeId.....	11-3
%ExternalAuthInfo	11-3
%Import.....	11-4
%IsMultiLanguageEnabled.....	11-4
%Language.....	11-4
%Language_Base.....	11-4
%Market.....	11-4
%MaxInterlinkSize.....	11-5
%MaxMessageSize	11-6
%Menu	11-6
%MessageAgent.....	11-6
%Mode	11-6
%NavigatorHomePermissionList.....	11-6
%OperatorClass.....	11-7
%OperatorId.....	11-7
%OperatorRowLevelSecurityClass.....	11-7
%Page.....	11-7
%Panel.....	11-7
%PanelGroup	11-8
%PasswordExpired	11-8
%PermissionLists	11-8
%PrimaryPermissionList.....	11-8
%ProcessProfilePermissionList	11-8
%PSAuthResult.....	11-8
%Request.....	11-8
%Response	11-9
%ResultDocument.....	11-9
%Roles	11-9
%RowSecurityPermissionList.....	11-9

%ServerTimeZone	11-9
%Session	11-10
%SignonUserId	11-10
%SignOnUserPswd	11-10
%SQLRows.....	11-10
%Time	11-11
%UserDescription	11-11
%UserId.....	11-11
%WLInstanceID.....	11-11
%WLName.....	11-11

Chapter 12

Cheat Sheets

Class List Cheat Sheet	12-1
Summary of PeopleCode Classes.....	12-1
Summary of Session Class Properties and Methods	12-14
Summary of Component Interface API Properties and Methods.....	12-16
Summary of Search API Methods and Properties.....	12-19
Summary of Repository Methods	12-20
Summary of Repository Properties	12-20
Summary of Methods and Properties for Internet Script Classes	12-21
Summary of Classes, Methods and Properties for PortalRegistry Classes ...	12-23
Summary of Classes, Methods and Properties for Query Classes	12-27
Summary of Methods for Tree Classes API.....	12-34
Summary of Properties for Tree Classes API	12-36
Chart, TreeView, and ImageList Cheat Sheets.....	12-40
Chart Control Objects, Properties and Methods.....	12-40
TreeView Objects, Properties and Methods.....	12-52
ImageList Objects, Properties and Methods.....	12-54
Data Buffer Access Classes Cheat Sheet.....	12-55
Mapping of Functions to Methods and Properties.....	12-60
Mapping of Old Names to New Names.....	12-62
PeopleCode Syntax Cheat Sheet.....	12-63

Index

ABOUT THIS PEOPLEBOOK

This PeopleBook is a complete reference to PeopleCode, the proprietary scripting language used in the development of PeopleSoft applications. Its chapters describe the syntax and fundamental elements of the PeopleCode language.

The accompanying book, *PeopleCode Developer's Guide*, describes techniques for adding PeopleCode to applications, tips for using PeopleCode, the interaction of PeopleCode and the Component Processor, and a number of other specialized topics, such as the use of the PeopleCode debugger and referencing data in the component buffer.



For more information see PeopleCode Developer's Guide.

Audience

This book is written for technical users, project leaders, and programmers who will be customizing or developing applications using PeopleTools. To take full advantage of the information covered in this book, we recommend that you have a basic understanding of how to use PeopleSoft applications. In other words, you should be familiar with how to navigate your way around the system and how to add, update, and delete information using PeopleSoft tables and panels. You should also be comfortable using Microsoft® Windows.

PeopleCode is closely integrated with objects that you create and modify in Application Designer. This PeopleBook assumes that you are familiar with Application Designer, and that you understand the structure and relationship of the PeopleSoft application components developed in Application Designer. It also assumes a basic familiarity with structured programming languages, relational database concepts, and SQL.



For more information about Application Designer see Application Designer.

In this book, you'll find detailed reference information about PeopleCode, that is, basic descriptions of functions, methods and properties. For information on **using** PeopleCode, see the PeopleCode Developer's Guide. For information specific to your application, please refer to your PeopleSoft application documentation.

PeopleCode Built-in Functions describes all the built-in functions available in the PeopleCode language. The functions are arranged alphabetically.

PeopleCode Classes describes all the classes and objects you can access using PeopleCode. Each class contains a reference section detailing all the properties and methods available for that class.

ActiveX Controls in PeopleTools describes the ActiveX controls that are sponsored by PeopleSoft. Each description contains a reference section detailing all the events, properties and methods available for that control.

Meta-SQL describes the meta-SQL constructs that are available in PeopleCode. The meta-SQL constructs are arranged alphabetically. It also contains links to the additional meta-SQL constructs that can only be used in Application Engine programs. The end of the section contains a list of “short-cut” meta-SQL statements that act like short-hand for the more lengthy constructs.

System Variables describes the system variables that are available in PeopleCode. The variables are arranged alphabetically.

Cheat Sheets contains a number of tables that can provide a quick reference for classes and the different properties and methods available for that class. It also contains some short code examples.

Before You Begin

To benefit fully from the information covered in this book, you need to have a basic understanding of how to use PeopleSoft applications. We recommend that you complete at least one PeopleSoft introductory training course.

You should be familiar with navigating around the system and adding, updating, and deleting information using PeopleSoft windows, menus, and pages. You should also be comfortable using the World Wide Web and the Microsoft® Windows or Windows NT graphical user interface.

Related Documentation

To add to your knowledge of PeopleSoft applications and tools, you may want to refer to the documentation of the specific PeopleSoft applications your company uses. You can access additional documentation for this release from PeopleSoft Customer Connection (www.peoplesoft.com). We post updates and other items on Customer Connection, as well. In addition, documentation for this release is available on CD-ROM and in hard copy.



Important! Before upgrading, it is *imperative* that you check PeopleSoft Customer Connection for updates to the upgrade instructions. We continually post updates as we refine the upgrade process.

Documentation on the Internet

You can order printed, bound versions of the complete PeopleSoft documentation delivered on your PeopleBooks CD-ROM. You can order additional copies of the PeopleBooks CDs through the Documentation section of the PeopleSoft Customer Connection Web site:
<http://www.peoplesoft.com/>

You'll also find updates to the documentation for this and previous releases on Customer Connection. Through the Documentation section of Customer Connection, you can download files to add to your PeopleBook library. You'll find a variety of useful and timely materials, including updates to the full PeopleSoft documentation delivered on your PeopleBooks CD.

Documentation on CD-ROM

Complete documentation for this PeopleTools release is provided in HTML format on the PeopleTools PeopleBooks CD-ROM. The documentation for the PeopleSoft applications you have purchased appears on a separate PeopleBooks CD for the product line.

Hardcopy Documentation

To order printed, bound volumes of the complete PeopleSoft documentation delivered on your PeopleBooks CD-ROM, visit the PeopleSoft Press Web site from the Documentation section of PeopleSoft Customer Connection. The PeopleSoft Press Web site is a joint venture between PeopleSoft and Consolidated Publications Incorporated (CPI), our book print vendor.

We make printed documentation for each major release available shortly after the software is first shipped. Customers and partners can order printed PeopleSoft documentation using any of the following methods:

Internet

From the main PeopleSoft Internet site, go to the Documentation section of Customer Connection. You can find order information under the Ordering PeopleBooks topic. Use a Customer Connection ID, credit card, or purchase order to place your order.

PeopleSoft Internet site: <http://www.peoplesoft.com/>.

Telephone

Contact Consolidated Publishing Incorporated (CPI) at **800 888 3559**.

Email

Email CPI at callcenter@conpub.com.

Typographical Conventions and Visual Cues

To help you locate and interpret information, we use a number of standard conventions in our online documentation. We also use standard conventions in PeopleCode syntax.

Please take a moment to review the following typographical cues:

`monospace font`

Indicates a PeopleCode program or other example

Bold

In PeopleCode syntax, boldface items indicate function names, method names, language constructs, and PeopleCode reserved words that must be included literally in the function call.

Throughout the rest of this PeopleBook bold indicates field names and other page elements, such as buttons and group box labels, when these elements are documented below the page on which they appear. When we refer to these elements elsewhere in the documentation, we set them in Normal style (not in bold).

We also use boldface when we refer to navigational paths, menu names, or process actions (such as **Save** and **Run**).

Italics

In PeopleCode syntax, italic items are placeholders for arguments that your program must supply.

Throughout the rest of this PeopleBook italics indicates a PeopleSoft or other book-length publication. We also use italics for *emphasis* and to indicate specific field values. When we cite a field value under the page on which it appears, we use this style: *field value*.

We also use italics when we refer to words as words or letters as letters, as in the following: Enter the number *0*, not the letter *O*.

...

In PeopleCode syntax, ellipses indicate that the preceding item or series can be repeated any number of times.

{**Option1**|**Option2**}

In PeopleCode syntax, when there is a choice between two options, the options are enclosed in curly braces and separated by a pipe.

[]

In PeopleCode syntax optional items are enclosed in square brackets.

&Parameter

In PeopleCode syntax an ampersand before a parameter indicates that the parameter is an already instantiated object.

KEY+KEY

Indicates a key combination action. For example, a plus sign (+) between keys means that you must hold down the first key while you press the second key. For ALT+W, hold down the ALT key while you press W.

Jump links

Indicates a jump (also called a link, hyperlink, or hypertext link). Click a jump to move to the jump destination or referenced section.

Cross-references

The phrase For more information indicates where you can find additional documentation on the topic at hand. We include the navigational path to the referenced topic, separated by colons (:). Capitalized titles in *italics* indicate the title of a PeopleBook; capitalized titles in normal font refer to sections and specific topics within the PeopleBook. Cross-references typically begin with a jump link. Here's an example:

For more information, see Documentation on CD-ROM in *About These PeopleBooks*: Related Documentation.

- Topic list

Contains jump links to all the topics in the section. Note that these correspond to the heading levels you'll find in the Contents window.



Name of Page or
Dialog Box

Opens a pop-up window that contains the named page or dialog box. Click the icon to display the image. Some screen shots may also appear inline (directly in the text).



Text in this bar indicates information that you should pay particular attention to as you work with your PeopleSoft system. If the note is preceded by **Important!**, the note is crucial and includes information that concerns what you need to do for the system to function properly.



Text in this bar indicates For more information cross-references to related or additional information.



Text within this bar indicates a crucial configuration consideration. Pay very close attention to these warning messages.

Comments and Suggestions

Your comments are important to us. We encourage you to tell us what you like, or what you would like changed about our documentation, PeopleBooks, and other PeopleSoft reference and training materials. Please send your suggestions to:

PeopleTools Product Documentation Manager
PeopleSoft, Inc.
4460 Hacienda Drive
Pleasanton, CA 94588

Or send comments by email to the authors of the PeopleSoft documentation at:

DOC@PEOPLESOFT.COM

While we cannot guarantee to answer every email message, we will pay careful attention to your comments and suggestions. We are always improving our product communications for you.

CHAPTER 1

PeopleCode Built-in Functions

This section provides a reference to PeopleCode built-in functions and language constructs.

Throughout this section, we use typographical conventions to distinguish between different elements of the PeopleCode language, such as bold to indicate function names, italics for arguments, and so on.



For more information, see [Typographical Conventions and Visual Cues](#).

Functions by Category

The following topics subdivide the PeopleCode built-in functions by functional category and provide links from within each category to the reference entries.

ActiveX Controls

GetControl
GetControlOccurrence

APIs

GetSession
%Session

Application Engine

CallAppEngine
CommitWork

GetAESession

Application Messages

AddSystemPauseTimes
CreateMessage
DeleteSystemPauseTimes

GetMessage
GetMessageInstance
GetPubContractInstance
GetSubContractInstance
ReturnToServer
PingNode
SetChannelStatus
%MaxMessageSize

Arrays

CreateArray
CreateArrayRept
Split

Attachment

AddAttachment
DeleteAttachment
GetAttachment
PutAttachment
ViewAttachment



See also Files and Images.

Business Interlink

GetBiDoc
GetInterlink
%MaxInterlinkSize

Component Buffer

ActiveRowCount
AddKeyListItem
CompareLikeFields
ClearKeyList
ComponentChanged
CopyFields
CopyRow
CurrentLevelNumber
CurrentRowNumber
DeleteRecord
DeleteRow
DiscardRow
ExpandBindVar
ExpandEnvVar
ExpandSqlBinds
FetchValue
FieldChanged

GetMethodNames
GetNextNumberWithGaps
GetRelField
GetSetId
InsertRow
PriorValue
RecordChanged
RecordDeleted
RecordNew
RowFlush
SetChannelStatus
SetDefault
SetDefaultAll
SetTempTableInstance
StopFetching
TotalRowCount
TreeDetailInNode
UpdateSysVersion
UpdateValue
%Component
%Menu
%Mode
%OperatorClass
%Table
%TruncateTable



See also Data Buffer Access

Component Interface

GetMethodNames
GetSession
%CompIntfcName

Conversion

Char
Code
Codeb
ConvertChar
String
Value

Currency and Financial

Amortize
ConvertCurrency
RoundCurrency
SinglePaymentPV
UniformSeriesPV

%Currency

Current Date and Time

%CurrentDateIn
%CurrentDateOut
%CurrentDateTimeIn
%CurrentDateTimeOut
%CurrentTimeIn
%CurrentTimeOut



See also Date and Time and SQL Date and Time

Custom Display Formats

GetStoredFormat
SetDisplayFormat

Database and Platform

%DbName
%DbType

Data Buffer Access

CreateRecord
CreateRowset
FlushBulkInserts
GetField
GetLevel0
GetRecord
GetRow
GetRowset



See also Component Buffer.

Date and Time

AddToDate
AddToDateTime
AddToTime
ConvertDatetimeToBase
Date

Date3
DatePart
DateTime6
DateTimeToLocalizedString
DateTimeToTimeZone
DateTimeValue
DateValue
Day
Days
Days360
Days365
FormatDateTime
GetCalendarDate
Hour
IsDaylightSavings
Minute
Month
Second
Time
Time3
TimePart
TimeToTimeZone
TimeValue
TimeZoneOffset
Weekday
Year
%AsOfDate
%ClientDate
%ClientTimeZone
%Date
%DateAdd
%DateDiff
%DateTime
%DateTimeDiff
%DateTimeIn
%DateTimeOut
%DTTM
%PermissionLists
%ServerTimeZone
%Time
%TextIn



See also Current Date and Time and SQL Date and Time

Defaults, Setting

SetDefault
SetDefaultAll
SetDefaultNext
SetDefaultNextRel
SetDefaultPrior
SetDefaultPriorRel

DOS

ChDir
ChDrive
ExpandEnvVar
GetCwd
GetEnv

Double-Byte Characters

CharType
ContainsCharType
ContainsOnlyCharType
ConvertChar
DBCSTrim

Dynamic Tree Controls

GetSelectedTreeNode
GetSQL
GetTreeNodeRecordName
GetTreeNodeValue
RefreshTree
TreeDetailInNode



Dynamic Tree Controls are not the same as ActiveX Tree View Controls, and the functions used with the Dynamic Tree Controls can't be used with ActiveX Tree View Controls.

Effective Date and Effective Sequence

CurrEffDt
CurrEffRowNum
CurrEffSeq
NextEffDt
NextRelEffDt
PriorEffDt
PriorRelEffDt
SetDefaultNext
SetDefaultNextRel
SetDefaultPrior
SetDefaultPriorRel
%EffDtCheck

Executable Files, Running

Exec
WinExec



See also OLE.

Files

FileExists
FindFiles
GetFile



See also Attachment and Images.

Financial

AccruableDays
AccrualFactor
BlackScholesCall
BlackScholesPut
BootstrapYTMs
ConvertRate
CubicSpline
HermiteCubic
HistVolatility
LinearInterp

Grids

GetGrid

Images

DeleteImage
GetImageExtents
InsertImage



See also Attachment and Files.

Import Manager

%ExternalAuthInfo

Internet

EncodeURL
EncodeURLForQueryString
EscapeHTML
EscapeJavascriptString
EscapeWML
GenerateTree
GetHTMLText
GetMethodNames
GetURL
Unencode
ViewURL
%EmailAddress
%Request
%Response

Java

CreateJavaArray
CreateJavaObject
GetJavaClass

Language Preference and Locale

SetLanguage
%IsMultiLanguageEnabled
%IsMultiLanguageEnabled
%Language_Base
%Market

Language Constructs

Break
Component
Declare Function
Evaluate
Exit
For
Function
Global
If
Local
Repeat
Return
While

Logical (Tests for Blank Values)

All
AllOrNone
None
OnlyOne

OnlyOneOrNone

Mail

SendMail
%EmailAddress

Math

Abs
Acos
Asin
Atan
Cos
Cot
Degrees
Exp
Fact
GetMethodNames
GetNextNumberWithGaps
Int
Ln
Log10
Mod
Product
Radians
Rand
Round
Sign
Sin
Sqrt
Tan
Truncate
%Abs
%DecDiv
%DecMult
%Round
%Truncate

Menu Appearance

CheckMenuItem
DisableMenuItem
EnableMenuItem
HideMenuItem
UnCheckMenuItem

Message Agent

%MessageAgent

Message Catalog and Message Display

Error
MessageBox
MsgGet
MsgGetExplainText
MsgGetText
Prompt
Warning
WinMessage

Message, Application



See Application Messages

Modal Transfers

DoModalComponent
IsModalComponent



See also Secondary Pages, Transfers

Object-Based

CreateArray
CreateArrayRept
CreateJavaArray
CreateJavaObject
CreateMessage
CreateProcessRequest
CreateRecord
CreateRowset
CreateSQL
DeleteSQL
FetchSQL
GetAESection
GetControl
GetField
GetFile
GetGrid
GetHTMLText
GetInterlink
GetLevel0
GetJavaClass
GetMessage

GetMessageInstance
GetPubContractInstance
GetRecord
GetRow
GetRowset
GetSession
GetSQL
GetSubContractInstance
ReturnToServer
Split
StoreSQL

OLE

CreateObject
ObjectDoMethod
ObjectGetProperty
ObjectSetProperty



The OLE functions are used to access and manipulate external OLE objects only. Do not use them to access or manipulate PeopleCode objects such as arrays, rowsets, and so on. You can use them in special circumstances to access the TreeView and Chart ActiveX controls.



See also ActiveX Controls

Page

GetPage

Page Control Appearance

GetImageExtents
Gray
Hide
HideRow
HideScroll
IsHidden
SetCursorPos
SetLabel
Ungray
Unhide
UnhideRow
UnhideScroll

Process Scheduler

CreateProcessRequest
ScheduleProcess

Remote Call

DoSaveNow
RemoteCall

Saving and Canceling

DoCancel
DoSave
DoSaveNow
WinEscape

Scroll Select

RowFlush
RowScrollSelect
RowScrollSelectNew
ScrollFlush
ScrollSelect
ScrollSelectNew
SortScroll

Search Dialog

ClearSearchDefault
ClearSearchEdit
IsSearchDialog
SetSearchDefault
SetSearchDialogBehavior
SetSearchEdit
%Mode

Secondary Pages

DoModal
EndModal
IsModal



See also Modal Transfers and Transfers.

SQL

CreateSQL
DeleteSQL
ExpandBindVar
ExpandSqlBinds
FetchSQL
FlushBulkInserts
GetSQL
SQLExec
StoreSQL
%FirstRows
%InsertSelect
%InsertValues
%Join
%KeyEqual
%KeyEqualNoEffDt
%Like
%LikeExact
%List
%ListBind
%ListEqual
%OldKeyEqual
%SQL
%SignonUserId
%Table
%UpdatePairs



See also Scroll Select and Data Buffer Access.

SQL Date and Time

%DateAdd
%DateDiff
%DateIn
%DateOut
%DateTimeDiff
%DateTimeIn
%DateTimeOut
%DTTM
%TimeAdd
%TextIn
%TimeIn
%TimeOut



See also Current Date and Time and Date and Time

SQL Shortcuts

%Delete(
%Insert(
%SelectAll(
%SelectByKey(
%SelectByKeyEffDt(
%Update(

String

Clean
Code
Codeb
DBCSTrim
Exact
ExpandBindVar
ExpandEnvVar
Find
Findb
GetHTMLText
Left
Len
Lenb
Lower
LTrim
Proper
Replace
Rept
Right
RTrim
String
Substitute
Substring
Substringb
Upper
%Abs
%Substring
%TrimSubstr
%Upper

Subrecords

%SUBREC

TimeZone

ConvertDatetimeToBase
ConvertTimeToBase
DateTimeToTimeZone
FormatDateTime
IsDaylightSavings
TimeToTimeZone
TimeZoneOffset

%ClientTimeZone
%ServerTimeZone

Trace Control

SetTracePC
SetTraceSQL

Transfers

AddKeyListItem
ClearKeyList
SetNextPage
Transfer
TransferPage



See also Modal Transfers and Secondary Pages.

User Information

%EmailAddress
%EmployeeId
%UserDescription
%UserId

User Security

AllowEmplIdChg
Decrypt
Encrypt
ExecuteRolePeopleCode
ExecuteRoleQuery
ExecuteRoleWorkflowQuery
Hash
IsMenuItemAuthorized
IsUserInPermissionList
IsUserInRole
ReturnToServer
SetAuthenticationResult
SetPasswordExpired
SwitchUser
%AuthenticationToken
%EmailAddress
%ExternalAuthInfo
%NavigatorHomePermissionList
%PasswordExpired
%PermissionLists
%PrimaryPermissionList
%ProcessProfilePermissionList

%PSAuthResult
%ResultDocument
%Roles
%RowSecurityPermissionList
%SignonUserId
%SignOnUserPswd
%UserId

Validation

Error
IsMenuItemAuthorized

ReturnToServer
SetCursorPos
SetReEdit
Warning

Workflow

GetWLFieldValue
MarkWLItemWorked
TriggerBusinessEvent
%BPName
%WLInstanceId
%WLName

PeopleCode Built-in Functions and Language Constructs A-D

Abs

Syntax

Abs (*x*)

Description

Abs returns a decimal value equal to the absolute value of a number *x*.

Example

The example returns the absolute value of the difference between &NUM_1 and &NUM_2:

```
&RESULT = Abs (&NUM_1 - &NUM_2) ;
```

Related Topics

Sign, %Abs

AccruableDays

Syntax

AccruableDays(*StartDate*, *EndDate*, *Accrual_Conv*)

Description

The **AccruableDays** function returns the number of days during which interest can accrue in a given range of time according to the *Accrual_Conv* parameter.

Parameters

<i>StartDate</i>	The beginning of the time period for determining the accrual. This parameter takes a date value.
<i>EndDate</i>	The end of the time period for determining the accrual. This parameter takes a date value.
<i>Accrual_Conv</i>	The accrual convention. This parameter takes either a number or a constant value. The valid values for this parameter are:

Value	Constant	Description
0	%Accrual_30DPM	30/360: all months 30 days long according to NASD rules for date truncation
1	%Accrual_30DPME	30E/360: all months 30 days long according to European rules for date truncation
2		30N/360: all months but February are 30 days long according to SIA rules
3	%Accrual_Fixed360	Act/360: months have variable number of days, but years have fixed 360 days
4	%Accrual_Fixed365	Act/365: months have variable number of days, but years have fixed 365 days
5	%Accrual_ActualDPY	Act/Act: months and years have a variable number of days

Returns

An integer representing a number of days.

Related Topics

AccrualFactor

AccrualFactor

Syntax

```
AccrualFactor(StartDate, EndDate, Accrual_Conv)
```

Description

The **AccrualFactor** function computes a factor that's equal to the number of years of interest accrued during a date range, according to *Accrual_Conv* parameter.

Parameters

<i>StartDate</i>	The beginning of the time period for determining the accrual. This parameter takes a date value.
<i>EndDate</i>	The end of the time period for determining the accrual. This parameter takes a date value.
<i>Accrual_Conv</i>	The accrual convention. This parameter takes either a number or constant value. The valid values for this parameter are:

Value	Constant	Description
0	%Accrual_30DPM	30/360: all months 30 days long according to NASD rules for date truncation
1	%Accrual_30DPME	30E/360: all months 30 days long according to European rules for date truncation
2		30N/360: all months but February are 30 days long according to SIA rules
3	%Accrual_Fixed360	Act/360: months have variable number of days, but years have fixed 360 days
4	%Accrual_Fixed365	Act/365: months have

Value	Constant	Description
		variable number of days, buy years have fixed 365 days
5	%Accrual_ActualDPY	Act/Act: months and years have a variable number of days

Returns

A floating point number representing a number of years.

Related Topics

AccruableDays

Acos

Syntax

`Acos (value)`

Description

The **Acos** function calculates the arccosine of the given value, that is, the size of the angle whose cosine is that value.

Parameters

value

Any real number between **-1.00** and **1.00** inclusive—the range of valid cosine values. If the input value is outside this range, you'll see an error message at runtime ("Decimal arithmetic error occurred. (2,110)"). Adjust your code to provide a valid input value.

Returns

A value in radians between **0** and **pi**.

Example

The following example returns the size in radians of the angle whose cosine is **0.5**:

```
&MY_ANGLE = Acos (0.5) ;
```

Related Topics

Asin, Atan, Cos, Cot, Degrees, Radians, Sin, Tan

ActiveRowCount

Syntax

ActiveRowCount (*Scrollpath*)

Where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]]
RECORD.target_recname
```

To prevent ambiguous references, you can use **SCROLL.scrollname**, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Description

ActiveRowCount returns the number of active (non-deleted) rows for a specified scroll area in the active page.



This function remains for backward compatibility only. Use the ActiveRowCount Rowset class property instead. See also Data Buffer Access.

ActiveRowCount is often used to get a limiting value for a For statement. This allows you to loop through the active rows of a scroll area, performing an operation on each active row. Rows that have been marked as deleted will not be affected in a **For** loop delimited by **ActiveRowCount**. If you want to loop through all the rows of a scroll area, including deleted rows, use TotalRowCount.

ActiveRowCount can be used with CurrentRowNumber to determine whether the user is on the last row of a record.

Returns

Returns a Number value equal to the total active (non-deleted) rows in the specified scroll area in the active page.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
-------------------	---



For more information on scroll path syntax, see [References Using Scroll Path Syntax and Dot Notation](#).

Example

In this example **ActiveRowCount** is used to delimit a **For** loop through a level-one scroll:

```
&CURRENT_L1 = CurrentRowNumber(1);
&ACTIVE_L2 = ActiveRowCount(RECORD.ASSIGNMENT, &CURRENT_L1,
RECORD.ASGN_HOME_HOST);
&HOME_HOST = FetchValue(RECORD.ASSIGNMENT, &CURRENT_L1,
ASGN_HOME_HOST.HOME_HOST, 1);
If All(&HOME_HOST) Then
    For &I = 1 To &ACTIVE_L2
        DeleteRow(RECORD.ASSIGNMENT, &CURRENT_L1, RECORD.ASGN_HOME_HOST, 1);
    End-For;
End-If;
```

Related Topics

CurrentRowNumber, TotalRowCount

AddAttachment

Syntax

```
AddAttachment(URLDestination, DirAndFilename, FileType, UserFile, MaxSize)
```

where *URLDestination* can have *one* of the following forms:

```
URL.URLName
```

OR a string URL, such as

```
ftp://user:password@ftp.ps.com/
```

Description

The **AddAttachment** function enables you to transfer a file from one server to another. You could use this to associate a file with a row in a record, or with a page of data.

Specifying non-existing Subdirectories Considerations

If the specified subdirectories don't exist, in the PeopleSoft Internet Architecture, this function tries to create them. In Windows Client, the subdirectories are **not** created, and the functions does not complete.

File Name Considerations

When the file is transferred, the following characters are replaced with an underscore:

- space
- semi-colon
- plus sign
- percent sign
- ampersand
- apostrophe
- exclamation point
- @ sign
- pound sign
- dollar sign

Restrictions on Use in PeopleCode Events

AddAttachment is a "think-time" function, which means that it shouldn't be used in any of the following PeopleCode events:

- SavePreChange
- SavePostChange
- Workflow
- RowSelect
- Any PeopleCode event that fires as a result of a Select or SelectNew method, or any of the ScrollSelect functions.



For more information, see Think-Time Functions.

If you want to transfer files in a "non-interactive" mode, with functions that aren't think-time, see the GetAttachment and PutAttachment functions.

Parameters

URLDestination

A reference to a URL. This can be either a URL name, in the form `URL.URLName`, or a string. This is where the file is transferred to.

DirAndFileName

A directory and filename. This is appended to the *URLDestination* to make up the full URL where the file is transferred to.



The *URLDestination* requires "/" (forward) slashes. Because the *DirAndFileName* parameter is appended to the URL, it also requires only "/" slashes. "\" (backward) slashes are NOT supported in anyway for either the *URLDestination* or the *DirAndFileName* parameter.

FileType

A string to use as a suggestion for the extension file type, such as ".doc", ".gif", ".properties" and so on. Passing in a null string (that is, two double-quotes with no space ("")) will invoke all files (*.*)

UserFile

The name of the file on the source system.

MaxSize

Specify, in bytes, the maximum size of the attachment.

Returns

An integer value. You can check for either the integer or the constant:

Number	Constant	Description
0	%Attachment_Success	File was transferred successfully.
1	%Attachment_Failed	File was not successfully transferred.
2	%Attachment_Cancelled	File transfer didn't complete because the operation was canceled by the user.
3	%Attachment_FileTransferFailed	File transfer didn't succeed.
4	%Attachment_NoDiskSpaceAppServ	No disk space on the application server.
5	%Attachment_NoDiskSpaceWebServ	No disk space on the web server.
6	%Attachment_FileExceedsMaxSize	File exceeds maximum size.
7	%Attachment_DestSystNotFound	Cannot locate destination system for ftp.
8	%Attachment_DestSystFailedLogin	Unable to login to destination system for ftp.

Example

```

&uniquefilename = "";

&REC = GetRecord();

For &I = 1 To &REC.FieldCount

    &Fld = &REC.getfield(&I);

    If (&Fld.IsKey = True) Then

        If (&Fld.value = "" And

            &Fld.IsRequired = True) Then

            MessageBox(0, "File Attachment Status", 0, 0, "Please attach the file
after filling out the required fields on this page.");

            Return;

        Else

            &uniquefilename = &uniquefilename | &Fld.value;

        End-If;

    End-If;

End-For;

&temp = Substitute(&uniquefilename, " ", ""); /* delete spaces */

ATTACHSYSFILENAME = Left(&uniquefilename, 64); /* make uniq part a max of 64
chars */

/* If you have created a subdirectory on the ftp server for this page, prepend
it here:

    ATTACHSYSFILENAME = "HR/ABSENCEHIST/" | ATTACHSYSFILENAME;

*/

&retcode = AddAttachment(URL.MYFTP, ATTACHSYSFILENAME, "", ATTACHUSERFILE, 0);

/* The actual name on the ftp file server is the following */

```

```

ATTACHSYSFILENAME = ATTACHSYSFILENAME | ATTACHUSERFILE;

&temp = Substitute(ATTACHSYSFILENAME, " ", ""); /* delete spaces */
ATTACHSYSFILENAME = &temp;

If (&retcode = %Attachment_Success) Then

    MessageBox(0, "File Attachment Status", 0, 0, "AddAttachment succeeded");

    Hide(ATTACHADD);

    UnHide(ATTACHVIEW);

    UnHide(ATTACHDELETE);

End-If;

If (&retcode = %Attachment_Failed) Then

    MessageBox(0, "File Attachment Status", 0, 0, "AddAttachment failed");

    ATTACHUSERFILE = "";

    ATTACHSYSFILENAME = "";

End-If;

If (&retcode = %Attachment_Cancelled) Then

    MessageBox(0, "File Attachment Status", 0, 0, "AddAttachment cancelled");

    ATTACHUSERFILE = "";

    ATTACHSYSFILENAME = "";

End-If;

If (&retcode = %Attachment_FileTransferFailed) Then

    MessageBox(0, "File Attachment Status", 0, 0, "AddAttachment failed: File
Transfer did not succeed");

    ATTACHUSERFILE = "";

    ATTACHSYSFILENAME = "";

End-If;

/* following error message only in iclient mode */

If (&retcode = %Attachment_NoDiskSpaceAppServ) Then

```

```
    MessageBox(0, "File Attachment Status", 0, 0, "AddAttachment failed: No disk
space on the app server");

    ATTACHUSERFILE = "";

    ATTACHSYSFILENAME = "";

End-If;

/* following error message only in iclient mode */

If (&retcode = %Attachment_NoDiskSpaceWebServ) Then

    MessageBox(0, "File Attachment Status", 0, 0, "AddAttachment failed: No disk
space on the web server");

    ATTACHUSERFILE = "";

    ATTACHSYSFILENAME = "";

End-If;

If (&retcode = %Attachment_FileExceedsMaxSize) Then

    MessageBox(0, "File Attachment Status", 0, 0, "AddAttachment failed: File
exceeds the max size");

    ATTACHUSERFILE = "";

    ATTACHSYSFILENAME = "";

End-If;

If (&retcode = %Attachment_DestSystNotFound) Then

    MessageBox(0, "File Attachment Status", 0, 0, "AddAttachment failed: Cannot
locate destination system for ftp");

    ATTACHUSERFILE = "";

    ATTACHSYSFILENAME = "";

End-If;

If (&retcode = %Attachment_DestSysFailedLogin) Then

    MessageBox(0, "File Attachment Status", 0, 0, "AddAttachment failed: Unable
to login into destination system for ftp");

    ATTACHUSERFILE = "";

    ATTACHSYSFILENAME = "";

End-If;
```


Related Topics

DeleteAttachment, GetAttachment, PutAttachment, ViewAttachment, Using the Attachment Functions

AddKeyListItem

Syntax

```
AddKeyListItem(field, value)
```

Description

AddKeyListItem adds a new key field and its value to the current list of keys. It enables PeopleCode to help users navigate through related pages without being prompted for key values. A common use of **AddKeyListItem** is to add a field to a key list and then transfer to a page which uses that field as a key.

Returns

Returns a Boolean value indicating whether it completed successfully.

Parameters

<i>field</i>	The field to add to the key list.
<i>value</i>	The value of the added key field used in the search.

Example

The following example sets up a key list using **AddKeyListItem** and transfers the user to a page named VOUCHER_INQUIRY_FS.

```
AddKeyListItem(VNDR_INQ_VW_FS.BUSINESS_UNIT, ASSET_ACQ_DET.BUSINESS_UNIT_AP);  
AddKeyListItem(VNDR_INQ_VW_FS.VOUCHER_ID, ASSET_ACQ_DET.VOUCHER_ID);  
TransferPage("VOUCHER_INQUIRY_FS");
```

Related Topics

ClearKeyList, TransferPage, Transfer

AddSystemPauseTimes

Syntax

```
AddSystemPauseTimes(StartDay, StartTime, EndDay, EndTime)
```

Description

The **AddSystemPauseTimes** function allows you to set when pause times occur on your system by adding a row to the system pause-times tables.

This function is used in the PeopleCode for the Application Message Monitor. Pause times are set up in the Application Message Monitor.

Setting System Pause Times in the Application Message Monitor



For more information, see Application Message Monitor.

Parameters

StartDay Specify a number from 0-6. The valid values are:

Value	Description
0	Sunday
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday

StartTime Specify a time, in seconds, since midnight.

EndDay Specify a number from 0-6. The valid values are:

Value	Description
0	Sunday
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday

EndTime Specify a time, in seconds, since midnight.

Returns

A boolean value: True if the system pause time specified was added, False otherwise.

Example

```

Declare Function SetTime PeopleCode REFRESH_BTN FieldFormula;

Component boolean &spt_changed;

Function GetSecond(&time) Returns number ;

    Return Hour(&time) * 3600 + Minute(&time) * 60 + Second(&time);

End-Function;

/* initialize; */

STARTDAY = "0";

AMM_STARTTIME = SetTime(0);

ENDDAY = "0";

AMM_ENDTIME = SetTime(0);

```

```

If DoModal (Panel.AMM_ADD_SPTIMES, MsgGetText (117, 13, ""), - 1, - 1) = 1 Then

    If AddSystemPauseTimes (Value (STARTDAY), GetSecond (AMM_STARTTIME),
    Value (ENDDAY), GetSecond (AMM_ENDTIME)) Then

        &spt_changed = True;

        DoSave ();

    Else

        MessageBox (16, MsgGetText (117, 13, ""), 117, 14, "");

    End-If;

End-If;

```

Related Topics

Application Message Monitor, DeleteSystemPauseTimes

AddToDate

Syntax

```
AddToDate (date, num_years, num_months, num_days)
```

Description

The **AddToDate** function adds the specified number of years, months, and days to the *date* provided.

Suppose, for example, that you wanted to find a date six years from now. You couldn't just multiply 6 times 365 and add the result to today's date, because of leap years. And, depending on the current year, there may be one or two leap years in the next six years. **AddToDate** takes care of this for you.

You can subtract from dates by passing the function negative numbers.

Returns

Returns a Date value equal to the original date plus the number of years, months, and days passed to the function.

Parameters

<i>date</i>	The input date to be adjusted.
<i>num_years</i>	The number of years by which to adjust the specified <i>date</i> . <i>num_years</i> can be a negative number.
<i>num_months</i>	The number of months by which to adjust the specified <i>date</i> . <i>num_months</i> can be a negative number.

num_days The number of days by which to adjust the specified *date*.
num_days can be a negative number.

Example

The following example finds the date one year, three months, and 16 days after a field called BEGIN_DT:

```
AddToDate(BEGIN_DT, 1, 3, 16);
```

The next example finds the date two months ago prior to BEGIN_DT:

```
AddToDate(BEGIN_DT, 0, -2, 0);
```

Related Topics

DateValue, Day, Days, Days360, Days365, Month, Weekday

AddToDateTime

Syntax

```
AddToDateTime(datetime, years, months, days, hours, minutes, seconds)
```

Description

The **AddToDateTime** function adds the specified number of years, months, days, hours, seconds, and minutes to the *datetime* provided. You can subtract from datetimes by passing the function negative numbers.

Returns

Returns a Datetime value equal to the original date plus the number of years, months, days, hours, minutes, and seconds passed to the function.

Parameters

<i>datetime</i>	The Datetime value to which you want to add.
<i>years</i>	An integer representing the number of years to add to <i>datetime</i> .
<i>months</i>	An integer representing the number of months to add to <i>datetime</i> .
<i>days</i>	An integer representing the number of days to add to <i>datetime</i> .
<i>hours</i>	An integer representing the number of hours to add to <i>datetime</i> .

<i>minutes</i>	An integer representing the number of minutes to add to <i>datetime</i> .
<i>seconds</i>	An integer representing the number of seconds to add to <i>datetime</i> .

Example

The following example postpones an interview scheduled in the INTRTime field by two days and two hours:

```
INTRTIME = AddToDateTime(INTRTIME, 0, 0, 2, 2, 0, 0);
```

Related Topics

AddToTime, DateValue, DateTimeValue, TimeValue

AddToTime

Syntax

```
AddToTime(time, hours, minutes, seconds)
```

Description

AddToTime adds *hours*, *minutes*, and *seconds* to *time*, and returns the result as a Time value. To subtract from *time*, use negative numbers for *hours*, *minutes*, and *seconds*. The resulting value is always adjusted such that it represents an hour less than 24 (a valid time of day.)

Returns

A Time value equal to time increased by the number of hours, minutes, and seconds passed to the function.

Parameters

<i>time</i>	A time value that you want to subtract from or add to.
<i>hours</i>	An integer representing the number of hours to add to <i>time</i> .
<i>minutes</i>	An integer representing the number of minutes to add to <i>time</i> .
<i>seconds</i>	An integer representing the number of seconds to add to <i>time</i> .

Example

Assume that a time, &BREAKTime, is 0:15:00. The following moves the time &BREAKTime back by one hour, resulting in 23:15:00:

```
&BREAKTime = AddToTime(&BREAKTime, -1, 0, 0);
```

Related Topics

AddToDateTime, DateValue, DateTimeValue, TimeValue

All

Syntax

```
All(fieldlist)
```

Where *fieldlist* is an arbitrary-length list of field names in the form:

```
[recordname.] fieldname1 [, [recordname.] fieldname2] ...
```

Description

All checks to see if a field contains a value, or if all the fields in a list of fields contain values. If any of the fields are Null, then **All** returns False.

A blank character field, or a zero (0) numeric value in a required numeric field is considered a null value.

Returns

Returns a Boolean value based on the values in *fieldlist*. **All** returns True if all of the specified fields have a value; it returns False if any one of the fields does not contain a value.

Related Functions

None	Checks that a field or list of fields have no value.
AllOrNone	Checks if either all the field parameters have values, or none of them have values. Use this in examples where if an user fills in one field, she must fill in all the other related values.
OnlyOne	Checks if exactly one field in the set has a value. Use this when the user must fill in only one of a set of mutually exclusive fields.
OnlyOneOrNone	Checks if no more than one field in the set has a value. Use this in examples when a set of fields is both optional and mutually exclusive; that is, if the user can put a value into one field in a set of fields, or leave them all empty.

Example

All is commonly used in SaveEdit PeopleCode to ensure that a group of related fields are all entered. For example:

```

If All (RETURN_DT, BEGIN_DT) and
    8 * (RETURN_DT - BEGIN_DT) (DURATION_DAYS * 8 + DURATION_HOURS)
Then
    Warning MsgGet(1000, 1, "Duration of absence exceeds standard hours for
number of days absent.");
End-if;

```

Related Topics

AllOrNone, None, OnlyOne, OnlyOneOrNone, SetDefault, SetDefaultAll

AllOrNone

Syntax

AllOrNone(*fieldlist*)

Where *fieldlist* is an arbitrary-length list of field references in the form:

[*recordname.*] *fieldname1* [, [*recordname.*] *fieldname2*] ...

Description

AllOrNone takes a list of fields and returns True if either of these conditions is true:

- All of the fields have values (that is, are not Null)
- None of the fields has a value.

For example, if field1 = 5, field2 = "Mary", and field3 = null, **AllOrNone** returns False.

This function is useful, for example, where you have a set of page fields, and if any one of the fields contains a value, then all of the other fields are required as well.

A blank character field, or a zero (0) numeric value in a required numeric field is considered a null value.

Returns

Returns a Boolean value: True if all of the fields in fieldlist or none of the fields in fieldlist has a value, False otherwise.

Related Functions

All	Checks to see if a field contains a value, or if all the fields in a list of fields contain values. If any of the fields is Null, then All returns False.
None	Checks that a field or list of fields have no value. None is the opposite of All.

OnlyOne	Checks if exactly one field in the set has a value. Use this when the user must fill in only one of a set of mutually exclusive fields.
OnlyOneOrNone	Checks if no more than one field in the set has a value. Use this in cases when a set of fields is both optional and mutually exclusive; that is, if the user can put a value into one field in a set of fields, or leave them all empty.

Example

You could use **AllOrNone** as follows:

```
If Not AllOrNone(STREET1, CITY, STATE) Then
    WinMessage("Address should consist of at least Street (Line 1), City, State,
and Country.");
End-if;
```

Related Topics

All, None, OnlyOne, OnlyOneOrNone

AllowEmplIdChg

Syntax

```
AllowEmplIdChg(is_allowed)
```

Description

By default, the Component Processor does not allow an user to make any changes to a record if a record contains an EMPLID key field and its value matches the value of the user's EMPLID. In some situations, though, such changes are warranted. For example, you would want employees to be able to change information about themselves when entering time sheet data.

AllowEmplIdChg enables the user to change records whose key matches the user's own EMPLID, or prevents the user from changing these records. The function takes a single Boolean parameter that when set to True allows the employee to update their own data. When the parameter is set to False, the employee is prevented from updating this data.

Once permission is granted, it stays through the life of the *component*, not the page. Once a user switches to another component, the default value (not being able to make changes) is reapplied.

Returns

Optionally returns a Boolean value: True if the function executed successfully, False otherwise.

Parameters

<i>is_allowed</i>	A Boolean value indicating whether the user is permitted to change the user's own data.
-------------------	---

Example

```
If Substring (%Page, 1, 9) = Substring(PAGE.TimeSHEET_PNL_A, 1, 9) Then
    AllowEmpIdChg(true);
End-if;
```

Amortize

Syntax

```
Amortize(intr, pb, pmt, pmtnbr, payintr, payprin, balance)
```

Description

Amortize computes the amount of a loan payment applied towards interest (*payintr*), the amount of the payment applied towards principal (*payprin*), and the remaining balance *balance*, based on the principal balance (*pb*) at the beginning of the loan term, the amount of one payment *pmt*, the interest rate charged during one payment period (*intr*), and the payment number *pmtnbr*.

Returns

None.

Parameters

Note that *payintr*, *payprin*, and *balance* are "outvars": you need to pass variables in these parameters, which **Amortize** will then fill with values. The remaining parameters are "invars" containing data the function needs to perform its calculation.

<i>intr</i>	Number indicating the percent of interest charged per payment period.
<i>pb</i>	Principal balance at the beginning of the loan term (generally speaking, the amount of the loan).
<i>pmt</i>	The amount of one loan payment.
<i>pmtnbr</i>	The number of the payment.
<i>payintr</i>	The amount of the payment paid toward interest.
<i>payprin</i>	The amount of the payment paid toward principal.
<i>balance</i>	The remaining balance after the payment.

Example

Let's say you want to calculate the principal, interest, and remaining balance after the 24th payment on a loan of \$15,000, at an interest rate of 1% per loan payment period, and a payment amount of \$290.

```
&INTRST_RT=1;
&LOAN_AMT=15000;
&PYMNT_AMNT=290;
```

```

&PYMNT_NBR=24;
Amortize(&INTRST_RT, &LOAN_AMT, &PYMNT_AMNT, &PYMNT_NBR, &PYMNT_INTRST,
&PYMNT_PRIN, &BAL);
&RESULT = "Int=" | String(&PYMNT_INTRST) | " Prin=" | String(&PYMNT_PRIN) | "
          Bal=" | String(&BAL);

```

This example sets &RESULT equal to "Int=114 Prin=176 Bal=11223.72".

Asin

Syntax

Asin(*value*)

Description

The **Asin** function calculates the arcsine of the given value, that is, the size of the angle whose sine is that value.

Parameters

<i>value</i>	Any real number between -1.00 and 1.00 inclusive—the range of valid sine values. If the input value is outside this range, you'll see an error message at runtime ("Decimal arithmetic error occurred. (2,110)"). Adjust your code to provide a valid input value.
--------------	--

Returns

A value in radians between **-pi/2** and **pi/2**.

Example

The following example returns the size in radians of the angle whose sine is **0.5**:

```
&MY_ANGLE = Asin(0.5);
```

Related Topics

Acos, Atan, Cos, Cot, Degrees, Radians, Sin, Tan

Atan

Syntax

Atan(*value*)

Description

The **Atan** function calculates the arctangent of the given value, that is, the size of the angle whose tangent is that value.

Parameters

value Any real number.

Returns

A value in radians between **-pi/2** and **pi/2**.

Example

The following example returns the size in radians of the angle whose tangent is **0.5**:

```
&MY_ANGLE = Atan(0.5);
```

Related Topics

Acos, Asin, Cos, Cot, Degrees, Radians, Sin, Tan

BlackScholesCall

Syntax

```
BlackScholesCall(Asset_Price, Strike_Price, Interest_Rate, Years, Volatility)
```

Description

The **BlackScholesCall** function returns the value of a call against an equity underlying according to the Black-Scholes equations.

Parameters

<i>Asset_Price</i>	The asset price. This parameter takes a decimal value.
<i>Strike_Price</i>	The strike price. This parameter takes a decimal value.
<i>Interest_Rate</i>	The risk-free interest rate. This parameter takes a decimal value.
<i>Years</i>	The number of years to option expiration. This parameter takes a number value (decimal).
<i>Volatility</i>	The volatility of underlying. This parameter takes a decimal value.

Returns

A number representing the value of a call against an equity.

Related Topics

BlackScholesPut

BlackScholesPut

Syntax

```
BlackScholesPut(Asset_Price, Strike_Price, Interest_Rate, Years, Volatility)
```

Description

The **BlackScholesPut** function returns the value of a put against an equity underlying according to the Black-Scholes equations.

Parameters

<i>Asset_Price</i>	The asset price. This parameter takes a decimal value.
<i>Strike_Price</i>	The strike price. This parameter takes a decimal value.
<i>Interest_Rate</i>	The risk-free interest rate. This parameter takes a decimal value.
<i>Years</i>	The number of years to option expiration. This parameter takes a number (decimal) value.
<i>Volatility</i>	The volatility of underlying. This parameter takes a decimal value.

Returns

A number representing the value of a call against an equity.

Related Topics

BlackScholesCall

BootstrapYTM

Syntax

```
BootstrapYTM(Date, MktInst, Accrual_Conv)
```

Description

The **BootstrapYTM** function creates a zero-arbitrage implied zero-coupon curve from a yield-to-maturity curve using the integrated discount factor method, based on the *Accrual_Conv*.

Parameters

Date

The trading date of the set of market issues. This parameter takes a date value.

MktInst

The market instrument properties. This parameter takes an array of array of number. The elements in the array specify the following:

<i>Elements</i>	<i>Description</i>
1	tenor in days
2	yield in percent
3	price per 100 par
4	coupon rate (zero for spot instruments)
5	frequency of coupon payments
6	unit of measure for coupon frequency, 0 for days, 1 for months, and 2 for years
7	coefficient a of a curve interpolating the dataset
8,9,10	coefficients b , c , and d of a curve interpolating the dataset

Returns

An array of array of number. The elements in the array have the same type as the elements in the array for the *MktInst* parameter.

Related Topics

Array Class

Break

Syntax

Break

Description

A **Break** statement takes no parameters and can appear within a loop or an **Evaluate** statement. It terminates execution of the loop or **Evaluate**, and resumes execution immediately after the end of the statement. If the loop or **Evaluate** is nested in another statement, only the innermost statement is terminated.

Example

In the following example, **Break** is used to terminate the **Evaluate** statement, while staying within the outermost **If** statement:

```
If CURRENCY_CD = PriorEffdt(CURRENCY_CD) Then
    Evaluate ACTION
    When = "PAY"
        If ANNUAL_RT = PriorEffdt(ANNUAL_RT) Then
            Warning MsgGet(1000, 27, "Pay Rate Change action is chosen and Pay
Rate has not been changed.");
            End-if;
            Break;
        When = "DEM"
            If ANNUAL_RT >= PriorEffdt(ANNUAL_RT) Then
                Warning MsgGet(1000, 29, "Demotion Action is chosen and Pay Rate has
not been decreased.");
                End-if;
                Break;
            When-other
            End-evaluate;
        WinMessage("This message appears after executing either of the BREAK
statements or after all WHEN statements are false");
    End-if;
```

Related Topics

Evaluate, Exit, For, While

CallAppEngine

Syntax

```
CallAppEngine(applid [, statereclist]);
```

Where *statereclist* is list of record objects in the form:

```
&staterecord1 [, &staterecord2] . . .
```

There can only be as many record objects in *statereclist* as there are state records for the Application Engine program. Additional record objects will be ignored.



For more information on state records, see Advanced Development.

Description

CallAppEngine starts the Application Engine program named *applid*. This is a way of starting your Application Engine programs synchronously from a page. (Prior to PeopleTools Release 8.0, you could do only this using **RemoteCall**.) Normally, you won't run your Application

Engine programs from PeopleCode in this manner. Rather, the bulk of your Application Engine execution will be run using the Process Scheduler, and the exception would be done using **CallAppEngine**.

The *statercord* can be the hard-coded name of a record, but generally you will use a record object to pass in values to seed particular state fields. The record name must match the state record name exactly.



If you use this function, you shouldn't use the %TruncateTable or %Execute meta-SQL statement in any of your Application Engine steps. This is because on some platforms an implicit commit occurs after these statements, and all online processing should be done as a single logical unit of work.

After you use **CallAppEngine**, you may want to refresh your page. The **Refresh** method, on a rowset object, reloads the rowset (scroll) using the current page keys. This causes the page to be redrawn. GetLevel0().Refresh() refreshes the entire page. If you only want a particular scroll to be redrawn, you can refresh just that part.



For more information, see Refresh rowset class method.

Runtime Considerations

Online PeopleCode that uses CallAppEngine **must** be set to run on the Application Server. If you run this PeopleCode on the client, you will have serious performance issues, because every SQL statement issued by Application Engine will have to be serialized and sent over to the Application Server for execution.

If you're running in the PeopleSoft Internet Architecture, this isn't an issue, since all PeopleCode automatically runs on a server in the PeopleSoft Internet Architecture.

PeopleCode Event Considerations

You need to include the CallAppEngine PeopleCode function within events that allow database updates because generally, if you're calling Application Engine, you're intending to perform database updates. This includes the following PeopleCode events:

- SavePreChange (Page)
- SavePostChange (Page)
- Workflow
- Message Subscription
- FieldChange

CallAppEngine cannot be used in an Application Engine PeopleCode action. If you need to access one Application Engine program from another Application Engine program, use the **CallSection** action.



For more information on CallSection, see Call Section Actions.

If **CallAppEngine** results in a failure, all database updates will be rolled back. All information the user entered into the component will be lost, as if the user pressed ESC.

Save Events Considerations

If you want to execute the Application Engine program based on an end user Save, use the **CallAppEngine** function within a Save event. When you use **CallAppEngine**, you should keep the following items in mind:

- No commits will occur during the entire program run.
- During **SavePreChange**, any modified rows in the page have not been written to the database.
- During **SavePostChange**, the modified rows have been written to the database. The Page Process issues one commit at the end of the Save cycle.

FieldChange Considerations

If you don't want the **CallAppEngine** call to depend on a Save event, you can also trigger **CallAppEngine** from a **FieldChange** event. When having a **FieldChange** event trigger **CallAppEngine**, keep the following items in mind:

- No commits will occur within the program called by **CallAppEngine**. The called program will remain a synchronous execution in the same unit of work.
- The normal **FieldChange** commit occurs, which frees any locks that the Application Engine program might have acquired.
- For best performance, PeopleSoft recommends running programs called from a **FieldChange/CallAppEngine** combination on the server and not on the client.
- Do not include a **DoSave** or **DoSaveNow** function in the same **FieldChange** event. Not only is this not allowed, but it also indicates that you should be including the **CallAppEngine** within a Save event.

Returns

None.

Parameters

<i>applid</i>	Specify the name of the Application Engine program you want to start.
---------------	---

statereclist

Specify an optional record object that provides initial values for a state record.

Example

The following calls the Application Engine program named MYAPPID, and passes initialization values.

```
&REC = CreateRecord(RECORD.INIT_VALUES);  
  
&REC.FIELD1.Value = "XYZ";  
  
/* set the initial value for INIT_VALUES.FIELD1 */  
  
CallAppEngine("MYAPPID", &REC);
```

Related Topics

ProcessRequest Class, Application Engine

Char

Syntax

Char(*n*)

Description

Char converts a numeric value to the corresponding character in the current character code set.

Returns

Returns a String representing the ANSI character corresponding to the number *n*.

Example

The examples set &LETTER to "P", then "S".

```
&LETTER = Char(80);  
&LETTER = Char(83);
```

Related Topics

Code, String, %Substring

CharType

Syntax

CharType(*source_str*, *char_code*)

Description

CharType determines whether the first character in *source_str* is of type *char_code*. The *char_code* is a number value representing a character set (see Parameters for details).

Japanese character sets can be recognized only if a Japanese operating system is running. If the function is unable to determine whether the first character in *source_str* belongs to the specified character set, the function returns a value of UNKNOWN (-1). This will occur if the application is running on a non-Japanese version of Windows or, if on a UNIX application server, the CharSet parameter in the PSADMIN table isn't set to SJIS.



For more information about character sets, see [Character Sets and Language Input/Output](#).

Returns

CharType returns one of the following Number values. You can check for the constant values instead of the numeric values if you prefer:

<i>Return Value</i>	<i>Constant</i>	<i>Description</i>
1	%CharType_Matched	Character is of type <i>char_code</i> .
0	%CharType_NotMatched	Character is not of type <i>char_code</i> .
-1	%CharType_Unknown	UNKNOWN: unable to determine whether character is of set <i>char_code</i> . This will occur if the application is running on a non-Japanese version of Windows or on a UNIX application server when the CharSet parameter of the PSADMIN table is not set to SJIS.

Parameters

<i>source str</i>	A String, the first character of which will be tested.
-------------------	--

<i>char code</i>	A Number representing the character type to be tested for.
------------------	--

The following table shows valid values for *char_code*. You can specify either a Character Code or a Constant:

Character Code	Constant	Character Set
0	%CharType_Alphanumeric	Alphanumeric (7-bit ASCII codes; A-Z, a-z, 1-9, punctuation)
1	%CharType_ExtendedLatin1	Extended Latin-1 characters (ISO8859-1 accents for Spanish, French, etc.)
2	%CharType_HankakuKatakana	Hankaku Katakana (single-byte Japanese Katakana)
3	%CharType_ZenkakuKatakana	Zenkaku Katakana (double-byte Japanese Katakana)
4	%CharType_Hiragana	Hiragana (Japanese)
5	%CharType_Kanji	Kanji (Japanese)
6	%CharType_DBAAlphaNumeric	Double-byte Alphanumeric (Japanese)
7,8,9		Reserved for future use
10	%CharType_JapanesePunctuation	Japanese punctuation



For more information about character sets, see Character Sets and Language Input/Output.

Example

This example tests to see if a character is Kanji:

```

&ISKANJI = CharType(&STRTOTEST, 5);

If &ISKANJI = 1 Then

    WinMessage("Character type is Kanji");

Else

    If &ISKANJI = 0 Then

        WinMessage("Character type is Kanji");

    Else

        WinMessage("Character type is UNKNOWN");

    End-If

End-If

```

Related Topics

ContainsCharType, ContainsOnlyCharType, ConvertChar

ChDir

Syntax

```
ChDir(path)
```

Description

ChDir, like the DOS ChDir command, changes the current directory on a drive. The drive and the directory are both specified in a *path* string.

Restrictions on Use in Three-Tier Mode and PeopleSoft Internet Architecture

Because it is specific to systems running MS-DOS or Windows, **ChDir** is a client-only function, which limits its use in three-tier mode and in PeopleSoft Internet Architecture applications.

- In three-tier mode **ChDir** can be used only in processing groups set to run on the client. If the **ChDir** function is called in a processing group running on the application server, a runtime error will occur.
- In PeopleSoft Internet Architecture applications, where all PeopleCode runs on the application server, **ChDir** cannot be used at all.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

Returns

Optionally returns a Boolean value indicating whether the function succeeded.

Example

Assume that the current directory is C:\USER\WINWORD. The following command is issued from a PeopleCode program:

```
ChDir("N:\PS\SQR");
```

The working directory is still C:\USER\WINWORD, but the current directory on drive N: is now \PS\SQR. The next time the user switches to N:, they will be in directory N:\PS\SQR.

Related Topics

ChDrive, GetCwd

ChDrive

Syntax

```
ChDrive(str_dr)
```

Description

ChDrive changes the current disk drive to the drive specified by *str_dr*, which is a string consisting of a valid drive letter followed by a colon, for example "C:".

Restrictions on Use in Three-Tier Mode and PeopleSoft Internet Architecture

Because it is specific to systems running MS-DOS or Windows, **ChDrive** is a client-only function, which limits its use in three-tier mode and in PeopleSoft Internet Architecture applications.

- In three-tier mode **ChDrive** can be used only in processing groups set to run on the client. If the **ChDrive** function is called in a processing group running on the application server, a runtime error will occur.
- In PeopleSoft Internet Architecture applications, where all PeopleCode runs on the application server, **ChDrive** cannot be used at all.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

Returns

Optionally returns a Boolean value indicating whether the function succeeded.

Example

Assume that the current directory is C:\USER\WINWORD. After the following statements execute:

```
ChDir("N:\PS\SQR");
```

```
ChDrive("N:");
```

the working directory is N:\PS\SQR.

Related Topics

CharType, GetCwd

CheckMenuItem

Syntax

```
CheckMenuItem(BARNAME.menubar_name, ITEMNAME.menuitem_name)
```

Description

CheckMenuItem changes the menu state by placing a check mark beside the menu item. It is useful for PeopleCode menu items that act as a toggle. To apply this function to a pop-up menu, use the PrePopup Event of the field with which the pop-up menu is associated. If you're using this function with a pop-up menu associated with a page (not a field), the earliest event you can use is the **PrePopup** event for the first "real" field on the page (that is, the first field listed in the **Order** view of the page in Application Designer.)

Restrictions on Use in Three-Tier Mode and PeopleSoft Internet Architecture

CheckMenuItem is a client-only function, which limits its use in three-tier mode and in PeopleSoft Internet Architecture applications.

- In three-tier mode **CheckMenuItem** can be used only in processing groups set to run on the client. If the **CheckMenuItem** function is called in a processing group running on the application server, a runtime error will occur.
- In PeopleSoft Internet Architecture applications, where all PeopleCode runs on the application server, **CheckMenuItem** cannot be used at all.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```
If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;
```

Returns

None.

Parameters

<i>menubar_name</i>	Name of the menu bar that owns the menu item, or, in the case of pop-up menus, the name of the pop-up menu that owns the menu item.
<i>menuitem_name</i>	Name of the menu item.

Example

```
CheckMenuItem(BARNAME.MYPOPUP1, ITEMNAME.DO_JOB_TRANSFER);
```

Related Topics

UnCheckMenuItem, DisableMenuItem, EnableMenuItem, HideMenuItem

Clean

Syntax

```
Clean(string)
```

Description

Clean removes all non-printable characters from a text *string* and returns the result as a String value. It is intended for use on text imported from other applications that contains characters that may not print correctly under Windows. Frequently, low-level characters appear at the beginning and end of each line of imported data, and they cannot be printed.

Returns

Returns a String value purged of non-printable characters.

Example

Because **Char(7)** is a non-printable character, the following **Clean** function returns a null string:

```
&CLEANSTR = Clean(Char(7));
```

Related Topics

Char, String, %Substring

ClearKeyList

Syntax

```
ClearKeyList( )
```


Description

ClearKeyList clears the current key list. This function is useful for programmatically setting up keys before transferring to another component.

Returns

Optionally returns a Boolean value indicating whether the function succeeded.

Example

The following example sets up a key list and then transfers the user to a page named PAGE_2.

```
ClearKeyList( );
AddKeyListItem(OPRID, OPRID);
AddKeyListItem(REQUEST_ID, REQUEST_ID);
SetNextPage("PAGE_2");
TransferPage( );
```

Related Topics

AddKeyListItem

ClearSearchDefault

Syntax

```
ClearSearchDefault([recordname.]fieldname)
```

Description

ClearSearchDefault disables default processing for the specified field, reversing the effects of a previous call to **SetSearchDefault**. If search default processing is cleared for a record field, the default value specified in the record field properties for that field will not be assigned when the field appears in a search dialog box. This function is effective only when used in SearchInit PeopleCode.



This function remains for backward compatibility only. Use the SearchDefault Field class property instead. See Data Buffer Access.

Parameters

[*recordname.*]*fieldname*

The name of the target field, which is a search key or alternate search key that is about to appear in a search dialog box. The *recordname* prefix is required if the call to **ClearSearchDefault** is in a record definition other than *recordname*.

Returns

Optionally returns a Boolean value indicating whether the function succeeded.

Related Topics

ClearSearchEdit, SetSearchDefault, SetSearchEdit, SetSearchDialogBehavior, and Search Processing in Update Modes

ClearSearchEdit

Syntax

```
ClearSearchEdit([recordname.]fieldname)
```

Description

ClearSearchEdit reverses the effects of a previous call to **SetSearchEdit**. If **ClearSearchEdit** is called for a specific field, the edits specified in the record field properties will not be applied to the field when it occurs in a search dialog.



This function remains for backward compatibility only. Use the SearchEdit Field class property instead. See Data Buffer Access.

Parameters

[*recordname.*] *fieldname*

The name of the target field, which is a search key or alternate search key about to appear in a search dialog box. The *recordname* prefix is required if the call to **ClearSearchEdit** is in a record definition other than *recordname*.

Returns

Optionally returns a Boolean value indicating whether the function succeeded.

Related Topics

SetSearchEdit, SetSearchDefault, ClearSearchDefault, SetSearchDialogBehavior, and Search Processing in Update Modes

Code

Syntax

```
Code(str)
```

Description

Code returns a numeric code for the first character in a text *str*. (Normally you would pass this function a single character.) The return code depends on the character set in use by the operating system. If the first character of *str* consists of a lead byte and trailing byte (in a double-byte character set), **Code** returns the numeric equivalent of both bytes.

Returns

Returns a Number value equal to the character code for the first character in *str*.

Related Topics

Char, String, %Substring, Codeb

Codeb

Syntax

```
Code(str)
```

Description

Codeb is a byte-oriented variant of **Code**, which determines the value of the first byte in *str*.

Returns

Returns a Number value equal to the first byte in *str*.

Related Topics

Char, String, %Substring, Code

CommitWork

Syntax

```
CommitWork()
```

Description

The **CommitWork** function commits pending changes (inserts, updates, and deletes) to the database.

Considerations for using CommitWork

- This function is only available in Application Engine PeopleCode. If you use it any other PeopleCode program, you'll receive a runtime error.
- This function only applies to an Application Engine program that's running in batch (not

online). If the program is invoked via CallAppEngine, the **CommitWork** will be ignored. The same is true for commit settings at the section or step level.

- This function can only be used in an Application Engine program that has restart disabled. If you try to use this function in a program that doesn't have restart disabled, you'll receive a runtime error.



For more information about restart and batch in Application Engine, see Restarting Application Engine Programs.

The **CommitWork** function is only useful when you are doing row-at-a-time SQL processing in a single PeopleCode program, and you need to commit without exiting the program. In a typical Application Engine program, SQL commands are split between multiple Application Engine actions that fetch, insert, update, or delete application data. Therefore, you would use the section or step level commit settings to manage the commits. This is the recommended approach.

However, with some types of Application Engine programs that are PeopleCode intensive, it can be difficult to exit the PeopleCode in order to perform a commit. This is the only time when the **CommitWork** function should be used.

Restart Considerations

Disabling restart on a particular program means that the application **itself** is intrinsically self-restartable: it can be re-run from the start after an abend, and it will perform any initialization, cleanup, and filtering of input data to ensure that everything gets processed once and only once, and that upon successful completion, the database is in the same state it would have been if no abend occurred.

Set-based applications should always use Application Engine's restart. Only row-by-row applications that have restart built into them can benefit from disabling Application Engine's restart.

Consider the following aspects to managing restarts in a self-restarting program:

- **Locking input transactions** (optional). If the input data can change, and if it's important not to pick up new data during a restart, there should be logic to lock transactions at the start of the initial run (such as updating rows with current Process Instance). The program should first check whether any rows have the current Process Instance (that is, is the process being restarted from the top after an abend?). If no rows found, do the update.

In some cases it is acceptable for a restarted process to pick up new rows, so that locking is not necessary. It depends on your application.

Also, if you don't lock transactions, you must provide some other way to manage concurrent processing of the same program. You don't want two simultaneous runs of the same program to use the same data, so you must have some strategy for dividing up the data such that there is no overlap.

- **Filtering input transactions** (required). After an input transaction is processed, the row should be updated accordingly (that is, setting a "processed" flag). The SELECT statement

that drives the main processing loop should include a WHERE condition to filter out rows that have already been processed.



For more information about restarting Application Engine programs, see Advanced Development.

Returns

A boolean value, True if data was successfully committed, False otherwise.

Example

The following example fetches rows and processes them one at a time, committing every one hundred iterations. Because restart is disabled, you must have a marker indicating which rows have been processed, and use it in a conditional clause that filters out those rows.

```
Local SQL &SQL;

Local Record &REC;

Local Number &COUNT;

&REC = CreateRecord(RECORD.TRANS_TBL);

&SQL = CreateSQL("%SelectAll(:1) WHERE PROCESSED <> 'Y'");

&COUNT = 0;

&SQL.Execute(&REC);

While &SQL.Fetch(&REC)

    If (&COUNT > 99) Then

        &COUNT = 0;

        CommitWork(); /* commit work once per 100 iterations */

    End-if;

    &COUNT = &COUNT + 1;

    /* do processing */

    ...

/* update transaction as "processed" */
```

```
&REC.PROCESSED.Value = 'Y';

&REC.Update();

End-While;
```

Related Topics

Introducing Application Engine

CompareLikeFields

Syntax

```
CompareLikeFields (from, to)
```

where *from* and *to* are constructions that reference rows of data on specific source and target records in the component buffer; each have the following syntax:

```
level, scrollpath, target_row
```

and where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]]  
RECORD.target_recname
```

To prevent ambiguous references, you can use **SCROLL**.*scrollname*, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation

Description

CompareLikeFields compares fields in a row on a specified source record to similarly named fields on a specified target record. If all of the like-named fields have the same data value, **CompareLikeFields** returns True; otherwise it returns False.



This function remains for backward compatibility only. Use the CompareFields record class method instead. See also Data Buffer Access.

Returns

Returns a Boolean value indicating whether all of the like-named fields in the two records have the same data value.

Parameters

<i>from</i>	A placeholder for a construction (<i>level</i> , <i>scrollpath</i> , <i>target_row</i>) that references the first row in the comparison.
<i>to</i>	A placeholder for a construction (<i>level</i> , <i>scrollpath</i> , <i>target_row</i>) that references the second row in the comparison.
<i>level</i>	Specifies the scroll level for the target level scroll.
<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
<i>target_row</i>	Specifies the row number of each target row on its scroll level.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation

Example

The following example compares the like-named fields in two rows on levels 1 (&L1_ROW) and 2 (&L2_ROW) and returns True if all of the like-named fields in the two rows have the same value.

```
&L1_ROW = 1;
&L2_ROW = 1;
if CompareLikeFields(1, RECORD.BUS_EXPENSE_PER, &L1_ROW, 2,
RECORD.BUS_EXPENSE_PER, 1, RECORD.BUS_EXPENSE_DTL, &L2_ROW) then
    WinMessage("The fields match.");
end-if;
```

Related Topics

CopyFields

Component

Syntax

```
Component data_type &var_name
```

Description

The **Component** statement enables you to declare PeopleCode Component variables. A Component variable, once declared in any PeopleCode program, will remain in scope throughout the life of the component.

The variable must be declared with the Component statement in *every* PeopleCode program in which it is used.

Declarations appear at the beginning of the program, intermixed with function declarations.



Because a function can be called from anywhere, you cannot declare any variables within a function. You will get a design time error if you try.

The system automatically initializes temporary variables. Declared variables always have values appropriate to their declared type. Undeclared variables are initialized as null strings.

Not all PeopleCode data types can be declared as Component. For example, ApiObject or Interlink data types can only be declared as Local.



For more information on different PeopleCode data types, see Data Types.

Parameters

<i>data_type</i>	Specify a PeopleCode data type.
<i>&var_name</i>	A legal variable name.

Example

```
Component string &PG_FIRST;
```

Related Topics

Local, Global

ComponentChanged

Syntax

```
ComponentChanged()
```

Description

ComponentChanged determines whether a component has changed since the last save, whether by the user or by PeopleCode.

Returns

Returns a Boolean value: True if the component has changed.

Example

```
If ComponentChanged() Then
    /* do some stuff */
End-if;
```

ContainsCharType

Syntax

```
ContainsCharType(source_str, char_code)
```

Description

ContainsCharType determines if any of the characters in *source_str* are of type *char_code*. The *char_code* is a number value representing a character set (see Parameters for details).

Japanese character sets can be recognized only if a Japanese operating system is running. UNIX application servers support only alphanumeric and extended Latin character sets. If the function is unable to determine whether *source_str* contains any characters belonging to the specified character set, the function returns a value of UNKNOWN (-1). This will occur if the application is running on a non-Japanese version of Windows or, if on a UNIX application server, the CharacterSet parameter in the PSADMIN table isn't set to SJIS.

Returns

ContainsCharType returns one of the following Number values. You can check for the constant instead of the numeric value if you prefer:

Return Value	Constant	Description
1	%CharType_Matched	String contains at least one character of set <i>char_code</i> .
0	%CharType_NotMatched	String contains no characters of set <i>char_code</i> .
-1	%CharType_Unknown	UNKNOWN: unable to determine whether one or more characters in string is of set <i>char_code</i> . This will occur if the application is running on a non-Japanese version of Windows or on a UNIX application server when the CharacterSet parameter of the PSADMIN table is not set to SJIS.

Parameters

source_str String to be examined.

char_code A Number value representing the character set to be tested for. The following table shows valid values. You can specify either a Character Code or a Constant:

<i>Character Code</i>	<i>Constant</i>	<i>Character Set</i>
0	%CharType_Alphanumeric	Alphanumeric (7-bit ASCII codes; A-Z, a-z, 1-9, punctuation)
1	%CharType_ExtendedLatin1	Extended Latin-1 characters (ISO8859-1 accents for Spanish, French, etc.)
2	%CharType_HankakuKatakana	Hankaku Katakana (single-byte Japanese Katakana)
3	%CharType_ZenkakuKatakana	Zenkaku Katakana (double-byte Japanese Katakana)
4	%CharType_Hiragana	Hiragana (Japanese)
5	%CharType_Kanji	Kanji (Japanese)
6	%CharType_DBAAlphaNumeric	Double-byte Alphanumeric (Japanese)
7,8,9		Reserved for future use
10	%CharType_JapanesePunctuation	Japanese punctuation



For more information about character sets, see Character Sets and Language Input/Output.

Example

This example tests to see if the string contains any Kanji:

```
&ANYKANJI = ContainsCharType(&STRTOTEST, 5);

If &ANYKANJI = 1 Then

    WinMessage("There are Kanji characters");

Else
```

```

If &ANYKANJI = 0 Then

    WinMessage("There are no Kanji characters");

Else

    WinMessage("UNKNOWN");

End-If

End-If

```

Related Topics

ContainsCharType, ContainsOnlyCharType, ConvertChar

ContainsOnlyCharType

Syntax

```
ContainsOnlyCharType(source_str, char_code_list)
```

Where ***char_code_list*** is a list of character set codes in the form:

```
char_code_1 [, char_code_2]...
```

Description

ContainsOnlyCharType determines whether every character in ***source_str*** belongs to one or more of the character sets in ***char_code_list***. See Parameters for a list of valid character code values.

Japanese character sets can be recognized only if a Japanese operating system is running. UNIX application servers support only alphanumeric and extended Latin character sets. If the function is unable to recognize one or more characters in ***source_str***, the function returns a value of UNKNOWN (-1). This will occur if the application is running on a non-Japanese version of Windows or, if on a UNIX application server, the CharacterSet parameter in the PSADMIN table isn't set to SJIS.

Returns

ContainsOnlyCharType returns one of the following Number values. You can check for the constant instead of the numeric value, if you prefer:

<i>Return Value</i>	<i>Constant</i>	<i>Description</i>
1	%CharType_Matched	String contains only characters belonging to the sets listed in <i>char_code_list</i> .
0	%CharType_NotMatched	String contains one or more characters that do not belong to sets listed in

		<i>char_code_list.</i>
-1	%CharType_Unknown	UNKNOWN: unable to determine whether one or more characters in string is of set <i>char_code</i> . This will occur if the application is running on a non-Japanese version of Windows or on a UNIX application server when the CharSet parameter in the PSADMIN table is not set to SJIS.



If any character is determined to be UNKNOWN, the return value is UNKNOWN.

Parameters

Source_str String to be examined.

char_code_list A comma-separated list of character set codes.

char_code_n Either a Number value identifying a character set, or a constant. The following table shows valid values. You can specify either a character code or a constant:

Character Code	Constant	Character Set
0	%CharType_Alphanumeric	Alphanumeric (7-bit ASCII codes; A-Z, a-z, 1-9, punctuation)
1	%CharType_ExtendedLatin1	Extended Latin-1 characters (ISO8859-1 accents for Spanish, French, etc.)
2	%CharType_HankakuKatakana	Hankaku Katakana (single-byte Japanese Katakana)
3	%CharType_ZenkakuKatakana	Zenkaku Katakana (double-byte Japanese Katakana)
4	%CharType_Hiragana	Hiragana (Japanese)
5	%CharType_Kanji	Kanji (Japanese)
6	%CharType_DBAAlphaNumeric	Double-byte Alphanumeric (Japanese)

7,8,9		Reserved for future use
10	%CharType_JapanesePunctuation	Japanese punctuation



For more information about character sets, see Character Sets and Language Input/Output.

Example

This example tests to see if the string is only Hiragana or Punctuation:

```
&ONLYHIRAGANA = ContainsOnlyCharType(&STRTOEST, 4, 10);

If &ONLYHIRAGANA = 1 Then

    WinMessage("There are only Hiragana and Punctuation characters");

Else

    If &ONLYHIRAGANA = 0 Then

        WinMessage("Mixed characters");

    Else

        WinMessage("UNKNOWN");

    End-If

End-If
```

Related Topics

CharType, ContainsCharType, ConvertChar

ConvertChar

Syntax

```
ConvertChar(source_str, source_str_category, output_str, target_char_code)
```

Description

ConvertChar converts every character in *source_str* to type *target_char_code*, if possible, and places the converted string in *output_str*. **ConvertChar** supports the following conversion:

- Conversion among the Japanese Hankaku Katakana, Zenkaku Katakana, and Hiragana character sets.

- Conversion of Japanese Hankaku Katakana, Zenkaku Katakana, and Hiragana character sets to ASCII single-byte alphanumeric characters.
- Conversion of Japanese double-byte alphanumeric characters to their ASCII single-byte alphanumeric equivalents.
- Conversion of Japanese double-byte punctuation characters to their ASCII single-byte equivalents.

Other *source_str* and *target_char_code* combinations are either passed through without conversion, or not supported. Character types 0 and 1 (alphanumeric and extended Latin-1) are always passed through to *output_str* without conversion. See the Supported Conversions section later in this reference entry for details.

Japanese character sets can be recognized only if the application is running on a Japanese version of Windows, or, if on a UNIX application server, the CharacterSet parameter in the PSADMIN table is set to SJIS.

If **ConvertChar** is unable to determine whether the characters in *source_str* belong to the specified character set, the function returns a value of UNKNOWN (-1). This will occur if the application is running on a non-Japanese version of Windows or on a UNIX application server when the CharacterSet parameter in the PSADMIN table has not been set to SJIS. UNKNOWN characters are echoed as-is to the output string. If *source_str* can be partially converted, **ConvertChar** will partially convert string, echo the remaining characters to the output string as-is, and return a value of -2 (Completed with Issues).

Returns

Returns either a Number or a constant with one of the following values, depending on what you're checking for:

<i>Value</i>	<i>Constant</i>	<i>Description</i>
1	%ConvertChar_Success	String successfully converted.
0	%ConvertChar_NotConverted	String not converted.
-1	%ConvertChar_Unknown	UNKNOWN condition: this will occur if the application is running on a non-Japanese version of Windows or on a UNIX application server when the CharacterSet parameter of the PSADMIN table is not set to SJIS.
-2	%ConvertChar_Issues	Completed with issues. Conversion executed but there were one or more characters encountered that were either not recognized,

		or whose conversion is not supported.
--	--	---------------------------------------



If any character cannot be translated, it is echoed as-is to **output_str**. **Output_str** could therefore be a mixture of converted and non-converted characters.

Parameters

Source_str String to be converted.

Source_str_category Language category of input string. You can specify either a number or a constant.

Category	Constant	Description
0	%ConvertChar_Alphanumeric	Alphanumeric (7-bit ASCII codes: A-Z, a-z, 1-9, punctuation)
1	%ConvertChar_ExtendedLatin1	Extended Latin-1 Characters (ISO8859-1 accents, Spanish, French etc.)
2	%ConvertChar_Japanese	Japanese (any)

Output_str A String variable to receive the converted string.

Target_char_code Either a Number or a constant representing the conversion target character type. You can specify either a Character Code or a Constant:

Character Code	Constant	Description
0	%CharType_Alphanumeric	Alphanumeric (7-bit ASCII codes: A-Z, a-z, 1-9, punctuation)
2	%CharType_HankakuKatakana	Hankaku Katakana (Single-byte Japanese Katakana)
3	%CharType_ZenkakuKatakana	Zenkaku Katakana (Double-byte Japanese Katakana)

4	%CharType_Hiragana	Hiragana (Japanese)
6	%CharType_DBAAlphaNumeric	Double-byte AlphaNumeric (Japanese)

The following target values are not supported; if the source string is of the same type as any of these values, then the string is passed through without conversion.

Character Code	Constant	Description
1	%CharType_ExtendedLatin1	Extended Latin-1 characters (ISO8859-1 accents for Spanish, French, etc.)
5	%CharType_Kanji	Kanji (Japanese)
10	%CharType_JapanesePunctuation	Japanese punctuation

Supported Conversions

The following table shows which conversions are supported, which are passed through without conversion, and which are not supported:

Source	Target	Conversion
0 (Alphanumeric ASCII)	0-6 (All supported character types)	Pass through without conversion
1 (Extended Latin-1 characters)	0-6 (All supported character sets)	Pass through without conversion
2 (Hankaku Katakana)	0 (Alphanumeric)	Conversion supported
	1 (Extended Latin)	Not supported
	2 (Hankaku Katakana)	Pass through without conversion
	3 (Zenkaku Katakana)	Conversion supported
	4 (Hiragana)	Conversion supported
	5 (Kanji)	Not supported
	6 (Double-byte alphanumeric)	Not supported
3 (Zenkaku Katakana)	0 (Alphanumeric)	Conversion supported
	1 (Extended Latin)	Not supported
	2 (Hankaku Katakana)	Conversion supported
	3 (Zenkaku Katakana)	Pass through without

		conversion
	4 (Hiragana)	Conversion supported
	5 (Kanji)	Not supported
	6 (Double-byte alphanumeric)	Not supported
4 (Hiragana)	0 (Alphanumeric)	Conversion supported
	1 (Extended Latin)	Not supported
	2 (Hankaku Katakana)	Conversion supported
	3 (Zenkaku Katakana)	Conversion supported
	4 (Hiragana)	Pass through without conversion
	5 (Kanji)	Not supported
	6 (Double-byte alphanumeric)	Not supported
5 (Kanji)	0-4, 6	Not supported
	5 (Kanji)	Pass through without conversion
6 (Double-byte alphanumeric)	0 (Alphanumeric)	Conversion supported
	1-5	Not supported
	6 (Double-byte alphanumeric)	Pass through without conversion
10 (Japanese punctuation)	0 (Alphanumeric)	Conversion supported
	1 (Extended Latin)	Not supported
	3-6, 10	Pass through without conversion



For more information about character sets, see Character Sets and Language Input/Output.

Example

This example attempts to convert a string to Hiragana:

```
&RETVALUE = ConvertChar(&INSTR, 2, &OUTSTR, 4);

If &RETVALUE = 1 Then

    WinMessage("Conversion to Hiragana successful");

Else
```

```

If &RETVALUE = 0 Then

    WinMessage("Conversion to Hiragana failed");

Else

    If &RETVALUE = - 1 Then

        WinMessage("Input string is UNKNOWN character type.");

    Else

        WinMessage("Some characters could not be converted.");

    End-If

End-If

End-If

```

Related Topics

CharType, ContainsCharType, ContainsOnlyCharType

ConvertCurrency

Syntax

```

ConvertCurrency(amt, currency_cd, exchn_g_to_currency, exchn_g_rt_type, effdt,
converted_amt [, error_process [, round] [, rt_index]])

```

Description

Use the **ConvertCurrency** function to convert between currencies. The result of the conversion is placed in a variable passed in *converted_amt*.



For more information about currency exchange, see Controlling Currency Display Format.

Returns

ConvertCurrency returns a Boolean value where True means a conversion rate was found and *converted_amt* calculated, and False means a conversion rate was not found and a value of one (1) was used.

Parameters

<i>Amt</i>	The currency amount to be converted.
<i>Currency_cd</i>	The currency in which the <i>amt</i> is currently expressed.

<i>Exchng_to_currency</i>	The currency to which the <i>amt</i> should be converted.
<i>Exchng_rt_type</i>	The currency exchange rate to be used. This is the value of the RT_TYPE field in the RT_RATE table of RT_DFLT_VW.
<i>Effdt</i>	The effective date of the conversion to be used.
<i>Converted_amt</i>	The resulting converted amount. You must supply a variable for this parameter. If a conversion rate cannot be found, <i>converted_amt</i> is set equal to <i>amt</i> .
<i>Error_process</i>	<p>An optional string that, if specified, contains one of the following values:</p> <p>"F" - Produce a fatal error if a matching conversion rate is not found.</p> <p>"W" - Produce a warning message box if a matching conversion rate is not found.</p> <p>"I" - Or other – return without producing a message box.</p> <p>If <i>error_process</i> is not specified, it defaults to Fatal ("F").</p>
<i>Round</i>	Optional Boolean value indicating whether to round <i>converted_amt</i> to the smallest currency unit. If omitted, round defaults to False.
<i>rt_index</i>	An optional string to indicate which exchange rate index should be used to retrieve the exchange rate. If omitted, the Default Exchange Rate index (as specified on the Market Rate index definition) will be used.



If the currency exchange rate is changed in a PeopleSoft table, this change will *not* be reflected in an already open page until the user closes the page, then opens it again.

Example

```

rem **-----**;
```

```

rem *   Convert the cost & accum_depr fields if books *;
```

```

rem * use different currencies.  *;
```

```

rem **-----**;
```

```

rem;
```

```

    If &FROM_CUR <> &PROFILE_CUR_CD Then
```

```

&CON_COST_FROM = &COST_COST;

&CON_ACC_DEPR_FROM = &COST_ACCUM;

ConvertCurrency(&CON_COST_FROM, &FROM_CUR, &PROFILE_CUR_CD, RT_TYPE,
TRANS_DT, &CON_COST_TO, "F");

UpdateValue(COST_NON_CAP.COST, &COST_ROW_CUR, &CON_COST_TO);

Else

UpdateValue(COST_NON_CAP.COST, &COST_ROW_CUR, &COST_COST);

End-If;

UpdateValue(COST_NON_CAP.FROM_CUR, &COST_ROW_CUR, &PROFILE_CUR_CD);

UpdateValue(COST_NON_CAP.OPRID, &COST_ROW_CUR, %UserIdId);

```

ConvertDatetimeToBase

Syntax

```
ConvertDatetimeToBase(textdatetime, {timezone | "Local" | "Base"});
```

Description

ConvertDatetimeToBase converts the text value *textdatetime* to a datetime value, then further converts it to the base time. This function automatically calculates whether daylight savings time is in effect for the given *textdatetime*.

The system's base time zone is specified on the PSOPTIONS table.



For more information about setting the base time, see PeopleTools Utilities.

Parameters

<i>textdatetime</i>	Specify a datetime value represented as text (e.g., "01/01/99 3:00 PM")
<i>timezone</i> Local Base	Specify a value for converting <i>textdatetime</i> . The valid values are: <i>timezone</i> - a time zone abbreviation or a field reference to be used for converting <i>textdatetime</i> . Local - use the local time zone for converting <i>textdatetime</i> .

Base - use the base time zone for converting *textdatetime*.

Returns

Returns a datetime value in the base time zone.

Example

In the following example, `&DATETIMEVAL` would have a datetime value of "01-01-1999-07:00:00.000000", assuming the Base time (as defined in `PSOPTIONS`) was PST.

```
&DATETIMEVAL= ConvertDateTimeToBase("01/01/99 10:00:00AM", "EST");
```

Related Topics

`ConvertTimeToBase`, `FormatDateTime`, `IsDaylightSavings`, `DateTimeToTimeZone`, `TimeToTimeZone`, `TimeZoneOffset`

ConvertRate

Syntax

```
ConvertRate(Rate, In_Frequency, Out_Frequency)
```

Description

The **ConvertRate** function converts a rate between various compounding frequencies.

Parameters

<i>Rate</i>	The rate to be converted. This parameter takes a number value.
<i>In_Frequency</i>	The frequency of the rate to be converted from. This parameter takes an array of number, with two elements. The first element is periodicity, (for example, if you chose daily compounding, 1 would represent daily while 7 would represent weekly.) The second element is the unit of measure of frequency. The valid values for the second element are:

<i>Value</i>	<i>Description</i>
0	continuous compounding
1	daily compounding
2	monthly compounding

Value	Description
3	yearly compounding

Out_Frequency

The frequency of the rate to be converted to. This parameter takes an array of number, with two elements. The first element is periodicity, (for example, if you chose daily compounding, 1 would represent daily while 7 would represent weekly.) The second element is the unit of measure of frequency. The valid values for the second element are:

Value	Description
0	continuous compounding
1	daily compounding
2	monthly compounding
3	yearly compounding

Returns

A number representing the converted rate.

Example

The following example converts from days to years.

```
Local array of number &In, &Out;  
  
Local number &rate, &NewRate;  
  
&rate = 0.01891;  
  
&In = CreateArray(0, 0);  
  
&In[1] = 1; /* daily */  
  
&In[2] = 1; /* compound_days */  
  
&Out = CreateArray(0, 0);  
  
&Out[1] = 1; /* one year */  
  
&Out[2] = 3; /* compound_years */
```

```
&NewRate = ConvertRate(&rate, &In, &Out);
```

Related Topics

RoundCurrency

ConvertTimeToBase

Syntax

```
ConvertTimeToBase(texttime,{timezone | "Local" | "Base"});
```

Description

ConvertTimeToBase converts the text value *texttime* to a Time value and converts it to the base time. This function automatically calculates whether daylight savings time is in effect for the given *texttime*.

This function is useful for users to convert constant times in specific time zones into database time. For example, there is a deadline for completing Federal Funds transfers by 3:00 PM Eastern Time. **ConvertTimeToBase** will do this conversion, taking into account daylight savings time. The date used to calculate whether daylight savings time is in effect is the current date.

The system’s base time zone is specified on the PSOPTIONS table.



For more information about setting the base time, see PeopleTools Utilities.

Parameters

<i>texttime</i>	Specify a time value represented as text (e.g., "3:00 PM")
<i>timezone</i> Local Base	<p>Specify a value for converting <i>textdatetime</i>. The valid values are:</p> <p><i>timezone</i> - a time zone abbreviation or a field reference to be used for converting <i>textdatetime</i>.</p> <p>Local - use the local time zone for converting <i>textdatetime</i>.</p> <p>Base - use the base time zone for converting <i>textdatetime</i>.</p>

Returns

Returns a time value in the base time zone.

Example

In the following example, `&TIMEVAL` would have a time value of "07:00:00.000000", assuming the Base time (as defined in `PSOPTIONS`) was PST.

```
&TEXTTIME = ConvertTimeToBase("01/01/99 10:00:00AM", "EST");
```

Related Topics

`ConvertDatetimeToBase`, `FormatDateTime`, `IsDaylightSavings`, `DateTimeToTimeZone`, `TimeToTimeZone`, `TimeZoneOffset`

CopyFields

Syntax

```
CopyFields(from, to)
```

where *from* and *to* are constructions that reference rows of data on specific source and target records in the component buffer; each have the following syntax:

```
level, scrollpath, target_row
```

and where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]]  
RECORD.target_recname
```

To prevent ambiguous references, you can also use `SCROLL.scrollname`, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see [References Using Scroll Path Syntax and Dot Notation](#).

Description

CopyFields copies like-named fields from a row on the specific source record to a row on the specific target record.



This function remains for backward compatibility only. Use the `CopyFieldsTo` or `CopyChangedFieldsTo` record class methods instead. See [Data Buffer Access](#). Also see `CopyTo Row` class method, as well as the `CopyTo Rowset` class method.

Returns

Optionally returns a Boolean value indicating whether the function succeeded.

Parameters

<i>from</i>	A placeholder for a construction (<i>level</i> , <i>scrollpath</i> , <i>target_row</i>) that references the first row in the comparison.
<i>to</i>	A placeholder for a construction (<i>level</i> , <i>scrollpath</i> , <i>target_row</i>) that references the second row in the comparison.
<i>level</i>	Specifies the scroll level for the target level scroll.
<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
<i>target_row</i>	Specifies the row number of each target row on its scroll level.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Example

The following example copies fields from PO_RECEIVED_INV (level 1 scroll) from row &ROW to PO_RECV_INV_VW (level 1 scroll), row &LOC_ROW:

```
CopyFields(1, RECORD.PO_RECEIVED_INV, &ROW, 1, RECORD.PO_RECV_INV_VW, &LOC_ROW);
```

Related Topics

CompareLikeFields

CopyRow

Syntax

```
CopyRow(destination_row, source_row)
```

Description

CopyRow copies the data from one row to another row. *destination_row* is the row number to which you want the *source_row* data values copied. The two rows, as well as the PeopleCode function call, must all be located on the same scroll level.



This function remains for backward compatibility only. Use the CopyTo Row class method instead. See Data Buffer Access. Also see CopyTo Rowset class method, as well as CopyFieldsTo and CopyChangedFieldsTo record class methods.

Parameters

<i>destination_row</i>	Row number of row to which to copy data.
<i>source_row</i>	Row number of row from which to read data.

Example

This example uses **CopyRow** to give an inserted row the same values as the previous row:

```
/* Get the row number of the inserted row and the previous row */
&NEW_ROW_NUM = CurrentRowNumber();
&LAST_ROW_NUM = &NEW_ROW_NUM - 1;
/* Copy the data from the previous row into the inserted row */
CopyRow(&NEW_ROW_NUM, &LAST_ROW_NUM);
```

Cos**Syntax**

Cos(*angle*)

Description

Cos calculates the cosine of the given angle (adjacent / hypotenuse).

Parameters

<i>angle</i>	A value in radians.
--------------	---------------------

Returns

A real number between **-1.00** and **1.00**.

Example

The following example returns the cosine of an angle measuring **1.2** radians:

```
&MY_RESULT = Cos(1.2);
```

Related Topics

Acos, Asin, Atan, Cot, Degrees, Radians, Sin, Tan

Cot**Syntax**

Cot(*angle*)

Description

Cot calculates the cotangent of the given angle (adjacent / opposite, that is, the reciprocal of the tangent).

Parameters

<i>angle</i>	A value in radians, excluding 0 . If the input value is 0 , you'll see an error message at runtime ("Decimal arithmetic error occurred. (2,110)"). Adjust your code to provide a valid input value.
--------------	---



In theory, all values of *angle* such that *angle mod pi* = **0** are not valid for this function, because inputs approaching such values produce results that tend toward infinity. In practice, however, no computer system can represent such values exactly. Thus, for example, the statement **Cot(Radians(180))** will produce a number close to the largest value PeopleCode can represent, rather than an error.

Returns

A real number.

Example

The following example returns the cotangent of an angle measuring **1.2** radians:

```
&MY_RESULT = Cot(1.2);
```

Related Topics

Acos, Asin, Atan, Cos, Degrees, Radians, Sin, Tan

CreateArray

Syntax

```
CreateArray(paramlist)
```

Where *paramlist* is an arbitrary-length list of values in the form:

```
param1 [, param2] ...
```

Description

CreateArray constructs an array and returns a reference to it.

The type of the first parameter is used to determine the type of array that is built. That is, if the first parameter is of type NUMBER, an array of number is built. If there is no first parameter an empty array of ANY is built.

The **CreateArray** function uses flattening and promotion as required to convert subsequent values into suitable elements of the array.

Parameters

paramlist Specify a list of values to be used as the elements of the array.

Returns

Returns a reference to the array.

Example

```
Local Array of Array of Number &AAN;

Local Array of Number &AN;

&AAN = CreateArray(CreateArray(1, 2), CreateArray(3, 4), 5);

&AN = CreateArray(6, &AAN[1]);
```

&AAN is a two dimensional array with three elements: a one-dimensional array with 1 and 2 as elements, a one-dimensional array with 3 and 4, and a one-dimensional array with only the element 5. The last parameter to Array was promoted to a one-dimensional array. &AN is a one-dimensional array with 3 elements: 6, 1, and 2. The last parameter to Array in the last line was flattened into its two elements.



For more information, see Flattening and Promotion.

Related Topics

CreateArrayRept, Split, and Array Class

CreateArrayRept

Syntax

```
CreateArrayRept(val, count)
```

Description

CreateArrayRept creates an array that contains *count* copies of *val*. If *val* is itself an array, the created array has one higher dimension, and each element (sub-array) is the array reference *val*.

The type of the first parameter (*val*) is used to determine the type of array that is built. That is, if the first parameter is of type NUMBER, an array of number is built. If *count* is zero, **CreateArrayRept** creates an empty array, using the *val* parameter for the type.

If you are making an array that is multi-dimensional, *val* will be the subarray used as the elements.

To make the subarrays distinct, use the Clone method. For example:

```
&A = CreateArrayRept (&AN, 4).Clone();
```



For more information, see Clone method.

Parameters

<i>val</i>	A value of any type.
<i>count</i>	The number of copies of <i>val</i> contained in the array.

Returns

Returns a reference to the array.

Example

The following code sets &A to a new empty array of string:

```
&A = CreateArrayRept ("", 0);
```

The following code sets &A to a new array of ten zeroes:

```
&A = CreateArrayRept (0, 10);
```

The following code sets &AAS to a new array of array of strings, with two subarrays:

```
&AAS = CreateArrayRept (CreateArray ("one", "two"), 2);
```

Note that in this case, both elements (rows) of &AAS contain the *same* subarray, and changing the value of an element in one of them will change it in *both* of them. To get distinct subarrays, use the Clone method:

```
&AAS = CreateArrayRept (CreateArray ("one", "two"), 2).Clone();
```

Related Topics

CreateArray, Split, and Array Class

CreateJavaArray

Syntax

CreateJavaArray (*ElementClassName*[], *NumberOfElements*)

Description

The **CreateJavaArray** function allows you to create a Java array without knowing the number or value of the elements.

When you create an array in Java, you already know the number of elements in the array. If you don't know the number of elements in the array, but you want to use a Java array, use the **CreateJavaArray** function in PeopleCode. This will create a Java object that is a Java array, and you can pass in the number of elements that are to be in the array.

The first index in a Java array is 0. PeopleCode arrays start at 1.

You would do the following to specify this type of array in Java:

```
new ElementClassName[NumberOfElements] ;
```

Parameters

<i>ElementClassName</i> []	Specify the array class name. This parameter takes a string value.
<i>NumberOfElements</i>	Specify the number of elements in the array. This parameter takes a number value.

Returns

A Java object

Related Topics

Internet Script Classes, CreateJavaObject, GetJavaClass

CreateJavaObject

Syntax

CreateJavaObject (*ClassName* [*ConstructorParams*])

Where *ConstructorParams* has the form

```
Argument1 [, argument2] . . .
```

Description

The **CreateJavaObject** function creates a Java object that can be manipulated in your PeopleCode.



Note. If you create a class that you want to call using CreateJavaObject, it must be located in a directory specified in the PS_CLASSPATH environment variable.



For more information, see Internet Script Classes

You can use the CreateJavaObject function to create a Java array. If *ClassName* is the name of an array class (ending with []), *ConstructorParams* are used to initialize the array.

In Java, you would do the following to initialize an array:

```
intArray = new int[]{1, 2, 3, 5, 8, 13};
```

Do the following to initialize a Java array from PeopleCode:

```
&IntArray = CreateJavaObject("int[]", 1, 2, 3, 5, 8, 13);
```

If you want to initialize a Java array without knowing the number of parameters until runtime, use the CreateJavaArray function.

Parameters

<i>ClassName</i>	Specify the name of an already existing class.
<i>ConstructorParams</i>	Specify any construction parameters required for the class. Constructors are matched by construction parameter type and placement.

Returns

A Java object.

Example

The following example shows using dot notation and CreateJavaObject.

```
&CHARACTER.Value = CreateJavaObject(&java_path).GetField(&java_newchar).Value;

&NUMBER.Value = CreateJavaObject(&java_path).GetField(&java_newnum).Value;

&DATE.Value = CreateJavaObject(&java_path).GetField(&java_newdate).Value;
```

Related Topics

Internet Script Classes, CreateJavaArray, GetJavaClass

CreateMessage

Syntax

`CreateMessage (MESSAGE.messageName)`

Description

CreateMessage instantiates a message object that refers to a message definition created in Application Designer. The CreateMessage function sets the following properties for the resulting message object, based on the values set for the message definition:

- Name
- ChannelName
- Active

Other properties are set when the message is published or subscribed to (PubID, SubName, etc.), or are dynamically generated at other times (Size, EditError, etc.)

CreateMessage will also set the LANGUAGE_CD field in the level 0 PSCAMA record for a message based on the OPERID default language group. If the message is being published from a component, the language code is set to the OPERID language code for the component. If CreateMessage is called from an Application Engine program, the language code of the user who started the batch process will be used.

Parameters

MESSAGE.messageName	Specify the name of the message definition you want to create a message object for.
---------------------	---

Returns

Returns a reference to a message object.

Example

The following example creates a message &MSG based on the message definition PURCHASE_ORDER.

```
Local message &MSG;  
  
&MSG = CreateMessage (MESSAGE.PURCHASE_ORDER) ;
```

Related Topics

GetMessage, GetPubContractInstance, GetSubContractInstance, Internet Script Classes

CreateObject

Syntax

```
CreateObject (str_class_name)
```

Where *str_class_name* identifies a class of OLE Automation object, in the form:

```
app_name.object_name
```

Description

CreateObject returns an instance of an OLE Automation object as a variable of type Object.



The OLE functions are used to access and manipulate external OLE objects only. You can *not* use them to access or manipulate PeopleCode objects such as arrays, rowsets, and so on. However, the ObjectSetProperty function *can* be used with the Tree View and Chart ActiveX control in special cases. See ActiveX Controls in PeopleTools.

The *str_class_name* argument uses the syntax *app_name.object_type*, which consists of: *app_name* (the name of the application providing the object) and *object_type* (the class or type of the object to create), separated by a period (dot).

Any application that supports OLE Automation exposes at least one type of object. For example, a spreadsheet application may provide an application object, a worksheet object, and a toolbar object.

To create an OLE Automation object, you assign the object returned by **CreateObject** to a variable of type Object:

```
local object &WORKSHEET;

&WORKSHEET = CreateObject ("Excel.Sheet");
```

Once an object is created, you can reference it using the object variable. In the above example, you access properties and methods of the new object using the **ObjectGetProperty**, **ObjectSetProperty**, and **ObjectDoMethod** functions.



If an object has registered itself as a single-instance object, only one instance of the object can be created, even if **CreateObject** is executed more than once. Note also that an object assigned to a global variable is not valid across processes: that is, the scope and lifetime of the global is the same as the scope and lifetime of the instance of PeopleTools in which the object was created.

Restrictions on Use in PeopleCode Events

CreateObject is a "think time" function, and therefore cannot be used in any of the following PeopleCode events:

- SavePreChange
- Workflow
- SavePostChange
- RowSelect

Restrictions on Use in Three-Tier Mode and PeopleSoft Internet Architecture

Because it is specific to Windows, **CreateObject** is a client-only function, which limits its use in three-tier mode and in PeopleSoft Internet Architecture applications.

- In three-tier mode **CreateObject** can be used only in processing groups set to run on the client. If the **CreateObject** function is called in a processing group running on the application server, a runtime error will occur.
- In PeopleSoft Internet Architecture applications, where all PeopleCode runs on the application server, **CreateObject** cannot be used at all.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

Example

This simple example instantiates an Excel worksheet object, makes it visible, names it, saves it, and displays its name. Note the use of **ObjectGetProperty** in the example to return the Excel.Sheet.Application object. This technique is used instead of dot notation, which is not supported in PeopleTools 6, to reference the object:

```
&WORKAPP = CreateObject("Excel.Application");

&WORKBOOKS = ObjectGetProperty(&WORKAPP, "Workbooks");

ObjectDoMethod(&WORKBOOKS, "Add", "C:\TEMP\INVOICE.XLT"); /* This associates the
INVOICE template w/the workbook */

ObjectDoMethod(&WORKAPP, "Save", "C:\TEMP\TEST1.XLS");

ObjectSetProperty(&WORKAPP, "Visible", True);
```

Related Topics

ObjectDoMethod, ObjectGetProperty, ObjectSetProperty

CreateProcessRequest

Syntax

```
CreateProcessRequest()
```

Description

The **CreateProcessRequest** function creates a ProcessRequest object. After you've created this object, you can assign values to its properties then use the Schedule method to submit the process request for scheduling.

Restrictions on Use in Three-Tier Mode and PeopleSoft Internet Architecture

The RunLocation ProcessRequest class property specifies whether the scheduled process is to be run on the client or on the server.

If the ProcessRequest object is called in a program running on the application server, the process cannot run on the client; that is, you can't set the RunLocation property to "CLIENT". This will cause a runtime error and the transaction will be canceled.

If the PeopleCode program where the ProcessRequest is called runs on the application server, then COBOL and SQR processes must be set to run on the server. If the PeopleCode program runs on the client (which could happen in either 2-tier or 3-tier mode), then COBOL or SQR processes can run on either the client or the server.

Parameters

None.

Returns

A reference to a ProcessRequest object.

Example

```
Local ProcessRequest &MYRQST;  
  
&MYRQST = CreateProcessRequest();
```

Related Topics

Process Scheduler, Schedule ProcessRequest method, PortalRegistry Classes

CreateRecord

Syntax

```
CreateRecord(RECORD.recname)
```

Description

CreateRecord creates a **freestanding** record definition and its component set of field objects. The specified record must have been defined previously, that is, it must have a record definition. However, if you are calling this function from PeopleCode associated with a page, the record does not have to be included on the current page.

The record and field objects created by this function are accessible only within PeopleCode. They can be used with any of the record and field object methods and properties. The record and field objects will be automatically deleted when there are no remaining references to them stored in any variables.

The fields created by this function will be initialized to null values. Default processing will not be performed. No data associated with the record definition's SQL table will be brought in: only the record definition.

You can select into a record object created this way using the **SelectByKey** method. You can also select into it using **SQLExec**.

Returns

This function returns a record object that references a new record buffer and set of fields.

Parameters

RECORD.*recname* Specify a record definition that already exists.

Examples

```
Local Record &REC2;
```

```
&REC2 = CreateRecord(RECORD.OPC_METH);
```

In the following example, a free-standing record is created (&PSBATREPREQRES). Values are assigned to the fields associated with the record. Then a second record is created (&PUBHDR), and the values from the first record are used to populate the second record.

```
&PSBATREPREQRES = CreateRecord(RECORD.PSBATREPREQRES);
```

```
&PSBATREPREQRES.BATREPID.Value = &BATREPID;
```

```
&PSBATREPREQRES.PUBID.Value = &MSG.Pubid;
```

```
&PSBATREPREQRES.CHNLNAME.Value = &MSG.ChannelName;
```

```
&PSBATREPREQRES.PUBNODE.Value = &MSG.PubNodeName;
```

```
&PSBATREPREQRES.MSGNAME.Value = &MSG.Name;
```

```
&PUBHDR = CreateRecord(RECORD.PSAPMSGPUBHDR);
```

```
&PSBATREPREQRES.CopyFieldsTo(&PUBHDR);
```

If you want to create a PeopleCode record object for a record whose name is unknown when the PeopleCode is written, you can do the following.

Suppose a record name is in the PeopleCode variable &RECNAME. Use the @ operator to convert the string to a component name. The following code will create a corresponding record object:

```
&RECNAME = "RECORD." | Upper(&RECNAME);

&REC = CreateRecord(@ &RECNAME);
```

The following example uses SQLExec to select into a record object, based on the effective date.

```
Local Record &DST;

&DST = CreateRecord(RECORD.DST_CODE_TBL);

&DST.SETID.Value = GetSetId(FIELD.BUSINESS_UNIT, DRAFT_BU, RECORD.DST_CODE_TYPE,
    "");

&DST.DST_ID.Value = DST_ID_AR;

SQLExec("%SelectByKeyEffDt(:1,:2)", &DST, %Date, &DST);

/* do further processing using record methods and properties */
```

Related Topics

GetRecord, GetField, ProcessRequest Class, and Data Buffer Access

CreateRowset

Syntax

```
CreateRowset({RECORD.recname | &Rowset} [, {FIELD.fieldname, RECORD.recname |
    &Rowset}] . . .)
```

Description

The **CreateRowset** function creates an **unpopulated**, standalone rowset.

A standalone rowset is a rowset that has the specified structure, but is not tied to any data (that is, to the component buffer or to a message.) In addition, a standalone rowset isn't tied to the Component Processor. When you fill it with data, no PeopleCode will run (like RowInsert, FieldDefault, and so on.)



For more information see Using Standalone Rowsets.

The first parameter determines the structure of the rowset to be created.

If you specify a record as the first parameter, it's used as the primary level 0 record. If you don't specify any other parameters, you create a rowset containing one row, with one unpopulated record. To populate this type of rowset with data, you should only use the following.

- the Fill or FillAppend methods
- a record method (**SelectByKey**)
- SQLExec

If you specify a rowset object, you are creating a new rowset based on the structure of the specified rowset object, including any child rowsets. It will not contain any data. If you want to populate this type of rowset with data, use the **CopyTo** method or a SQL statement.



You should *not* use the rowset **Select** or **SelectNew** methods for populating rowsets created using **CreateRowset**. Use **Fill** or **FillAppend** instead.

Restrictions on Using CreateRowset

The following methods and properties don't work with a rowset created using CreateRowset:

- Select
- SelectNew
- Any GUI methods (like HideAllRows)
- Any effective date methods or properties (like EffDt, EffSeq, or GetCurrEffRow)

In addition, rowsets created using CreateRowset are *not* automatically tied to the database. This means if you insert or delete rows, the rows will *not* be inserted or deleted in the database when you save the page.

Parameters

RECORD.*recname* |
&Rowset Specify either a record name or an existing rowset object.

FIELD.*fieldname*,
RECORD.*recname* |
&Rowset Use FIELD.*fieldname*, RECORD.*recname* to specify a related display **record**. FIELD.*fieldname* refers to the controlling field, (not the related display field) while RECORD.*recname* refers to the related display record.

If you specify &rowset, you are adding a child rowset object to the newly created rowset. This must be an existing rowset object.

Returns

An unpopulated, standalone rowset object.

Example

The following creates a simple rowset of just a single record per row:

```
&RS = CreateRowset (RECORD.QA_MYRECORD) ;
```

The following creates a rowset with the same structure as the specified rowset:

```
&RS2 = CreateRowset (&RS) ;
```

The following code creates a rowset structure composed of four records in an hierarchical structure, that is,

QA_INVEST_HDR

 QA_INVEST_LN

 QA_INVEST_TRANS

 QA_INVEST_DTL

Note that you have to start at the *bottom* of the hierarchy, and add the upper levels, not the other way around.

```
Local Rowset &RS, &RS2, &RS_FINAL;
```

```
&RS2 = CreateRowset (RECORD.QA_INVEST_DTL) ;
```

```
&RS = CreateRowset (RECORD.QA_INVEST_TRANS, &RS2) ;
```

```
&RS2 = CreateRowset (RECORD.QA_INVEST_LN, &RS) ;
```

```
&RS_FINAL = CreateRowset (RECORD.QA_INVEST_HDR, &RS2) ;
```

The following example reads all of the QA_MYRECORD records into a rowset, and returns the number of rows read:

```
&RS = CreateRowset (RECORD.QA_MYRECORD) ;
```

```
&NUM_READ = &RS.Fill() ;
```

In order to make a clone of an existing rowset, that is, to make two distinct copies, you can do the following:

```
&RS2 = CreateRowset (&RS) ;
```

```
&RS.CopyTo (&RS2) ;
```

Related Topics

GetRowset, [GetLevel0](#), GetRecord, GetField

CreateSQL

Syntax

```
CreateSQL(sqlstring [, paramlist])
```

Where *paramlist* is an arbitrary-length list of values in the form:

```
inval1 [, inval2] ...
```

Description

The **CreateSQL** function instantiates a SQL object from the SQL class and opens it on the given *sqlstring* and input values. *sqlstring* is a PeopleCode string value giving the SQL statement.

Any errors in the SQL processing cause the PeopleCode program to be terminated with an error message.

Opening and Processing *sqlstring*

If *sqlstring* is a SELECT statement, it is immediately bound with the *inval* input values and executed. The SQL object should subsequently be the subject of a series of **Fetch** method calls to retrieve the selected rows. If you want to fetch only a single row, use the **SQLExec** function instead. If you want to fetch a single row into a PeopleCode record object, use the record **Select** method.

If *sqlstring* is not a SELECT statement, and *either*: there are some *inval* parameters, *or* there are no bind placeholders in the SQL statement, the statement is immediately bound and executed. This means that there is nothing further to be done with the SQL statement and the **IsOpen** property of the returned SQL object will be False. In this case, using the **SQLExec** function would generally be better. If you want to delete, insert or update a record object, use the record **Delete**, **Insert**, or **Update** methods.

If *sqlstring* is not a SELECT statement, there are no *inval* parameters, *and* there are bind placeholders in the SQL statement, the statement is neither bound nor executed. The resulting SQL object should subsequently be the subject of a series of **Execute** method calls to affect the desired rows.

Parameters

<i>sqlstring</i>	Specify a SQL string.
<i>paramlist</i>	Specify input values for the SQL string.

Returns

None.

Example

This SQL object should be used in a series of Fetch method calls:

```
Local SQL &SQL;
```



```
&SQL = CreateSQL("%Select(:1) where EMPLID = :2", RECORD.ABSENCE_HIST, &EMPLID);
```

This SQL object has been opened, bound and is already closed again:

```
&SQL = CreateSQL("Delete from %Table(:1) where EMPLID = :2",  
RECORD.ABSENCE_HIST, &EMPLID);
```

This SQL object should be used in a series of Execute method calls:

```
&SQL = CreateSQL("Delete from %Table(:1) where EMPLID = :2");
```

Related Topics

DeleteSQL, FetchSQL, GetSQL, SQLExec, StoreSQL and SQL Class, Open SQL method

CubicSpline

Syntax

```
CubicSpline(DataPoints, Control_Option, Left_Constraint, Right_Constraint)
```

Description

The **CubicSpline** function computes a cubic spline interpolation through a set of at least four datapoints.

Parameters

DataPoints

This parameter takes an array of array of number. The array's contents are an array of six numbers. The first two of these six numbers are the x and y points to be fit. The last four are the four coefficients to be returned from the function: **a**, **b**, **c** and **d**. **a** is the coefficient of the x^0 term, **b** is the coefficient of the x^1 term, **c** is the coefficient of the x^2 term, and **d** is the coefficient of the x^3 term.

Control_Option

Specifies the control option. This parameter takes either a number or constant value. The valid values are:

Value	Constant	Description
0	%SplineOpt_SlEstEst	Generate an internal estimate of the beginning and ending slope of the cubic piecewise equations
1	%SplineOpt_SlSetEst	Use the user-specified value

Value	Constant	Description
		for the slope of the leftmost point, and generate an estimate for the rightmost point
2	%SplineOpt_SlEstSet	Use the user-specified value for the slope of the rightmost point, and generate an estimate for the leftmost point
3	%SplineOpt_SlSetSet	Use the user-specified values for the slopes of the leftmost and rightmost points
4	%SplineOpt_CvEstEst	Generate an internal estimate of the beginning and ending curvature of the cubic piecewise equations
5	%SplineOpt_CvSetEst	Use the user-specified value for the curvature of the leftmost point, and generate an estimate for the rightmost point
6	%SplineOpt_CvEstSet	Use the user-specified value for the curvature of the rightmost point, and generate an estimate for the leftmost point
7	%SplineOpt_CvSetSet	Use the user-specified values for the curvatures of the leftmost and rightmost points
8	%SplineOpt_ClEstEst	Generate an internal estimate of the beginning and ending curl of the cubic piecewise equations
9	%SplineOpt_ClSetEst	Use the user-specified value for the curl of the leftmost point, and generate an estimate for the rightmost point
10	%SplineOpt_ClEstSet	Use the user-specified value for the curl of the rightmost point, and generate an

Value	Constant	Description
		estimate for the leftmost point
11	%SplineOpt_ClSetSet	Use the user-specified values for the curls of the leftmost and rightmost points
12	%SplineOpt_Natural	Generate a "natural" spline
13	%SplineOpt_ContCurl	Generate a spline wherein the equation for the first segment is the exact same equation for the second segment, and where the equation for the penultimate segment is the same as the equation for the last segment.

Left_Constraint

A single number for the constraint for the left point.
Specify a zero if this parameter isn't needed.

Right_Constraint

A single number for the constraint for the right point.
Specify a zero if this parameter isn't needed.

Returns

A modified array of array of numbers. The elements in the array correspond to the elements in the array used for *DataPoints*.

Related Topics

HermiteCubic, LinearInterp

CurrEffDt**Syntax**

```
CurrEffDt ( [level_num] )
```

Description

CurrEffDt returns the effective date of the specified scroll level as a Date value. If no level is specified, **CurrEffDt**, returns the effective date of the current scroll level.



This function remains for backward compatibility only. Use the EffDt rowset class property instead. See Data Buffer Access. Also see GetCurrEffRow Rowset method and EffSeq rowset property .

Returns

Returns a Date value equal to the current effective date of the specified scroll level.

Example

```

If INSTALLATION.POSITION_MGMT = "P" Then
  If All (POSITION_NBR) Then
    If (EFFDT = CurrEffDt (1) and
        EFFSEQ >= CurrEffSeq (1)) or
        (EFFDT > CurrEffDt (1) and
        EFFDT = %Date) Then
      Gray_employment ( );
    End-if;
  End-if;
End-if;

```

Related Topics

CurrEffRowNum, CurrEffSeq, CurrentLevelNumber

CurrEffRowNum

Syntax

```
CurrEffRowNum( [level_num] )
```

Description

CurrEffRowNum returns the effective row number of the selected scroll level. If no level is specified, it returns the effective row number of the current level.



This function remains for backward compatibility only. Use the RowNumber row class property, in combination with the GetCurrEffRow Rowset method instead. See also Data Buffer Access.

Example

```
&ROW = CurrEffRowNum (1) ;
```

Related Topics

CurrEffSeq , CurrentLevelNumber , CurrEffRowNum

CurrEffSeq

Syntax

```
CurrEffSeq([level_num])
```

Description

CurrEffSeq determines the effective sequence of a specific scroll area. If no level is specified, **CurrEffSeq** returns the effective sequence of the current scroll level.



This function remains for backward compatibility only. Use the EffSeq rowset class property instead. See Data Buffer Access. Also see GetCurrEffRow Rowset method and DeleteEnabled rowset property .

Returns

Returns a Number representing the effective sequence of the specified scroll level.

Example

```
If INSTALLATION.POSITION_MGMT = "P" Then
  If All(POSITION_NBR) Then
    If (EFFDT = CurrEffDt(1) and
        EFFSEQ >= CurrEffSeq(1)) or
        (EFFDT > CurrEffDt(1) and
        EFFDT = %Date) Then
      Gray_employment( );
    End-if;
  End-if;
End-if;
```

Related Topics

CurrEffDt, CurrentLevelNumber, CurrEffRowNum

CurrentLevelNumber

Syntax

```
CurrentLevelNumber( )
```

Description

The **CurrentLevelNumber** function returns the scroll level where the function call is located.

Returns

Returns a Number value equal to the scroll level where the function is being called. The function returns 0 if the field where the function is called is not in a scroll area.

Example

```
&LEVEL = CurrentLevelNumber();
```

Related Topics

CurrentRowNumber, FetchValue

CurrentRowNumber

Syntax

```
CurrentRowNumber([level])
```

Description

Use **CurrentRowNumber** to determine the row number of the row currently displayed in a specific scroll area.

This function can determine the current row number on the level where the function call resides, or on a higher scroll level. It won't work on a scroll level below the one where the PeopleCode program resides.



This function remains for backward compatibility only. Use the RowNumber Row property instead. See Data Buffer Access. Also see GetRow function.

Returns

Returns a Number value equal to the current row number on the specified scroll level. The current number is the row where the PeopleCode program is being processed, or, if level specifies a higher level scroll, **CurrentRowNumber** returns the row number of the parent or grandparent row.

Parameters

<i>level</i>	A Number specifying the scroll level from which the function returns the current row number. If the level parameter is omitted, it defaults to the scroll level where the function call resides.
--------------	--

Example

CurrentRowNumber is commonly used in component buffer functions to return the current row of the parent scroll of the target:

```
&VAL = FetchValue(RECORD.BUS_EXPENSE_PER, CurrentRowNumber(),  
BUS_EXPENSE_DTL.CHARGE_DT, &COUNT);
```

The following example checks if the current row number is equal to the active row count (that is, whether the active row is the last record on the scroll):

```
If CurrentRowNumber() = ActiveRowCount(EMPLID) Then  
    det_employment_dt();  
End-if;
```

Related Topics

ActiveRowCount, CurrentLevelNumber, FetchValue

Date

Syntax

Date(*date_num*)

Description

Date takes a number in the form YYYYMMDD and returns a corresponding Date value. If the date is invalid, **Date** displays an error message.



Make sure that you pass a four-digit year in the year parameter of this function. Two-digit values will be interpreted literally: 93, for example, represents the year 93 AD.

Returns

Returns a date equal to the date specified in *date_num*.

Example

Set the temporary variable &HIREDate to a date field containing the date July 1, 1997:

```
&HIREDate = Date(19970701);
```

Related Topics

Date3, DateValue, Day, Days360, Days365, Month, Weekday, Year

Date3

Syntax

Date3(*year, month, day*)

Description

The **Date3** function accepts a date expressed as three integers: *year*, *month*, and *day*. It returns a corresponding Date value. If the date is invalid, the **Date3** displays an error message.



Make sure that you pass a four-digit year in the year parameter of this function. Two-digit values will be interpreted literally: 93, for example, represents the year 93 AD.

Returns

Returns a Date value equal to the date specified in the function parameters.

Parameters

<i>year</i>	An integer for the year in the form YYYY.
<i>month</i>	An integer from 1 to 12 designating the month.
<i>day</i>	An integer from 1 to 31 designating the day of the month.

Example

The following PeopleCode **Date3** function returns the first day of the year in which the employee was hired:

```
Date3 (HIRE_YEAR, 1, 1);
```

Related Topics

Date, DateValue, Day, Days360, Days365

DatePart

Syntax

```
DatePart(datetime_value)
```

Description

Use **DatePart** to determine a date based on a provided DateTime value.

Returns

Returns a Date value equal to the date part of a specified DateTime value.

Example

The following statement would set &D2 to a Date value for 11/12/97:

```
&D1 = DateTimeValue("11/12/97 10:23:15 AM");  
&D2 = DatePart(&D1);
```

DateTime6

Syntax

```
DateTime6(year, month, day, hour, minute, second)
```


Description

DateTime6 returns a DateTime value based on integer values for the *year*, *month*, *day*, *hour*, *minute*, and *second*. If the result of this function is not an actual date, there will be a runtime error.



Make sure that you pass a four-digit year in the year parameter of this function. Two-digit values will be interpreted literally: 93, for example, represents the year 93 AD.

Returns

Returns a DateTime value based on the integers provided.

Parameters

<i>year</i>	A four-digit number representing the year.
<i>month</i>	A number between 1 and 12 representing the month.
<i>day</i>	A number representing the day of the month.
<i>hour</i>	A number from 0 to 23 representing the hour of the day.
<i>minute</i>	A number from 0 to 59 representing the minute of the hour.
<i>second</i>	A number from 0 to 59.999999 representing seconds.

Example

The following example sets &DT to a DateTime value equal to 10:09:20 on March 15, 1997:

```
&DT = DateTime6(1997, 3, 15, 10, 9, 20);
```

DateTimeToLocalizedString

Syntax

```
DateTimeToLocalizedString({datetime | date}, [Pattern])
```

Description

The **DateTimeToLocalizedString** function converts either *datetime* or *date* to a localized string. You can also specify a particular pattern to convert *datetime* or *date* to.

The *Pattern* is optional. Only specify *Pattern* if necessary.

If you need to change the pattern for each language, change the first message in Message Catalog set number 138. This is a format for each language.

Parameters

Datetime | *Date*

Specify either the datetime or date that you want to convert.

Pattern

Specify the pattern you want to the localized datetime or date to be converted to. For more information see Using the Pattern Parameter.

Using the Pattern Parameter

Pattern takes a string value, and indicates how you want the date or datetime converted.

The valid values for *Pattern* are as follows. They are case-sensitive:

Symbol	Definition	Type	Example
G	Era designator	Text	AD
y	Year	Number	1996
M	Month in year	Text&Number	July&07
d	Day in month	Number	10
h	Hour in am/pm	Number (1-12)	12
H	Hour in day	Number (0-23)	0
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
E	Day in week	Text	Tuesday
a	am/pm marker	Text	PM
k	Hour in day	Number (1-24)	24
K	Hour in am/pm	Number (0-11)	0
'	Escape for text	Delimiter	
"	Single quote	Literal	'

The number of pattern letters determine the format.

Type	Pattern Format
Text	If 4 or more pattern letters are used, the full form is used. If less than 4 pattern letters are used, the short or abbreviated form is used if one exists.
Number	Use the minimum number of digits. Shorter numbers are zero-padded to this amount. The year is handled specially; that is, if the

Type	Pattern Format
	count of 'y' is 2, the year is truncated to 2 digits.
Text&Number	If 3 or more pattern letters are used, text is used, otherwise, a number is used.

Any characters in *Pattern* are not in the ranges of ['a'..'z'] and ['A'..'Z'] are treated as quoted text. For instance, characters like ':', '.', ',', '#', and '@' appear in the resulting string even they're not within single quotes.

A pattern containing any invalid pattern letter results in a runtime error.

Examples using a United States locale:

Pattern	Result
"yyyy.MM.dd G 'at' hh:mm:ss z"	1996.07.10 AD at 15:08:56 PDT
"EEE, MMM d, 'yy"	Wed, July 10, '96
"h:mm a"	12:08 PM
"hh 'o'clock' a, zzzz"	12 o'clock PM, Pacific Daylight Time
"K:mm a, z"	0:00 PM, PST
"yyyyy.MMMMM.dd GGG hh:mm aaa"	1996.July.10 AD 12:08 PM

Returns

A string.

Related Topics

FormatDateTime

DateTimeToTimeZone

Syntax

```
DateTimeToTimeZone(OldDateTime, SourceTimeZone, DestinationTimeZone);
```

Description

DateTimeToTimeZone converts datetimes from the datetime specified by *SourceTimeZone* to the datetime specified by *DestinationTimeZone*.

Considerations using this Function

Typically, this function is used in PeopleCode, *not* for displaying time. If you take a datetime value, convert it from base time to client time, then try to display this time, depending on the user settings, when the time is displayed the system might try to do a *second* conversion on an already

converted datetime. This function could be used as follows: suppose a user wanted to check to make sure a time was in a range of times on a certain day, in a certain timezone. If the times were between 12 AM and 12PM in EST, these resolve to 9 PM and 9AM PST, respectively. The start value is *after* the end value, which makes it difficult to make a comparison. This function could be used to do the conversion for the comparison, in temporary fields, and not displayed at all.

Parameters

<i>OldDateTime</i>	Specify the datetime value to be converted.
<i>SourceTimeZone</i>	Specify the time zone that <i>OldDateTime</i> is in. Valid values are: <i>timezone</i> - a time zone abbreviation or a field reference to be used for converting <i>OldDateTime</i> . Local - use the local time zone for converting <i>OldDateTime</i> . Base - use the base time zone for converting <i>OldDateTime</i> .



For more information about setting the base time, see PeopleTools Utilities.

<i>SourceTimeZone</i>	Specify the time zone that you want to convert <i>OldDateTime</i> to. Valid values are: <i>timezone</i> - a time zone abbreviation or a field reference to be used for converting <i>OldDateTime</i> . Local - use the local time zone for converting <i>OldDateTime</i> . Base - use the base time zone for converting <i>OldDateTime</i> .
-----------------------	---

Returns

A converted datetime value.

Example

The following example. TESTDTTM, is a datetime field with a value 01/01/99 10:00:00. This example converts TESTDTTM from Pacific standard time (PST) to eastern standard time (EST).

```
&NEWDATETIME = DateTimeToTimeZone(TESTDTTM, "PST", "EST");
```

&NEWDATETIME will have the value 01/01/99 13:00:00 because EST is three hours ahead of PST on 01/01/99, so three hours are added to the datetime value.

Related Topics

ConvertDatetimeToBase, ConvertTimeToBase, FormatDateTime, IsDaylightSavings, TimeToTimeZone, TimeZoneOffset

DateTimeValue

Syntax

```
DateTimeValue(string)
```

Description

Use **DateTimeValue** to derive a DateTime value from a string representing a date and time. The string has the form:

```
MM/DD/YY[YY] hh:mm:ss.ssssss [{AM|PM}]
```

or

```
MM.DD.YY[YY] hh:mm:ss.ssssss [{AM|PM}]
```

If the AM|PM part is omitted, the value is assumed to be based on a 24-hour clock. This function is language-sensitive. However, be aware that it returns date values based on Windows settings, not based on PeopleSoft language preference settings. It is therefore unaffected by calls to **SetLanguage**.

If the Windows Regional Settings Date properties are set to DD/MM/YY, then

```
&DTM = DateTimeValue("10/09/97 10:34:36");
```

will return a Datetime value equal to September 10, 1997, 10:34:36.

If the Windows Regional Settings Date properties are set to MM/DD/YY, then the same function call will return a value equal to October 9, 1997, 10:34:36.

Using this Function in Fields without a Default Century Setting

This function may derive the wrong century setting if passed a 2-character year **and** DateTimeValue is executing in a PeopleCode event not associated with a field that has a default century setting.

For example, assume that TEST_DATE is a date field with a default century setting of 10. TEST_FIELD is a field with *no* default century setting. If the following PeopleCode program is executing in TEST_FIELD, the date will be calculated incorrectly:

```
TEST_DATE = DateTimeValue("10/13/11 15:34:25");
```

Though TEST_DATE has a century setting, it isn't used because the PeopleCode fired in TEST_FIELD. Instead, the code uses the 50/50 rule and calculates the year to be 2011 (instead of 1911).

Returns

Returns a DateTime value equal to the datetime specified in the input string.

Example

Both of the following example set `&Date_TIME` to a Datetime value equal to 10/13/97 10:34:25 PM. These examples assume that the Windows Regional Date properties are set to the US English defaults:

```
&Date_TIME = DateTimeValue("10/13/97 10:34:25 PM");
&Date_TIME = DateTimeValue("10/13/97 22:34:25");
```

Related Topics

Date, Date3, DateValue, Day, Days360, Days365, Month, Weekday, Year

DateValue

Syntax

```
DateValue(date_str)
```

Description

DateValue converts a date string and returns the result as a Date type. *date_str* must be a string in the active date format, based on the Windows Regional Date property settings.

Be aware that it returns date values based on Windows settings, not based on PeopleSoft language preference settings. It is therefore unaffected by calls to **SetLanguage**.

If the Windows Regional Settings Date properties are set to DD.MM.YY, then

```
&DTM = DateValue("10/09/97");
```

will return a Datetime value equal to September 10, 1997.

If the Windows Regional Settings Date properties are set to MM/DD/YY, then the same function call will return a value equal to October 9, 1997.

Using this Function in Fields without a Default Century Setting

This function may derive the wrong century setting if passed a 2-character year *and* DateValue is executing in a PeopleCode event not associated with a field that has a default century setting.

For example, assume that TEST_DATE is a date field with a default century setting of 10. TEST_FIELD is a field with *no* default century setting. If the following PeopleCode program is executing in TEST_FIELD, the date will be calculated incorrectly:

```
TEST_DATE = DateValue("10/13/11");
```

Though TEST_DATE has a century setting, it isn't used because the PeopleCode fired in TEST_FIELD. Instead, the code uses the 50/50 rule and calculates the year to be 2011 (instead of 1911).

Returns

Returns a Date value equal to the date specified in *date_str*.

Example

If the Windows client is set up with standard US date settings, the following **DateValue** function call converts the date string to a Date value equal to October 24, 1997:

```
&DT = DateValue("10/24/97");
```

If a workstation is set up with the typical European date settings, the following **DateValue** function converts the date string to the equivalent Date value:

```
&DT = DateValue("24.10.97");
```

Related Topics

Date, Date3, DateTimeValue, Day, Days360, Days365, Month, Weekday, Year

Day

Syntax

```
Day(dt_val)
```

Description

The **Day** function determines an integer representing the day of the month based on a Date or DateTime value.

Returns

Returns a Number value equal to the day of the month for *dt_val*. The return value is an integer from 1 to 31.

Example

If HIRE_DATE is November, 1, 1997, the following Day function returns the integer 1:

```
&FIRST_DAY = Day(HIRE_DATE);
```

Related Topics

Date, Date3, DateValue, Days, Days360, Days365, Month, Weekday, Year

Days

Syntax

```
Days(dt_val)
```

Description

The **Days** function returns the Julian date format for the *dt_val* specified. This function accepts a Date, DateTime, or Time value parameter.

Returns

Returns a Number value equal to the Julian date format for *dt_val*.

Example

To find the number of days between two dates, use the **Days** function on both dates, and subtract one from the other:

```
&NUM_DAYS = Abs(Days(HIRE_Date) - Days(RELEASE_Date));
```

Related Topics

DateValue, Days360, Days365, Month, Weekday, Year

Days360

Syntax

```
Days360(date_val1, date_val2)
```

Description

The **Days360** function returns the number of days between the Date values *date_val1* and *date_val2* using a 360-day year (twelve 30-day months). You can use this function to help compute payments if your accounting system is based on twelve 30-day months.

If *date_val2* occurs before *date_val1*, **Days360** returns a negative number.

Example

The following example sets &NUMDAYS to the number of days between TERM_START_DT and PMT_DT based on a 360-day calendar:

```
&NUMDAYS = Days360(TERM_START_DT, PMT_DT);
```

Related Topics

Date, Date3, DateValue, Day, Days, Days365, Month, Weekday, Year

Days365

Syntax

```
Days365(date_val1, date_val2)
```


Description

Returns the number of days between the Date values *date_val1* and *date_val2* using a 365-day year. You could use this function to help compute payments if your accounting system is based on a 365-day year.

If *date_val2* occurs before *date_val1*, **Days365** returns a negative number.

Returns

Returns a Number value equal to the number of days between the two dates in a 365-day calendar.

Example

The following example sets &NUMDAYS to the number of days between and TERM_START_DT and PMT_DT, based on a 365-day calendar:

```
&NUMDAYS = Days360 (TERM_START_DT, PMT_DT) ;
```

Related Topics

Date, Date3, DateValue, Day, Days360, Month, Weekday, Year

DBCSTrim

Syntax

```
DBCSTrim(source_str)
```

Description

DBCSTrim removes a trailing DBCS lead byte at the end of the string. Use this function to clean up a DBCS string after a byte-oriented operation (such as **Substringb**) that may have split a DBCS character at the end of the string.

Returns

Returns a String formed by trimming the trailing DBCS lead byte. If there is no trailing DBCS lead byte, the string returned is identical to the source string.

Related Topics

Substringb

Declare Function

PeopleCode Function Syntax

```
Declare Function function_name PeopleCode record_name.field_name event_type
```

External Function Syntax

```

Declare Function function_name Library lib_name
    [ALIAS module_name ]
    [paramlist]
    [RETURNS ext_return_type [As pc_type]]

```

Where

```

paramlist ([ext_param1 [, ext_param2] ...])

ext_paramn ext_datatype [{REF|VALUE}] [As pc_return_type]

```

Description

PeopleCode can call PeopleCode functions defined in any field on any record definition. You can create special record definitions whose sole purpose is to serve as function libraries. By convention, PeopleCode functions are stored in FieldFormula PeopleCode, in record definitions with names beginning in FUNCLIB_.

PeopleCode can also call external programs that reside in dynamic link libraries. You must declare either of these types of functions at the top of the calling program using the **Declare Function** statement.

To support processes running on an application server, it is possible to declare and call functions compiled in dynamic link libraries on windows (.DLL files) and shared libraries on UNIX (.so files.) The PeopleCode declaration and function call syntax is the same regardless of platform, but UNIX libraries must be compiled with an interface function.



For more information, see Calling Dynamic Link Library Functions on the Application Server.

PeopleCode Functions

You can call a PeopleCode function defined on any record definition, provided you declare it at the top of the calling program. The declaration identifies the function name, as well as the record, field, and event type where the function definition resides. The function parameters and return type are not declared; they are determined from the function definition.



You can only define functions in Record Field PeopleCode. You can't define functions in Component PeopleCode, Component Record Field PeopleCode, and so on.

External Library Functions

Function declarations define routines in an external (C-callable) library. The function declaration provides the name of the library, an optional alias *module_name*, a list of parameters to pass to the function, an optional **Returns** clause specifying the type of any value returned by the external function, and the PeopleCode data type into which to convert the returned value. The library must be a DLL accessible by Windows or a shared library accessible by UNIX.

Once you have declared an external DLL function, you can call it the same way as an external PeopleCode function. Like PeopleCode functions, you must pass the number of parameters the library function expects. Calls to external functions suspend processing; this means that you should exercise caution to avoid "think-time" errors when calling the function in the following PeopleCode events:

- SavePreChange
- SavePostChange
- Workflow
- RowSelect
- Any PeopleCode event that fires as a result of a ScrollSelect, ScrollSelectNew, RowScrollSelect or RowScrollSelectNew function call.



For more information, see Think-Time Functions.

PeopleCode Function Parameters

<i>function_name</i>	Name of the function.
PeopleCode	Reserved word that identifies the function as a PeopleCode function.
<i>recordname.fieldname</i>	Specifies the record and field where the function is located.
<i>event_type</i>	Component Processor event with which the function is associated.



event_type can only be used to specify **record field** events. You can't specify a component record field event, a component record event, and so on.

External Function Parameters

<i>function_name</i>	Name of the function.
Library	Reserved word that identifies the function as an external library function.
<i>lib_name</i>	A string representing the name of the external library. The external routine must be located in a DLL named <i>lib_name</i> accessible by Windows, or an equivalent shared library in a UNIX system.

Alias *module_name* Optionally specifies alias of module. The external module is invoked using the stdcall calling convention on Windows.

paramlist List of parameters expected by the function, each in the form:

ext_datatype [{**Ref**|**Value**}] [**As** *pc_type*]

ext_type The datatype of the parameter expected by the function. To specify the type you can use any of the following:

BOOLEAN
INTEGER
LONG
UINTeger
ULONG
STRING
LSTRING
FLOAT
DOUBLE

{**Ref**|**Value**} Optionally use one of these two reserved words to specify whether the parameter is passed by reference or by value. If **Ref** is specified, it is passed by pushing a reference (pointer) on the stack. If **Value** is specified the value is pushed on the stack (for integers, and so on.). If neither is specified, **Ref** is assumed.

AS *pc_type* Specifies PeopleCode data type of the value passed to the function. You can choose between PeopleCode data types String, Number, Date, Boolean, and Any.

Returns *ext_return_type* Specifies the data type of any value returned by the function. The **Returns** clause is omitted if the function is void (returns no value). To specify the return type you can use any of the following:

BOOLEAN
INTEGER
LONG
UINTeger
ULONG
FLOAT
DOUBLE

The types String and LString are not allowed for the result type of a function.

As *pc_return_type*

Specifies the PeopleCode data type of the variable or field into which to read the returned value. You can choose between PeopleCode data types String, Number, Date, Boolean, and Any. If the **As** clause is omitted, PeopleTools selects an appropriate type based on the type of value returned by the external function (for example, all integer and floating point types are converted to Number).

Example

Assume you have defined a PeopleCode function called **VerifyZip**. The function definition is located in the record definition FUNCLIB_MYUTILS, in the record field ZIP_EDITS, attached to the FieldFormula event. You would declare the function using the following statement:

```
declare function verifyzip PeopleCode FUNCLIB_MYUTILS.ZIP_EDITS FieldFormula;
```

Now assume you want to declare a function called **PCTest**, in PSUSER.DLL. It takes an integer and returns an integer. You would write this declare statement:

```
declare function pctest library "psuser.dll" (integer value as number) returns
integer as number;
```

Related Topics

Function

Decrypt**Syntax**

```
Decrypt(KeyString, EncryptedString)
```

Description

The **Decrypt** function decrypts a string previously encrypted with the **Encrypt** function. This function is generally used with merchant passwords. In order for this function to decrypt a string successfully, you must use the same *KeyString* value used to encrypt the string.

Parameters*KeyString*

Specify the key used for encrypting the string. You can specify a NULL value for this parameter, that is, two quotation marks with no blank space between them ("").

EncryptedString

Specify the string you want decrypted.

Returns

A clear text string.

Example

Encrypt and Decrypt only support strings.

```
&AUTHPARMS.WRKTOKEN.Value = Decrypt("",  
RTrim(LTrim(&MERCHANTID_REC.CMPAUTHNCTOKEN.Value)));
```

Related Topics

Encrypt, Hash, Security

Degrees

Syntax

Degrees (*angle*)

Description

Degrees converts the given angle from radian measurement to degree measurement.

Parameters

<i>angle</i>	The size of an angle in radians.
--------------	----------------------------------

Returns

The size of the given angle in degrees.

Example

The following example returns the equivalent size in degrees of an angle measuring **1.2** radians:

```
&DEGREE SIZE = Degrees(1.2);
```

Related Topics

Acos, Asin, Atan, Cos, Cot, Radians, Sin, Tan

DeleteAttachment

Syntax

DeleteAttachment(URLSource, DirAndFileName)

where *URLSource* can have **one** of the following forms:

URL.*URLname*

OR a string URL, such as

```
ftp://user:password@ftp.ps.com/
```

Description

The **DeleteAttachment** function enables you to delete a file on a specified system.

DeleteAttachment does *not* generate any type of "Are you sure" message. If you want the end-user to verify the deletion before it's performed, you must write your own checking code in your application.

File Name Considerations

When the file is transferred using AddAttachment or PutAttachment, the following characters are replaced with an underscore:

- space
- semi-colon
- plus sign
- percent sign
- ampersand
- apostrophe
- exclamation point
- @ sign
- pound sign
- dollar sign

Parameters

<i>URLSource</i>	A reference to a URL. This can be either a URL name, in the form URL. <i>URLName</i> , or a string. This will be where the file will end up.
<i>DirAndFileName</i>	A directory and filename. This is appended to the <i>URLSource</i> to make up the full URL of the file.



The *URLSource* requires "/" slashes. Because the *DirAndFileName* parameter is appended to the URL, it also requires only "/" slashes. "\" are *not* supported in anyway for either the *URLSource* the *DirAndFileName* parameter.

Returns

An integer value. You can check for either the integer or the constant:

Number	Constant	Description
0	%Attachment_Success	File was attached successfully.
1	%Attachment_Failed	File was not successfully attached.
2	%Attachment_Cancelled	File attachment didn't complete because the operation was canceled by the user.
7	%Attachment_DestSystNotFound	Cannot locate destination system for ftp.
8	%Attachment_DestSystFailedLogin	Unable to login to destination system for ftp.

Example

```
&retcode = DeleteAttachment(URL.BKFTP, ATTACHSYSFILENAME);
```

```
If (&retcode = %Attachment_Success) Then
```

```
    MessageBox(0, "File Attachment Status", 0, 0, "DeleteAttachment succeeded");
```

```
    UnHide(ATTACHADD);
```

```
    Hide(ATTACHVIEW);
```

```
    Hide(ATTACHDELETE);
```

```
    ATTACHUSERFILE = "";
```

```
    ATTACHSYSFILENAME = "";
```

```
End-If;
```

```
If (&retcode = %Attachment_Failed) Then
```

```
    MessageBox(0, "File Attachment Status", 0, 0, "DeleteAttachment failed");
```

```
End-If;
```

```
If (&retcode = %Attachment_Cancelled) Then
```

```
    MessageBox(0, "File Attachment Status", 0, 0, "DeleteAttachment cancelled");
```

```
End-If;
```



```
If (&retcode = %Attachment_FileTransferFailed) Then

    MessageBox(0, "File Attachment Status", 0, 0, "DeleteAttachment failed: File
Transfer did not succeed");

End-If;


/* following error message only in PeopleSoft Internet Architecture */


If (&retcode = %Attachment_NoDiskSpaceAppServ) Then

    MessageBox(0, "File Attachment Status", 0, 0, "DeleteAttachment failed: No
disk space on the app server");

End-If;


/* following error message only in PeopleSoft Internet Architecture */


If (&retcode = %Attachment_NoDiskSpaceWebServ) Then

    MessageBox(0, "File Attachment Status", 0, 0, "DeleteAttachment failed: No
disk space on the web server");

End-If;


If (&retcode = %Attachment_FileExceedsMaxSize) Then

    MessageBox(0, "File Attachment Status", 0, 0, "DeleteAttachment failed: File
exceeds the max size");

End-If;


If (&retcode = %Attachment_DestSystNotFound) Then

    MessageBox(0, "File Attachment Status", 0, 0, "DeleteAttachment failed:
Cannot locate destination system for ftp");

End-If;


If (&retcode = %Attachment_DestSysFailedLogin) Then

    MessageBox(0, "File Attachment Status", 0, 0, "DeleteAttachment failed:
Unable to login into destination system for ftp");

End-If;
```

Related Topics

AddAttachment, GetAttachment, PutAttachment, ViewAttachment, Using the Attachment Functions

DeleteImage

Syntax

```
DeleteImage(scrollpath, target_row, [recordname.]fieldname)
```

where *scrollpath* is:

```
[SCROLL.level1_recname, level1_row, [SCROLL.level2_recname, level2_row,]]
SCROLL.target_recname
```

Description

The **DeleteImage** function enables a user to remove an image associated with a record field.

Restrictions for Use with Windows Client

This function is only available in the PeopleSoft Internet Architecture, and not with windows client.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll area in the component buffer.
<i>target_row</i>	The row number of the target row.
<i>[recordname.]fieldname</i>	The name of the field associated with the image file. The field can be on scroll level one, two, or three of the active page. The <i>recordname</i> prefix is required if the function call is in a record definition other than <i>recordname</i> .

Returns

Returns a Boolean value: True if image was successfully deleted, False otherwise.

Example

```
&Rslt = DeleteImage(EMPL_PHOTO.EMPLOYEE_PHOTO);
```

Related Topics

InsertImage

DeleteRecord

Syntax

```
DeleteRecord(level_zero_recfield)
```

Description

Use **DeleteRecord** to remove a high-level row of data and all dependent rows in other tables from the database. **DeleteRecord** deletes the component's level-zero row from the database, deletes any dependent rows in other tables from the database, and exits the component.



This function remains for backward compatibility only. Use the Delete Record class method instead. See Data Buffer Access.

This function, like **DeleteRow**, initially marks the record or row as needing to be deleted. At save time the row is actually deleted from the database and cleared from the buffer.

This function will only work if the PeopleCode is on a level-zero field. It cannot be used from SavePostChange or WorkFlow PeopleCode.

Returns

Optionally returns a Boolean value indicating whether the deletion was completed successfully.

Parameters

<i>level_zero_recfield</i>	A <i>recordname.fieldname</i> reference identifying any field on the level-zero area of the page.
----------------------------	---

Example

The following example, which is in SavePreChange PeopleCode on a level-zero field, deletes the high-level row and all dependent rows in other tables if the current page is EMPLOYEE_ID_DELETE.

```
if %Page = PAGE.EMPLOYEE_ID_DELETE then
    &success = DeleteRecord(EMPLID);
end-if;
```

Related Topics

DeleteRow

DeleteRow

Syntax

```
DeleteRow(scrollpath, target_row)
```

Where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]]
RECORD.target_recname
```

To prevent ambiguous references, you can also use **SCROLL**.*scrollname*, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Description

Use **DeleteRow** to delete rows programmatically. A call to this function causes the RowDelete event sequence to fire, as if an user had manually deleted a row.



This function remains for backward compatibility only. Use the DeleteRow Rowset class method instead. See Data Buffer Access.

DeleteRow cannot be executed from the same scroll level where the deletion will take place, or from a lower scroll level. Place the PeopleCode in a higher scroll level record.

When **DeleteRow** is used in a loop, you have to process rows from high to low to achieve the correct results, that is, you must delete from the bottom up rather than from the top down. This is necessary because the rows are renumbered after they are deleted (if you delete row one, row two becomes row one).

Returns

Boolean (optional). **DeleteRow** returns a Boolean value indicating whether the deletion was completed successfully.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
<i>target_row</i>	The row number of the row to delete.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Example

In the following example **DeleteRow** is used in a **For** loop. The example checks values in each row, then conditionally deletes the row. Note the syntax of the **For** loop, including the use of the -1 in the **Step** clause to loop from the highest to lowest values. This ensures that the renumbering of the rows will not affect the loop.

```
For &L1 = &X1 To 1 Step - 1
    &SECTION = FetchValue(AE_STMT_TBL.AE_SECTION, &L1);
    &STEP = FetchValue(AE_STMT_TBL.AE_STEP, &L1);
    If None(&SECTION, &STEP) Then
        DeleteRow(RECORD.AE_STMT_TBL, &L1);
    End-If;
End-For;
```

Related Topics

InsertRow

DeleteSQL

Syntax

```
DeleteSQL([SQL.]sqlname[, dbtype[, effdt]])
```

Description

The **DeleteSQL** function enables you to programmatically delete a SQL definition. The SQL definition must have already been created and saved, either using the CreateSQL and StoreSQL functions, or by using Application Designer.

When you create a SQL definition, you must create a base statement before you can create other types of statements, that is, one that has a *dbtype* as **GENERIC** and *effdt* as the null date (or Date(19000101)). If you specify a base (generic) statement to be deleted, *all* statements as well as the generic statement will be deleted.

If you specify a non-generic statement that ends up matching the generic statement, DeleteSQL will *not* delete anything, and will return False.

Parameters

<i>sqlname</i>	Specify the name of a SQL definition. This is either in the form SQL. <i>sqlname</i> or a string value giving the <i>sqlname</i> .
<i>dbtype</i>	Specify the database type associated with the SQL definition. This parameter takes a string value. If <i>dbtype</i> isn't specified or is null(""), it defaults to the current database type (the value returned from the %DbName system variable.)

Valid values for *dbtype* are as follows. These values are not case sensitive:

- DEFAULT
- DB2ODBC
- DB2UNIX
- INFORMIX
- MICROSOFT
- ORACLE
- SYBASE



Database platforms are subject to change.

effdt

Specify the effective date associated with the SQL definition. If *effdt* isn't specified, it defaults to the current as of date, that is, the value returned from the %AsOfDate system variable.

Returns

A boolean value: True if the delete was successful, False if the specified SQL statement wasn't found, and terminates with an error message if there was another problem (that is, date in incorrect format, and so on.)

Example

The following code deletes the ABCD_XY SQL definition for the current DBType and as of date:

```
&RSLT = DeleteSQL(SQL.ABC_XY);

If NOT(&RSLT) Then

    /* SQL not found - do error processing */

End-if;
```

The following code deletes the ABCD_XY SQL Definition for the current DBType and the third of November, 1998:

```
&RSLT = DeleteSQL(SQL.ABCD_XY, "", Date(19981103));
```

Related Topics

CreateSQL, FetchSQL, GetSQL, StoreSQL and SQL Class

DeleteSystemPauseTimes

Syntax

```
DeleteSystemPauseTimes(StartDay, StartTime, EndDay, EndTime)
```

Description

The **DeleteSystemPauseTimes** function enables you to delete pause times occur on your system by adding a row to the system pause times table.

This function is used in the PeopleCode for the Application Message Monitor. Pause times are set up in the Application Message Monitor.

[Overview](#)
[Message Instances](#)
[Pub Contracts](#)
[Sub Contracts](#)
[Channel Status](#)
[Node Status](#)
[Queries](#)

Scheduled System Pause Times For Local Node: QE_LOCAL

Start Day	Start Time	End Day	End Time
View All First ◀ 1 of 1 ▶ Last			

Add Pause

Test Node

Ping a Node to Determine It's Availability

Message Node Name:

Location	Result
View All First ◀ 1 of 1 ▶ Last	

Setting System Pause Times in the Application Message Monitor



For more information, see Application Message Monitor.

Parameters

StartDay Specify a number from 0-6. The valid values are:

Value	Description
0	Sunday

Value	Description
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday

StartTime Specify a time, in seconds, since midnight.

EndDay Specify a number from 0-6. The valid values are:

Value	Description
0	Sunday
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday

EndTime Specify a time, in seconds, since midnight.

Returns

A boolean value: True if the system pause time specified was deleted, False otherwise.

Example

```
Component boolean &spt_changed;  
  
/* deleting a system pause time interval; */
```



```

If Not DeleteSystemPauseTimes(PSSPTIMES.STARTINGDAY, PSSPTIMES.STARTINGSECOND,
PSSPTIMES.ENDINGDAY, PSSPTIMES.ENDINGSECOND) Then

    Error MsgGetText(117, 15, "");

Else

    &spt_changed = True;

/* to force a save; */

PSSPTIMES.MSGSPTNAME = " ";

DoSave();

End-If;

```

Related Topics

[Application Message Monitor](#), [AddSystemPauseTimes](#)

DisableMenuItem

Syntax

```
DisableMenuItem(BARNAME.menubar_name, ITEMNAME.menuitem_name)
```

Description

DisableMenuItem disables (makes unavailable) the specified menu item. To apply this function to a pop-up menu, use the PrePopup Event of the field with which the pop-up menu is associated.

If you're using this function with a pop-up menu associated with a page (not a field), the earliest event you can use is the **PrePopup** event for the first "real" field on the page (that is, the first field listed in the **Order** view of the page in Application Designer.)

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```

If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

```

```

Else
    /* do regular processing */
    . . .
End-if;

```

Returns

None.

Parameters

<i>menubar_name</i>	Name of the menu bar that owns the menu item, or, in the case of pop-up menus, the name of the pop-up menu that owns the menu item.
<i>menuitem_name</i>	Name of the menu item.

Example

```
DisableMenuItem(BARNAME.MYPOPUP1, ITEMNAME.DO_JOB_TRANSFER);
```

Related Topics

CheckMenuItem, UnCheckMenuItem, EnableMenuItem, HideMenuItem

DiscardRow

Syntax

```
DiscardRow()
```

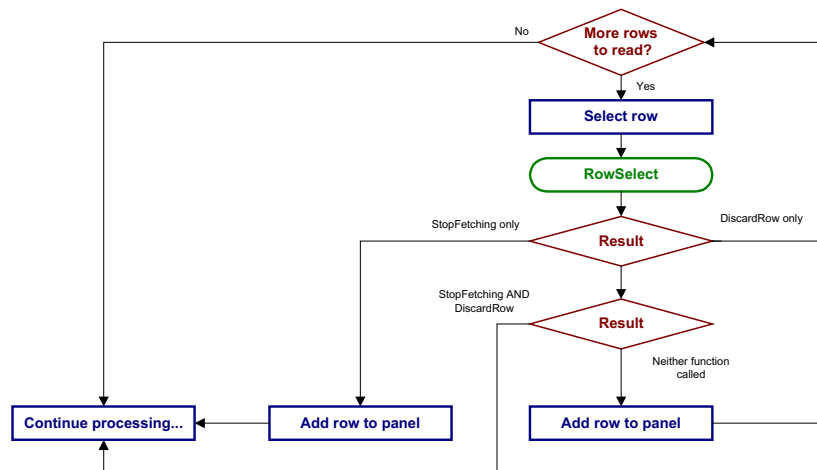
Description

DiscardRow prevents a row from being added to a page scroll during Row Select processing. This function is valid only in RowSelect PeopleCode. When **DiscardRow** is called during RowSelect processing, the current row is skipped (not added to the scroll). Processing then continues on the next row, unless **StopFetching** has also been called, in which case no more rows are added to the page.

DiscardRow has the same functionality as Warning in the RowSelect event. The anomalous behavior of Warning is supported for compatibility with previous releases of PeopleTools.



Row Select processing is used infrequently, because it is more efficient to filter out rows of data using a search view or an effective-dated record before the rows are selected down to the client from the database server.



Flow of Row Select Processing

Returns

None.

Related Topics

StopFetching and Row Select Processing

DoCancel

Syntax

```
DoCancel ( )
```

Description

DoCancel cancels the current page as though the user had pressed ESC. On the Windows client, it terminates the current component and returns the user to the current menu with no component active (that is, to reset state). In Web client applications, **DoCancel** does one of the following:

- In the Page Applet **DoCancel** terminates the current component and returns the user to the search dialog box.
- In the Menu Applet **DoCancel** terminates the current component and returns the user to the current menu with no component active. This is the same behavior as on the Windows client.

DoCancel terminates any PeopleCode programs executing prior to a save action. It will not stop processing of PeopleCode in SaveEdit, SavePreChange, and SavePostChange events.

Returns

None.

Related Topics

DoSave, DoSaveNow, DoModal, EndModal, WinEscape

DoModal

Syntax

```
DoModal (PAGE.pagename, title, xpos, ypos,
        [level, scrollpath, target_row])
```

where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]]
RECORD.target_recname
```

To prevent ambiguous references, you can also use **SCROLL**.*scrollname*, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Description

The **DoModal** function displays a secondary page. Secondary pages are modal, meaning that the user must dismiss the secondary page before continuing work in the page from which the secondary page was called.

If you call **DoModal** without specifying a level number or any record parameters, the function uses the current context as the parent.

You can alternately specify a secondary page in a command push button definition without using PeopleCode. This may be preferable for performance reasons, especially with Web client code.



For more information, see Scroll Area or Scroll Bar.

DoModal can display on a single page. If you want to display an entire component modally, use DoModalComponent.

Any variable declared as a Component variable will still be defined after using a **DoModal** function.

Returns

Returns a number that indicates how the secondary page was terminated. A secondary page can be terminated by the user clicking a built-in **OK** or **Cancel** button, or by a call to the **EndModal** function in a PeopleCode program. In either case, the return value of **DoModal** is one of the following:

- 1 if the user clicked **OK** in the secondary page, or if 1 was passed in the **EndModal** function call that terminated the secondary page.
- 0 if the user clicked **Cancel** in the secondary page, or if 0 was passed in the **EndModal** function call that terminated the secondary page.

Parameters

<i>pagename</i>	The name of the secondary page.
<i>title</i>	The text that will be displayed in the caption of the secondary page.
<i>xpos, ypos</i>	The pixel coordinates of the top left corner of the secondary page, offset from the top left corner of the parent page (the default of -1, -1 means centered).
<i>level</i>	Specifies the level of the scroll level on the parent page that contains the row corresponding to level 0 on the secondary page.
<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
<i>target_row</i>	The row number of the row in the parent page corresponding to the level 0 row in the secondary page.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Restrictions on Use in PeopleCode Events

Control does not return to the line after **DoModal** until after the user has dismissed the secondary page. This interruption of processing makes **DoModal** a "think-time" function, which means that it shouldn't be used in any of the following PeopleCode events:

- SavePreChange
- SavePostChange
- Workflow
- RowSelect
- Any PeopleCode event that fires as a result of a ScrollSelect, ScrollSelectNew, RowScrollSelect or RowScrollSelectNew function call.



For more information, see Think-Time Functions.

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```
If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;
```

Example

```
DoModal(PAGE.EDUCATION_DTL, MsgGetText(1000, 167, "Education Details - %1",
EDUCATN.DEGREE), - 1, - 1, 1, RECORD.EDUCATN, CurrentRowNumber());
```

Related Topics

EndModal, IsModal, DoModalComponent

DoModalComponent

Syntax

```
DoModalComponent(MENUNAME.menuname, BARNAME.barname, ITEMNAME.menuitem_name,
PAGE.page_group_item_name, action, RECORD.shared_record_name [, keylist])
```

where *keylist* is a list of field references in the form:

```
[recordname.]field1 [, [recordname.]field2]...
```

Or

```
&RecordObject1 [, &RecordObject2]. . .
```

Description

DoModalComponent launches a modal component. The modal component launches from within an originating component. Once the modal component displays, the user can't proceed with changes to the originating component until either accepting or canceling the modal component.

Modal components can be displayed in any of the following action modes: Add, Update/Display, Update/Display All, Correction. A modal component can be launched from any component, including another modal component. You can use **DoModalComponent** from a secondary page.

The originating component and the modal component share data, including search keys, via a Shared Work Records. If valid search keys are provided in the shared work record and populated with valid values before launching the modal component, the search is conducted using the provided search key values. If no search keys are provided, or if search key fields contain invalid values, then the user accesses the modal component via a search dialog.

In the *page_group_item_name* parameter, make sure to pass the component item name for the page, not the page name. The component item name is specified in the Component Definition, in the Item Name column on the row corresponding to the specific page, as shown here:

	Panel Name	Item Name	Hidden	Item Label	Folder Tab Label
1	PERSONAL_DATA1	PERSONAL_DATA_1	<input type="checkbox"/>	&Personal Data 1	
2	PERSONAL_DATA2	PERSONAL_DATA_2	<input type="checkbox"/>	&Personal Data 2	
3	SCRTY_TBL_GBL_WRK	SCRTY_TBL_GBL_WRK	<input checked="" type="checkbox"/>	ScrtY Tbl Gbl Wrk	

Component Item Name Parameter

Shared Work Records

The originating component and the modal component share fields in a Derived/Work record called a *shared work record*. Shared fields from this record must be placed at level zero of both the originating component and the modal component.

You can use the shared fields to:

- Pass values that are assigned to the search keys in the modal component search record. If these fields are missing or not valid, the search dialog box appears, allowing the user to enter search keys.
- Optionally pass other values from the originating component to the modal component.
- Pass values back from the modal component to the originating component for processing.

To make this happen, you have to write PeopleCode that:

- Assigns values to fields in the shared work record in the originating page at some point before the modal transfer takes place.
- Accesses and changes the values, if necessary, in the modal component.
- Accesses the values from the shared work record from the originating component after the modal component is dismissed.

Parameters

menuname Name of the menu through which the modal component is accessed.

<i>barname</i>	Name of the menu bar through which the modal component is accessed.
<i>menuitem_name</i>	Name of the menu item through which the modal component is accessed.
<i>page_group_item_name</i>	The component item name of the page to be displayed on top of the modal component when it displays. The component item name is specified in the component definition.
<i>action</i>	String representing the action mode in which to start up the component. You can use either a character value (passed in as a string) or a constant:

Value	Constant	Description
A	%Action_Add	Add
U	%Action_UpdateDisplay	Update/Display
L	%Action_UpdateDisplayAll	Update/Display All
C	%Action_Correction	Correction
E	%Action_DataEntry	Data Entry
P	%Action_Prompt	Prompt

If only one action mode is allowed for the component, that action mode is used. If more than one action mode is allowed, the user can choose which mode to come up in.

<i>shared_record_name</i>	<p>The record name of the shared work record (preceded by the reserved word Record). This record must include:</p> <ul style="list-style-type: none"> • fields that are search keys in the modal component search record; if search key fields are not provided, or if they are invalid, the user will access the modal component via the search dialog • other fields to pass to the modal component • fields to get back from the modal component after it has finished processing
---------------------------	--

keylist

A list of field specifications used to select a unique row at level zero in the page you are transferring to, by matching keys in the page you are transferring from. It can also be an already instantiated record object. The keys in fieldlist must uniquely identify a row in the "to" page search record, and they must be shared by the "from" and "to" pages. If a keylist is needed but not specified, or if no unique row is identified, the search dialog appears. If a record object is specified, any field of that record object *that is also a field of the search record for the destination component* will be added to *keylist*.

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```
If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;
```

Example

The following example shows how to structure a **DoModalComponent** function call:

```
DoModalComponent (MENUNAME.MAINTAIN_ITEMS_FOR_INVENTORY, BARNAME.USE_A,
ITEMNAME.ITEM_DEFINITION, PAGE.ESTABLISH_AN_ITEM, "C", RECORD.NEW7_WRK);
```

Supporting PeopleCode is required if you need to assign values to fields in the shared work record or access those values, either from the originating component, or from the modal component.

Related Topics

IsModalComponent, Transfer, TransferPage, and Implementing Modal Transfers

DoModalPanelGroup

Syntax

```
DoModalPanelGroup (MENUNAME.menuname, BARNAME.barname, ITEMNAME.menuitem_name,
    PANEL.panel_group_item_name, action, RECORD.shared_record_name)
```

Description

DoModalPanelGroup launches a modal component.



The **DoModalPanelGroup** function is supported for compatibility with previous releases of PeopleTools. Future applications should use DoModalComponent instead.

DoSave

Syntax

```
DoSave ( )
```

Description

DoSave saves the current page. **DoSave** defers processing to the end of the current PeopleCode program event, as distinct from DoSaveNow, which causes save processing (including SaveEdit, SavePreChange, SavePostChange, and Workflow PeopleCode) to be executed immediately.

DoSave can be used only in FieldEdit, FieldChange, or MenuItemSelected PeopleCode.

Returns

None.

Example

The following example sets up a key list with AddKeyListItem, saves the current page and then transfers the user to a page named PAGE_2.

```
ClearKeyListItem( );
AddKeyListItem(OPRID, OPRID);
AddKeyListItem(REQUEST_ID, REQUEST_ID);
SetNextPage("PAGE_2");
DoSave( );
TransferPage( );
```

Related Topics

DoCancel, DoSaveNow, TransferPage

DoSaveNow

Syntax

`DoSaveNow ()`

Description

DoSaveNow is designed primarily for use with remote calls. It enables a PeopleCode program to save page data to the database before running a remote process (most frequently a COBOL process) that will access the database directly. It is generally necessary to call **DoSaveNow** before calling `RemoteCall`.

DoSaveNow causes the current page to be saved immediately. Save processing (including `SaveEdit`, `SavePreChange`, `SavePostChange`, and Workflow PeopleCode) is executed before continuing execution of the program where **DoSaveNow** is called. **DoSaveNow** differs from the `DoSave` function in that **DoSave** defers saving the component until after any running PeopleCode is completed.

DoSaveNow can only be called from a `FieldEdit` or `FieldChange` event.

Errors in DoSaveNow Save Processing

DoSaveNow initiates save processing. It handles errors that occur during save processing as follows:

- If save processing encounters a `SaveEdit` error, it displays the appropriate message box. **DoSaveNow** immediately exits from the originating `FieldChange` or `FieldEdit` program. The user can correct the error and continue.
- If save processing encounters a fatal error, it displays the appropriate fatal error. **DoSaveNow** handles the error by immediately exiting from the originating `FieldChange` or `FieldEdit` program. The user must then cancel the page.
- If save processing completes with no errors, PeopleCode execution continues on the line after the **DoSaveNow** call in `FieldChange` or `FieldEdit`.

Restrictions on Use of DoSaveNow

The following restrictions apply:

- **DoSaveNow** can only be executed from a `FieldEdit` or `FieldChange` event.
- Deferred operations should not be called before the **DoSaveNow**. Deferred operations include **DoSave**, **TransferPage**, **SetCursorPos**, **EndModal**.
- Components that use **DoSaveNow** must not have **DoCancel**, **Transfer**, **TransferPage**, or **WinEscape** calls in PeopleCode attached to save action events (`SaveEdit`, `SavePreChange`, and `SavePostChange`), because these functions terminate the component, which would cause the program from which **DoSaveNow** was called to terminate prematurely.



Application developers should be aware that **DoSaveNow** may result in unpredictable behavior if PeopleCode in save events deletes rows or inserts rows into scrolls. PeopleCode that runs after **DoSaveNow** must be designed around the possibility that rows were deleted or inserted (which causes row number assignments to change). Modifying data on a deleted row may cause it to become "undeleted."

Returns

None.

Example

The following example calls **DoSaveNow** to save the component prior to running a remote process in the **remote_depletion** declared function:

```
Declare Function remote_depletion PeopleCode FUNCLIB_ININTFC.RUN_DEPLETION
FieldFormula;

/*
Express Issue Page - run Depletion job through RemoteCall()
*/
If %Component = COMPONENT.EXPRESS_ISSUE_INV Then
    DoSaveNow();
    &BUSINESS_UNIT = FetchValue(SHIP_HDR_INV.BUSINESS_UNIT, 1);
    &SHIP_OPTION = "S";
    &SHIP_ID = FetchValue(SHIP_HDR_INV.SHIP_ID, 1);
    remote_depletion(&BUSINESS_UNIT, &SHIP_OPTION, &SHIP_ID, &PROGRAM_STATUS);
End-If
```

CHAPTER 2

PeopleCode Built-in Functions and Language Constructs E-M

EnableMenuItem

Syntax

```
EnableMenuItem(BARNAME.menuubar_name, ITEMNAME.menuitem_name)
```

Description

EnableMenuItem enables (makes available for selection) the specified menu item. To apply this function to a pop-up menu, use the PrePopup Event of the field with which the pop-up menu is associated.

If you're using this function with a pop-up menu associated with a page (not a field), the earliest event you can use is the **PrePopup** event for the first "real" field on the page (that is, the first field listed in the **Order** view of the page in Application Designer.)

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```
If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;
```

Returns

None.

Parameters

<i>menubar_name</i>	Name of the menu bar that owns the menu item, or, in the case of pop-up menus, the name of the pop-up menu that owns the menu item.
<i>menuitem_name</i>	Name of the menu item.

Example

```
EnableMenuItem(BARNAME.MYPOPUP1, ITEMNAME.DO_JOB_TRANSFER);
```

Related Topics

CheckMenuItem, UnCheckMenuItem, DisableMenuItem, HideMenuItem

EncodeURL

Syntax

```
EncodeURL(URLString)
```

Description

The **EncodeURL** function applies URL encoding rules, including escape characters, to the string specified by *URLString*. The method used to encode the *URLString* is the standard defined by W3C. This function returns a string that contains the encoded URL. Special characters (/, :, and so on) are assumed to be used for their special purposes (that is, as part of the URI/protocol information) and are not encoded. If you need to encode such characters, use the `EncodeURLForQueryString` function.

Parameters

<i>URLString</i>	Specify the string you want encoded. This parameter takes a string value.
------------------	---

Returns

An encoded string.

Example

```
&MYSTRING = EncodeURL("http://corp.office.com/hr/benefits/401k/401k_home.htm");
```

Related Topics

Internet Script Classes, EncodeURLForQueryString, Unencode

EncodeURLForQueryString

Syntax

```
EncodeURLForQueryString(URLString)
```

Description

The **EncodeURLForQueryString** function encodes *URLString* for use in a querystring parameter in an URL. It encodes &, ? and = characters, along with all other unsafe characters.

If the link is constructed in a PeopleSoft Internet Architecture page, and the value of a link field, you should not call EncodeURL to encode the entire URL, as the PIA architecture does this for you. You will still need to unencode the parameter value when you retrieve it, however.

Parameters

<i>URLString</i>	Specify the string you want encoded. This parameter takes a string value.
------------------	---

Returns

An encoded URL string.

Example

In an Internet Script, to construct an anchor with a URL in a querystring parameter, you should do the following

```
&url = "http://host/iclientservlet.jrun/peoplesoft8?emplid=1111&mkt=usa"
```

```
&href = %Request.RequestURI | "?" | %Request.QueryString | "&myurl=" |  
EncodeURLForQueryString(&url);
```

```
%Response.WriteLine("<a href= " | EncodeURL(&href) | ">My Link</a>");
```

The following uses a generic method to find, then encode, the URL, for the external link:

```
&StartPos = Find("?", &URL, 1);  
  
&CPColl = &Portal.ContentProviders;  
  
&strHREF = EncodeURLForQueryString(Substring(&URL, &StartPos + 1, Len(&URL) -  
  &StartPos));  
  
&LINK = &Portal.GetQualifiedURL("PortalServlet", "PortalOriginalURL=" |  
  &CPColl.ItemByName(&CP_NAME).URI | "?" | &strHREF);
```

Related Topics

EncodeURL, EncodeURLForQueryString, Unencode, Internet Script Classes

Encrypt

Syntax

```
Encrypt (KeyString, ClearTextString)
```

Description

The **Encrypt** function encrypts a string. This function is generally used with merchant passwords.

The value you use for *KeyString* must be the same for **Decrypt** and **Encrypt**.

Parameters

<i>KeyString</i>	Specify the key used for encrypting the string. You can specify a NULL value for this parameter, that is, two quotation marks with no blank space between them ("").
<i>ClearTextString</i>	Specify the string you want encrypted.

Returns

An encrypted string.

Example

The following encrypts a field if it contains a value. It also removes any blanks either preceding or trailing the value.

```
If All (PSCNFMRCHTOKEN.WRKTOKEN) Then  
  
    CMPAUTHTOKEN = Encrypt ("", RTrim (LTrim (PSCNFMRCHTOKEN.WRKTOKEN)) );  
  
End-If;
```

Related Topics

Decrypt, Hash, Security

EndMessage

Syntax

```
EndMessage (message, messagebox_title)
```


Description



EndMessage is obsolete and is supported only for backward compatibility. **MessageBox**, which can now be used to display informational messages in any PeopleCode event, should be used instead.

The **EndMessage** function displays a message at the end of a transaction, at the time of the database COMMIT. This function can only be used in SavePostChange PeopleCode.

When an **EndMessage** function executes, PeopleTools:

- Verifies that the function is in SavePostChange; if it is not, an error will occur and the function will terminate
- Displays the message
- Terminates the SavePostChange PeopleCode program

Because it terminates the SavePostChange program, **EndMessage** will always be the last statement executed in the program on the specific field and row where the **EndMessage** is called. For this reason, you need to write the SavePostChange program so that all necessary processing takes place before the **EndMessage** statement. PeopleCode programs on other fields and rows will execute as usual.

Returns

None.

Parameters

<i>message</i>	A string that must be enclosed in quotes containing the message text you want displayed.
<i>messagebox_title</i>	A string that must be enclosed in quotes containing the title of the message—it appears in the message box title bar.

Example

The following example is from SavePostChange event PeopleCode. It checks to see whether a condition is true, and if so, it displays a message and terminates the SavePostChange program. If the condition is false, then processing continues in the **Else** clause:

```
If BUSINESS_UNIT = BUS_UNIT_WRK.DEFAULT_SETID Then
    EndMessage(MsgGet(20000, 12, "Message not found in Message Catalog"), "
");
Else
/* any other SavePostChange processing in Else clause */
```

Related Topics

MessageBox, WinMessage

EndModal

Syntax

```
EndModal(returnvalue)
```

Description

The EndModal function closes a currently open secondary page. It is required only for secondary pages that do not have **OK** and **Cancel** buttons. If the secondary page has **OK** and **Cancel** buttons, then the function for exiting the page is built in and no PeopleCode is required.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```
If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;
```

Returns

None.

Parameters

returnvalue

A Number value that determines whether the secondary page data is copied back to the parent page. A positive value runs **SaveEdit** PeopleCode and copies the data (this is the same as pressing the **OK** button). A value of zero skips **SaveEdit** and discards buffer changes made in the secondary page (this is the same as pressing the **Cancel** button). This value becomes the return value of the **DoModal** function that started the secondary page, and it can be tested after the secondary page is closed.

Example

The following statement acts as an **OK** button:

```
EndModal (1) ;
```

The following statement acts as a **Cancel** button:

```
EndModal (0) ;
```

Related Topics

DoModal, IsModal

Error

Syntax

```
Error str
```

Description

Error is generally used in FieldEdit or SaveEdit PeopleCode to stop processing and display an error message. It is distinct from **Warning**, which displays a warning message, but does not stop processing. **Error** is also used in RowDelete and RowSelect PeopleCode events.



The behavior of the Error function in the RowSelect event is very different from its normal behavior.



For more information, see Error in RowSelect .

The text of the error message (the *str* parameter), should always be stored in the Message Catalog and retrieved using the **MsgGet** or **MsgGetText** function. This makes it much easier for the text to be translated, and it also lets you include more detailed Explain text about the error.

When **Error** executes in a PeopleCode program, the program terminates immediately and no statements after the **Error** are executed. In other respects behavior of **Error** differs, depending on which PeopleCode event the function occurs in.

Errors in FieldEdit and SaveEdit

The primary use of **Error** is in FieldEdit and SaveEdit PeopleCode:

- In FieldEdit, **Error** stops processing, displays a message, and highlights the relevant field.
- In SaveEdit, **Error** stops all save processing and displays a message, but does not highlight any field. You can move the cursor to a specific field using the **SetCursorPos** function, but be sure to call **SetCursorPos** *before* calling **Error**, otherwise **Error** will stop processing before

SetCursorPos is called. Note that an **Error** on any field in SaveEdit will stop all save processing, and no page data will be saved to the database.

Errors in RowDelete

When the user attempts to delete a row of data, the system first prompts for confirmation. If the user confirms, the RowDelete event fires. An **Error** in the RowDelete event displays a message and prevents the row from being deleted.

Error in RowSelect

The behavior of **Error** in RowSelect is totally anomalous, and is supported only for backward compatibility. It is used to filter rows that are being added to a page scroll after the rows have been selected and brought down to the client. *No message is displayed.* **Error** causes the Component Processor to add the current row (the one where the PeopleCode is executing) to the page scroll, then stops adding any additional rows to the page scroll.

The behavior of **Error** in the RowSelect event permits you to filter out rows that are above or below some limiting value. In practice this technique is rarely used, because it is more efficient to filter out rows of data before they are brought down to the client. This can be accomplished with search views or effective date processing.

Errors in Other Events

Do not use the **Error** function in any of the remaining events, which include:

- FieldDefault
- FieldFormula
- RowInit
- FieldChange
- Prepopup
- RowInsert
- SavePreChange
- SavePostChange

The user has no control over processing that occurs in these events. If the Component Processor encounters an **Error** in one of these events, the user can't fix it. The Component Processor requires the user to cancel the component to avoid unpredictable results, which results in loss of any changes that the user has made.

Returns

None.

Parameters

Str A string containing the text of the error message. This string should always be stored in the Message Catalog and retrieved via the **MsgGet** or **MsgGetText** function. This makes translation much easier and also enables you to provide detailed Explain text about the error.

Example

The following example, from SaveEdit PeopleCode, displays an error message, stops all save processing, and places the cursor in the QTY_ADJUSTED field. Note that **SetCursorPos** must be called before **Error**.

```
If PAGES2_INV_WRK.PHYS_CYC_INV_FLG = "Y" Then
    SetCursorPos(%Page, PHYSICAL_INV.INV_LOT_ID, CurrentRowNumber(1),
QTY_ADJUSTED, CurrentRowNumber());
    Error MsgGet(11100, 180, "Message not found.");
End-If;
```

Related Topics

MsgGet, MsgGetText, SetCursorPos, Warning, WinMessage

EscapeHTML

Syntax

EscapeHTML (*HTMLString*)

Description

The **EscapeHTML** function takes *HTMLString* and replaces all characters that are significant to a browser with special HTML escape sequences.

The characters that are replaced are ones that cause the browser to misinterpret the HTML if they aren't encoded. This function is intended to make the text "browser safe".

For example, the less-than symbol (<) is replaced with <., a single quotation mark (') is replaced with '., and so on.

This function is intended to be used with strings that display in an HTML area.

Parameters

HTMLString Specify a string of HTML that contains characters that need to be replaced with HTML escape sequences.

Returns

A string containing the original text plus HTML escape sequences.

Related Topics

EscapeJavaScriptString, EscapeWML

EscapeJavaScriptString

Syntax

```
EscapeJavaScriptString(String)
```

Description

The **EscapeJavaScriptString** function takes *String* and replaces the characters that have special meaning in a JavaScript string with special escape sequences.

For example, a single quotation mark (') is replaced by \', a new line character is replaced by \n, and so on.

The characters that are replaced are ones that cause the browser to misinterpret the JavaScript if they aren't encoded. This function is intended to make the text "browser safe".

This function is intended to be used with test that becomes part of a JavaScript program.

Parameters

<i>String</i>	Specify a string that contains character that need to be replaced with JavaScript escape sequences.
---------------	---

Returns

A string containing the original text plus JavaScript escape sequences.

Related Topics

EscapeHTML, EscapeWML

EscapeWML

Syntax

```
EscapeWML(String)
```

Description

The EscapeWML function escapes special characters that are significant to WML. This includes <, >, \$ (escaped as \$\$), &, ' and ".

This function is intended to be used with strings that display on an WML browser.

Parameters

String Specify a string that contains characters that need to be replaced with WML escape sequences.

Returns

A string containing the original plus text plus WML escape sequences.

Related Topics

EscapeHTML, EscapeJavascriptString

Evaluate

Syntax

```

Evaluate left_term
    When [relop_1] right_term_1
        [statement_list]
    .
    .
    .

    [When [relop_n] right_term_n
        [statement_list]]
    [When-other
        [statement_list]]
End-evaluate

```

Description

Use the **Evaluate** statement in cases where you want to check multiple conditions. It takes an expression, *left_term*, and compares it to compatible expressions (*right_term*) using the relational operators (*relop*) in a sequence of **When** clauses. If *relop* is omitted, then = is assumed. If the result of the comparison is True, it executes the statements in the **When** clause, then moves on to evaluate the comparison in the following **When** clause. It executes the statements in all of the **When** clauses for which the comparison evaluates to True. If and only if none of the **When** comparisons evaluates to True, it executes the statement in the **When-other** clause (if one is provided).

To end the **Evaluate** after the execution of a **When** clause, you can add a **Break** statement at the end of the clause:

Example

The following is an example of a **When** statement taken evaluates ACTION and performs various statements based on its value:

```

&PRIOR_STATUS = PriorEffdt(EMPL_STATUS);
Evaluate ACTION
When "HIR"
    If %Mode = "A" Then

```

```

Warning MsgGet(1000, 13, "You are hiring an employee and Action is not set
to Hire.");
End-if;
Break;
When = "REH"
If All(&PRIOR_STATUS) and
    not (&PRIOR_STATUS = "T" or
        &PRIOR_STATUS = "R" ) Then
    Error MsgGet(1000, 14, "Hire or Rehire action is valid
    only if employee status is Terminated or Retired.");
End-if;
Break;
When-Other
/* default code */
End-evaluate;

```

Exact

Syntax

```
Exact(string1, string2)
```

Description

Exact compares two text strings and returns True if they are the same, False otherwise. **Exact** is case-sensitive because it uses the internal character codes.

Returns

Returns a Boolean value: True if the two strings match in a case-sensitive comparison.

Example

The examples set &MATCH to True, then False:

```

&MATCH = Exact("PeopleSoft", "PeopleSoft");
&MATCH = Exact("PeopleSoft", "Peoplesoft");

```

Related Topics

Len, String, %Substring

Exec

Syntax

```
Exec(command_str [, synch_exec])
```


Description

Exec is a cross-platform function that executes an external program on either UNIX or Windows. **Exec**, unlike WinExec, can be called from PeopleCode running on either the client or the application server.

If your PeopleCode is running on the client, **Exec** runs on the client. If your PeopleCode is running on the server, **Exec** runs on the server. Make sure you know which environment your PeopleCode is executing in.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

The function can make either a synchronous or asynchronous call. Synchronous execution acts as a "modal" function, suspending the PeopleSoft application until the called executable completes. This is appropriate if you want to force the user (or the PeopleCode program) to wait for the function to complete its work before continuing processing. Asynchronous processing, which is the default, launches the executable and immediately returns control to the calling PeopleSoft application.

If **Exec** is unable to execute the external program, the PeopleCode program will terminate with a fatal error.

Returns

Returns a Number value equal to the process ID of the called process.

Parameters

<i>command_str</i>	A String containing the path and name of the executable file.
--------------------	---



PS_HOME is always prefixed to *command_str*.

<i>synch_exec</i>	Optional Boolean value indicating whether to execute the external program synchronously (True) or asynchronously (False). If this parameter is omitted, the program executes asynchronously.
-------------------	--

Restrictions on Use in PeopleCode Events

When **Exec** is used to execute a program synchronously (that is, if its *synch_exec* parameter is set to True) it behaves as a think-time function, which means that it can't be used in any of the following PeopleCode events:

- SavePreChange

- SavePostChange
- Workflow
- RowSelect
- Any PeopleCode event that fires as a result of a ScrollSelect, ScrollSelectNew, RowScrollSelect or RowScrollSelectNew function call.



For more information, see Think-Time Functions.

Related Topics

Declare Function, RemoteCall, ScheduleProcess, WinExec

ExecuteRolePeopleCode

Syntax

```
ExecuteRolePeopleCode (RoleName)
```

Description

The **ExecuteRolePeopleCode** function executes the PeopleCode Rule for the Role *RoleName*. This function returns an array of string containing dynamic members (UserIds).

Typically, this function is used by an Application Engine process that runs periodically and executes the role rules for different roles. It could then write the results of the rules (a list of users) into the security tables, effectively placing users in certain roles based on the rule.



For more information see Security.

Parameters

<i>RoleName</i>	Specify the name of an existing role.
-----------------	---------------------------------------

Returns

An array of string containing the appropriate UserIds.

Example

The following saves valid users to a temporary table:

```
Local array of string &pcode_array_users;
```

```

SQLExec("delete from ps_dynrole_tmp where ROLENAME=:1", &ROLENAME);

If &pcode_rule_status = "Y" Then

    SQLExec("select RECNAME, FIELDNAME, PC_EVENT_TYPE, PC_FUNCTION_NAME from
    PSROLEDEFN where ROLENAME= :1", &ROLENAME, &rec, &fld, &pce, &pcf);

    If (&rec <> "" And

        &fld <> "" And

        &pce <> "" And

        &pcf <> "") Then

        &pcode_array_users = ExecuteRolePeopleCode(&ROLENAME);

        &pcode_results = True;

    Else

        &pcode_results = False;

    End-If;

    &comb_array_users = &pcode_array_users;

End-If;

```

Related Topics

ExecuteRoleQuery, ExecuteRoleWorkflowQuery, IsUserInPermissionList, IsUserInRole, %Roles, Security

ExecuteRoleQuery

Syntax

```
ExecuteRoleQuery(RoleName, BindVars)
```

where *BindVars* is an arbitrary-length list of bind variables that are stings in the form:

```
bindvar1 [, bindvar2] . . .
```

Description

The **ExecuteRoleQuery** function executes the Query rule for the role *rolename*, passing in *BindVars* as the bind variables. This function returns an array object containing the appropriate user members (UserIds).



For more information see Security.

Parameters

<i>RoleName</i>	Specify the name of an existing role.
<i>BindVars</i>	A list of bind variables to be substituted in the query. These bind variables must be strings. You can't use numbers, dates, and so on.

Returns

An array object containing the appropriate UserIds.

Related Topics

ExecuteRolePeopleCode, ExecuteRoleWorkflowQuery, IsUserInPermissionList, IsUserInRole, %Roles, Security

ExecuteRoleWorkflowQuery

Syntax

```
ExecuteRoleWorkflowQuery(RoleName, BindVars)
```

where *BindVars* is an arbitrary-length list of bind variables in the form:

```
bindvar1 [, bindvar2] . . .
```

Description

The **ExecuteRoleWorkflowQuery** function executes the Workflow Query rule for the role *rolename*, passing in *BindVars* as the bind variables. This function returns an array object containing the appropriate user members (UserIds).



For more information see Security.

Parameters

<i>RoleName</i>	Specify the name of an existing role.
<i>BindVars</i>	A list of bind variables to be substituted in the query.

Returns

An array object containing the appropriate UserIds.

Related Topics

ExecuteRolePeopleCode, ExecuteRoleQuery, IsUserInPermissionList, IsUserInRole, %Roles, Security

Exit

Syntax

```
Exit ([1])
```

Description

Exit causes immediate termination of a PeopleCode program. If the **Exit** statement is executed within a PeopleCode function, the main program terminates.

Parameters

- | | |
|---|---|
| 1 | Use this parameter to rollback database changes. Generally, this parameter is used in PeopleCode programs that affect application messages. When used with an application message, all database changes are rolled back, errors for the subscription contract are written to the subscription contract error table, and the status of the message is marked to Error . All errors that have occurred for this message will be viewable in the message monitor: even those errors detected by ExecuteEdits . |
|---|---|

Returns

None.

Example

The following example terminates the main program from a **For** loop:

```
For &I = 1 To ActiveRowCount(RECORD.SP_BUIN_NONVW)
  &ITEM_SELECTED = FetchValue(ITEM_SELECTED, &I);
  If &ITEM_SELECTED = "Y" Then
    &FOUND = "Y";
    Exit;
  End-If;
End-For;
```

Related Topics

Break, Return, Processing Messaging Errors

Exp

Syntax

Exp (*n*)

Description

Exp returns **e** raised to the power of *n* where *n* is a number. The constant **e** equals 2.71828182845904, the base of natural logarithms. The number *n* is the exponent applied to the base **e**.

Exp is the inverse of **Ln**, which is the natural logarithm of x.

Returns

Returns a Number value equal to the constant e raised to the power of *n*.

Example

The examples set &NUM to 2.71828182845904, then 7.389056099 (e²):

```
&NUM = Exp (1) ;  
&NUM = Exp (2) ;
```

Related Topics

Ln, Log10

ExpandBindVar

Syntax

ExpandBindVar (*str*)

Description

Inline bind variables can be included in any PeopleCode string. An inline bind variable is a field reference (in the form **recordname.fieldname**), preceded by a colon. The inline bind variable references the value in the field.

ExpandBindVar expands any inline bind variables that it finds in *str* into strings (converting the data type of non-character fields) and returns the resulting string. This will work with inline bind variables representing fields containing any data type except Object. It also expands bind variables specified using additional parameters.



For more information about bind variables and inline bind variables, see Bind Variables in SQLExec and Inline Bind Variables in SQLExec.

Returns

Returns a String value equal to the input string with all bind variables expanded.

Parameters

str A string containing one or more inline bind variables.

Example

A bind variable is included in the string &TESTSTR, which is then expanded into a new string containing the current value of BUS_EXPENSE_PER.EMPLID in place of the bind variable. If this program runs on the row for EMPLID 8001, the message displayed will read "This is a test using EmplID 8001".

```
&TESTSTR = "This is a test using EmplID :bus_expense_per.emplid";  
&RESULT = ExpandBindVar(&TESTSTR);  
WinMessage(&RESULT);
```

Related Topics

MessageBox, SQLExec

ExpandEnvVar

Syntax

```
ExpandEnvVar (str)
```

Description

ExpandEnvVar converts any environment variables that it finds within *str* into String values and returns the entire resulting string.

Restrictions on use in Three-Tier Mode and PeopleSoft Internet Architecture

Because it is specific to systems running MS-DOS or Windows, **ExpandEnvVar** is a client-only function, which limits its use in three-tier mode and in PeopleSoft Internet Architecture applications.

- In three-tier mode **ExpandEnvVar** can be used only in processing groups set to run on the client. If the **ExpandEnvVar** function is called in a processing group running on the application server, a runtime error will occur.
- In PeopleSoft Internet Architecture applications, where all PeopleCode runs on the application server, **ExpandEnvVar** cannot be used at all.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

Returns

Returns a string equal to `str` with any enclosed environment variables expanded to their values.

Parameters

str A string containing a DOS environment variable.

Example

Assume that the DOS environment variable `%NETDRIVE%` is equal to `"N:"`. The following PeopleCode sets `&newstring` equal to `"The network drive is equal to N:"`:

```
&newstring = ExpandEnvVar("The network drive is equal to %netdrive%.");
```

Related Topics

ChDrive, ChDir, ExpandBindVar, GetEnv

ExpandSqlBinds

Syntax

```
ExpandSqlBinds(string)
```

Description

Prior to PeopleTools 8.0, the PeopleCode replaced runtime parameter markers in SQL strings with the associated literal values. For databases that offer SQL statement caching, a match was never found in the cache so the SQL had to be re-parsed and re-assigned a query path.

In order to handle skipped parameter markers, each parameter marker is assigned a unique number. This doesn't change the values associated with the parameter markers.

However, *some SQL statements can't contain parameter markers because of database compatibility.*

To handle these exceptions, use the **ExpandSqlBinds** function. This function does the bind variable reference expansion, and can be used within a `SQLExec` statement or on its own.

You should *only* use `ExpandSQLBinds` for those parts of the string that you want to put literal values into. For example, you wouldn't use `ExpandSQLBinds` with any meta-SQL statements. The following code shows how to use `ExpandSQLBinds` with `%Table`:

```
SQLExec(ExpandSqlBinds("Insert.... Select A.Field, :1, :2 from ", "01", "02") |  
"%table(:1)", Record.MASTER_ITEM_TBL);
```

Parameters

string Specify the string you want to do the bind variable reference expansion on.

Returns

A string.

Example

The following example shows both the original string and what it expands to.

```
&NUM = 1;

&STRING = "My String";

&STR2 = ExpandSqlBinds("This :2 is an expanded string(:1)", &STRING, &NUM);
```

The above code produces the following value for &STR2:

```
This 1 is an expanded string(My String)
```

If you're having problems with an old SQL statement binds, you can use ExpandSqlBinds with it. For example, if your SQLExec looks like this:

```
SQLExec("String with concatenated bindrefs 'M':2, 'M':1", &VAR1, &VAR2),
&FETCHRESULT1, &FETCHRESULT2);
```

you can make it work by expanding it as follows:

```
SQLExec(ExpandSqlBinds("String with concatenated bindrefs 'M':2, 'M':1", &VAR1,
&VAR2), &FETCHRESULT1, &FETCHRESULT2);
```

Related Topics

SQLExec, SQL Class

Fact

Syntax

Fact (x)

Description

Fact returns the factorial of a positive integer *x*. The factorial of a number *x* is equal to 1*2*3*...**x*. If *x* is not an integer, it is truncated to an integer.

Returns

Returns a Number equal to the factorial of *x*.

Example

The example sets &X to 1, 1, 2, then 24. **Fact**(2) is equal to 1*2; **Fact**(4) is equal to 1*2*3*4:

```
&X = Fact (0);
&X = Fact (1);
```

```
&X = Fact (2) ;  
&X = Fact (4) ;
```

Related Topics

Product

FetchSQL

Syntax

```
FetchSQL( [SQL.]sqlname[, dbtype[, effdt]] )
```

Description

The **FetchSQL** function returns the SQL definition with the given *sqlname* as *SQL.sqlname* or a string value, matching the *dbtype* and *effdt*. If *sqlname* is a literal name, it must be in quotes.

Parameters

sqlname

Specify the name of a SQL definition. This is either in the form *SQL.sqlname* or a string value giving the *sqlname*.

dbtype

Specify the database type associated with the SQL definition. This parameter takes a string value. If *dbtype* isn't specified or is null(""), it defaults to the current database type (the value returned from the %DbName system variable.)

Valid values for *dbtype* are as follows. These values are not case sensitive:

- DEFAULT
- DB2ODBC
- DB2UNIX
- INFORMIX
- MICROSOFT
- ORACLE
- SYBASE

Note. Database platforms are subject to change.

effdt

Specify the effective date associated with the SQL definition. If *effdt* isn't specified, it defaults to the current as of date, that is, the value returned from the %AsOfDate system variable.

Returns

The SQL statement associated with *sqlname* as a string.

Example

The following code gets the text associated with the ABCD_XY SQL Definition for the current DBType and as of date:

```
&SQLSTR = FetchSQL(SQL.ABC_XY);
```

The following code gets the text associated with the ABCD_XY SQL Definition for the current DBType and the third of November, 1998:

```
&SQLSTR = FetchSQL(SQL.ABCD_XY, "", Date(19981103));
```

Related Topics

CreateSQL, DeleteSQL, SQLExec, GetSQL, StoreSQL and SQL Class

FetchValue

Syntax

```
FetchValue(scrollpath, target_row, [recordname.]fieldname)
```

where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]]
RECORD.target_recname
```

To prevent ambiguous references, you can also use **SCROLL**.*scrollname*, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Description

FetchValue returns the value of a buffer field in a specific row of a scroll level.



This function remains for backward compatibility only. Use the Value Field class property instead. See Data Buffer Access. See also LongTranslateValue, ShortTranslateValue Field properties .

This function is generally used to retrieve the values of buffer fields outside the current context; if a buffer field is in the current context, you can reference it directly using a **[recordname.]fieldname** expression.



For more information, see Referencing Data in the Component Buffer.

Returns

Returns the field value as an Any data type.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
<i>target_row</i>	An integer specifying the row on the target scroll level where the referenced buffer field is located.
[recordname.]fieldname	The name of the field where the value to fetch is located. The field can be on scroll level one, two, or three of the active page. The <i>recordname</i> prefix is required if the call to FetchValue is in a record definition other than <i>recordname</i> .

Example

The following example retrieves the value from field **DEPEND_ID** in record **DEPEND** on row **&ROW_CNT** from scroll level one:

```
&VAL = FetchValue(Scroll.Depend, &ROW_CNT, Depend.Depend_ID);
```

Related Topics

ActiveRowCount, CopyRow, CurrentRowNumber, PriorValue, UpdateValue

FieldChanged

Syntax

The syntax of **FieldChanged** varies, depending on whether it uses a scroll path reference or a contextual reference to specify the field.

If it uses a scroll path reference, the syntax is:

```
FieldChanged(scrollpath, target_row, [recordname.]fieldname)
```

where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]]
RECORD.target_recname
```

To prevent ambiguous references, you can also use **SCROLL.scrollname**, where *scrollname* is the same as the scroll level's primary record name.

If it uses a contextual reference, the syntax is:

```
FieldChanged([recordname.]fieldname)
```

In this construction the scroll level and row number are determined based on the current context.



For more information on scroll path and contextual references, see Referencing Data in the Component Buffer.

Description

FieldChanged returns True if the referenced buffer field has been modified since being retrieved from the database either by a user or by a PeopleCode program. This is useful during SavePreChange or SavePostChange processing for checking whether to make related updates based on a change to a field.



This function remains for backward compatibility only. Use the IsChanged Field class property instead. See Data Buffer Access.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
<i>[recordname.]fieldname</i>	The name of the field where the value to check is located. The field can be on scroll level one, two, or three of the active page. The <i>recordname</i> prefix is required if the call to FieldChanged is in a record definition other than <i>recordname</i> .
<i>target_row</i>	The row number of the target row. If this parameter is omitted, the function assumes the row on which the PeopleCode program is executing.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Related Functions

Save PeopleCode programs (SaveEdit, SavePreChange, SavePostChange) normally process each row of data in the component. The following four functions allow you to control more precisely when the Component Processor should perform save PeopleCode:

- FieldChanged
- RecordChanged
- RecordDeleted
- RecordNew

These functions allow you to restrict save program processing to specific rows.

Example

The following example checks three fields and sets a flag if any of them has changed:

```
/* Set the net change flag to 'Yes' if the scheduled date, scheduled */  
/* time or quantity requested is changed */  
If FieldChanged(QTY_REQUESTED) Or  
    FieldChanged(SCHED_DATE) Or  
    FieldChanged(SCHED_TIME) Then  
    NET_CHANGE_FLG = "Y";  
End-If;
```

Related Topics

RecordChanged, RecordDeleted, RecordNew

FileExists

Syntax

```
FileExists(filename [, pathtype])
```

Description

The **FileExists** function determines whether a particular external file is present on your system, so you can decide which mode to use when you open the file for writing.



If you want to open a file for reading, you should use GetFile function or the Open file class method with the "e" mode, which prevents another process from deleting or renaming the file between the time you tested for the file and when you open it.

Parameters

filespec

Specify the name, and optionally, the path, of the file you want to test.

pathtype

If you have prepended a path to the file name, use this parameter to specify whether the path is an absolute or relative path. The valid values for this parameter are:

- %FilePath_Relative (default)
- %FilePath_Absolute

If you don't specify *pathtype* the default is %FilePath_Relative.

If you don't specify a path, the file is assumed to be in one of two locations, depending on where your PeopleCode program is executing.

- When your program is executing on the client, the location is the directory specified by the **TEMP** environment variable.

Note. Your system security should verify if a user has the correct permissions before allowing access to a drive (for example, if a user changes their **TEMP** environmental variable to a network drive they don't normally have access to, your system security should detect this.)

- When your program is executing on the server, the location is the "**files**" directory under the directory specified by the PeopleSoft **PS_SERVDIR** environment variable.

If you specify a relative path, that path will be appended to the TEMP or PS_SERVDIR variable (depending on where your PeopleCode is executing.) You must make sure to not include a drive letter when using a relative path.

If the path is an absolute path, whatever path you specify is used verbatim. You must specify a drive letter as well as the complete path. You can't use any wildcards when specifying a path.

The Component Processor automatically converts platform-specific separator characters to the appropriate form for where your PeopleCode program is executing. On a WIN32 system, UNIX "/" separators are converted to "\", and on a UNIX system, WIN32 "\" separators are converted to "/".



The syntax of the file path does *not* depend on the file system of the platform where the file is actually stored; it depends only on the platform where your PeopleCode is executing.



For more information about setting a program's processing location, see PeopleCode and PeopleSoft Internet Architecture.

Returns

A boolean value: True if the file exists, False if it doesn't.

Example

The following example opens a file for appending if it exists on the system:

```
If FileExists("c:\work\item.txt", %FilePath_Absolute) Then

    &MYFILE = GetFile("c:\work\item.txt", "A");

    /* Process the file */

    &MYFILE.Close();

End-If;
```

Related Topics

FindFiles, GetFile, and File Class

Find

Syntax

```
Find(string, within_string [, number])
```

Description

Find locates one string of text within another string of text and returns the character position of the string as an integer. **Find** is case-sensitive and does not allow wildcards.

Returns

Returns a Number value indicating the starting position of *string* in *within_string*.

Find returns with 0 if *string* does not appear in *within_string*, if *number* is less than or equal to zero, or if *number* is greater than the length of *within_string*.

Parameters

<i>string</i>	The text you are searching for.
<i>within_string</i>	The text string you are searching within.
<i>number</i>	The position of <i>within_string</i> at which you want to start your search. If you omit <i>number</i> , Find starts at the first character of <i>within_string</i> .

Example

The first statement below returns 1; the second statement returns 6.

```
&POS = Find("P", "PeopleSoft")
&POS = Find("e", "PeopleSoft", 4)
```

Related Topics

Exact, Len

Findb

Syntax

```
Findb(string, within_string [, number])
```

Description

Findb locates one string of text within another string of text and returns the byte of the string as an integer. **Findb** is byte-oriented, whereas **Find** is character-oriented; this distinction is significant whenever the source string contains double-byte characters.

Findb is case-sensitive and does not allow wildcards.

Findb displays an error message if *string* does not appear in *within_string*, if *number* is less than zero, or if *number* is greater than the length of *within_string*.

Returns

Returns a Number value indicating the starting position of *string* in *within_string*.

Parameters

<i>string</i>	The text you are searching for.
<i>within_string</i>	The text string you are searching within.
<i>number</i>	The position of <i>within_string</i> at which you want to start your search. If you omit <i>number</i> , Find starts at the first byte of <i>within_string</i> .

Examples

The first statement below returns 1; the second statement returns 7.

```
&POS = Find("P", "PeopleSoft")
&POS = Find("S", "PeopleSoft", 4)
```

Related Topics

Lenb

FindFiles

Syntax

```
FindFiles(filespec_pattern [, pathtype])
```

Description

The **FindFiles** function returns a list of the external filenames that match the filename pattern you provide, in the location you specify.

Parameters

filespec_pattern

Specify the path and filename pattern for the files you want to find. The path can be any string expression that represents a single relative or absolute directory location. The filename pattern—but not the path—can include two wildcards:

- * Asterisk—matches zero or more characters at its position.
- ? Question mark—matches exactly one character at its position.

pathtype

If you have prepended a path to the file name, use this parameter to specify whether the path is an absolute or relative path. The valid values for this parameter are:

- %FilePath_Relative (default)
- %FilePath_Absolute

If you don't specify *pathtype* the default is %FilePath_Relative.

If you don't specify a path, the file is assumed to be in one of two locations, depending on where your PeopleCode program is executing.

- When your program is executing on the client, the location is the directory specified by the **TEMP** environment variable.

Note. Your system security should verify if a user has the correct permissions before allowing access to a drive (for example, if a user changes their **TEMP** environmental variable to a network drive they don't normally have access to, your system security should detect this.)

- When your program is executing on the server, the location is the "**files**" directory under the directory specified by the PeopleSoft **PS_SERVDIR** environment variable.

If you specify a relative path, that path will be appended to the TEMP or PS_SERVDIR variable (depending on where your PeopleCode is executing.) You must make sure you do not include a drive letter when using a relative path.

If the path is an absolute path, whatever path you specify is used verbatim. You must specify a drive letter as well as the complete path. You can't use any wildcards when specifying a path.

The Component Processor automatically converts platform-specific separator characters to the appropriate form for where your PeopleCode program is executing. On a WIN32 system, UNIX "/" separators are converted to "\", and on a UNIX system, WIN32 "\" separators are converted to "/".



The syntax of the file path does *not* depend on the file system of the platform where the file is actually stored; it depends only on the platform where your PeopleCode is executing.



For more information about setting a program's processing location, see PeopleCode and PeopleSoft Internet Architecture.

Returns

A string array whose elements are filenames qualified with the same relative or absolute path you specified in the input parameter to the function.

Example

The following example finds all files in the system's TEMP location whose names end with ".txt", then opens and processes each one in turn:

```
Local array of string &FNAMES;

Local file &MYFILE;

&FNAMES = FindFiles("\*.txt");

while &FNAMES.Len > 0

    &MYFILE = GetFile(&FNAMES.Shift(), "R"); /* Open each file */

    /* Process the file contents */

    &MYFILE.Close();

end-while;
```

Related Topics

FileExists, GetFile, and File Class, Array Class

FlushBulkInserts

Syntax

```
FlushBulkInserts()
```

Description

The **FlushBulkInserts** function moves the bulk inserted rows from the bulk insert buffer(s) of the PeopleSoft process to the physical table(s) on the database. This flushes all open SQL objects that have pending bulk inserts, but performs no COMMITs. If the flush fails, the PeopleCode program terminates.

When executing a SQL insert using a SQL object with the BulkMode property set to True, the rows being inserted cannot be selected by this database connection until the bulk insert is flushed. For another connection to the database to be able to select those rows, both a flush and a COMMIT are required. To let your process see the bulk inserted rows without committing and without closing the SQL object or its cursor (that is, maintaining reuse for the SQL object), use FlushBulkInserts.

An example of using this function would be in preparation for a database commit where you do not wish to close the SQL insert statement, but need to ensure that all the rows you have inserted up to this point are in fact in the database and not in the buffer.

Another example would be when another SQL statement in the same PeopleSoft process needs to select rows that have been inserted using bulk insert and you do not wish to close the SQL insert statement. The SELECT cannot read rows in the bulk insert buffer, so you need to flush them to the table from which the SELECT is reading.

Parameters

None.

Returns

None. If the flush fails, the PeopleCode program terminates.

Example

```
&CM_DEPLETION_REC = CreateRecord(Record.CM_DEPFIFO_VW);

&CM_DEPLETE_REC = CreateRecord(Record.CM_DEPLETE);

&DEPLETE_FIFO_SEL = GetSQL(SQL.CM_DEPLETE_FIFO_SEL);

&ONHAND_FIFO_SEL = GetSQL(SQL.CM_ONHAND_FIFO_SEL);

DEPLETE_INS = GetSQL(SQL.CM_DEPLETE_INS);

&DEPLETE_INS.BulkMode = True;

&DEPLETE_FIFO_SEL.Execute(&CM_DEPLETION_REC, CM_COSTING_AET.BUSINESS_UNIT,
CM_COSTING_AET.CM_BOOK);

While &DEPLETE_FIFO_SEL.Fetch(&CM_DEPLETION_REC);

    /* Call functions that populate &CM_DEPLETE_REC.values */

    . . .

    &DEPLETE_INS.Execute(&CM_DEPLETE_REC);

    . . .

    If &CM_DEPLETION_REC.CM_COST_PROC_GROUP.Value = "BINTOBIN" Then

        /* Bin to Bin transfers are both a deplete and receipt, call
        functions to create the receipt */

        . . .

        /* Flush Bulk Insert to be able to see the current on hand quantities
        in CM_ONHAND_VW */
```

```

        FlushBulkInserts();

    End-if;

End-While;

. . .

```

Related Topics

SQL Class

For

Syntax

```

For count = expression1 To expression2
    [Step i];
    statement_list
End-for

```

Description

The **For** loop causes the statements of the *statement_list* to be repeated until *count* is equal to *expression2*. **Step** specifies the value by which *count* will be incremented each iteration of the loop. If you do not include **Step**, *count* will be incremented by 1 (or -1 if the start value is greater than the end value.) Any kind of statements are allowed in the loop, including other loops.

A **Break** statement inside the loop causes execution to continue with whatever follows the loop. If the **Break** occurs in a nested loop, the **Break** does not apply to the outside loop.

Example

The following example loops through all of the rows for the FIELDNAME scroll area:

```

&FIELD_CNT = ActiveRowCount(DBFIELD_VW.FIELDNAME);
For &I = 1 to &FIELD_CNT;
    WinMessage(MsgGetText(21000, 1, "Present Row Number is: %1", &I));
End-for;

```

FormatDateTime

Syntax

```

FormatDateTime(datetime, {timezone | "Local" | "Base"}, displayTZ);

```

Description

FormatDateTime takes a *datetime* value and converts it to text. If a specific time zone abbreviation, or a field reference, is passed in *timezone*, **FormatDateTime** adjusts the datetime to

the user's local time zone instead of the specified time zone. The system's base time zone is specified on the PSOPTIONS table. The value *datetime* is assumed to be in base time.



For more information about setting the base time, see PeopleTools Utilities.

If "**Local**" is specified, FormatDateTime adjusts the datetime to the user's local time zone instead of a specific time zone.

If "**Y**" is specified for *displayTZ* FormatDateTime appends the time zone abbreviation to the returned string.

Parameters

<i>datetime</i>	Specify the datetime value to be formatted.
<i>timezone</i> Local Base	Specify a value for converting <i>datetime</i> . The valid values are: <i>timezone</i> - a time zone abbreviation or a field reference to be used for converting <i>textdatetime</i> . Local - use the local time zone for converting <i>textdatetime</i> . Base - use the base time zone for converting <i>textdatetime</i> .
<i>displayTZ</i>	Specify whether the time zone abbreviation should be appended to the returned string. Valid values are "Y" or "N".

Returns

A formatted string value.

Example

The following example populates the &DISPDATE variable with a string containing the datetime value in the ORDER_DATE field adjusted to the user's local time zone, and with the time zone abbreviation.

```
&DISPDATE=FormatDateTime(ORDER_DATE, "Local", Y);
```

The following example populates the &DISPDATE variable with a string containing the datetime value in the SHIP_DATE field adjusted to the time zone stored in the SHIP_TZ field, and does not include the time zone abbreviation in the output.

```
&DISPDATE=FormatDateTime(SHIP_DATE, SHIP_TZ, "N");
```

Related Topics

ConvertDatetimeToBase, ConvertTimeToBase, DateTimeToLocalizedString, IsDaylightSavings, DateTimeToTimeZone, TimeToTimeZone, TimeZoneOffset

Function

Syntax

```

Function name [(paramlist)] [Returns data_type]
    [statements]
End-function

where

paramlist is      &param1 [As data_type] [, &param2
                  [As data_type]]...

data_type is      Number, String, Date, Time, Datetime,
                  Boolean, Object, or Any

statements is      a list of PeopleCode statements

```

Description

PeopleCode functions can be defined in any PeopleCode program. Function definitions must be placed at the top of the program, along with any variable and external function declarations.

Functions can be called from the program in which they are defined, in which case they don't need to be declared, and they can be called from another program, in which case they need to be declared at the top of the program where they are called.



Because a function can be called from anywhere, you cannot declare any variables within a function. You will get a design time error if you try.

By convention, external PeopleCode functions are stored in records whose names begin in FUNCLIB_, and they are always placed in the FieldFormula event (which is convenient because this event should no longer be used for anything else).



Functions can only be stored in the FieldFormula event for record fields, **not** for component record fields.

A function definition consists of:

- The keyword **Function** followed by the name of the function and an optional list of parameters. The name of the function can be up to 100 characters in length.

- An optional **Returns** clause specifying the data type of the value returned by the function.
- The statements to be executed when the function is called.
- The **End-function** keyword.

The parameter list, which must be enclosed in parentheses, is a comma-separated list of variable names, each prefixed with the & character. Each parameter is optionally followed by the keyword **As** and the name for one of the PeopleCode data types (Number, String, Date, DateTime, Time, Boolean, or Any). If you specify data types for parameters, then function calls will be checked to ensure that values passed to the function are of the appropriate type. If data types are not specified, then the parameters, like other temporary variables in PeopleCode, take on the type of the value that is passed to them.

PeopleCode parameters are always passed by reference. This means that if you pass the function a variable from the calling routine and change the value of the variable within the function, the value of the variable will be changed when the flow of execution returns to the calling routine.

If the function is to return a result to the caller, the optional **Returns** part must be included to specify the data type of the returned value. You have seven choices of value types: Number, String, Date, Time, DateTime, Boolean, or Any.

PeopleCode internal subroutines are part of the enclosing program and can access the same set of variables as the other statement-lists of the program, in addition to local variables created by the parameters.

Returning a Value

You can optionally return a value from a PeopleCode function. To do so, you must include a **Return** statement in the function definition, as described in the preceding section. For example, the following statement returns a Number value:

```
function calc_something(&parm1 as number, &parm2 as number) returns number
```

In the code section of your function, use the **Return** statement to return a value to the calling routine. When the **Return** statement executes, the function ends and the flow of execution goes back to the calling routine.

Example

This example returns a Boolean value based on the return value of a **SQLExec**:

```
function run_status_upd(&PROCESS_INSTANCE, &RUN_STATUS) returns boolean;
    &UPDATEOK = SQLExec("update PS_PRCS_RQST set run_status = :1 where
process_instance = :2", &RUN_STATUS, &PROCESS_INSTANCE);
    If &UPDATEOK Then
        Return True;
    Else
        Return False;
    End-if;
End-function;
```

Related Topics

Declare Function, Return

GenerateTree

Syntax

```
GenerateTree(&rowset [, TreeEventField])
```

Description

The GenerateTree function is used to display data in a tree format, with nodes and leaves. The result of the GenerateTree function is an HTML string, which can be displayed in an HTML area control. The tree generated by GenerateTree is called an *HTML tree*.

The GenerateTree function can be used in conjunction with the Tree Classes to display data from trees created using Tree Manager.

The GenerateTree function works with both an HTML area control and a hidden field. The *TreeEventField* parameter contains the contents of the invisible character field used to process the HTML tree events.

When an end-user selects a node, expands a node, collapses a node, or uses one of the navigation links, that event (end-user action) is passed to the invisible field, and the invisible field's FieldChange PeopleCode is executed.



For more information, see HTML Tree End-User Actions (Events).

Restrictions on use with Windows Client

The **GenerateTree** function is not available for Windows Client.

Parameters

<i>&rowset</i>	Specify the name of the rowset you've populated with tree data.
<i>TreeEventField</i>	Specify the contents of the invisible character field used to process the HTML tree events. The first time the GenerateTree function is used, that is, to generate the initial tree, you don't need to include this parameter. Subsequent calls require this parameter.

Returns

A string that contains HTML code that can be used with the HTML control to display a tree.

Example

In the following example, TREECTLEVENT is the name of the invisible control field that contains the event string that was passed from the browser.

```
HTMLAREA = GenerateTree(&TREECTL, TREECTLEVENT);
```



For a complete example, see PostBuild PeopleCode Example or FieldChange PeopleCode Example.

Related Topics

Using the GenerateTree Function

GetAERSection

Syntax

```
GetAERSection(ae_applid, ae_section [, effdt])
```

Description

Opens and associates an AERSection PeopleCode object with the **base** section, as specified. If no base section by the specified name is found, one is created. This allows you to create base sections as needed.



When you open or get an AERSection object, (that is, the base section) any existing steps in the section will be deleted.

After you've instantiated the AERSection object, set the **template** section using the **SetTemplate** method. You can copy steps *from* the template section *to* the base section before you start your Application Engine program. This is useful for applications that let users input their "rules" into a user-friendly page, then convert these rules, at save time, into Application Engine constructs.



For more information, see AERSection Class.

When an AERSection is opened (or accessed), the system first looks to see if it exists with the given input parameters. If such a section doesn't exist, the system looks for a similar section based on market, database platform, and effective date.



For more information, see How an AERSection is Accessed.

The AERSection Object is designed for use within online. Typically, dynamic sections should be constructed in response to an end user action.



Do not call an AERSection object from an Application Engine PeopleCode Action. If you need to access another section, use the CallSection action instead.

Parameters

<i>ae_applid</i>	Specifies the application ID of the section you want to modify.
<i>ae_section</i>	Specifies the section name of the base section you want to modify. If no base section by the specified name is found, one is created.
<i>effdt</i>	Specifies the effective date of the section you want to modify (optional).

Returns

An AERSection object is returned.

Example

See AERSection Example.

Related Topics

Open AERSection class method, AERSection Class

GetAttachment

Syntax

```
GetAttachment(URLSource, SysFileName, LocalFile [, LocalDirEnvVar])
```

where *URLSource* can have **one** of the following forms:

URL.URLname

OR a string URL, such as

```
ftp://user:password@ftp.ps.com/
```

Description

The **GetAttachment** function transfers a file from the file server to the application sever.



If you're using Windows Client, this function transfers the file to the client.

File Name Considerations

When the file is transferred using `AddAttachment` or `PutAttachment`, the following characters are replaced with an underscore:

- space
- semi-colon
- plus sign
- percent sign
- ampersand
- apostrophe
- exclamation point
- @ sign
- pound sign
- dollar sign

Parameters

URLSource

A reference to a URL. This can be either a URL name, in the form `URL.URLName`, or a string. This is where the file is transferred to.

SysFileName

The relative path and filename of the file on the file server. This is appended to *URLSource* to make up the full URL where the file is transferred from. This parameter takes a string value.



The *URLSource* requires "/" slashes. Because *SysFileName* is appended to the URL, it also requires only "/" slashes. "\" are NOT supported in anyway for either the *URLSource* or the *SysFileName* parameter.

LocalFile

The name, and possible full path, to the destination file on the application server. This parameter takes a string value.

LocalDirEnvVar

The name of the environment variable containing path information. If specified, the path value in the environment variable is prepended to *LocalFile*. Use this if you don't want to hard-code path names in the *LocalFile* parameter. This parameter takes a string value.

Returns

An integer value. You can check for either the integer or the constant:

Number	Constant	Description
0	%Attachment_Success	File was transferred successfully.
1	%Attachment_Failed	File was not successfully transferred.

Example

The following transfers the file "resume.txt" from the file server to the application server using ftp.

```
GetAttachment("ftp://anonymous:hobbit1@ftp.ps.com/HRarchive/",
"NewHire/11042000resume.txt", "c:/NewHires/resume.txt");
```

The following example, all resumes are in the directory NewHires, which is pointed to by the environment variable RESUMES.

```
&SysFileName = %DATE | &LASTNAME;

GetAttachment(URL.MYFTP, &SysFileName, "resume.txt", "RESUMES");
```

The following example gets employee photos from an ftp archive, and writes them to a record.

```
Local File &FILE;

Local Record &REC;

Local SQL &SQL;

&REC = CreateRecord(Record.EMPL_PHOTO);

&SQL = CreateSQL("%SelectAll(:1)", Record.EMPL_PHOTO);

While &SQL1.Fetch(&REC)

/* photos are stored in FTP archive by EmplID */
```

```

    &EmplId = &REC.EmplId.Value;

    &SysFileName = &EmplId | "_PHOTO.GIF";

    /* Store files with same name on local system */

    GetAttachment(URL.FTPARCHIVE, &SysFileName, &SysFileName, "PHOTOS");

    /* get the file */

    &FileName = "C:\HRArchive\Photos" | &SysFileName;

    &FILE = GetFile(&FileName, "w", "a", %FilePath_Absolute);

    /* Write file to record */

    &FILE.WriteRaw(&REC.EMPLOYEE_PHOTO.Value);

End-While;

&FILE.Close();

```

Related Topics

AddAttachment, DeleteAttachment, PutAttachment, ViewAttachment, Using the Attachment Functions

GetBiDoc

Syntax

```
GetBiDoc(XMLstring)
```

Description

The **GetBiDoc** function creates a BiDocs structure. You can populate the structure with data from *XMLstring*. This is part of the incoming Business Interlink functionality, which allows PeopleCode to receive an XML request and return an XML response.



For more information, see PeopleSoft Business Interlink Application Developer Guide.

Parameters

XMLstring

A string containing XML. You can specify a NULL string for this parameter, that is two quotation marks (""), without a space between them.

Return Value

A BiDocs object.

Example

The following example gets an XML request, puts it into a text string, and puts that into a BiDoc. Once this is done, the GetDoc method and the GetValue method can get the value of the skills XML element, which is contained within the postreq XML element in the XML request.

```
Local BiDocs &rootInDoc, &postreqDoc;

Local string &blob;

Local number &ret;

&blob = %Request.GetContentBody();

/* process the incoming xml(request)- Create a BiDoc and fill with the request*/

&rootInDoc = GetBiDoc(&blob);

&postreqDoc = &rootInDoc.GetDoc("postreq");

&ret = &postreqDoc.GetValue("skills", &skills);
```

You can also create an empty BiDoc with GetBiDoc, as in the following example.

```
Local BiDocs &rootInDoc, &postreqDoc;

Local string &blob;

Local number &ret;

&blob = %Request.GetContentBody();

/* process the incoming xml(request)- Create a BiDoc and fill with the request*/

&rootInDoc = GetBiDoc("");

&ret = &rootInDoc.ParseXMLString(&blob);

&postreqDoc = &rootInDoc.GetDoc("postreq");

&ret = &postreqDoc.GetValue("skills", &skills);
```

GetCalendarDate

Syntax

```
GetCalendarDate(comparedate, periods, periodadjustment,  
               outfieldname, company, paygroup)
```

Description

GetCalendarDate returns the value of a Date field from the PS_PAY_CALENDAR table. If a table entry is not found, **GetCalendarDate** returns 1899-01-01.

Processing Rules

1. The function SELECTs all the values for *outfieldname*, PAY_BEGIN_DT and PAY_END_DT from PS_PAY_CALENDAR. The result set is sorted in increasing PAY_END_DT order.
2. A SQL SELECT statement is generated in the following form:
3.

```
SELECT outfieldname, PAY_BEGIN_DT, PAY_END_DT  
FROM PS_PAY_CALENDAR  
WHERE COMPAny=:1 AND PAYGROUP=:2  
ORDER BY PAY_END_DT;
```
4. Rows are fetched from the result set until the value of *comparedate* falls between PAY_BEGIN_DT and PAY_END_DT. The value of *outfieldname* is stored in a storage stack.
5. A work variable equal to the value in *periods* is set.
6. If the value of *outfieldname* in the located result row is equal to *comparedate*, then the value in *periodadjustment* is added to the work variable. Because *periodadjustment* may be negative, the result may be negative.
7. If the work variable is negative then the saved value of *outfieldname* is returned from the storage stack at the level specified by the work variable. If the work variable is positive then fetch forward the number of times specified by the work variable. The value of *outfieldname* is returned from the most recently fetched (current) row.

Returns

Returns a Date value from the PS_PAY_CALENDAR table.

Parameters

<i>comparedate</i>	A date field set by the caller as the date of interest, for example, "1997-02-17."
<i>periods</i>	A numeric variable set by the caller specifying the number of periods forward or backward to be returned.

<i>periodadjustment</i>	A numeric variable that adjusts the <i>periods</i> if the <i>comparedate</i> equals the period end date. This is typically used to adjust for period end dates. Usually the <i>periodadjustment</i> is either -1, 0, or 1.
<i>outfieldname</i>	The name of a date field in the PS_PAY_CALENDAR table. For example PAY_BEGIN_DT. The value of this field is not referenced or modified by the routine, but the name of the field is used to build a SQL SELECT statement and to indicate which value from the table to return in the return date.
<i>company</i>	A field set by the caller to be equal to the company code of interest, for example, "CCB".
<i>paygroup</i>	A variable set by the caller to be equal to the PayGroup code of interest, for example, "M01".

Example

The following examples use the sample PS_PAY_CALENDAR entries in the table below. In the example, *comparedate* and the result date are Date type fields defined in some record.

COMPANY	PAYGROUP	PAY_END_DT	PAY_BEGIN_DT	CHECK_DT
CCB	MO1	1997-01-31	1997-01-01	1997-01-31
CCB	MO1	1997-02-28	1997-02-01	1997-02-28
CCB	MO1	1997-03-31	1997-03-01	1997-03-29
CCB	MO1	1997-04-30	1997-04-01	1997-04-30
CCB	MO1	1997-05-31	1997-05-01	1997-05-31
CCB	MO1	1997-06-30	1997-06-01	1997-06-28
CCB	MO1	1997-07-31	1997-07-01	1997-07-31
CCB	MO1	1997-08-31	1997-08-01	1997-08-30
CCB	MO1	1997-09-30	1997-09-01	1997-09-30
CCB	MO1	1997-10-31	1997-10-01	1997-10-31
CCB	MO1	1997-11-30	1997-11-01	1997-11-27
CCB	MO1	1997-12-31	1997-12-01	1997-12-31
CCB	SM1	1997-01-15	1997-01-01	1997-01-15

Find the begin date of the pay period containing the date 1997-05-11 (the value of &COMPAREDate). The result date returned would be 1997-05-01.

```
&RESULT_Date = GetCalendarDate(&COMPAREDate, 0, 0,
    PAY_BEGIN_DT, COMPAny, PAYGROUP);
```

Or:

```
&RESULT_Date = GetCalendarDate(&COMPAREDate, 1, -1,
    PAY_BEGIN_DT, COMPAny, PAYGROUP);
```

GetControl

Syntax

```
GetControl([pagename, controlname [, occurs])
```

Description

Use the **GetControl** function to access an ActiveX control at runtime. The **GetControl** function returns a reference to the ActiveX object, so it can be used directly with the properties and methods associated with that control. That is, following the **GetControl** function call, you can use dot notation to access the properties and methods of that control.



The **GetControl** function returns a reference to an ActiveX control. You can't access any controls until after the Component Processor has loaded the page. You shouldn't use this function in an event prior to the Activate Event.

You can only use this function with no parameters when your code is an ActiveX event associated with the control you want to manipulate, that is, if your code is in **PSControlInit**, **PSLostFocus**, or an event specific for that ActiveX control. If your PeopleCode program is in any other event, like **RowInit** or **FieldChange**, you *must* specify *pagename* and *controlname*.

If you've placed your ActiveX control in a scroll that has an occurs count greater than one, you can determine which occurs number an ActiveX control has by using the **GetControlOccurrence** function.

After you place an ActiveX control on a page, you must go into the properties and give the ActiveX control a name before you can save the page. Choose a unique name, like the name of your company combined with the task of the ActiveX control, like **FORD_HIST_CHART** or **MORGAN_EMPL_DATA_TREE**. The name *must* be unique, otherwise the control and its associated PeopleCode may be lost during upgrade (if PeopleSoft places an ActiveX control on a page and gives it the exact same name as the one your company has created.)

Restrictions on Use in Three-Tier Mode and PeopleSoft Internet Architecture

GetControl is a client-only function, which limits its use in three-tier mode and in PeopleSoft Internet Architecture applications.

- In three-tier mode **GetControl** can be used only in processing groups set to run on the client. If the **GetControl** function is called in a processing group running on the application server, a runtime error will occur.
- In PeopleSoft Internet Architecture applications, where all PeopleCode runs on the application server, **GetControl** cannot be used at all.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```

If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;

```

Parameters

<i>pagename</i>	Specify the name of the page that the ActiveX control is on. The preferred form is <code>PAGE.pagename</code> . You can also use the system variable <code>%page</code> .
<i>controlname</i>	Specify the name of the ActiveX control you want to change.
<i>occurs</i>	If you have more than one occurrence of a control (that is., the control is located in a scroll bar that has its occurs count set higher than 1), you can specify which occurrence of the control you want to change. The default value for <i>occurs</i> is 1.

Returns

A reference to an ActiveX control (object).

Example

The following loops through every column (series) and row of a chart.

```

&CHART = GetControl();

For &I = 1 To &CHART.ColumnCount

    &CHART.column = &I;

    For &J = 1 to &CHART.RowCount

```

```

&CHART.row = &J;

/* do processing */

End-For;

End-For;

```

This code is in the DURATION_DAYS record field FieldChange event. It loads the new value into the chart:

```

&ROW = CurrentRowNumber();

&CHART = GetControl(PAGE.ABSENCE_HISTORY, "ACTIVEX1");

&CHART.Row = &ROW;

&CHART.RowLabel = BEGIN_DT;

&CHART.Data = DURATION_DAYS;

```

Related Topics

GetControlOccurrence, Implementing ActiveX Controls, ActiveX Controls in PeopleTools

GetControlOccurrence

Syntax

```
GetControlOccurrence()
```

Description

The **GetControlOccurrence** function returns the occurrence number of the current ActiveX control. This function can only be used with the events associated with an ActiveX control, such as PSControlInit, OnClick, and so on. You would use this function when your control is in a scroll bar that has an occurs count greater than one.



If you place an ActiveX control in a scroll that has an occurs count greater than one, the PSControlInit event will fire once for every occurrence. That is, if the scroll has an occurs count of 3, PSControlInit will fire 3 times.

Restrictions on Use in Three-Tier Mode and PeopleSoft Internet Architecture

GetControlOccurrence is a client-only function, which limits its use in three-tier mode and in PeopleSoft Internet Architecture applications.

- In three-tier mode **GetControlOccurrence** can be used only in processing groups set to run on the client. If the **GetControlOccurrence** function is called in a processing group running on the application server, a runtime error will occur.

- In PeopleSoft Internet Architecture applications, where all PeopleCode runs on the application server, **GetControlOccurrence** cannot be used at all.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```

If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;

```

Parameters

None.

Returns

The occurrence number of the current ActiveX control. If the occurs count for the scroll is 1, this function will return 1.

Example

In the following example, an ActiveX control has been placed in a scroll that has an occurs count of 2. The control will be initialized with the exact same data, but will display it to the user in two different forms: first as a bar chart, second, as a three-dimensional line chart.

```

Function PSControlInit()

    &CHART = GetControl();

    &NUM = GetControlOccurrence();

    If &NUM = 1 Then

```

```
&CHART.ChartType = 1;

Else

    &CHART.ChartType = 2;

End-if;

/* fill chart with data */

End-Function;
```

Related Topics

GetControl, Implementing ActiveX Controls, ActiveX Controls in PeopleTools

GetCwd

Syntax

```
GetCwd( )
```

Description

GetCwd is used to determine the current working directory on the client.

Restrictions on use in Three-Tier and PeopleSoft Internet Architecture

GetCwd is a client-only function, which limits its use in three-tier mode and in PeopleSoft Internet Architecture applications.

- In three-tier mode **GetCwd** can be used only in processing groups set to run on the client. If the **GetCwd** function is called in a processing group running on the application server, a runtime error will occur.
- In PeopleSoft Internet Architecture applications, where all PeopleCode runs on the application server, **GetCwd** cannot be used at all.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

Returns

Returns a string containing the path of the current working directory.

Example

The example stores a string specifying the current working directory in &CWD.

```
&CWD = GetCwd();
```

Related Topics

ChDir, ChDrive, GetEnv, ExpandEnvVar

GetEnv

Syntax

```
GetEnv(env_var_str)
```

Description

This function returns a DOS environment variable specified by *env_var_str* as a string. If the environment variable does not exist, a null string is returned.

Restrictions on use in Three-Tier Mode and PeopleSoft Internet Architecture

Because it is specific to systems running MS-DOS or Windows, **GetEnv** is a client-only function, which limits its use in three-tier mode and in PeopleSoft Internet Architecture applications.

- In three-tier mode **GetEnv** can be used only in processing groups set to run on the client. If the **GetEnv** function is called in a processing group running on the application server, a runtime error will occur.
- In PeopleSoft Internet Architecture applications, where all PeopleCode runs on the application server, **GetEnv** cannot be used at all.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

Example

Assume that the DOS environment string NETDRIVE is equal to "N:". The following statement will return "N:" in &NETDRIVE:

```
&NETDRIVE = GetEnv("netdrive");
```

Related Topics

GetCwd, ExpandEnvVar

GetField

Syntax

```
GetField([recname.fieldname])
```


Description

GetField creates a reference to a field object for the current context; that is, from the row containing the currently executing program.

If you don't specify *recname.fieldname* the current field executing the PeopleCode is returned.



For PeopleCode programs located in events that are not associated with a specific row, record and field at the point of execution this function is invalid. That is, you cannot use this function in PeopleCode programs on events associated with high-level objects like ActiveX controls, pages or components. For events associated with record level programs (like component record), this function is valid, but it must be specified with a field name, as there is an assumed record, but no assumed field name.

When GetField is used with an associated record and field name on component buffer data, the following is assumed:

```
&FIELD = GetRow().recname.fieldname;
```

Returns

This function returns a field object that references the field from the specified record.

Parameters

<i>recname.fieldname</i>	If you don't want to refer to the field executing the PeopleCode, specify a record name and field name.
--------------------------	---

Example

```
Local Field &CHARACTER;

&CHARID = GetField(FIELD.CHAR_ID);
```

Related Topics

Field Class, GetField record class method, Data Buffer Access

GetFile

Syntax

```
GetFile(filename, mode [, charset] [, pathtype])
```

Description

The **GetFile** function instantiates a new file object from the File class, associates it with an external file, and opens the file so you can use File class methods to read from or write to it.

GetFile is similar to the Open file class method, but Open associates an *existing* file object with an external file. If you plan to access only one file at a time, you only need one file object. Use GetFile to instantiate a file object for the first external file you access, then use Open to associate the same file object with as many different external files as you want. You'll need to instantiate multiple file objects with **GetFile** only if you expect to have multiple files open at the same time.

Parameters

<i>filespec</i>	Specify the name, and optionally, the path, of the file you want to open.
<i>mode</i>	A string indicating the manner in which you want to access the file. The mode can be one of the following: <p>"R" Read mode—opens the file for reading, starting at the beginning.</p> <p>"W" Write mode—opens the file for writing.</p>



When you specify Write mode, any existing content in the file is discarded.

"A" Append mode—opens the file for writing, starting at the end. Any existing content is retained.

"U" Update mode—opens the file for reading or writing, starting at the end. Any existing content is retained. Use this mode and the GetPosition and SetPosition file class methods to maintain checkpoints of the current read/write position in the file.



In Update mode, any write operation will clear the file of all data that follows the position you set.



Currently, the effect of the Update mode and the GetPosition and SetPosition methods is not well defined for Unicode files. Use the Update mode only on files stored with the ANSI character set.



For more information about using Update mode, see Recovering from File Access Interruptions.

"E" Conditional "exist" read mode—opens the file for reading only if it exists, starting at the beginning. If it doesn't exist, **Open** has no effect. Before attempting to read from the file, use the `IsOpen` property to confirm that it's open.

"N" Conditional "new" write mode—opens the file for writing, only if it doesn't already exist. If a file by the same name already exists, **Open** has no effect. Before attempting to write to the file, use the `IsOpen` property to confirm that it's open.

You can insert an asterisk (`*`) in the filename to ensure that a new file is created. The system replaces the asterisk with numbers starting at **1** and incrementing by 1, and checks for the existence of a file by each resulting name in turn. It uses the first name for which a file *doesn't* exist. In this way you can generate a set of automatically numbered files. If you insert more than one asterisk, all but the first one are discarded.

charset

A string indicating the character set you expect when you read the file, or the character set you want to use when you write to the file. You can abbreviate ANSI to "A" and Unicode to "U". All other character sets must be spelled out in full, for example, ASCII, THAI, or UTF8. If you don't specify a character set, the default value is ANSI. You can also use a record field value to specify the character set (for example, `RECORD.CHARSET`.)



If you attempt to read data from a file using a different character set than was used to write that data to the file, the methods used will generate a runtime error or the data returned will be unusable.

pathtype

If you have prepended a path to the file name, use this parameter to specify whether the path is an absolute or relative path. The valid values for this parameter are:

`%FilePath_Relative` (default)

`%FilePath_Absoulte`

If you don't specify *path*type the default is %FilePath-
_Relative.

If you don't specify a path, the file is assumed to be in one of two locations, depending on where your PeopleCode program is executing.

- When your program is executing on the client, the location is the directory specified by the **TEMP** environment variable.

Note. Your system security should verify if a user has the correct permissions before allowing access to a drive (for example, if a user changes their **TEMP** environmental variable to a network drive they don't normally have access to, your system security should detect this.)

- When your program is executing on the server, the location is the "**files**" directory under the directory specified by the PeopleSoft **PS_SERVDIR** environment variable.

If you specify a relative path, that path will be appended to the TEMP or PS_SERVDIR variable (depending on where your PeopleCode is executing.) You must make sure not to include a drive letter when using a relative path.

If the path is an absolute path, whatever path you specify is used verbatim. You must specify a drive letter as well as the complete path. You can't use any wildcards when specifying a path.

The Component Processor automatically converts platform-specific separator characters to the appropriate form for where your PeopleCode program is executing. On a WIN32 system, UNIX "/" separators are converted to "\", and on a UNIX system, WIN32 "\" separators are converted to "/".



The syntax of the file path does *not* depend on the file system of the platform where the file is actually stored; it depends only on the platform where your PeopleCode is executing.



For more information about setting a program's processing location, see PeopleCode and PeopleSoft Internet Architecture.

Returns

A file object.

Example

The following example opens an existing Unicode file for reading:

```
&MYFILE = GetFile(&SOMENAME, "E", "U");

If &MYFILE.IsOpen Then

    while &MYFILE.ReadLine(&SOMESTRING);

        /* Process the contents of each &SOMESTRING */

    End-While;

    &MYFILE.Close();

End-If;
```

The following example opens a numbered file for writing in ANSI format, without overwriting any existing files:

```
&MYFILE = GetFile("c:\temp\item*.txt", "N", %FilePath_Absolute);

If &MYFILE.IsOpen Then

    &MYFILE.WriteLine("Some text.");

    &MYFILE.Close();

End-If;
```

The following example uses the CHARSET field to indicate the character set to be used:

```
&MYFILE = GetFile("INPUT.DAT", "R", RECORD.CHARSET);
```

Related Topics

FileExists, FindFiles, GetFile, and File Class

GetGrid

Syntax

```
GetGrid(PAGE.pagename, gridname [, occursnumber])
```

Description

The **GetGrid** function instantiates a grid object from the Grid class, and populates it with the grid specified by *gridname*, which is the Page Field Name on the General tab of that grid's page field properties.

Specify a grid name consisting of any combination of upper case letters, digits and "#", "\$", "@", and "_".



PeopleSoft builds a page grid one row at a time. Because the Grid class applies to a complete grid, you can't attach PeopleCode that uses the Grid class to events that occur before the grid is built; the earliest event you can use is the page Activate Event.



For more information, see File Class.

Using the Occurs Count

The occurs number (*occursnumber*) parameter that you specify with the GetGrid function does **not** refer to the row number of the grid. Rather, it's the occurrence of the grid within a scroll. If your grid is contained within a scroll, and the occurs number of the scroll is greater than 1, then to specify the correct grid within the scroll you must use the *occursnumber* parameter.

Using the Grid Name

When you place a grid on a page, the grid is automatically named the same as the name of the primary record of the scroll for the grid (in the Page Field Name.)

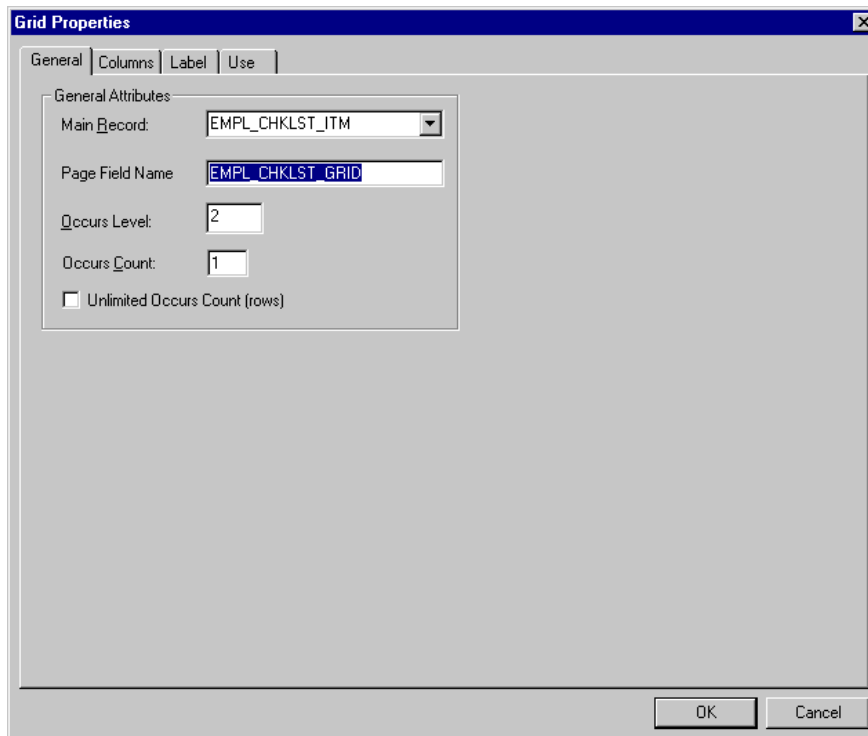


If the name of the record changes, the Page Field Name is *not* automatically updated. You will have to go in and change this name if you want the name of the grid to reflect the name of the record.

This is the name you use with the GetGrid function. You can change this name on the Record tab of the Grid properties.

To change a grid name:

1. Open the page in Application Designer, select the grid and access the page field properties.
2. On the General tab, type the new grid name in Page Field Name.



Changing a Grid Name



Every grid on a page must have a unique name.

Parameters

PAGE, *pagename*

Specify the name of the page definition containing the grid you want.

gridname

Specify the Page Field Name on the General tab of the grid's page field properties.

occursnumber

Optionally, specify the occurs number to identify the grid. Since you can use occurs count to generate multiple instances of a grid within a scroll just as you can with other page fields, *occursnumber* provides a way to identify which one of those grids you want. The default value is **1**.

Returns

A Grid object populated with the requested grid.

Example

This example retrieves the second grid named "EMPL_GRID" within a scroll:

```
local Grid &MYGRID;
```

```
&MYGRID = GetGrid(PAGE.EMPLOYEE_CHECKLIST, "EMPL_GRID", 2);
```

Related Topics

GetColumn grid class method

GetHTMLText

Syntax

```
GetHTMLText (HTML.textname [, paramlist]
```

Where *paramlist* is an arbitrary-length list of values of undetermined (Any) data type in the form:

```
inval1 [, inval2] ...
```

Description

The **GetHTMLText** function retrieves a pre-defined HTML text from an HTML definition. If any values are included in *paramlist*, they are substituted into the HTML text based on positional reference (for example, %BIND(:1) is the first parameter, %BIND(:2) is the second, and so on.)



For more information on creating an HTML definition, see HTML Area Control.



Use the GetHTMLText function only to retrieve HTML, or HTML that contains a JavaScript program, from an HTML definition. If you have an HTML definition that *only* contains JavaScript, use the GetJavaScriptURL Response object method to access it.

Restrictions on Use

This function is supposed to be used with the PeopleSoft Internet Architecture. If run from a 2-tier environment, the parameter substitution does *not* take place.

Parameters

HTML.textname	Specify the name of an existing HTML text from an HTML definition.
----------------------	--

Returns

The resulting HTML text is returned as a string.

Example

The following is the text in the HTML definition TEST_HTML:

```
This is a %BIND(:1) and %BIND(:2) test.
```

The following is the PeopleCode program:

```
Local Field &HTMLfield;

&string = GetHTMLText(HTML.TEST_HTML, "good", "simple");

&HTMLfield = GetRecord(Record.CHART_DATA).HTMLAREA;

&HTMLfield.Value = &string;
```

The output from &string (displayed in an HTML area control) looks like this:

```
This is a good and simple test.
```

Related Topics

Internet Script Classes, Using HTML Definitions and the GetHTMLText Function

GetImageExtents

Syntax

```
GetImageExtents (IMAGE.ImageName)
```

Description

The **GetImageExtents** function returns the width and height of the image specified with *ImageName*.

Parameters

<i>ImageName</i>	Specify the name of the image on the page. This image must already be existing on the page.
------------------	---

Returns

An array of data type number, where element 1 is the image height and element 2 is the image width.

Example

```
Local array of number &ImageExtents;
```

```
&ImageExtents = GetImageExtents(Image.PT_TREE_EXPANDED);
```

```
WinMessage("Height is " | &ImageExtents[1] | " and width is " |  
&ImageExtents[2]);
```

Related Topics

Image Field

GetInterlink

Syntax

```
GetInterlink(Interlink.name)
```

Description

The **GetInterlink** function instantiates a Business Interlink definition object based on a Business Interlink definition created in Application Designer. The Business Interlink object can provide a gateway for PeopleSoft applications to the services of any external system.



For more information, see the PeopleSoft Business Interlink Application Developer Guide.

After you use this function, you may want to refresh your page. The **Refresh** method, on a rowset object, reloads the rowset (scroll) using the current page keys. This causes the page to be redrawn. `GetLevel0().Refresh()` refreshes the entire page. If you only want a particular scroll to be redrawn, you can refresh just that part.



For more information, see Refresh rowset class method.

Generally, you will not use the **GetInterlink** function in a program you create from scratch. If you drag a Business Interlink definition from the project workspace (in Application Designer) to an open PeopleCode editor window, a "template" is created, with values filled in based on the Business Interlink definition you dragged in.

The following is the template created from dragging the Business Interlink definition LDAP_SEARCHBIND to an open PeopleCode editor window.

```
/* ==>
```

```

    This is a dynamically generated PeopleCode template to be used only as a
    helper to the application developer. You need to replace all references to
    '<*>' OR default values with references to PeopleCode variables and/or a
    Rec.Fields.*/
```

```

/* ==> Declare and instantiate: */

Local Interlink &LDAP_SEARCHBI_1;

Local BIDocs &inDoc;

Local BIDocs &outDoc;

Local boolean &RSLT;

Local number &EXECSRSLT;

&LDAP_SEARCHBI_1 = GetInterlink(INTERLINK.LDAP_SEARCHBIND);


/* ==> You can use the following assignments to set the configuration
parameters.

*/

&LDAP_SEARCHBI_1.Server = "jtsay111198.peoplesoft.com";

&LDAP_SEARCHBI_1.Port = 389;

&LDAP_SEARCHBI_1.User_DN = "cn=Admin,o=PeopleSoft";

&LDAP_SEARCHBI_1.Password = &password;

&LDAP_SEARCHBI_1.UserID_Attribute_Name = "uid";

&LDAP_SEARCHBI_1.URL = "file://psio_dir.dll";

&LDAP_SEARCHBI_1.BIDocValidating = "Off";


/* ==> You might want to call the following statement in a loop if there is
more than one row of data to be added. */


/* ==> Add inputs: */

&inDoc = &LDAP_SEARCHBI_1.GetInputDocs("");

&ret = &inDoc.AddValue("User_ID", <*>);

&ret = &inDoc.AddValue("User_Password", <*>);

&ret = &inDoc.AddValue("Connect_DN", <*>);

```

```

&ret = &inDoc.AddValue("Connect_Password", <*>);

&Directory_Search_ParmsDoc = &inDoc.AddDoc("Directory_Search_Parms");

&ret = &Directory_Search_ParmsDoc.AddValue("Host", <*>);

&ret = &Directory_Search_ParmsDoc.AddValue("Port", <*>);

&ret = &Directory_Search_ParmsDoc.AddValue("Base", <*>);

&ret = &Directory_Search_ParmsDoc.AddValue("Scope", <*>);

&ret = &Directory_Search_ParmsDoc.AddValue("Filter", <*>);


/* ===> The following statement executes this instance: */

&EXECSRSLT = &LDAP_SEARCHBI_1.Execute();

If ( &EXECSRSLT <> 1 ) Then

    /* The instance failed to execute */

Else

    &outDoc = &LDAP_SEARCHBI_1.GetOutputDocs("");

    &ret = &outDoc.GetValue("Distinguished_Name", <*>);

    &ret = &outDoc.GetValue("return_status", <*>);

    &ret = &outDoc.GetValue("return_status_msg", <*>);

End-If; /* If NOT &RSLT ... */

```



For more information, see [Generating the PeopleCode Template](#).

Parameters

Interlink. <i>name</i>	Specify the name of the Business Interlink definition from which to instantiate a Business Interlink object.
-------------------------------	--

Returns

A Business Interlink object.

Example

The following example instantiates a Business Interlink object based on the Business Interlink definition QE_RP_SRAALL.

```
Local Interlink &SRA_ALL_1;  
  
&SRA_ALL_1 = GetInterlink(Interlink.QE_RP_SRAALL);
```

Related Topics

Business Interlink Class, PeopleSoft Business Interlink Application Developer Guide

GetJavaClass

Syntax

```
GetJavaClass(ClassName)
```

Description

The **GetJavaClass** function finds a Java class you can manipulate in PeopleCode. This is used for those classes that have static members, where it isn't appropriate to instantiate an object of the class. You can only call static methods, that is, class methods, with the object created with this function.

In Java, you access such static members of a class by using the class name:

```
result = java.class.name.SomeStaticMethod();
```

To do this in PeopleCode, do the following:

```
&Result = GetJavaClass("java.class.name").SomeStaticMethod();
```



Note. If you create a class that you want to call using GetJavaClass, it must be located in a directory specified in the PS_CLASSPATH environment variable.



For more information see Internet Script Classes.

Parameters

<i>ClassName</i>	Specify the name of an already existing class. This parameter takes a string value.
------------------	---

Returns

A `JavaObject` that refers to the named Java class.

Example

The Java class `java.lang.reflect.Array` has no public constructors and only has static methods. The methods are used to manipulate Java array objects. One of these static methods is `GetInt`:

```
public static int getInt(Object array, int index)
```

To use this method, get the class by using `GetJavaClass`. This code illustrates accessing the value of the fifth element of an integer array.

```
Local JavaObject &RefArray, &MyArray;

. . .

&RefArray = GetJavaClass("java.lang.reflect.Array");

. . .

&MyArray = CreateJavaArray("int []", 24);

. . .

&FifthElement = &RefArray.getInt(&MyArray, 4);
```

Related Topics

`CreateJavaArray`, `CreateJavaObject`, Internet Script Classes

GetLevel0

Syntax

```
GetLevel0()
```

Description

GetLevel0 creates a rowset object that corresponds to level 0 of the component buffer. If used from PeopleCode that isn't associated with a page, it returns the base rowset from the current context.

Parameters

GetLevel0 has no parameters. However, it does have a default method, `GetRow`, and a shortcut. Specifying `GetLevel0()(1)` is the equivalent of specifying `GetLevel0().GetRow(1)`.

Returns

This function returns a rowset object that references the base rowset. For a component, this is the level 0 of the page. For an Application Engine program, this will be the state record rowset. For an application message, this will be the base rowset.



You can also get the base rowset for an application message using the message object `GetRowset` method, that is, `&MSG.GetRowset()`.

Example

The following code sample returns the level one rowset.

```
Local Rowset &ROWSET;

&ROWSET = GetLevel0().GetRow(1).GetRowset(SCROLL.LEVEL1_REC);
```

The following is equivalent to the above.

```
Local Rowset &ROWSET;

&ROWSET = GetLevel0()(1).GetRowset(SCROLL.LEVEL1_REC);
```

To reference a level 2 rowset you would have code similar to this:

```
Local Rowset &ROWSET_LEVEL2, &ROWSET_LEVEL0, &ROWSET_LEVEL1;

&ROWSET_LEVEL2 = GetLevel0().GetRow(1).GetRowset(SCROLL.LEVEL1_REC).GetRow(5).
GetRowset(RECORD.LEVEL2_REC);

/* or */

&ROWSET_LEVEL0 = GetLevel0();

&ROWSET_LEVEL1 = &ROWSET_LEVEL0.GetRow(1).GetRowset(SCROLL.LEVEL1_REC);

&ROWSET_LEVEL2 = &ROWSET_LEVEL1.GetRow(5).GetRowset(SCROLL.LEVEL2_REC);

/* or */

&ROWSET_LEVEL2 = GetLevel0()(1).LEVEL1_REC(5).GetRowset(SCROLL.LEVEL2_REC);
```

Related Topics

[GetRowset](#), [Rowset Class](#), and [Data Buffer Access](#)

GetMethodNames

Syntax

`GetMethodNames (Type, Name)`

Description

The **GetMethodNames** function returns either the method names for a Component Interface, or the function names of a WEBLIB record.

Parameters

<i>Type</i>	Specify the type of methods or functions you want returned. This parameter takes a string value. The valid values are: <ul style="list-style-type: none">• WebLib• CompIntfc
<i>Name</i>	Specify the name of the Component Interface or WEBLIB record that you want to know the methods or functions for.

Returns

An array of string containing the method or function names.

Example

```
Local array of string &Array;  
  
&Array = GetMethodNames("CompIntfc", CompIntfc.USER_PROFILE);  
  
&Array = GetMethodNames("WebLib", Record.WEBLIB_PORTAL);
```

Related Topics

Component Interface Classes, Creating Web Libraries

GetMessage

Syntax

`GetMessage ()`

Description

The **GetMessage** function is used in any of the application messaging PeopleCode events. It retrieves a message from the application message queue for the current message being processed.

It creates and loads a data tree for the default message version, and returns NULL if not successful.

Parameters

None.

Returns

A reference to a message object if successful, NULL if not successful.

Example

```
Local message &MSG;

&MSG = GetMessage();
```

Related Topics

[CreateMessage](#), [GetMessageInstance](#), [GetPubContractInstance](#), [GetSubContractInstance](#), and [Internet Script Classes](#)

GetMessageInstance

Syntax

```
GetMessageInstance(pub_id, pub_nodename, channelname)
```

Description

The **GetMessageInstance** function gets a message from the application message queue. It creates and loads a data tree for the default message version, and returns NULL if not successful.



This function should *not* be used in standard application message processing. It should only be used when correcting or debugging a publication or subscription contract that is in error.

Parameters

<i>pub_id</i>	Specifies the PubID of the message.
<i>pub_nodename</i>	Specifies the PubNodeName of the message.
<i>channelname</i>	Specifies the name of the message.

Returns

A reference to a message object if successful, NULL if not successful.

Example

```
Local message &MSG;
```

```
&MSG = GetMessageInstance(&PUBID, &PUBNODE, &CHANNEL);
```

Related Topics

[CreateMessage](#), [GetMessage](#), [GetPubContractInstance](#), [GetSubContractInstance](#), and Internet Script Classes

GetNextNumber

Syntax

```
GetNextNumber ({record.field | record_name, field_name}, max_number)
```

Description

The **GetNextNumber** function increments the value in a record for the field you specify by one and returns that value. You might use this function to increment an employee ID field by one when you are adding a new employee. If the new value generated exceeds *max_number*, a negative value is returned and the field value isn't incremented.

Parameters

record.field

Specify the record and field identifiers for the field for which you want the new number. This is the recommended way to identify the field.

record_name

Specify as a string the name of the record containing the field for which you want the new number. This parameter with *field_name* was used prior to PeopleTools 8.

field_name

Specify as a string the name of the field for which you want the new number. This parameter with *record_name* was used prior to PeopleTools 8.

Note. If you use the older syntax (*record_name*, *field_name*), you have to manually update these two parameters in your programs whenever that record or field is renamed. The new syntax (*record.field*) is automatically updated, so you won't have to maintain it.

max_number

Specify the highest allowed value for the field you're incrementing.

Returns

A Number value equal to the highest value of the field specified plus one.

GetNextNumber returns an error if the value to be returned would be greater than *max_number*. The function returns one of the following:

<i>Return</i>	<i>Constant</i>	<i>Description</i>
Number		The new number
-1	%GetNextNumber_SQLFailure	SQL failure
-2	%GetNextNumber_TooBig	Number too big, beyond <i>max_number</i>
-3	%GetNextNumber_NotFound	No number found, invalid data format

Example

```
If %PanelGroup = "RUN_AR33000" Then
    DUN_ID_NUM = GetNextNumber(INSTALLATION_AR.DUN_ID_NUM, 99999999);
End-if;
```

The following uses the constant to check for the value returned:

```
&VALUE = GetNextNumber(INSTALLATION_AR.DUN_ID_NUM, 999);

Evaluate &VALUE

When = %GetNextNumber_SQLFailure
    /* do processing */

When = %GetNextNumber_TooBig
    /* do processing */

When = %GetNextNumber_NotFound
    /* Do processing */

When-other
    /* do other processing */

End-Evaluate;
```

Related Topics

GetNextNumberWithGaps

GetNextNumberWithGaps

Syntax

```
GetNextNumberWithGaps(record.field, max_number, increment [, WHERE_Clause,  
paramlist])
```

Where *paramlist* is an arbitrary-length list of values in the form:

```
var1 [, var2] ...
```

Description

The **GetNextNumberWithGaps** function determines the highest value in a table for the field you specify, and returns that value plus *increment*. This function also allows you to specify a SQL WHERE clause as part of the function for maintaining multiple sequence numbers in a single record.



GetNextNumberWithGaps also issues a COMMIT after incrementing the sequence number if no other database updates have occurred since the last COMMIT. This limits the time a database lock is held on the row and so may improve performance.

Parameters

<i>record.field</i>	Specify the record and field identifiers for the field for which you want the new number. This is the recommended way to identify the field.
<i>max_number</i>	Specify the highest allowed value for the field you're incrementing.
<i>increment</i>	Specify the value you want the numbers incremented by.
<i>WHERE_Clause</i>	Specify a WHERE clause for maintaining multiple sequence numbers.
<i>paramlist</i>	Parameters for the WHERE clause.

Returns

A Number value equal to the highest value of the field specified plus one.

GetNextNumberWithGaps returns an error if the value to be returned would be greater than *max_number*. The function returns one of the following:

Return	Constant	Description
Number		The new number
-1	%GetNextNumber_SQLFailure	SQL failure

Return	Constant	Description
-2	%GetNextNumber_TooBig	Number too big, beyond <i>max_number</i>
-3	%GetNextNumber_NotFound	No number found, invalid data format

Example

The following PeopleCode

```
&greg = GetNextNumberWithGaps(GREG.DURATION_DAYS, 999999, 50, "where emplid = :1", 8001);
```

results in the following:

```
2-942   21.53.09   0.000 Cur#4.PTTST81B RC=0 Dur=0.000 Connect=PTTST81B/sa/
2-943   21.53.09   0.000 Cur#4.PTTST81B RC=0 Dur=0.000 COM Stmt=UPDATE PS_GREG
SET DURATION_DAYS = DURATION_DAYS + 50 where emplid = 8001
2-944   21.53.09   0.000 Cur#4.PTTST81B RC=0 Dur=0.000 EXE
2-945   21.53.09   0.000 Cur#4.PTTST81B RC=0 Dur=0.000 COM Stmt=SELECT
DURATION_DAYS FROM PS_GREG where emplid = 8001
2-946   21.53.09   0.000 Cur#4.PTTST81B RC=0 Dur=0.000 EXE
2-947   21.53.09   0.000 Cur#4.PTTST81B RC=0 Dur=0.000 Fetch
2-948   21.53.09   0.010 Cur#4.PTTST81B RC=0 Dur=0.010 Commit
2-949   21.53.09   0.010 Cur#4.PTTST81B RC=0 Dur=0.010 Disconnect
```

Related Topics

GetMethodNames

GetPage

Syntax

```
GetPage (PAGE.pagename)
```

Description

The **GetPage** function returns a reference to a page object. Generally, page objects are used to hide or unhide pages in a component.

Generally, the PeopleCode used to manipulate a page object would be associated with PeopleCode in the Activate event.



The page object shouldn't be used until after the Component Processor has loaded the page: that is, don't instantiate this object in RowInit PeopleCode, use it in PostBuild or Activate instead.



For more information, see Page Class.

Note

An expression of the form

PAGE.name.property

is equivalent to **GetPage**(*name*).*property*.

Parameters

PAGE,*pagename*

The name of the page for which you want to create an object reference. Must be a page in the current context.



For more information, see Understanding Current Context.

Returns

A page object that references the page.

Example

In the following example, a page is hidden based on the value of the current field.

```
If PAYROLE_TYPE = "Global" Then  
    GetPage(PAGE.JOB_EARNINGS).Visible = False;  
End-If;
```

Related Topics

Page Class

GetPubContractInstance

Syntax

```
GetPubContractInstance(pub_id, pub_nodename, channelname, sub_nodename)
```

Description

The **GetPubContractInstance** function gets a message from the application message queue. It creates and loads a data tree for the default message version, and returns NULL if not successful. This function is used for publication contract error correction when the error correction process needs to fetch a particular message instance for the publication contract in error. SQL on the Publication Contract table is used to retrieve the key fields.



This function should *not* be used in standard application message processing. It should only be used when correcting or debugging a publication contract that is in error.

Parameters

<i>pub_id</i>	Specifies the PubID of the message.
<i>pub_nodename</i>	Specifies the PubNodeName of the message.
<i>ChannelName</i>	Specifies the channel name of the message.
<i>sub_nodename</i>	Specifies the subscribing node of the message.

Returns

A reference to a message object if successful, NULL if not successful.

Example

```
Local message &MSG;
```

```
&MSG = GetPubContractInstance(&PUBID, &PUBNODE, &CHANNEL, &SUBNODENAME);
```

Related Topics

[CreateMessage](#), [GetMessage](#), [GetMessageInstance](#), [GetSubContractInstance](#), and Internet Script Classes

GetRecord

Syntax

```
GetRecord([RECORD.recname])
```

Description

GetRecord creates a reference to a record object for the current context, that is, from the row containing the currently executing program.



For more information, see Understanding Current Context.

The following code:

```
&REC = GetRecord();
```

is equivalent to:

```
&REC = GetRow().GetRecord(Record.recname);
```

or

```
&REC = GetRow().recname;
```



This function is invalid for PeopleCode programs located in events that aren't associated with a specific row and record at the point of execution. That is, you cannot use this function in PeopleCode programs on events associated with high-level objects like ActiveX controls, pages (the *Activate* event), or components (component events).

Returns

GetRecord returns a record object.

Parameters

With no parameters, this function returns a record object for the current context (the record containing the program that is running).

If a parameter is given, **RECORD.recname** must specify a record in the current row.

Example

In the following example, the level 2 rowset (scroll) has two records: EMPL_CHKLIST_ITM, (the primary record) and CHKLST_ITM_TBL. If the code is running from a field on the EMPL_CHKLIST_ITM record, the following will return a reference to that record:

```
&REC = GetRecord(); /*returns primary record */
```

The following returns the other record in the current row.

```
&REC2 = GetRecord(RECORD.CHKLST_ITM_TBL);
```

The following event uses the @ symbol to convert a record name that's been passed in as a string to a component name.


```

Function set_sub_event_info(&REC As Record, &NAME As string)

    &FLAGS = CreateRecord(RECORD.DR_LINE_FLG_SBR);

    &REC.CopyFieldsTo(&FLAGS);

    &INFO = GetRecord(@"RECORD." | &NAME);

    If All(&INFO) Then

        &FLAGS.CopyFieldsTo(&INFO);

    End-If;

End-Function;

```

Related Topics

CreateRecord, ProcessRequest Class, Data Buffer Access

GetRelField

Syntax

```
GetRelField(ctrl_field, related_field)
```

Description

GetRelField retrieves the value of a related display field and returns it as an unspecified (Any) data type.



This function remains for backward compatibility only. Use the GetRelated Field class method instead. See Data Buffer Access.

The field *ctrl_field* specifies the display control field, and *related_field* specifies the name of the related display field whose value is to be retrieved. In most cases, you could get the value of the field by referencing it directly. However, there are couple of cases where **GetRelField** can be useful:

- If there are two related display fields bound to the same record field, but controlled by different display control fields, you can use this function to specify which of the two related display fields you want.
- If all of a page's level-zero fields are search keys, the Component Processor does not load the entire row of level-zero data into the component buffer; it only loads the search keys. Adding a non-search-key level-zero field to the page would cause the Component Processor to load the entire row into the component buffer. To prevent a large row of data from being loaded into the buffer, you may occasionally want to make a level-zero display-only field a related display, even though the field is in the primary level-zero record. You won't be able to reference this related display field directly, but you can using **GetRelField**.

Using GetRelField with a Control Field

PeopleCode events on the Control Field can be triggered by the Related Edit field. When this happens, there can different behavior than with other types of fields:

- If the events are called from FieldEdit of the Control Field, and that FieldEdit is triggered by a change in the Related Edit field, the functions return the previous value.
- If the events are called from FieldChange of the Control Field, and that FieldChange is triggered by a change in the Related Edit field, the functions return the value entered into the Related Edit. This may be a partial value that will subsequently be expanded to a complete value when the processing is complete.

Example

In the following example, there are two related display fields in the page bound to PERSONAL_DATA.NAME. One is controlled by the EMPLID field of the high-level key, the other controlled by an editable DERIVED/WORK field in which the user can enter a new value. **GetRelField** is used to get the value of the related display controlled by EMPLID.

```
/* Use a related display of a required non-default field to verify
 * that the new Employee Id is not already in use */
If GetRelField(EMPLID, PERSONAL_DATA.NAME) <> "" Then
    Error MsgGet(1000, 65, "New Employee ID is already in use. Please
reenter.");
End-If;
```

Related Topics

FetchValue

GetRow

Syntax

```
GetRow()
```

Description

Use the **GetRow** function to obtain a row object for the current context, that is the row containing the currently executing program.



For more information, see Understanding Current Context.

Using the **GetRow()** function is equivalent to:

```
&ROW = GetRowset().GetRow(CurrentRowNumber());
```



For PeopleCode programs located in events that are not associated with a specific row at the point of execution, this function is invalid. That is, you cannot use this function in PeopleCode programs on events associated with high-level objects like ActiveX controls, pages, or components.

Parameters

None.

Returns

GetRow returns a row object that references the current row in the component buffers. If the program is not being run from a page (such as from Application Engine, or from as part of an Application Message program) it references that data.

Example

```
Local Row &ROW;  
  
&ROW = GetRow();
```

Related Topics

GetRowset and Row Class

GetRowset

Syntax

```
GetRowset ( [SCROLL.scrollname] )
```

Description

Use the **GetRowset** function to get a rowset object based on the current context. That is, the rowset is determined from the row containing the program that is running.



For more information, see Understanding Current Context.

Note

An expression of the form

```
RECORD.scrollname.property
```

or

```
RECORD.scrollname.method(...)
```

is converted to an object expression by using `GetRowset(SCROLL.scrollname)`.

Returns

With no parameters, **GetRowset** returns a rowset object for the rowset containing the currently running program. If a parameter is specified, it returns a rowset for that child scroll. *scrollname* must be the name of the primary record for the scroll.

Parameters

If a parameter is specified, it must be the name the primary record for the scroll that is a child of the current context.

Example

In the following example, RS1 is a level 1 rowset, and RS2 is a child rowset of RS1.

```
Local Rowset &RS1, &RS2;

&RS1 = GetRowset();

&RS2 = GetRowset(SCROLL.EMPL_CHKLST_ITM);
```

Related Topics

GetJavaClass and Rowset Class

GetSelectedTreeNode

Syntax

```
GetSelectedTreeNode(RECORD.recordname)
```

Description

GetSelectedTreeNode determines what node the user has selected in a dynamic tree control. The return value can then be passed to `GetTreeNodeValue`, `GetTreeNodeRecordName`, and `GetSQL` to retrieve other values.



Dynamic Tree Controls are not the same as ActiveX Tree View Controls, and the functions used with the Dynamic Tree Controls can't be used with ActiveX Tree View Controls.

Restrictions on Use in Three-Tier Mode and PeopleSoft Internet Architecture

GetSelectedTreeNode is a client-only function, which limits its use in three-tier mode and in PeopleSoft Internet Architecture applications.

- In three-tier mode **GetSelectedTreeNode** can be used only in processing groups set to run on the client. If the **GetSelectedTreeNode** function is called in a processing group running on the application server, a runtime error will occur.
- In PeopleSoft Internet Architecture applications, where all PeopleCode runs on the application server, **GetSelectedTreeNode** cannot be used.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```

If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;

```

Returns

Returns a value of type Object representing the node that the user has selected in a dynamic tree control.

Parameters

recordname The name of the record to which the root node of the tree control is bound.

Example

The following example exercises the tree control functions in a simplistic way using a three-level tree control. The hierarchy of the tree control is set up as follows. Note that the two level 3 record fields are siblings:

Level in tree hierarchy	Record field
level 0	APPR_RULE_HDR.BUSPROCNAME
level 1	APPR_RULE_LN.APPR_RULE_SET
level 2	APPR_RULE_DETL.DENY_ACTIVITYNAM

	E
level 3	APPR_RULE_AMT.MAX_CREDIT_AMT
level 3	APPR_RULE_FIELD.RTE_CNTL_TYPE

The general technique used here is to:

The example also demonstrates how to get the values of higher-level nodes.

```

/* initialize work fields to blanks*/
APPR_RULE_SET = " ";
DENY_ACTIVITYNAME = " ";
AMT1 = 0;
RTE_CNTL_TYPE1 = " ";
/* get selected node */
&TREENODE = GetSelectedTreeNode(RECORD.APPR_RULE_HDR);
/* make sure user has selected a node */
If All(&TREENODE) Then
/* get record name of selected node */
&RECNAME = GetTreeNodeRecordName(&TREENODE);
/* conditional on which record definition is bound to the selected node, display
the value of the node in the associated derived/work field */
If &RECNAME = "APPR_RULE_LN" Then
    APPR_RULE_SET = GetTreeNodeValue(&TREENODE, APPR_RULE_LN.APPR_RULE_SET);
End-If;
If &RECNAME = "APPR_RULE_DETL" Then
    DENY_ACTIVITYNAME = GetTreeNodeValue(&TREENODE,
APPR_RULE_DETL.DENY_ACTIVITYNAME);
End-If;
If &RECNAME = "APPR_RULE_AMT" Then
    AMT1 = GetTreeNodeValue(&TREENODE, APPR_RULE_AMT.MAX_CREDIT_AMT);
End-If;
If &RECNAME = "APPR_RULE_FIELD" Then
    RTE_CNTL_TYPE1 = GetTreeNodeValue(&TREENODE,
APPR_RULE_FIELD.RTE_CNTL_TYPE);
/* Get the node and value of the parent node and display its value. */
&PARENTNODE1 = GetTreeNodeParent(&TREENODE);
DENY_ACTIVITYNAME = GetTreeNodeValue(&PARENTNODE1,
APPR_RULE_DETL.DENY_ACTIVITYNAME);
/* Get and display values of higher nodes in tree. */
&PARENTNODE2 = GetTreeNodeParent(&PARENTNODE1);
APPR_RULE_SET = GetTreeNodeValue(&PARENTNODE2,
APPR_RULE_LN.APPR_RULE_SET);
End-If;
End-If;

```

Related Topics

GetTreeNodeValue, GetTreeNodeRecordName, GetSQL, RefreshTree and Implementing Dynamic Tree Controls

GetSession

Syntax

```
GetSession()
```

Description

The **GetSession** function gets a PeopleSoft session object.

After you use **GetSession**, you can instantiate many other types of objects, like Component Interfaces, data trees, and so on.



For more information, see Search Classes.

After you use **GetSession** you must connect to the system using the **Connect** property. If you are connecting to the existing session and not doing additional error checking, you may want to use the %Session system variable instead of GetSession. %Session returns a connection to the existing session.

Parameters

None.

Returns

A PeopleSoft session object.

Example

```
Local ApiObject &MYSESSION;  
  
&MYSESSION = GetSession();
```

Related Topics

Search Classes, Business Interlink Class, Tree Classes

GetSetId

Syntax

```
GetSetId({FIELD.fieldname | text_fieldname}, set_ctrl_fieldvalue,  
{RECORD.recname | text_recname}, treename)
```

Description

GetSetId returns a string containing a SetID based on a set control field (usually BUSINESS_UNIT), a set control value, and one of the following:

- The name of a control table (or view) belonging to a record group in the TableSet Control controlled by the set control value.
- The name of a tree in the TableSet Control controlled by the set control value.

If you want to pass a control recordname to the function, you must pass an empty string in the *treename* parameter. Conversely, if you want to pass a treename, you need to pass an empty string in the *text_recname* parameter. In practice, treenames are rarely used in this function.



For more information on tablesets, see Understanding Control Tables.

Returns

GetSetId returns a five-character SetID string.

Parameters

<i>fieldname</i>	Specify the set control field name as a FIELD reference. Use this parameter (recommended) or the <i>text_fieldname</i> parameter.
<i>text_fieldname</i>	Specify the name of the set control field as a string. Use this parameter or the <i>fieldname</i> parameter.
<i>set_ctrl_fieldvalue</i>	Specify the value of the set control field as a string.
<i>recname</i>	Specify as a RECORD reference the name of the control record belonging to the record group for which you want to obtain the SetID corresponding to the set control value. Use this parameter (recommended) or the <i>text_recname</i> parameter.
<i>text_recname</i>	Specify as a string the name of the control record belonging to the record group for which you want to obtain the SetID corresponding to the set control field value. Use this parameter or the <i>recname</i> parameter.
<i>treename</i>	Specify as a string the name of the tree for which you want to obtain the SetID corresponding to the set control field value.

Example

In this example, BUSINESS_UNIT is the Set Control Field, and PAY_TRMS_TBL is a control table belonging to a record group controlled by the current value of BUSINESS_UNIT. The function returns the SetID for the record group.

```
&SETID = GetSetId(FIELD.BUSINESS_UNIT, &SET_CTRL_VAL, RECORD.PAY_TRMS_TBL, "");
```

GetSQL

Syntax

```
GetSQL(SQL.sqlname [, paramlist])
```

Where *paramlist* is an arbitrary-length list of values in the form:

```
inval1 [, inval2] ...
```

Description

The **GetSQL** function instantiates a SQL object and associates it with the SQL definition specified by *sqlname*. The SQL definition must already exist, either created using Application Designer or the **StoreSQL** function.

Processing of the SQL definition is the same as for a SQL statement created by the **CreateSQL** function.



For more information, see Opening and Processing sqlstring.

Setting Data Fields to Null

This function will *not* set Component Processor data buffer fields to NULL after a row not found fetching error. However, it does set fields that aren't part of the Component Processor data buffers to NULL.

Parameters

SQL.sqlname	Specify the name of a SQL definition.
paramlist	Specify input values for the SQL string.

Returns

A SQL object.

Example

The following code creates and opens an SQL object on the SQL definition stored as ABCD_XY (for the current market, database type and as of date). It binds the given input values, and

executes the statement. If the SQL.ABCD is a SELECT, this should be followed by a series of Fetch method calls.

```
&SQL = GetSQL(SQL.ABCD_XY, ABSENCE_HIST, &EMPLID);
```

The following is a generic function that can be called from multiple places to retrieve a specific record using the SQL Objects.

```
Local SQL &SQL;

Local string &SETID, &TEMPLATE;

Local date &EFFDT;

Function FTP_GET_TEMPLATE(&REC As Record) Returns boolean ;

    &TEMPLATE = FTP_RULE_TEMPLATE;

    &EFFDT = EFFDT;

    &SETID = SETID;

    &SQL = GetSQL(SQL.FTP_TEMPLATE_SELECT, &SETID, &TEMPLATE, &EFFDT);

    If &SQL.Status = 0 Then

        If &SQL.Fetch(&REC) Then

            &SQL.Close();

            Return True;

        End-If;

    Else

        &TITLE = MsgGet(10640, 24, "SQL Error");

        MessageBox(64, &TITLE, 10640, 23, "SQL Object Not Found in SQL",
SQL.FTP_TEMPLATE_SELECT);

    End-If;

    &SQL.Close();

    Return False;

End-Function;
```

The SQL definition FTP_TEMPLATE_SELECT has the following code. Note that it uses the %List and %EFFDTCHECK meta-SQL statements. This makes the code easier to maintain: if there are any changes to the underlying record structure, this SQL definition *won't* have to change:

```
SELECT %List(FIELD_LIST,FTP_DEFAULT_TBL A)
```

```

FROM PS_FTP_TEMPLATE_TBL A

WHERE A.SETID = :1  AND A.FTP_RULE_TEMPLATE = :2

AND %EFFDTCHECK(FTP_DEFAULT_TBL A1,A, :3)  AND A.EFF_STATUS = 'A'

```

Related Topics

CreateSQL, DeleteSQL, FetchSQL, SQLExec, StoreSQL and SQL Class, Open SQL method

GetStoredFormat

Syntax

```

GetStoredFormat(scrollpath, target_row,
                [recordname.] fieldname)

```

where *scrollpath* is:

```

[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]]
RECORD.target_recname

```

To prevent ambiguous references, you can also use **SCROLL**.*scrollname*, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Description

GetStoredFormat returns the name of a field's custom stored format. To return the format for a field on level zero of the page, pass 1 in *target_row*.



This function remains for backward compatibility only. Use the StoredFormat Field class property instead. See Data Buffer Access.

Returns

Returns a String equal to the name of the stored custom format for the field.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
<i>target_row</i>	An integer specifying the row of the target field. If you are testing a field on level zero, pass 1 in this parameter.

[*recordname*.]*fieldname*

The name of the field from which to get the stored format name. The field can be on any level of the active page. The *recordname* prefix is required if the call to **GetStoredFormat** is in a record definition other than *recordname*.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Example

This example returns a string containing the custom format for postal codes on level zero of the page or on the current row of scroll level one. This function is called in the RowInit event, so no looping is necessary.

```
Function get_postal_format() Returns string
    &CURR_LEVEL = CurrentLevelNumber();
    Evaluate &CURR_LEVEL
    When = 0
        &FORMAT = GetStoredFormat(POSTAL, 1);
    When = 1
        &FORMAT = GetStoredFormat(POSTAL, CurrentRowNumber(1));
    End-Evaluate;
    Return (&FORMAT);
End-Function;
```

Related Topics

SetDisplayFormat

GetSubContractInstance

Syntax

```
GetSubContractInstance(pub_id, pub_nodename, channelname, messagename, sub_name)
```

Description

The **GetSubContractInstance** function gets a message from the application message queue. It creates and loads a data tree for the default message version, and returns NULL if not successful. This function is used for subscription contract error correction, when the error correction process needs to fetch a particular message instance for the subscription contract in error. SQL on the Subscription Contract table is used to retrieve the key fields.



This function should *not* be used in standard application message processing. It should only be used when correcting or debugging a subscription contract that is in error.

This function automatically populates the error properties at the Message, Rowset, Row, Record, and Field object levels. That is:

Message object (sets the IsEditError property to True if errors exist anywhere in this message)

Rowset (sets the IsEditError property to True if errors exist anywhere in this rowset)

Row (sets the IsEditError property to True if errors exist anywhere in this row)

Record (sets the IsEditError property to True if errors exist anywhere in this record)

Field (sets the EditError property to True if errors exist on this field, also sets the MessageNumber and MessageSetNumber properties)

Parameters

<i>pub_id</i>	Specifies the PubID of the message.
<i>ChannelName</i>	Specifies the PubNodeName of the message.
<i>MessageName</i>	Specified the name of the message to get.
<i>sub_name</i>	Specified the subscription name (SubName) of the message.

Returns

A reference to a message object if successful, NULL if not successful.

Example

The following is a complete program for getting messages from the error queue and populating a page based on the information in the message.

```

Local Message &ITEM_MESSAGE;

Local Rowset &MASTER_ROWSET, &DETAIL_ROWSET;

Local Record &MASTER_REC, &MASTER_REC_WRK;


&ITEM_MESSAGE = GetSubContractInstance (PSAPMSGSUBCON_I.PUBID,
PSAPMSGSUBCON_I.PUBNODE, PSAPMSGSUBCON_I.CHNLNAME, PSAPMSGSUBCON_I.MSGNAME,
PSAPMSGSUBCON_I.SUBNAME) ;


&MASTER_ROWSET = &ITEM_MESSAGE.GetRowset () ;

```

```

For &I = 1 To &MASTER_ROWSET.RowCount

    /*

    &MASTER_REC = &MASTER_ROWSET(&I).MASTER_ITEM_ECB;

    &MASTER_REC_WRK = CreateRecord(RECORD.MASTER_ITEMEWRK);

    &MASTER_REC.CopyFieldsTo(&MASTER_REC_WRK);

    */

    MASTER_ITEMEWRK.SETID = &MASTER_ROWSET(&I).MASTER_ITEM_ECB.SETID.Value;

    MASTER_ITEMEWRK.PROCESS_INSTANCE =
    &MASTER_ROWSET(&I).MASTER_ITEM_ECB.PROCESS_INSTANCE.Value;

    MASTER_ITEMEWRK.INV_ITEM_ID =
    &MASTER_ROWSET(&I).MASTER_ITEM_ECB.INV_ITEM_ID.Value;

    &DETAIL_ROWSET = &MASTER_ROWSET(&I).GetRowset(1);

    For &J = 1 To &DETAIL_ROWSET.RowCount

        InsertRow(RECORD.INV_ITEMS_ECWRK, &J);

        UpdateValue(INV_ITEMS_ECWRK.SETID, &J,
        &DETAIL_ROWSET(&J).INV_ITEMS_EC_BK.SETID.Value);

        UpdateValue(INV_ITEMS_ECWRK.PROCESS_INSTANCE, &J,
        &DETAIL_ROWSET(&J).INV_ITEMS_EC_BK.PROCESS_INSTANCE.Value);

        UpdateValue(INV_ITEMS_ECWRK.INV_ITEM_ID, &J,
        &DETAIL_ROWSET(&J).INV_ITEMS_EC_BK.INV_ITEM_ID.Value);

        UpdateValue(INV_ITEMS_ECWRK.INV_ITEM_SIZE, &J,
        &DETAIL_ROWSET(&J).INV_ITEMS_EC_BK.INV_ITEM_SIZE.Value);

        UpdateValue(INV_ITEMS_ECWRK.EFFDT, &J,
        &DETAIL_ROWSET(&J).INV_ITEMS_EC_BK.EFFDT.Value);

    End-For;

    DeleteRow(RECORD.INV_ITEMS_ECWRK, &DETAIL_ROWSET.RowCount + 1);

End-For;

```

Related Topics

[CreateMessage](#), [GetMessage](#), [GetMessageInstance](#), [GetPubContractInstance](#), and Internet Script Classes

GetTreeNodeParent

Syntax

GetTreeNodeParent (*node*)

Description

Use the **GetTreeNodeParent** function to access data from dynamic tree controls. It determines the parent node of the node currently selected by the user. If the *node* has no parent, the function returns a null Object value.



Dynamic Tree Controls are not the same as ActiveX Tree View Controls, and the functions used with the Dynamic Tree Controls can't be used with ActiveX Tree View Controls.

Restrictions on Use in Three-Tier Mode and PeopleSoft Internet Architecture

GetTreeNodeParent is a client-only function, which limits its use in three-tier mode and in PeopleSoft Internet Architecture applications.

- In three-tier mode **GetTreeNodeParent** can be used only in processing groups set to run on the client. If the **GetTreeNodeParent** function is called in a processing group running on the application server, a runtime error will occur.
- In PeopleSoft Internet Architecture applications, where all PeopleCode runs on the application server, **GetTreeNodeParent** cannot be used.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```

If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;
```

Returns

Returns an Object value equal to the parent node of the tree node currently selected by the user.

Parameters

<i>node</i>	A value of type OBJECT identifying a node in the tree control. This value can be returned by either <code>GetSelectedTreeNode</code> or <code>GetSQL</code> .
-------------	---

Example

See `GetSelectedTreeNode` Example.

Related Topics

`GetSelectedTreeNode`, `GetTreeNodeValue`, `GetTreeNodeRecordName`, `RefreshTree` and `Implementing Dynamic Tree Controls`

GetTreeNodeRecordName

Syntax

`GetTreeNodeRecordName (node)`

Description

Use the **GetTreeNodeRecordName** function in accessing data from dynamic tree controls. It determines the name of the record to which the tree node specified by *node* is bound.



Dynamic Tree Controls are not the same as ActiveX Tree View Controls, and the functions used with the Dynamic Tree Controls can't be used with ActiveX Tree View Controls.

Restrictions on Use in Three-Tier Mode and PeopleSoft Internet Architecture

GetTreeNodeRecordName is a client-only function, which limits its use in three-tier mode and in PeopleSoft Internet Architecture applications.

- In three-tier mode **GetTreeNodeRecordName** can be used only in processing groups set to run on the client. If the **GetTreeNodeRecordName** function is called in a processing group running on the application server, a runtime error will occur.
- In PeopleSoft Internet Architecture applications, where all PeopleCode runs on the application server, **GetTreeNodeRecordName** cannot be used.



For more information, see `PeopleCode` and `PeopleSoft Internet Architecture`.

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```

If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;

```

Returns

Returns a String value equal to the name of the record to which the specified tree node is bound.

Parameters

<i>node</i>	A value of type OBJECT identifying a node in the tree control. This value can be returned by either <code>GetSelectedTreeNode</code> or <code>GetSQL</code> .
-------------	---

Example

See `GetSelectedTreeNode` Example.

Related Topics

`GetTreeNodeValue`, `GetSelectedTreeNode`, `GetSQL`, `RefreshTree` and `Implementing Dynamic Tree Controls`

GetTreeNodeValue

Syntax

```
GetTreeNodeValue (node, [recordname.] fieldname)
```

Description

Use the **GetTreeNodeValue** function in accessing data from dynamic tree controls. It determines the value of a field specified by a node and record field. You may want to execute **GetTreeNodeValue** multiple times, passing different record fields to populate page fields with values from the row of data bound to the tree node.



Dynamic Tree Controls are not the same as ActiveX Tree View Controls, and the functions used with the Dynamic Tree Controls can't be used with ActiveX Tree View Controls.

Restrictions on Use in Three-Tier Mode and PeopleSoft Internet Architecture

GetTreeNodeValue is a client-only function, which limits its use in three-tier mode and in PeopleSoft Internet Architecture applications.

- In three-tier mode **GetTreeNodeValue** can be used only in processing groups set to run on the client. If the **GetTreeNodeValue** function is called in a processing group running on the application server, a runtime error will occur.
 - In PeopleSoft Internet Architecture applications, where all PeopleCode runs on the application server, **GetTreeNodeValue** cannot be used.
-



For more information, see PeopleCode and PeopleSoft Internet Architecture.

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```

If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;

```

Returns

Returns a value of type Any, equal to the value contained in the field specified by node and record field.

Parameters

<i>node</i>	A value of type OBJECT identifying a node in the tree control. This value can be returned by either GetSelectedTreeNode or GetSQL.
-------------	--

[recordname.]fieldname A field reference to the field to which the node is bound (in field properties).

Example

See **GetSelectedTreeNode** Example.

Related Topics

GetSelectedTreeNode, GetTreeNodeRecordName, GetSQL, RefreshTree and Implementing Dynamic Tree Controls

GetURL

Syntax

```
GetURL(URL.URLIdentifier)
```

Description

The **GetURL** function returns the URL, as a string, for the specified *URLIdentifier*. The *URLIdentifier* must exist and been created using URL Maintenance.



For more information, see URL Maintenance.



If the URL identifier has spaces in it, you must use quotation marks around *URLIdentifier*. For example, `GetURL(URL."My URL")`;

Parameters

URLIdentifier Specify a URL Identifier for a URL that already exists and was created using the URL Maintenance page.

Returns

A string containing the URL value for that URL Identifier, using the user's language preference.

Example

Suppose you have the following URL stored in URL Maintenance:

URL Maintenance

URL Identifier: PEOPLESOFT

Description: PeopleSoft Corporate Web Site ☐ PeopleSoft Internet App Server

URL: http://www.peoplesoft.com

Comments:

URL Maintenance Page

From the following code example

```
&PS_URL = GetURL(URL.PEOPLESOFT) ;
```

&PS_URL has the following value:

```
http://www.peoplesoft.com
```

Related Topics

URL Maintenance, ViewURL

GetWLFieldValue

Syntax

```
GetWLFieldValue(fieldname)
```

Description

When the user has opened a page from a worklist (by choosing one of the work items) **GetWLFieldValue** retrieves the value of a field from the current row of the application worklist record. You can use the %WLName system variable to check whether the page was accessed from a worklist.

Returns

Returns the value of a specified field in the worklist record as an Any data type.

Example

This example, from RowInit PeopleCode, populates page fields with values from the worklist record. The %WLName system variable is used to determine whether there is a currently active worklist (that is, whether the user accessed the page via a worklist).

```
&WL = %WLName;
If &WL > " " Then
```

```

&TEMP_NAME = "ORDER_NO";
ORDER_NO = GetWLFieldValue(&TEMP_NAME);
&TEMP_NAME = "BUSINESS_UNIT";
BUSINESS_UNIT = GetWLFieldValue(&TEMP_NAME);
&TEMP_NAME = "SCHED_Date";
&SCHED_Date = GetWLFieldValue(&TEMP_NAME);
SCHED_Date = &SCHED_Date;
&TEMP_NAME = "DEMAND_STATUS";
DEMAND_STATUS = GetWLFieldValue(&TEMP_NAME);
End-If;

```

Related Topics

MarkWLItemWorked, TriggerBusinessEvent

Global

Syntax

```
Global data_type &var_name
```

Description

The **Global** statement allows you to declare PeopleCode global variables. A global variable, once declared in any PeopleCode program, will remain in scope throughout the PeopleSoft session. The variable must be declared with the **Global** statement in any PeopleCode program in which it is used.

Declarations appear at the beginning of the program, intermixed with function declarations.

Not all PeopleCode data types can be declared as Global. For example, ApiObject data types can only be declared as Local.



For more information on different PeopleCode data types, see Data Types.



Because a function can be called from anywhere, you can't declare any variables within a function. You receive a design time error if you try.

Parameters

<i>data_type</i>	Specify a PeopleCode data type.
<i>&var_name</i>	A legal variable name.

Example

The following example declares a global variable and then assigns it the value of a field:

```
global string &AE_APPL_ID;

&AE_APPL_ID = AE_APPL_ID;
```

Related Topics

Local, Component

Gray

Syntax

```
Gray(scrollpath, target_row, [recordname.]fieldname)
```

where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]]
RECORD.target_recname
```

To prevent ambiguous references, you can also use **SCROLL**.*scrollname*, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation

Gray, **Hide**, **Ungray**, and **Unhide** usually appear in RowInit programs that set up the initial display of data, and in FieldChange programs that change field display based on changes the user makes to a field. Generally, you will want to put the functions on the same scroll level as the field that is being changed. This reduces the complexity of the function's syntax to:

```
Gray(fieldname)
```

The more complex syntax can be used to loop through a scroll on a lower level than the PeopleCode program.

Description

Gray grays out a page field, preventing the user from making changes to the field.



This function remains for backward compatibility only. Use the Enabled Field class property instead. See Data Buffer Access.

Gray makes a field display-only, while **Hide** makes it invisible. You can undo these effects using the built-in functions **Ungray** and **Unhide**.



If you specify a field as Display Only in Application Designer, using the PeopleCode functions **Gray**, followed by **Ungray**, will not make it editable.

Event Restrictions

This function shouldn't be used in any event prior to RowInit.

Returns

Optionally returns a Boolean value indicating whether the function succeeded.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
<i>target_row</i>	An integer specifying the row on the target scroll level where the referenced buffer field is located.
<i>[recordname.]fieldname</i>	The name of the field to gray. The field can be on scroll level one, two, or three of the active page. The <i>recordname</i> prefix is required if the call to Gray is in a record definition other than <i>recordname</i> .



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation

Example

This example, which would typically be found in the RowInit event, disables the page's address fields if the value of the SAME_ADDRESS_EMPL field is "Y".

```

If SAME_ADDRESS_EMPL = "Y" Then
    Gray(STREET1);
    Gray(STREET2);
    Gray(CITY);
    Gray(STATE);
    Gray(ZIP);
    Gray(COUNTRY);
    Gray(HOME_PHONE);
End-if;

```

Related Topics

Hide, Ungray, Unhide

GrayMenuItem

Syntax

```
GrayMenuItem(BARNAME.menubar_name, ITEMNAME.menuitem_name)
```

Description



The **GrayMenuItem** function is supported for compatibility with previous releases of PeopleTools. Future applications should use DisableMenuItem instead.

Hash

Syntax

```
Hash(ClearTextString)
```

Description

The **Hash** function returns a fixed length **unique** value, based on the input. Different input will always generate different output. The same input generates the same output. The output is always 30 characters.

Some of the original data is deliberately lost during the conversion process. This way, even if you know the algorithm, you can't "un-hash" the data.

Generally the Hash function is used like a checksum, to compare hashed values to ensure they match.

Parameters

<i>ClearTextString</i>	Specify the string you want converted.
------------------------	--

Returns

A hash string.

Example

```
MessageBox("Please confirm password");
```

```
&HASHPW = Hash(&PASSWD);
```

```
&OPERPSWD = USERDEFN.OPERPSWD.Value;
```



```

If not (&HASHPW = &OPERPSWD) Then

    /* do error handling */

End-if;

```

Related Topics

Decrypt, Encrypt, Security

HermiteCubic

Syntax

HermiteCubic(*DataPoints*)

Description

The **HermiteCubic** function computes a set of interpolating equations for a set of at least three datapoints. This particular Hermitian cubic is designed to mimic a hand-drawn curve.

Parameters

DataPoints

This parameter takes an array of array of number. The array's contents are an array of six numbers. The first two of these six numbers are the x and y points to be fit. The last four are the four coefficients to be returned from the function: **a**, **b**, **c** and **d**. **a** is the coefficient of the x^0 term, **b** is the coefficient of the x^1 term, **c** is the coefficient of the x^2 term, and **d** is the coefficient of the x^3 term.

Returns

A modified array of array of numbers. The elements in the array correspond to the elements in the array used for *DataPoints*.

Related Topics

CubicSpline, LinearInterp

Hide

Syntax

Hide(*scrollpath*, *target_row*, [*recordname.*]*fieldname*)

where *scrollpath* is:

```

[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]]
RECORD.target_recname

```

To prevent ambiguous references, you can also use **SCROLL**.*scrollname*, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation

Gray, **Hide**, **Ungray**, and **Unhide** usually appear in RowInit programs that set up the initial display of data, and in FieldChange programs that change field display based on changes the user makes to a field. Generally, you will want to put the functions on the same scroll level as the field that is being changed. This reduces the complexity of the function's syntax to:

```
Hide(fieldname)
```

The more complex syntax can be used to loop through a scroll on a lower level than the PeopleCode program.

Description

The **Hide** function makes a page field invisible. You can display the field again using **Unhide**, but **Unhide** will have no effect on a field that has been made display-only in the page definition.



This function remains for backward compatibility only. Use the Visible Field class property instead. See Data Buffer Access.

Event Restrictions

This function shouldn't be used in any event prior to RowInit.

Returns

Boolean (optional). **Hide** returns a Boolean value indicating whether it executed successfully.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
<i>target_row</i>	An integer specifying the row on the target scroll level where the referenced buffer field is located.
[<i>recordname.</i>] <i>fieldname</i>	The name of the field to hide. The field can be on scroll level one, two, or three of the active page. The <i>recordname</i> prefix is required if the call to Hide is in a record definition other than <i>recordname</i> .



For more information on scroll path syntax, see [References Using Scroll Path Syntax and Dot Notation](#)

Example

This example hides the page's address fields if SAME_ADDRESS_EMPL is equal to "Y":

```
If SAME_ADDRESS_EMPL = "Y" Then
  Hide (STREET1);
  Hide (STREET2);
  Hide (CITY);
  Hide (STATE);
  Hide (COUNTRY);
  Hide (HOME_PHONE);
End-if;
```

Related Topics

Gray, Ungray, Unhide

HideMenuItem

Syntax

```
HideMenuItem(BARNAME.menubar_name, ITEMNAME.menuitem_name)
```

Description

HideMenuItem hides a specified menu item. To apply this function to a pop-up menu, use the PrePopup Event of the field with which the pop-up menu is associated.

If you're using this function with a pop-up menu associated with a page (not a field), the earliest event you can use is the **PrePopup** event for the first "real" field on the page (that is, the first field listed in the **Order** view of the page in Application Designer.)

When a menu is first displayed, all menus are visible by default, so there is no need for a function to re-display a menuitem that has been hidden.

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```
If %CompIntfcName Then

  /* process is being called from a Component Interface */

  /* do CI specific processing */
```

```

Else
    /* do regular processing */
    . . .
End-if;

```

Returns

None.

Parameters

<i>menubar_name</i>	Name of the menu bar that owns the menuitem, or, in the case of pop-up menus, the name of the pop-up menu that owns the menuitem.
<i>menuitem_name</i>	Name of the menu item.

Example

```
HideMenuItem(BARNAME.MYPOPUP1, ITEMNAME.DO_JOB_TRANSFER);
```

Related Topics

CheckMenuItem, UnCheckMenuItem, DisableMenuItem, EnableMenuItem

HideRow

Syntax

```
HideRow(scrollpath [, target_row])
```

Where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]]
RECORD.target_recname
```

To prevent ambiguous references, you can also use **SCROLL.scrollname**, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation

Description

HideRow is used to hide a row occurrence programmatically. It hides the specified row and any associated rows at lower scroll levels.



This function remains for backward compatibility only. Use the Visible Row class property instead. See Data Buffer Access.

Hiding a row just makes the row invisible, it does not affect database processing such as inserting new rows, updating changed values, or deleting rows.

When you hide a row, it becomes the last row in the scroll or grid, and the other rows are renumbered accordingly. If you later use **UnHideRow** to make the row visible again, it is *not* moved back to its original position, but will remain in its new position. When **HideRow** is used in a loop, you have to process rows from the highest number to the lowest to achieve the correct results.



HideRow cannot be executed from the same scroll level as the row that is being hidden, or from a lower scroll level. Place the PeopleCode in a higher scroll level record.

Returns

Boolean (optional). **HideRow** returns a Boolean value indicating whether the function executed successfully.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
<i>target_row</i>	An integer specifying which row in the scroll to hide. If this parameter is omitted, the row on which the PeopleCode program is executing is assumed.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Example

This example hides all rows in scroll level 1 where the EXPORT_SW field is equal to "Y". Note that the loop has to count backwards from **ActiveRowCount** to 1.

```
For &ROW = ActiveRowCount(RECORD.EXPORT_OBJECT) to 1
step - 1
    &EXPORT_SW = FetchValue(EXPORT_OBJECT.EXPORT_SW, &ROW);
    If &EXPORT_SW "Y" Then
        HideRow(RECORD.EXPORT_OBJECT, &ROW);
    Else
        /* WinMessage("not hiding row " | &ROW);*/
```

```
End-if;
End-for;
```

Related Topics

UnhideRow, DeleteRow

HideScroll

Syntax

```
HideScroll(scrollpath)
```

Where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]]
RECORD.target_recname
```

To prevent ambiguous references, you can also use **SCROLL**.*scrollname*, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Description

HideScroll is used to programmatically hide a scroll bar and all data items within the scroll. Typically this function is used in RowInit and FieldChange PeopleCode to modify the page based on user action.



This function remains for backward compatibility only. Use the HideAllRows Rowset class method instead. See Data Buffer Access.

Returns

Boolean. **HideScroll** returns a Boolean value indicating whether the function executed successfully.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
-------------------	---



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Example

This example, from RowInit PeopleCode, initializes the visibility of the scroll based on a field setting:

```
If %Component = COMPONENT.APPR_RULE Then
  If APPR_AMT_SW = "N" Then
    HideScroll(RECORD.APPR_RULE_LN, CurrentRowNumber(1),
RECORD.APPR_RULE_DETL, CurrentRowNumber(2), RECORD.APPR_RULE_AMT);
  Else
    UnhideScroll(RECORD.APPR_RULE_LN, CurrentRowNumber(1),
RECORD.APPR_RULE_DETL, CurrentRowNumber(2), RECORD.APPR_RULE_AMT);
  End-If;
End-If;
```

The corresponding FieldChange PeopleCode dynamically changes the appearance of the page based on user changes to the APPR_AMT_SW field:

```
If APPR_AMT_SW = "N" Then
  HideScroll(RECORD.APPR_RULES_LN, CurrentRowNumber(1), RECORD.APPR_RULE_DETL,
CurrentRowNumber(2), RECORD.APPR_RULE_AMT);
  &AMT_ROWS = ActiveRowCount(RECORD.APPR_RULE_LN, CurrentRowNumber(1),
RECORD.APPR_RULE_DETL, CurrentRowNumber(2), RECORD.APPR_RULE_AMT);
  For &AMT_LOOP = &AMT_ROWS To 1 Step - 1
    DeleteRow(RECORD.APPR_RULE_LN, CurrentRowNumber(1), RECORD.APPR_RULE_DETL,
CurrentRowNumber(2), RECORD.APPR_RULE_AMT, &AMT_LOOP);
  End-For;
Else
  UnhideScroll(RECORD.APPR_RULE_LN, CurrentRowNumber(1), RECORD.APPR_RULE_DETL,
CurrentRowNumber(2), RECORD.APPR_RULE_AMT);
End-If;
```

Related Topics

HideRow, UnhideRow, UnhideScroll

HistVolatility

Syntax

```
HistVolatility(Closing_Prices, Trading_Days)
```

Description

The **HistVolatility** function computes the historical volatility of a market-traded instrument.

Parameters*Closing_Prices*

An array of number. The elements in this array contain a vector of closing prices for the instrument.

Trading_Days

The number of trading days in a year.

Returns

A number.

Related Topics

ConvertRate

Hour**Syntax**

```
Hour(time_value)
```

Description

Hour is used to extract a Number value for the hour of the day based on a Time or Datetime value. The value returned is a whole integer and is not rounded to the nearest hour.

Returns

Returns a Number equal to a whole integer value from 0 to 23 representing the hour of the day.

Parameters*time_value*

A Datetime or Time value.

Example

If &TIMEOUT contains a Time value equal to 04:59:59 PM, the following example sets &TIMEOUT_HOUR to 16:

```
&TIMEOUT_HOUR = Hour(&TIMEOUT);
```

Related Topics

Minute, Second

If**Syntax**

```
If condition Then  
    [statement_list_1]
```



```
[Else
    [statement_list_2]]
End-If
```

Description

Use the **If** statement to execute statements conditionally, depending on the evaluation of a conditional expression. The **Then** and **Else** clauses of an **If** consist of arbitrary lists of statements. The **Else** clause may be omitted. If *condition* evaluates to True, all statements in the **Then** clause are executed; otherwise, all statements in the **Else** clause are executed.

Example

The following example's first **If** statement checks for BEGIN_DT and RETURN_DT, and makes sure that RETURN_DT is greater (later) than BEGIN_DT. If this is True, the execution continues at the following line, otherwise execution continues at the line beginning with **WinMessage**:

```
If All (BEGIN_DT, RETURN_DT) and
    BEGIN_DT = RETURN_DT Then
    &DURATION_DAYS = RETURN_DT - BEGIN_DT;
    If &DURATION_DAYS 999 Then
        DURATION_DAYS = 999;
    Else
        DURATION_DAYS = &DURATION_DAYS;
    End-if;
Else
    WinMessage("The beginning date is later then the return date!");
End-if;
```

InsertImage

Syntax

```
InsertImage(scrollpath, target_row, [recordname.]fieldname)
```

where *scrollpath* is:

```
[SCROLL.level1_recname, level1_row, [SCROLL.level2_recname, level2_row,]]
SCROLL.target_recname
```

Description

The **InsertImage** function enables a user to associate an image file with a record field on a page. After the image file is associated with the record field, it can be saved to the database when the component is saved.

The following are the valid types of image files that can be associated with a record field:

- JPEG

- BMP

InsertImage uses a search page to allow the end-user to select the image file to be used. This is the same search page used to add an attachment.

Restrictions for Use with Windows Client

This function is only available in the PeopleSoft Internet Architecture, and not with windows client.

Restrictions on Use in PeopleCode Events

InsertImage is a "think-time" function, which means it shouldn't be used in any of the following PeopleCode events:

- SavePreChange
- SavePostChange
- Workflow
- RowSelect
- Any PeopleCode event that fires as a result of a Select or SelectNew method, or any of the ScrollSelect functions.



For more information, see Think-Time Functions.

Image Size Considerations

The size of the image that can be saved to the database depends on the database platform.

Database Platform	Size limitation
DB2/400	30 KB
DB2/MVS	31 KB
DB2/Unix	31 KB

Platforms other than those listed here are effectively without a size limit, that is, they allow for images equal to or greater than 2 GB.

Platform size limitations are subject to change.

Parameters

scrollpath A construction that specifies a scroll area in the component buffer.

target_row The row number of the target row.

[*recordname*].*fieldname*

The name of the field to be associated with the image file. The field can be on scroll level one, two, or three of the active page. The *recordname* prefix is required if the function call is in a record definition other than *recordname*.

Returns

The InsertImage function returns either a constant or a number:

Number	Constant	Description
0	%InsertImage_Success	Image was successfully associated with the record field.
1	%InsertImage_Failed	Image was not successfully associated with the record field. When the component is saved the image file will not be saved to the database.
2	%InsertImage_Canceled	User canceled the transaction so image file isn't associated with record field.
3	%InsertImage_ExceedsMaxSize	

Example

```
&RC = InsertImage(EMPL_PHOTO.EMPLOYEE_PHOTO);
```

Related Topics

DeleteImage

InsertRow

Syntax

```
InsertRow(scrollpath, target_row [, turbo])
```

where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]  
RECORD.target_recname
```

To prevent ambiguous references, you can also use **SCROLL**.*scrollname*, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Description

Use the **InsertRow** function to programmatically perform the ALT+7 and ENTER (**RowInsert**) function. InsertRow will insert a new row in the scroll buffer and cause a RowInsert PeopleCode event to fire, followed by the events that normally follow a RowInsert, as if the user had manually pressed ALT+7 and ENTER.



This function remains for backward compatibility only. Use the InsertRow Rowset method instead. See Data Buffer Access.

InsertRow can be executed in turbo or non-turbo mode. To execute the function in turbo mode, pass a value of True in the optional *turbo* parameter. Non-turbo mode is the default. In turbo mode, default processing is performed only for the row being inserted.

In scrolls that are not effective dated, the new row is inserted *after* the target row specified in the function call. However, if the scroll is effective dated, then the new row is inserted *before* the target row, and all the values from the previous current row are copied into the new row, except for EffDt, with is set to the current date.



InsertRow cannot be executed from the same scroll level where the insertion will take place, or from a lower scroll level. Place the PeopleCode in a higher scroll level record.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
<i>target_row</i>	The row number indicating the position where the new row will be inserted.
<i>turbo</i>	Specifies whether default processing is performed on the entire table or just the row being inserted. Pass a value of True to execute this parameter. Non-turbo mode is the default.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Example

The example inserts a row on the level-two page scroll. The PeopleCode has to be in the scroll-level-one record:

```
InsertRow(RECORD.BUS_EXPENSE_PER, &L1_ROW, RECORD.BUS_EXPENSE_DTL, &L2_ROW);
```

Related Topics

DeleteRow, HideRow, UnhideRow

Int

Syntax

```
Int(decimal)
```

Description

Int truncates a decimal number *x* to an integer and returns the result as a Number value.

Returns

Returns a Number equal to *decimal* truncated to a whole integer.

Parameters

<i>decimal</i>	A decimal number to be truncated.
----------------	-----------------------------------

Example

The following example sets &I1 to 1 and &I2 to -4:

```
&I1 = Int(1.975);  
&I2 = Int(-4.0001);
```

Related Topics

Mod, Round, Truncate, Value

IsDaylightSavings

Syntax

```
IsDaylightSavings(datetime, {timezone | "Local" | "Base"});
```

Description

IsDaylightSavings returns a Boolean value based on whether daylight savings is active in the specified time zone at the specified datetime. For time zones that don't observe daylight savings time, this function always returns False.

The system's base time zone is specified on the PSOPTIONS table.



For more information about setting the base time, see PeopleTools Utilities.

Parameters

<i>datetime</i>	The datetime value you want to check.
<i>timezone</i> Local Base	Specify a value for converting <i>datetime</i> . The valid values are: <i>timezone</i> - a time zone abbreviation or a field reference to be used for converting <i>textdatetime</i> . Local - use the local time zone for converting <i>textdatetime</i> . Base - use the base time zone for converting <i>textdatetime</i> .

Returns

A Boolean value: True if daylight savings is active in the specified time zone at the specified datetime. Returns False otherwise.

Example

In the first example, TESTDTTM has value of 01/01/99 10:00:00AM. &OUTPUT will be False.

```
&OUTPUT = IsDaylightSavings(TESTDTTM, "EST")
```

In this example, TESTDTTM has value of 04/05/99 12:00:00AM. &OUTPUT will have a value of True: 12:00am PST = 3:00am EST, so Daylight Savings Time has switched on.

```
&OUTPUT = IsDaylightSavings(TESTDTTM, "EST")
```

In this example, TESTDTTM has value of 04/05/99 12:00:00AM. &OUTPUT will return False: 12:00am PST = 1:00am MST, so Daylight Savings Time hasn't started yet.

```
&OUTPUT = IsDaylightSavings(TESTDTTM, "MST")
```

In this example, TESTDTTM has value of 07/07/99 10:00:00. &OUTPUT will return False: EST is Indiana time, where they do not observe Daily Savings Time.

```
&OUTPUT = IsDaylightSavings(TESTDTTM, "ESTA")
```

Related Topics

ConvertDatetimeToBase, ConvertTimeToBase, IsDaylightSavings, DateTimeToTimeZone, TimeToTimeZone, TimeZoneOffset

IsHidden

Syntax

```
IsHidden(scrollpath, target_row)
```

Where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]  
RECORD.target_recname
```

To prevent ambiguous references, you can also use **SCROLL**.*scrollname*, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Description

IsHidden checks to see whether a row is hidden or not. It returns True if the row is hidden, otherwise it returns False. **IsHidden** must be called in a PeopleCode program on a higher scroll level than one you are checking.



This function remains for backward compatibility only. Use the Visible Row class property instead. See Data Buffer Access.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
<i>target_row</i>	The row number indicating the position of the row.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Example

The following example tests whether a specific row on scroll level one is hidden:

```
&ROW_CNT = ActiveRowCount(RECORD.LD_SHP_INV_VW);  
&FOUND = True;  
If &ROW_CNT = 1 Then  
    &ORDER = FetchValue(LD_SHP_INV_VW.ORDER_NO, 1);
```

```

    If None(&ORDER) Then
        &FOUND = False;
    End-If;
End-If;
If &FOUND Then
    For &I = 1 To &ROW_CNT
        If Not IsHidden(RECORD.LD_SHP_INV_VW, &I) Then
            UpdateValue (ITEM_SELECTED, &I, "N");
        End-If;
    End-For;
End-If;

```

Related Topics

HideRow, UnhideRow

IsMenuItemAuthorized

Syntax

```

IsMenuItemAuthorized(MENUNAME.menuname, BARNAME.barname, ITEMNAME.menuitem_name,
PAGE.pagename [, action])

```

where *action* is one of the following constants:

Constant	Description
%Action_Add	Add
%Action_UpdateDisplay	Update/Display
%Action_UpdateDisplayAll	Update/Display All
%Action_Correction	Correction
%Action_DataEntry	Data Entry
%Action_Prompt	Prompt

If *action* is omitted, the current action is used.

Description

The **IsMenuItemAuthorized** function returns True if the current user is allowed to access the specified menu item.



You don't need to use this function to gray internal link pushbuttons/hyperlinks. This function will most often be used for transfers that are part of some PeopleCode processing.

Parameters

<i>menuname</i>	The name of the menu where the page is located, prefixed with the reserved word MENUNAME .
<i>barname</i>	The name of the menu bar where the page is located, prefixed with the reserved word BARNAME .
<i>menu_itemname</i>	The name of the menu item where the page is located, prefixed with the reserved word ITEMNAME .
<i>pagename</i>	The name of the page. This parameter must be prefixed with the keyword PAGE .
<i>action</i>	String representing the action mode in which to start up the page. If <i>action</i> is omitted, the current action is used. Valid values are:

Constant	Description
%Action_Add	Add
%Action_UpdateDisplay	Update/Display
%Action_UpdateDisplayAll	Update/Display All
%Action_Correction	Correction
%Action_DataEntry	Data Entry
%Action_Prompt	Prompt

Returns

A boolean value: True if the user is authorized to access the specified page, False otherwise.

Related Topics

DoModal, DoModalComponent, Transfer, TransferPage

IsModal

Syntax

```
IsModal()
```

Description

IsModal returns True if executed from PeopleCode running in a modal secondary page and False if executed elsewhere. This function is useful in separating secondary page-specific logic from general PeopleCode logic.

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```

If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;

```

Example

The following example executes logic specific to a secondary page:

```

If Not IsModal() Or
    Not (%Page = PAGE.PAY_OL_REV_RUNCTL Or
        %Page = PAGE.PAY_OL_RE_ASSGN_C Or
        %Page = PAGE.PAY_OL_RE_ASSGN_S) Then
    Evaluate COUNTRY
    When = "USA"
    When = "CAN"
        If Not AllOrNone(ADDRESS1, CITY, STATE) Then
            Warning MsgGet(1000, 5, "Address should consist of at least Street
(Line 1), City, State, and Country.")
            End-If;
            Break;
        When-Other;
        If Not AllOrNone(ADDRESS1, CITY, COUNTRY) Then
            Warning MsgGet(1000, 6, "Address should consist of at least Street
(Line 1), City, and Country.")
            End-If;
        End-Evaluate;
    End-If;

```

Related Topics

DoModal, EndModal

IsModalComponent

Syntax

```
IsModalComponent()
```

Description

IsModalComponent tests whether a modal component is currently executing, enabling you to write PeopleCode that only executes when a component has been called with DoModalComponent.

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```
If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;
```

Returns

Returns a Boolean value: True if the current program is executing from a modal component, False otherwise.

Example

```
If IsModalComponent() then
/* Logic that executes only if component is executing modally. */
end-if;
```

Related Topics

DoModalComponent

IsModalPanelGroup

Syntax

```
IsModalPanelGroup()
```

Description

IsModalPanelGroup tests whether a modal component is currently executing.



The **IsModalPanelGroup** function is supported for compatibility with previous releases of PeopleTools. Future applications should use **IsModalComponent** instead.

IsOperatorInClass

Syntax

```
IsOperatorInClass (operclass1 [, operclass2]...)
```

Description

IsInOperatorClass takes an arbitrary-length list of strings representing the names of operator classes and determines whether the current operator belongs to any class in a list of classes.



The **IsOperatorInClass** function is supported for compatibility with previous releases of PeopleTools. Future applications should use **IsUserInPermissionList** instead.

IsSearchDialog

Syntax

```
IsSearchDialog()
```

Description

IsSearchDialog determines whether a search dialog, add dialog, or data entry dialog box is currently executing. Use it to make processes conditional on whether a search dialog is running.

Returns

Returns a Boolean value: True if a search dialog is executing, False otherwise.

Example

```
If Not (IsSearchDialog()) Then
  If %Component = COMPONENT.SALARY_GRADE_TBL Then
    If All (SALARY_MATRIX_CD) Then
      Gray (RATING_SCALE)
    End-If;
    calc_range_spread();
  End-If;
End-If;
```

Related Topics

SetSearchDialogBehavior

IsUserInPermissionList

Syntax

```
IsUserInPermissionList(PermissionList1 [, PermissionList2]...)
```

Description

IsUserInPermissionList takes an arbitrary-length list of strings representing the names of Permission Lists and determines whether the current user belongs to any of the Permission Lists.

Parameters

<i>PermissionList</i>	An arbitrary-length list of string, each of which represents a Permission List.
-----------------------	---

Returns

Returns a Boolean value: True if the current user has access to one or more of the Permission Lists, False otherwise.

Related Topics

ExecuteRolePeopleCode, ExecuteRoleQuery, ExecuteRoleWorkflowQuery, IsUserInRole, %PermissionLists, %PrimaryPermissionList, %Roles, %RowSecurityPermissionList, Security

IsUserInRole

Syntax

```
IsUserInRole(rolename1 [, rolename2]...)
```

Description

IsInUserClass takes an arbitrary-length list of strings representing the names of roles and determines whether the current user belongs to any role in an array of roles.

Parameters

<i>rolename</i>	An arbitrary-length list of strings, each of which represents a role. PeopleCode Built-in Functions and Language Constructs E-M
-----------------	---

Returns

Returns a Boolean value: True if the current user belongs to one or more of the roles in the user role array, False otherwise.

Left

Syntax

```
Left(source_str, num_chars)
```

Description

Left returns a substring containing the leftmost number of characters in *source_str*. *num_chars* specifies how many characters to extract.

Returns

Returns a String value derived from *source_str*.

Parameters

<i>source_str</i>	A String from which to derive the substring.
-------------------	--

<i>num_chars</i>	A Number specifying how many characters to take from the left of <i>source_str</i> . The value of <i>num_chars</i> must be greater than or equal to zero. If <i>num_chars</i> is greater than the length of <i>source_str</i> , Left returns the entire string. If <i>num_chars</i> is omitted, it is assumed to be one.
------------------	---

Example

The following example sets &SHORT_ZIP to "90210":

```
&SHORT_ZIP = Left("90210-4455", 5);
```

Related Topics

Right

Len

Syntax

```
Len(str)
```

Description

Use the **Len** function to determine the number of characters in a string.

Returns

Returns a Number value equal to the number of characters, including spaces, in *str*.

Example

The following example sets &STRLEN to 10, then to 0:

```
&STRLEN = Len("PeopleSoft");  
&STRLEN = Len("");
```

Related Topics

Exact

Lenb

Syntax

```
Lenb(str)
```

Description

Lenb determines the number of bytes in a string. This is useful if you want to determine the size in bytes of a DBCS string.

Returns

Returns a Number value equal to the number of bytes in *str*.

Example

The following example sets &STRLEN to 20, if NAME is a field containing a string consisting of 10 double-byte characters:

```
&STRLEN = Lenb(NAME);
```

Related Topics

Len, Substringb

LinearInterp

Syntax

```
LinearInterp(DataPoints)
```

Description

The **LinearInterp** function computes a set of lines through a sequence of at least two points.

Parameters

DataPoints

This parameter takes an array of array of number. The array's contents are an array of six numbers. The first two of these six numbers are the x and y points to be fit. The last four are the four coefficients to be returned from the function: **a**, **b**, **c** and **d**. **a** is the coefficient of the x^0 term, **b** is the coefficient of the x^1 term, **c** is the coefficient of the x^2 term, and **d** is the coefficient of the x^3 term.

Returns

A modified array of array of numbers. The elements in the array correspond to the elements in the array used for *DataPoints*. The **c** and **d** elements will contain zeros.

Related Topics

CubicSpline, HermiteCubic

Ln

Syntax

`Ln(i)`

Description

The **Ln** function determines the natural logarithm of a number. Natural logarithms are based on the constant e, which equals 2.71828182845904. The number *i* must be a positive real number. **Ln** is the inverse of **Exp**.

Returns

A Number equal to the natural logarithm of *i*.

Example

The following examples set &I to 2.302585 and &J to 1:

```
&I = Ln(10);
&J = Ln(2.7182818);
```

Related Topics

Exp, Log10

Local

Syntax

`Local data_type &var_name`

Description

Use the **Local** statement to explicitly define local variables in PeopleCode.

Variable declarations appear at the start of the program, intermixed with function declarations. Local variables do not require explicit declaration. The scope of local variables is the PeopleCode program: they cannot be declared inside function definitions.

The system automatically initializes temporary variables. Declared variables always have values appropriate to their declared type. Undeclared local variables are initialized as null strings.



Because a function can be called from anywhere, you cannot declare any variables within a function. You will get a design time error if you try.

Parameters

data_type Any PeopleCode data type.



For more information on different PeopleCode data types, see Data Types.

&var_name A legal variable name.

Example

```
local string &LOC_FIRST;
```

Related Topics

Global, Component

Log10

Syntax

Log10 (*x*)

Description

Log10 returns the base-10 logarithm of a number *x* as a Number value. The number *x* must be a positive real number.

Returns

Returns a Number equal to the base-10 logarithm of *x*.

Example

The following example sets &X to 1 and &Y to 1.39794:

```
&X = Log10(10);  
&Y = Log10(25);
```

Related Topics

Exp, Ln

Lower

Syntax

```
Lower(string)
```

Description

Lower converts all uppercase characters in a text string to lowercase characters and returns the result as a String value. **Lower** does not change characters that are not letters. The lowercase conversion is based on the Windows language you have active. Refer to the Windows Control Panel to specify the language.

Returns

A String value equal to *string*, but in all lowercase format.

Example

The example sets &GOODKD to "k d lang":

```
&GOODKD = Lower("K D Lang");
```

Related Topics

Proper, Upper

LTrim

Syntax

```
LTrim(string1 [,string2])
```

Description

LTrim returns a string formed by deleting from the beginning of *string1*, all occurrences of each character in *string2*. If *string2* is omitted, " " is assumed; that is, leading blanks are trimmed.

Example

The following removes leading blanks from &NAME:

```
&TRIMMED = LTrim(&NAME);
```

The following removes leading punctuation marks from REC.INP:

```
&TRIMMED = LTrim(REC.INP, ".,:;!?" );
```

Related Topics

RTrim

MarkWLItemWorked

Syntax

```
MarkWLItemWorked()
```

Description

If you've invoked a page from the worklist, you can mark the current worklist entry as worked using this function. It should only be called in Workflow PeopleCode. You can use the %WLName system variable to check whether the page has been accessed via a worklist.

Returns

Returns a Boolean value indicating whether it executed successfully. The return value is not optional.

Example

This example, which would be in the WorkFlow event, checks to see whether a page check box MARK_WORKED_SW is selected, and if so, it marks the item in the worklist as complete:

```
If MARK_WORKED_SW = "Y" Then
    If MarkWLItemWorked() Then
        End-If;
    End-If;
```

Related Topics

GetTreeNodeValue, TriggerBusinessEvent

MessageBox

Syntax

```
MessageBox(style, title, message_set, message_num, default_txt [, paramlist])
```

where *paramlist* is an arbitrary-length list of parameters of undetermined (Any) data type to be substituted in the resulting text string, in the form:

```
param1 [, param2]...
```

Description

MessageBox displays a message box window. This function combines dialog-display ability with the text-selection functionality of `MsgGet`, `MsgGetText` and `MsgGetExplainText`. The *style* parameter selects the buttons to be included. *title* determines the title of message.



Note. *title* isn't available for messages displayed in the PeopleSoft Internet Architecture. Also, *style* is ignored if the message has any severity other than Message.

The remaining parameters are used to retrieve and process a text message selected from the Message Catalog.

MessageBox can be used for simple informational display, where the user reads the message, then clicks an **OK** button to dismiss the message box. **MessageBox** can also be used as a way of branching based on user choice, in which case the message box contains two or more buttons (such as **OK** and **Cancel** or **Yes**, **No**, and **Cancel**). The value returned by the function tells you which button the user clicked, and your code can branch based on that value.

In the **MessageBox** dialogs, both the Text and the Explanation, that is, more detailed information stored in the Message Catalog, are included.



Note. In Windows Client, **MessageBox** dialogs include an **Explain** button to display more detailed information stored in the Message Catalog.

If **MessageBox** displays buttons other than **OK**, it causes processing to stop while it waits for user response. This makes it a "user think-time" function, restricting its use to certain PeopleCode events.

Message Retrieval

MessageBox retrieves a message from the Message Catalog and substitutes the values of the parameters into the text message and explanation.

You can access and update the Message Catalog through PeopleTools Utilities, using the Message Catalog page located under the Use menu. You can enter message text in multiple languages. The *message_set* and *message_num* parameters specify the message to retrieve from the catalog. If the message is not found in the Message Catalog, the default message provided in *default_txt* will be used. Message sets 1 through 19,999 are reserved for use by PeopleSoft applications. Message sets 20,000 through 32,767 can be used by PeopleSoft users.

The optional *paramlist* is a comma-separated list of parameters; the number of parameters in the list is arbitrary. The parameters are referenced in the message text or message explanation using the % character followed by an integer corresponding to the position of the parameter in the *paramlist*. For example, if the first and second parameters in *paramlist* were `&FIELDNAME` and `&USERNAME`, they would be inserted into the message string as `%1` and `%2`. To include a literal percent sign in the string, use `%%`; `%\` is used to indicate an end-of-string and terminates the string at that point—this is generally used to specify fixed-length strings with trailing blanks.

Message Severity

MessageBox specifies processing for error handling functions based on the Message Severity of the message, which you can set in the Message Catalog. This enables you to change the severity of an error without changing the underlying PeopleCode, by setting the severity level for the message in the Message Catalog. The Message Severity settings and processing options are as follows:

Severity	Processing
Message	The message is displayed and processing continues.
Warning	The message is displayed and treated as a warning.
Error	The message is displayed and treated as an error.
Cancel	The message is displayed and forces a Cancel.

In addition, in the PeopleSoft Internet Architecture the Message Severity dictates how the message displays:

- If the message has a severity of Warning, Error or Cancel, the message is displayed in a pop-up dialog box with a single **OK** button *regardless* of the value of the *style* parameter.
- If the message has a severity of Message and *style* is %MsgStyle_OK (0), the message displays in a pop-up dialog box with the single **OK** button.
- If the message has a severity of Message and *style* is not %MsgStyle_OK (0), the message displays in a separate window.

Restrictions on Use in PeopleCode Events

If **MessageBox** displays any buttons other than **OK**, it returns a value based on the end-user response and interrupts processing until the end-user has clicked one of the buttons. This makes it a "user think-time" function, subject to the same restrictions as other think-time functions (see Think-Time Functions), which means that it cannot be used in any of the following PeopleCode events:

- SavePreChange
- Workflow
- RowSelect
- SavePostChange
- Any PeopleCode event that fires as a result of a ScrollSelect, ScrollSelectNew, RowScrollSelect or RowScrollSelectNew function call.

If the *style* parameter specifies a single button (that is, the **OK** button), the function can be called in any PeopleCode event.

Restrictions on Use with PeopleSoft Internet Architecture

In the PeopleSoft Internet Architecture, you can't change the icon of a message box. You can change the number and type of buttons, as well as the default button, but the message always displays with the warning icon (a triangle with an exclamation mark in it.)

In addition, you can't change the message box title.

Restrictions on Use with Application Engine

If you call `MessageBox` from a PeopleCode action in an Application Engine program, the syntax is the same. However, all GUI-related parameters like `style` and `title` are ignored. You should use 0 and "".



If you have an existing `MessageBox` in code called from a page, it should work as is.

The actual message data is routed to `PS_MESSAGE_LOG` at runtime, and you can view it from the Process Monitor by drilling down to the process details.

Parameters

Style

Either a numerical value or a constant specifying the contents and behavior of the dialog box. This parameter is calculated by cumulatively adding either a value or a constant from each category below:



In PeopleSoft Internet Architecture *style* is ignored if the message has any severity other than Message.

Category	Value	Constant	Meaning
Buttons	0	%MsgStyle_OK	The message box contains one pushbutton: OK.
	1	%MsgStyle_OKCancel	The message box contains two pushbuttons: OK and Cancel.
	2	%MsgStyle_AbortRetryIgnore	The message box contains three pushbuttons: Abort, Retry, and Ignore.
	3	%MsgStyle_YesNoCancel	The message box contains three pushbuttons: Yes, No, and Cancel.
	4	%MsgStyle_YesNo	The message box contains two push buttons: Yes and No.
	5	%MsgStyle_RetryCancel	The message box contains two push buttons: Retry and Cancel.



Note. The following values for *style* can only be used in Windows Client. They have no affect in PeopleSoft Internet Architecture.

Category	Value	Constant	Meaning
Default Button	0	%MsgDefault_First	The first button is the default.
	256	%MsgDefault_Second	The second button is the default.
	512	%MsgDefault_Third	The third button is the default.
Icon	0	%MsgIcon_None	None
	16	%MsgIcon_Error	A stop-sign icon appears in the message box.
	32	%MsgIcon_Query	A question-mark icon appears in the message box.
	48	%MsgIcon_Warning	An exclamation-point icon appears in the message box.
	64	%MsgIcon_Info	An icon consisting of a lowercase letter "i" in a circle appears in the message box.



For more information, see Restrictions on Use with a Component Interface.

title Title of message box. If a null string is specified, then PeopleTools will provide an appropriate value.



Note. *title* isn't available for messages displayed in the PeopleSoft Internet Architecture.

message_set The message set number of the message to be displayed. When message set and number are provided, it overrides the specified text. A value less than one indicates that the message will come from the provided text and not the Message Catalog.

message_num The message number of the message to be displayed.

default_txt

Default text to be displayed in the message box.

paramlist

A comma-separated list of parameters; the number of parameters in the list is arbitrary. The parameters are referenced in the message text using the % character followed by an integer corresponding to the position of the parameter in the *paramlist*.

Returns

Returns either a Number value or a constant. The return value is zero if there is not enough memory to create the message box. In other cases the following menu values are returned:

Value	Constant	Meaning
-1	%MsgResult_Warning	Warning was generated.
1	%MsgResult_OK	OK button was selected.
2	%MsgResult_Cancel	Cancel button was selected.
3	%MsgResult_Abort	Abort button was selected.
4	%MsgResult_Retry	Retry button was selected.
5	%MsgResult_Ignore	Ignore button was selected.
6	%MsgResult_Yes	Yes button was selected.
7	%MsgResult_No	No button was selected.



In PeopleSoft Internet Architecture, pressing the ESC key has no effect. In Windows client, if a message box has a **Cancel** button, the same value will be returned if either the ESC key is pressed or the **Cancel** button is selected. If the message box has no **Cancel** button, pressing ESC has no effect.

Example

Suppose the following string literal is stored in the Message Catalog as the message text:

```
Expenses of employee %1 during period beginning %2 exceed allowance.
```

The following is stored in the Explanation:

```
You don't have the authority to approve this expense. Only a director can approve this.
```

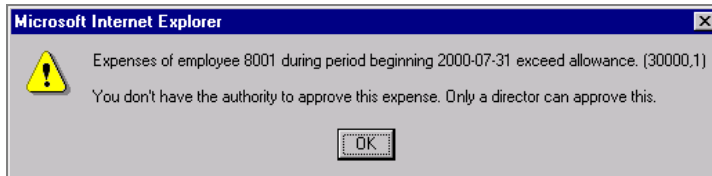
Here %1 is a placeholder for the employee ID and %2 a placeholder for the expense period date. The following MessageBox call provides bind variables corresponding to these placeholders at the end of its parameter list:


```

MessageBox(0, "", 30000, 1, "Message not found.", BUS_EXPENSE_PER.EMPLID,
BUS_EXPENSE_PER.EXPENSE_PERIOD_DT);

```

The call would display the a message box similar to this, if the message severity was Error or Warning.



Example of Message Box

Suppose the following is stored in the Message Catalog as the message text:

```
File not found.
```

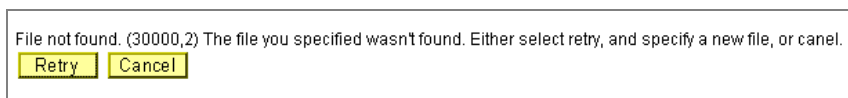
The following is stored in the Explanation:

```
The file you specified wasn't found. Either select retry, and specify a new
file, or cancel.
```

Suppose this message had a Severity of message, and you used the %MsgStyle_RetryCancel, in the following code:

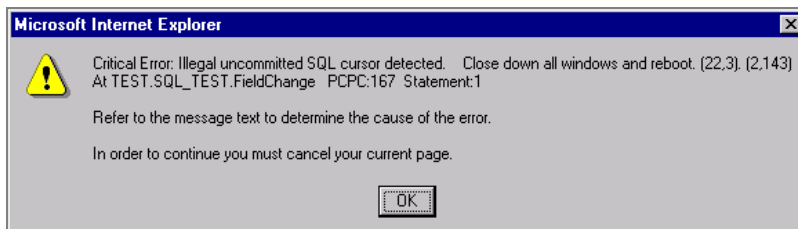
```
MessageBox(%MsgStyle_RetryCancel, "", 30000, 2, "Message not found.");
```

This is how the message displays:



Example Message with Retry and Cancel Buttons

If the message severity is of type Cancel, the message displayed looks like the following:



Critical type error message

Related Topics

MsgGet, MsgGetText, MsgGetExplainText

Minute

Syntax

```
Minute(timevalue)
```

Description

Use the **Minute** function to extract the minute component of a Time value.

Returns

Returns the minute part of *timevalue* as a Number data type.

Example

If &TIMEOUT contains "16:48:01" then the example sets &TIMEOUT_MINUTES to 48:

```
&TIMEOUT_MINUTES = Minute(&TIMEOUT) ;
```

Related Topics

Hour, Second

Mod

Syntax

```
Mod(x, divisor)
```

Description

Mod, the modulus function divides one number (*x*) by another (*divisor*) and returns the remainder.

Returns

Returns a Number equal to the remainder of the division of the number *x* by *divisor*.

Example

The example sets &NUM1 to one and &NUM2 to zero:

```
&NUM1 = Mod(10,3) ;  
&NUM2 = Mod(10,2) ;
```

Related Topics

Int, Round, Truncate

Month

Syntax

Month(*datevalue*)

Description

Month returns the month of the year as an integer from 1 to 12 for the specified *datevalue*. The **Month** function accepts a Date or DateTime value as a parameter.

Returns

Returns a Number value from 1 to 12 specifying the month of the year.

Parameters

<i>datevalue</i>	A Date or DateTime value on the basis of which to determine the month.
------------------	--

Example

This example sets &HIRE_MONTH to 3:

```
&HIREDATE = DateTime6(1997, 3, 15, 10, 9, 20);
&HIRE_MONTH = Month(&HIRE_DATE);
```

Related Topics

Date, Date3, DateValue, Day, Days360, Days365, Weekday, Year

MsgGet

Syntax

MsgGet(*message_set*, *message_num*, *default_msg_txt* [, *paramlist*])

where *paramlist* is an arbitrary-length list of parameters of undetermined (Any) data type to be substituted in the resulting text string, in the form:

param1 [, *param2*]...

Description

The **MsgGet** function retrieves a message from the PeopleCode Message Catalog and substitutes in the values of the parameters into the text message.

You can access and update the Message Catalog through the PeopleTools Utilities, using the Message Catalog page located under the Use menu. You can enter message text in multiple

languages. The Message Catalog also enables you to enter more detailed "Explain" text about the message. The *message_set* and *message_num* parameters specify the message to retrieve from the catalog. If the message is not found in the Message Catalog, the default message provided in *default_msg_txt* is used. Message sets 1 through 19,999 are reserved for use by PeopleSoft applications. Message sets 20,000 through 32,767 can be used by PeopleSoft users.

The optional *paramlist* is a comma-separated list of parameters; the number of parameters in the list is arbitrary. The parameters are referenced in the message text using the % character followed by an integer corresponding to the position of the parameter in the *paramlist*. For example, if the first and second parameters in *paramlist* were &FIELDNAME and &USERNAME, they would be inserted into the message string as %1 and %2. To include a literal percent sign in the string, use %%; %\ is used to indicate an end-of-string and terminates the string at that point—this is generally used to specify fixed-length strings with trailing blanks.

MsgGet suffixes the message with "[Message Set# and Message Error#]", so it can be processed by a user not conversant in the translated language.

Example

```
&MsgText = &MsgGet(30000, 2, "Message not found");
```

Related Topics

MsgGetText, MsgGetExplainText, MessageBox

MsgGetExplainText

Syntax

```
MsgGetExplainText(message_set, message_num, default_msg_txt [, paramlist])
```

where *paramlist* is an arbitrary-length list of parameters of undetermined (Any) data type to be substituted in the resulting text string, in the form:

```
param1 [, param2]...
```

Description

The **MsgGetExplainText** function retrieves the explain text of a message from the PeopleCode Message Catalog and substitutes the values of the parameters in *paramlist* into the explain text. It returns the resulting message explain text as a String data type.

You can access and update the Message Catalog through the PeopleTools Utilities window, using the Message Catalog panel located under the Use menu. You can enter messages in multiple languages.

Message sets 1 through 19,999 are reserved for use by PeopleSoft applications. Message sets 20,000 through 32,767 can be used by PeopleSoft users.

Unlike **MsgGet**, **MsgGetExplainText** returns the message without a message set and message number appended to the message.

Parameters

<i>message_set</i>	Specify the message set to be retrieved from the catalog. This parameter takes a number value.
<i>message_num</i>	Specify the message number to be retrieved from the catalog. This parameter takes a number value.
<i>default_msg_txt</i>	Specify the text to be displayed if the message isn't found. This parameter takes a string value.
<i>paramlist</i>	Specify values to be substituted into the message explain text.

The parameters listed in the optional *paramlist* are referenced in the message explain text using the % character followed by an integer referencing the position of the parameter in the function call. For example, if the first and second parameters in *paramlist* were &FIELDNAME and &USERNAME, they would be inserted into the message string as %1 and %2.

Note. This substitution only takes place in message explain text when the **MsgGetExplainText** function is used. If you use a message box, the parameter substitution will *not* occur in the explain text.

To include a literal percent sign in the string, use %%; %\ is used to indicate an end-of-string and terminates the string at that point—this is generally used to specify fixed-length strings with trailing blanks.

Example

Suppose the following explain text is stored in the Message Catalog:

```
A reference was made to a record.field (%1.%2) that is not defined within
Application Designer. Check for typographical errors in the specification of
the record.field or use Application Designer to add the new field or record.
```

Here %1 is a placeholder for the record name and %2 a placeholder for the field name. If the record.field in error was MyRecord.Field5, the above would resolve as follows:

```
A reference was made to a record.field (MyRecord.Field5) that is not defined
within Application Designer. Check for typographical errors in the
specification of the record.field or use Application Designer to add the new
field or record.
```

Related Topics

MsgGet, MsgGetText, MessageBox

MsgGetText

Syntax

```
MsgGetText (message_set, message_num, default_msg_txt [, paramlist])
```

where *paramlist* is an arbitrary-length list of parameters of undetermined (Any) data type to be substituted in the resulting text string, in the form:

```
param1 [, param2]...
```

Description

The **MsgGetText** function retrieves a message from the PeopleCode Message Catalog and substitutes the values of the parameters in *paramlist* into the text message. It returns the resulting message text as a String data type.

You can access and update the Message Catalog through the PeopleTools Utilities window, using the Message Catalog page located under the Use menu. You can enter message text in multiple languages. The *message_set* and *message_num* parameters specify the message to retrieve from the catalog. If the message is not found in the Message Catalog, the default message provided in *default_msg_txt* will be used. Message sets 1 through 19,999 are reserved for use by PeopleSoft applications. Message sets 20,000 through 32,767 can be used by PeopleSoft users.

The parameters listed in the optional *paramlist* are referenced in the message text using the % character followed by an integer referencing the position of the parameter in the function call. For example, if the first and second parameters in *paramlist* were &FIELDNAME and &USERNAME, they would be inserted into the message string as %1 and %2. To include a literal percent sign in the string, use %%; %\ is used to indicate an end-of-string and terminates the string at that point—this is generally used to specify fixed-length strings with trailing blanks.

Unlike **MsgGet**, **MsgGetText** returns the message without a message set and message number appended to the message.

Example

```
&MsgText = MsgGetText (30000, 2, "Message not found");
```

Related Topics

MsgGet, MsgGetExplainText, MessageBox

PeopleCode Built-in Functions and Language Constructs N-Z

NextEffDt

Syntax

```
NextEffDt(field)
```

Description

NextEffDt returns the value of the specified *field* from the record with the next effective date (and effective sequence number if specified). The return value is an Any data type. This function is only valid for effective dated records. If a next record does not exist, then the statement is skipped.

Related Topics

NextRelEffDt, PriorRelEffDt, PriorEffDt

NextRelEffDt

Syntax

```
NextRelEffDt(search_field, fetch_field)
```

where *fieldlist* is an arbitrary-length list of fields in the form:

```
field1 [, field2]...
```

Description

NextRelEffDt locates the next occurrence of the *search_field* with the next effective date (and effective sequence number if the record contains an effective sequence number). It then returns the value of the specified *fetch_field* corresponding to the *search_field*. The return value is an Any data type. Typically, this function is used to retrieve values for related display fields.

This function is only valid for effective dated records. If a next record does not exist, then the statement is skipped.

Related Topics

GetRelField, NextEffDt, PriorRelEffDt, PriorEffDt

None

Syntax

None(*fieldlist*)

where *fieldlist* is an arbitrary-length list of fields in the form:

[*recordname.*]*fieldname1* [, [*recordname.*]*fieldname2*] ...

Description

None takes an arbitrary number of field names as parameters and tests for values. **None** returns True if none of the specified fields contain a value. It returns False if any one of the fields contains a value.

A blank character field, or a zero (0) numeric value in a required numeric field is considered a null value.

Related Functions

All	Checks to see if a field contains a value, or if all the fields in a list of fields contain values. If any of the fields is Null, then All returns False.
AllOrNone	Checks if either all the field parameters have values, or none of them have values. Use this in cases where if an end-user fills in one field, she must all fill in the other related values.
OnlyOne	Checks if exactly one field in the set has a value. Use this when the end-user must fill in only one of a set of mutually exclusive fields.
OnlyOneOrNone	Checks if no more than one field in the set has a value. Use this in cases when a set of fields is both optional and mutually exclusive; that is, if the end-user puts can put a value into one field in a set of fields, or leave them all empty.

Example

The following example uses **None** to check whether REFERRAL_SOURCE has a value:

```
If None(REFERRAL_SOURCE) or
  REFERRAL_SOURCE = "EE" Then
  Gray(EMP_REFERRAL_ID);
End-if;
```

The following example uses **None** with a variable:

```
&ONETIME = FetchValue(POSN_INCUMB_WS.EMPLID, 1);
```



```

    If None(&ONETIME) Then

        /* do processing */

    End-if;

```

Related Topics

All, AllOrNone, OnlyOne, OnlyOneOrNone

ObjectDoMethod

Syntax

```
ObjectDoMethod(obj_this, str_method_name [, paramlist])
```

Where *paramlist* is a list of parameters of arbitrary length:

```
param1 [, param2]...
```

Description

ObjectDoMethod invokes a method *str_method_name* on an instance of an OLE automation object *obj_this*, passing it the parameters in *paramlist*.



The OLE functions are used to access and manipulate external OLE objects only. You can *not* use them to access or manipulate PeopleCode objects such as arrays, rowsets, etc. However, the ObjectSetProperty function *can* be used with the Tree View and Chart ActiveX control in special cases. See ActiveX Controls in PeopleTools.

Returns

None.

Parameters

<i>obj_this</i>	A variable of type Object, representing an instance of an OLE Automation object. This variable must have been instantiated previously with a call to CreateObject .
<i>str_method_name</i>	A string containing the name of an exposed method of <i>obj_this</i> .
<i>paramlist</i>	The parameter list to pass to the <i>str_method_name</i> method.

Restrictions on Use in PeopleCode Events

ObjectDoMethod is a "think time" function, and therefore cannot be used in any of the following PeopleCode events:

- SavePreChange
- Workflow
- SavePostChange
- RowSelect

Restrictions on Use in Three-Tier Mode and PeopleSoft Internet Architecture

Because it is specific to Windows, **ObjectDoMethod** is a client-only function, which limits its use in three-tier mode and in PeopleSoft Internet Architecture applications.

- In three-tier mode **ObjectDoMethod** can be used only in processing groups set to run on the client. If the **ObjectDoMethod** function is called in a processing group running on the application server, a runtime error will occur.
- In PeopleSoft Internet Architecture applications, where all PeopleCode runs on the application server, **ObjectDoMethod** cannot be used at all.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

Example

This simple example instantiates an Excel worksheet object, makes it visible, names it, saves it, and displays its name.

```
&WORKAPP = CreateObject("Excel.Application");

&WORKBOOKS = ObjectGetProperty(&WORKAPP, "Workbooks");

ObjectDoMethod(&WORKBOOKS, "Add", "C:\TEMP\INVOICE.XLT"); /* This associates the
INVOICE template w/the workbook */

ObjectDoMethod(&WORKAPP, "Save", "C:\TEMP\TEST1.XLS");

ObjectSetProperty(&WORKAPP, "Visible", True);
```

Related Topics

CreateObject, ObjectGetProperty, ObjectSetProperty

ObjectGetProperty

Syntax

```
ObjectGetProperty(obj_this, str_property_name [, index_param_list])
```

Description

ObjectGetProperty returns the value of a property *str_property_name* of the object *obj_this*; this object is an instance of an OLE automation object, which must have been instantiated previously with **CreateObject**.



"Default" OLE object properties are not supported. You must specify the object property that you want to retrieve explicitly.



The OLE functions are used to access and manipulate external OLE objects only. You can *not* use them to access or manipulate PeopleCode objects such as arrays, rowsets, etc. However, the **ObjectSetProperty** function *can* be used with the Tree View and Chart ActiveX control in special cases. See ActiveX Controls in PeopleTools.

Returns

Returns an Any value equal to the value of the *str_property_name* property of the *obj_this* object.

Parameters

<i>obj_this</i>	A variable of type Object, representing an instance of an OLE Automation object. This variable must have been instantiated previously with a call to CreateObject .
<i>str_property_name</i>	A string containing the name of an exposed property of <i>obj_this</i> .
<i>index_param_list</i>	A list for accessing the index property called Range used with Excel Worksheets.

Restrictions on Use in PeopleCode Events

ObjectGetProperty is a "think time" function, and therefore cannot be used in any of the following PeopleCode events:

- SavePreChange
- Workflow
- SavePostChange
- RowSelect

Restrictions on Use in Three-Tier Mode and PeopleSoft Internet Architecture

Because it is specific to Windows, **ObjectGetProperty** is a client-only function, which limits its use in three-tier mode and in PeopleSoft Internet Architecture applications.

- In three-tier mode **ObjectGetProperty** can be used only in processing groups set to run on the

client. If the **ObjectGetProperty** function is called in a processing group running on the application server, a runtime error will occur.

- In PeopleSoft Internet Architecture applications, where all PeopleCode runs on the application server, **ObjectGetProperty** cannot be used at all.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

Example

This simple example instantiates an Excel worksheet object, makes it visible, names it, saves it, and displays its name.

```
&WORKAPP = CreateObject("Excel.Application");

&WORKBOOKS = ObjectGetProperty(&WORKAPP, "Workbooks");

ObjectDoMethod(&WORKBOOKS, "Add", "C:\TEMP\INVOICE.XLT"); /* This associates the
INVOICE template w/the workbook */

ObjectDoMethod(&WORKAPP, "Save", "C:\TEMP\TEST1.XLS");

ObjectSetProperty(&WORKAPP, "Visible", True);
```

Excel Worksheets had an index property called Range that has the following signature:

```
Property Range (Cell1 [, Cell2]) as Range
```

In the following example, the range is A1:

```
&CELL = ObjectGetProperty(&SHEET, "Range", "A1");
```

Related Topics

CreateObject, ObjectDoMethod, ObjectSetProperty

ObjectSetProperty

Syntax

```
ObjectSetProperty(obj_this, str_property_name, val [, index_param_list])
```

Description

ObjectSetProperty sets the value of a property *str_property_name* of the object *obj_this* to *val*.



The OLE functions are used to access and manipulate external OLE objects only. You can *not* use them to access or manipulate PeopleCode objects such as arrays, rowsets, etc. However, the ObjectSetProperty function *can* be used with the Tree View and Chart ActiveX control in special cases. See ActiveX Controls in PeopleTools.

The object *obj_this* is an instance of an OLE automation object, which must have been instantiated previously by a call to **CreateObject**.



"Default" OLE object properties are not supported. You must specify the object property that you want to set explicitly.

Returns

None.

Parameters

<i>obj_this</i>	A variable of type Object, representing an instance of an OLE Automation object. This variable must have been instantiated previously with a call to CreateObject .
<i>str_property_name</i>	A string containing the name of an exposed property of <i>obj_this</i> .
<i>val</i>	<i>str_property_name</i> is set to this value.
<i>index_param_list</i>	A list for accessing the index property called Range used with Excel Worksheets.

Restrictions on Use in PeopleCode Events

ObjectSetProperty is a "think time" function, and therefore cannot be used in any of the following PeopleCode events:

- SavePreChange
- Workflow
- SavePostChange
- RowSelect

Restrictions on Use in Three-Tier Mode and PeopleSoft Internet Architecture

Because it is specific to Windows, **ObjectSetProperty** is a client-only function, which limits its use in three-tier mode and in PeopleSoft Internet Architecture applications.

- In three-tier mode **ObjectSetProperty** can be used only in processing groups set to run on the client. If the **ObjectSetProperty** function is called in a processing group running on the application server, a runtime error will occur.
- In PeopleSoft Internet Architecture applications, where all PeopleCode runs on the application server, **ObjectSetProperty** cannot be used at all.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

Example

This simple example instantiates an Excel worksheet object, makes it visible, names it, saves it, and displays its name.

```
&WORKAPP = CreateObject("Excel.Application");

&WORKBOOKS = ObjectGetProperty(&WORKAPP, "Workbooks");

ObjectDoMethod(&WORKBOOKS, "Add", "C:\TEMP\INVOICE.XLT"); /* This associates the
INVOICE template w/the workbook */

ObjectDoMethod(&WORKAPP, "Save", "C:\TEMP\TEST1.XLS");

ObjectSetProperty(&WORKAPP, "Visible", True);
```

Related Topics

CreateObject, ObjectDoMethod, ObjectGetProperty

OnlyOne

Syntax

```
OnlyOne(fieldlist)
```

where *fieldlist* is an arbitrary-length list of fields in the form:

```
[recordname.] fieldname1 [, [recordname.] fieldname2] ...
```

Description

OnlyOne checks a list of fields and returns True if one and only one of the fields has a value. If all of the fields are empty, or if more than one of the fields has a value, **OnlyOne** returns False. This function is used to validate that only one of a set of mutually exclusive fields has been given a value.

A blank character field, or a zero numeric value in a required numeric field is considered a Null value.

Related Functions

All	Checks to see if a field contains a value, or if all the fields in a list of fields contain values. If any of the fields is Null, then All returns False.
None	Checks that a field or list of fields have no value. None is the opposite of All .

AllOrNone	Checks if either all the field parameters have values, or none of them have values. Use this in cases where if an end-user fills in one field, she must all fill in the other related values.
OnlyOneOrNone	Checks if no more than one field in the set has a value. Use this in cases when a set of fields is both optional and mutually exclusive; that is, if the end-user puts can put a value into one field in a set of fields, or leave them all empty.

Example

You typically use **OnlyOne** as follows:

```
If OnlyOne(param_one, param_two)
  Then value_a = "Y";
End-if;
```

Related Topics

All, AllOrNone, None, OnlyOneOrNone

OnlyOneOrNone

Syntax

```
OnlyOneOrNone(fieldlist)
```

where *fieldlist* is an arbitrary-length list of fields in the form:

```
[recordname.]fieldname1 [, [recordname.]fieldname2] ...
```

Description

OnlyOneOrNone checks a list of fields and returns True if either of these conditions is true:

- Only one of the fields has a value.
- None of the fields has a value.

This function is useful when you have a set of mutually exclusive fields in a page and the entire set of fields is optional. The end-user can leave all the fields blank or enter a value in one of the fields only.

A blank character field, or a zero numeric value in a required numeric field is considered a null value.

Related Functions

All	Checks to see if a field contains a value, or if all the fields in a list of fields contain values. If any of the fields is Null, then All returns False.
None	Checks that a field or list of fields have no value. None is the opposite of All.
AllOrNone	Checks if either all the field parameters have values, or none of them have values. Use this in cases where if an end-user fills in one field, she must all fill in the other related values.
OnlyOne	Checks if exactly one field in the set has a value. Use this when the end-user must fill in only one of a set of mutually exclusive fields.

Example

You typically use **OnlyOneOrNone** as follows:

```
If OnlyOneOrNone(param_one, param_two)
    Then value_a = "Y";
End-if;
```

Related Topics

All, AllOrNone, None, OnlyOne

PanelGroupChanged

Syntax

```
PanelGroupChanged ( )
```

Description

PanelGroupChanged is used to determine whether a component has changed since the last save, whether by the user or by PeopleCode.



The **PanelGroupChanged** function is supported for compatibility with previous releases of PeopleTools. Future applications should use ComponentChanged instead.

PingNode

Syntax

PingNode (*MsgNodeName*)

Description

The **PingNode** function pings the specified node. It returns an array of numbers, containing the status codes for all locations. The status codes are ordered by SEQNO of the URL.

Parameters

MsgNodeName Specify the message node you want to ping.

Returns

An array of data type number, containing the status codes for all locations. A status of 200 means the ping completed successfully. Any other status means ping did not complete successfully.

Example

```
Local array of number &StatusCodes;

&StatusCodes = PingNode (&MyMsgNode) ;

For &I = 1 to &StatusCodes.Len

    If &StatusCodes[&I] = 200 then;

        /* ping was successful - do some processing */

    Else

        /* do error procesing */

    End-If;

End-For;
```

Related Topics

Array Class, Message Class, Introduction to Application Messaging

PriorEffDt

Syntax

PriorEffDt (*field*)

Description

PriorEffDt returns the value of the specified field from the record with the prior effective date. This function is only valid for effective-dated records.

If the record contains an effective sequence number field, the value of this field is compared along with the effective date field when the prior effective date/effective record sequence is determined. Therefore, if there is an effective sequence number, it's possible that the effective date field will be the same as the current record, but the sequence number would be earlier.

If a prior record does not exist, the statement is skipped.

Example

```
If CURRENCY_CD = PriorEffdt(CURRENCY_CD) Then
    Evaluate ACTION
    When = "PAY"
        If ANNUAL_RT = PriorEffdt(ANNUAL_RT) Then
            Warning MsgGet(1000, 27, "Pay Rate Change action is chosen and Pay
Rate has not been changed.");
            End-if;
            Break;
        When = "DEM"
            If ANNUAL_RT >= PriorEffdt(ANNUAL_RT) Then
                Warning MsgGet(1000, 29, "Demotion Action is chosen and Pay Rate has
not been decreased.");
            end-if;
        When-other
    End-evaluate;
    WinMessage("This message appears after executing either of the BREAK
statements or after all WHEN statements are false");
End-if;
```

Related Topics

NextEffDt, NextRelEffDt, PriorRelEffDt

PriorRelEffDt

Syntax

```
PriorRelEffDt(search_field, fetch_field)
```

Description

PriorRelEffDt locates the prior occurrence of the *search_field* with the prior effective date (and effective sequence number if specified) and then returns the value of the specified *fetch_field* corresponding to the *search_field*. The return value is an Any data type. Typically, this function is used to retrieve values for related display fields.

This function is only valid for effective dated records. If a prior record does not exist, then the statement is skipped.

Related Topics

NextEffDt, NextRelEffDt , PriorEffDt

PriorValue

Syntax

```
PriorValue(fieldname)
```

Description

PriorValue is used in FieldEdit and FieldChange PeopleCode to obtain the prior value of a buffer field that the user just changed. It returns the value that was in the buffer field before the user changed it, *not* the value of the field the last time the component was saved.

PriorValue gives correct results only in FieldEdit and FieldChange PeopleCode, and only for the buffer field where the function is executing. If you pass another field name to the function, it returns the current value of the buffer field, not the prior value.

Returns

Returns an Any value equal to the value that was in the current buffer field immediately prior to the last edit.

Parameters

<i>fieldname</i>	The name of the record field. For correct results, this must be the name of the field where the call to PriorValue executes.
-------------------------	---

Example

The following example from FieldChange PeopleCode gets the prior value of a field:

```
&PRIOR = PriorValue(QUANTITY);

DERIVED_TEST.TOTAL_AMT = (DERIVED_TEST.TOTAL_AMT - &PRIOR) + QUANTITY;
```

Related Topics

CurrentRowNumber

Product

Syntax

```
Product(numlist)
```

where *numlist* is an arbitrary-length list of numbers in the form

```
n1 [, n2]...
```

Description

Product multiplies all the numbers in *numlist* and returns the product as a Number data type. The numbers in the list can be any number expressed as a number, variable, or expression.

Returns

Returns a Number value equal to the product of the numbers in *numlist*.

Example

The example sets &N2 to 96:

```
&N2 = Product (4,80,0.3);
```

Related Topics

Fact

Prompt

Syntax

```
Prompt(title, heading, fieldlist)
```

where *fieldlist* is an arbitrary-length list of fields in the form:

```
field1 [, label1 [, tempvar1]] [, field2 [, label2 [, tempvar2]]]...
```

Note that the *label* parameter is now required before the temporary variable.

Description

Prompt displays a page prompting the user to insert values into one or more text boxes. If the user cancels the page, any entered values are discarded and the function returns False. When the prompt page is displayed, the text boxes are initially filled with default values from the fields in *fieldlist*. The user can change the values in the text boxes, then if the user clicks OK, the values are placed either into the buffer for the appropriate field, or into a temporary variable, if a *tempvar* for that field is provided in the function call.

Prompt is a think-time function, and as such cannot be used during the PeopleCode attached to the following events:

- SavePreChange
- Workflow
- RowSelect
- SavePostChange

Prompt should also not be called in any PeopleCode event that fires as a result of a **ScrollSelect**, **ScrollSelectNew**, **RowScrollSelect** or **RowScrollSelectNew** function call.

Returns

Optionally returns a Boolean value:

- False if the user clicks the Cancel button.
- True if the user clicks the OK button.

Parameters

<i>title</i>	Used as the title for the page.
<i>heading</i>	Displayed in the page above the fields. If a zero-length string ("") is passed, the heading line is omitted in the page.
<i>fieldlist</i>	A list of one or more fields; each field in the list consists of a <i>[recordname].fieldname</i> followed by an optional label and an optional temporary variable for storing the input value. The label parameter is required if you supply the temporary variable parameter.
<i>field</i>	The name of the field being prompted for, the form <i>[recordname].fieldname</i> .
<i>label</i>	Optional label for the prompted field. If this parameter is omitted, the field RFT Long value is used. This parameter is required before the <i>tempvar</i> parameter.
<i>tempvar</i>	Optional temporary variable to receive the user-entered value. If this parameter is omitted, the value is placed into the buffer for the field specified. Using a temp variable allows the PeopleCode program to inspect and process the entered value without affecting the buffer contents.

Example

The following example prompts for a single field, using calls to **MsgGetText** to retrieve values for the window title and prompt, and places the result into FISCAL_YEAR field:

```
Prompt (MsgGetText (5000, 182, " "), MsgGetText (5000, 184, " "), FISCAL_YEAR);
```

To following example places the results of the prompt into a temporary variable:

```
Prompt ("Change Voucher", "", VOUCHER_ID, "Change Voucher ID to",
&NEW_VOUCHER_ID);
```

The following code is in the USA push button FieldChange PeopleCode, and calls for the single field as shown in the page.

```
When = PAGE.PERSONAL_DATA1

/* Administer Global Personnel - USA Flag Btn on PERSONAL_DATA1 Page */
```

```
Prompt("US Social Security Number", "", PERSONAL_DATA.SSN);  
  
Break;
```

Proper

Syntax

Proper (*string*)

Description

Proper capitalizes the first letter in a text *string* and any other letters in a text *string* that follow any character other than another letter. It also converts all other letters in a text *string* to lowercase.

Returns

Returns a String value with the first character of each word capitalized.

Example

The example sets the value of &BADKD to "K. D. Lang".

```
&BADKD = Proper("k. d. LANG")
```

Related Topics

Lower, Upper

PutAttachment

Syntax

PutAttachment (*URLDestination*, *SysFileName*, *LocalFile* [, *LocalDirEnvVar*])

where *URLDestination* can have **one** of the following forms:

URL.*URLname*

OR a string URL, such as

```
ftp://user:password@ftp.ps.com/
```

Description

The **PutAttachment** function transfers a file from the application sever to the file server.



If you're using Windows Client, this function transfers the file from the client.

Specifying non-existing Subdirectories Considerations

If the specified subdirectories don't exist, in the PeopleSoft Internet Architecture, this function tries to create them. In Windows Client, the subdirectories are **not** created, and the functions does not complete.

File Name Considerations

When the file is transferred, the following characters are replaced with an underscore:

- space
- semi-colon
- plus sign
- percent sign
- ampersand
- apostrophe
- exclamation point
- @ sign
- pound sign
- dollar sign

Parameters

URLDestination

A reference to a URL. This can be either a URL name, in the form *URL.URLName*, or a string. This is where the file is transferred to.

SysFileName

The relative path and filename of the file on the file server. This is appended to *URLDestination* to make up the full URL where the file is transferred from. This parameter takes a string value.



The *URLDestination* requires "/" slashes. Because *SysFileName* is appended to the URL, it also requires only "/" slashes. "\" are NOT supported in anyway for either the *URLDestination* or the *SysFileName* parameter.

LocalFile

The name, and possible full path, to the destination file on the application server. This parameter takes a string value.


```
&RADIAN_SIZE = Radians(65.5);
```

The following example returns the value of **pi**, that is, **180** degrees expressed as radians:

```
&PI = Radians(180);
```



This example represents **pi** with a high degree of accuracy, but no computer system can represent irrational numbers exactly. Thus, the results of extended calculations based on **pi** have the potential for a cumulative reduction in precision.

Related Topics

Acos, Asin, Atan, Cos, Cot, Degrees, Sin, Tan

Rand

Syntax

```
Rand( )
```

Description

Rand generates a random number greater than or equal to 0 and less than 1. To generate a random integer that is greater than or equal to 0 but less than x, use **Int(Rand()*x)**.

Returns

Returns a random Number value greater than or equal to 0 and less than 1.

Example

The example sets &RANDOM_NUM to a random value less than 100.

```
&RANDOM_NUM = Int(Rand( ) * 100)
```

Related Topics

Int

RecordChanged

Syntax

The syntax of **RecordChanged** varies, depending on whether you use a scroll path reference or a contextual reference to designate the row being tested.

Using a scroll path reference, the syntax is:

```
RecordChanged(scrollpath, target_row)
```

where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row,]]
RECORD.target_recname
```

To prevent ambiguous references, you can also use **SCROLL**.*scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Using a contextual reference the syntax is:

```
RecordChanged(RECORD.target_recname)
```

A contextual reference specifies the current row on the scroll level designated by **RECORD**.*target_recname*.

An older construction, in which a record field expression is passed, is also supported. The record field is any field in the row where the PeopleCode program is executing (typically the one on which the program is executing).

```
RecordChanged(recordname.fieldname)
```



For more information on scroll path and contextual references, see References Using Scroll Path Syntax and Dot Notation and Understanding Current Context in the chapter Referencing Data in the Component Buffer.

Description

RecordChanged determines whether the data in a specific row has been modified since it was retrieved from the database either by the user or by a PeopleCode program.



This function remains for backward compatibility only. Use the IsChanged Record class property instead. See Data Buffer Access. See also DeleteEnabled Row class property .

This is useful during save processing for making updates conditional on whether rows have changed.



Note on terminology. The word "record" is used in this function name in a misleading way. Remember that this function (like the related functions **RecordDeleted** and **RecordNew**) checks the state of a *row*, not a *record*.

Returns

Returns a Boolean value:

- True if any data in the target row has been changed.
- False if no data in the target row has been changed.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
RECORD.target_recname	The primary scroll record of the scroll level where the row being referenced is located. As an alternative, you can use SCROLL.scrollname .



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Example

This example shows a **RecordChanged** call using a contextual reference:

```
If RecordChanged(RECORD.BUS_EXPENSE_DTL) Then
    WinMessage("Changed row msg from current row.", 64);
End-If;
```

The following example, which would execute on level one, checks rows on level two to determine which have been changed:

```
For &I = 1 To ActiveRowCount(RECORD.BUS_EXPENSE_PER, CurrentRowNumber(1),
RECORD.BUS_EXPENSE_DTL);

    If RecordChanged(RECORD.BUS_EXPENSE_PER, CurrentRowNumber(1),
RECORD.BUS_EXPENSE_DTL, &I) Then

        WinMessage("Changed row message from level one.", 64);

    End-If;

End-For;
```

Related Topics

FieldChanged, RecordDeleted, RecordNew

RecordDeleted

Syntax

The syntax of **RecordDeleted** varies, depending on whether you use a scroll path reference or a contextual reference to designate the row being tested.

Using a scroll path reference, the syntax is:

```
RecordDeleted(scrollpath, target_row)
```

where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row,]]
RECORD.target_recname
```

To prevent ambiguous references, you can also use **SCROLL**.*scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Using a contextual reference the syntax is:

```
RecordDeleted(RECORD.target_recname)
```

A contextual reference specifies the current row on the scroll level designated by **RECORD**.*target_recname*.

An older construction, in which a record field expression is passed, is also supported. The record field is any field in the row where the PeopleCode program is executing (typically the one on which the program is executing).

```
RecordDeleted(recordname.fieldname)
```



For more information on scroll path and contextual references, see References Using Scroll Path Syntax and Dot Notation and Understanding Current Context in the section Referencing Data in the Component Buffer.

Description

RecordDeleted checks to see if a row of data has been marked as deleted, either by an end-user row delete (F8) or by a call to DeleteRow. **RecordDeleted** is useful during save processing to make processes conditional on whether or not a row has been deleted.



This function remains for backward compatibility only. Use the IsDeleted Record class property instead. See Data Buffer Access. See also IsDeleted Row property.

Deleted rows are not actually removed from the buffer until after the component has been successfully saved, so you can check for deleted rows all the way through SavePostChange PeopleCode.

RecordDeleted is not typically used in a loop, because it is simpler to put it on the same scroll level as the rows being checked in SavePreChange or SavePostChange PeopleCode: these events execute PeopleCode on every row in the scroll, so no looping is necessary.



Note on terminology. To avoid confusion, note that this function (like the related functions RecordChanged and RecordNew) checks the state of a *row*, not a *record*.

Returns

Returns a Boolean value:

- True if the target row has been deleted.
- False if the target row has not been deleted.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
RECORD.target_recname	The primary scroll record of the scroll level where the row being referenced is located. As an alternative, you can use SCROLL.scrollname .



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Example

This example shows a **RecordDeleted** call using a contextual reference

```
If RecordDeleted(RECORD.BUS_EXPENSE_DTL) Then

    WinMessage("Deleted row msg from current row.", 64);

End-If;
```

The following example, which would execute on level zero, checks rows on level one to determine which have been deleted:

```
For &I = 1 To TotalRowCount(RECORD.BUS_EXPENSE_PER, CurrentRowNumber(1),
RECORD.BUS_EXPENSE_DTL);

    If RecordDeleted(RECORD.BUS_EXPENSE_PER, CurrentRowNumber(1),
RECORD.BUS_EXPENSE_DTL, &I) Then

        WinMessage("Deleted row message from level one.", 64);

    End-If;

End-For;
```

Note that the loop is delimited by TotalRowCount. **For** loops delimited by ActiveRowCount don't process deleted rows.

Related Topics

FieldChanged, RecordChanged, RecordNew

RecordNew

Syntax

The syntax of **RecordNew** varies, depending on whether you use a scroll path reference or a contextual reference to designate the row being tested.

Using a scroll path reference, the syntax is:

```
RecordNew(scrollpath, target_row)
```

where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row,]]  
RECORD.target_recname
```

To prevent ambiguous references, you can also use **SCROLL**.*scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Using a contextual reference the syntax is:

```
RecordNew(RECORD.target_recname)
```

A contextual reference specifies the current row on the scroll level designated by **RECORD**.*target_recname*.

An older construction, in which a record field expression is passed, is also supported. The record field is any field in the row where the PeopleCode program is executing (typically the one on which the program is executing).

```
RecordNew(recordname.fieldname)
```



For more information on scroll path and contextual references, see References Using Scroll Path Syntax and Dot Notation and Understanding Current Context in the section Referencing Data in the Component Buffer.

Description

RecordNew checks a specific row to determine whether it was added to the component buffer since the component was last saved.



This function remains for backward compatibility only. Use the IsNew Row class property instead. See Data Buffer Access.

This function is useful during save processing to make processes conditional on whether or not a row is new.



Note on terminology. To avoid confusion, remember that this function (like the related functions **RecordChanged** and **RecordDeleted**) checks the state of a *row*, not a *record*. In normal PeopleSoft usage, the word "record" denotes a table-level object (such as a table, view, or Derived/Work record).

Returns

Returns a Boolean value:

- True if the target row is new.
- False if the target row is not new.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
RECORD.target_recname	The primary scroll record of the scroll level where the row being referenced is located. As an alternative, you can use SCROLL.scrollname .



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation

Example

This example shows a **RecordNew** call using a contextual reference:

```
If RecordNew(RECORD.BUS_EXPENSE_DTL) Then

    WinMessage("New row msg from current row.", 64);

End-If;
```

The following example, which would execute on level one, checks rows on level two to determine which have been added:

```
For &I = 1 To ActiveRowCount(RECORD.BUS_EXPENSE_PER, CurrentRowNumber(1),
RECORD.BUS_EXPENSE_DTL);

    If RecordNew(RECORD.BUS_EXPENSE_PER, CurrentRowNumber(1),
RECORD.BUS_EXPENSE_DTL, &I) Then

        WinMessage("New row message from level one.", 64);

    End-If;
```

```
End-For;
```

Related Topics

FieldChanged, RecordChanged, RecordDeleted

RefreshTree

Syntax

```
RefreshTree (Record.bound_recname)
```

Description

RefreshTree updates a dynamic tree. If the value in the record field to which the root node of the tree is bound has changed, then the root node is updated to the new value. This function is used in pages containing an editable edit box bound to the same record field as the root of a dynamic tree control. If a call to **RefreshTree** is placed in the FieldChange event of the record field to which the edit box is bound, the user can update the tree by changing the value in the edit box.



For more information, see Controlling the Root Node of the Dynamic Tree.



Dynamic Tree Controls are not the same as ActiveX Tree View Controls, and the functions used with the Dynamic Tree Controls can't be used with ActiveX Tree View Controls.

Restrictions on Use in Three-Tier Mode and PeopleSoft Internet Architecture

RefreshTree is a client-only function, which limits its use in three-tier mode and in PeopleSoft Internet Architecture applications.

- In three-tier mode **RefreshTree** can be used only in processing groups set to run on the client. If the **RefreshTree** function is called in a processing group running on the application server, a runtime error will occur.
- In PeopleSoft Internet Architecture applications, where all PeopleCode runs on the application server, **RefreshTree** cannot be used.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```
If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;
```

Returns

None.

Parameters

<i>bound_recname</i>	The record field to which the root node of the dynamic tree control is bound.
----------------------	---

Example

This example reverts a dynamic tree control, the root node of which is bound to APPR_RULE_HDR, to its state at page startup:

```
RefreshTree(RECORD.APPR_RULE_HDR)
```

Related Topics

GetTreeNodeValue, GetTreeNodeRecordName, GetSelectedTreeNode, GetSQL and Implementing Dynamic Tree Controls

RemoteCall

Syntax

```
RemoteCall(dispatcher_name [, service_paramlist] [, user_paramlist])
```

where *service_paramlist* and *user_paramlist* are arbitrary-length lists of parameters in the form:

```
var1, val1 [, var2, val2]...
```

Description

RemoteCall is a PeopleCode API function that provides a means of calling a Tuxedo service from a PeopleSoft application. A typical use of Remote Call is to run data-intensive, performance-sensitive programs near or on the database server.



For PeopleTools 8 you can no longer use RemoteCall to start an Application Engine program. You must use CallAppEngine instead.

Because complex PeopleCode processes can now be run on the application server in three-tier mode, the **RemoteCall** PeopleCode function has more limited utility. However, RemoteCall can still be very useful, because it provides a way to take advantage of existing COBOL processes.

- In three-tier mode, **RemoteCall** always runs on the application server.
- In two-tier mode, **RemoteCall** always runs on the client.

This means that it is no longer necessary to set a location for the remote call in Configuration Manager.

Each RemoteCall service can have zero or more standard parameters and any number of user parameters. The standard parameters are determined by the RemoteCall dispatcher, the user parameters by the COBOL program being run.

There is only one RemoteCall dispatcher delivered with PeopleTools 7, PSRCCBL, which executes a COBOL program using the connect information of the current end-user.

RemoteCall can be used from any type of PeopleCode *except* SavePostChange, SavePreChange, Workflow, and RowSelect. However, remote programs that change data should not be run as part of the SaveEdit process, because the remote program may complete successfully even though an error occurs in a later part of the save process. For remote programs that change data, the normal place for them would be in the FieldChange PeopleCode behind a command push button, or in a pop-up menu item.

After you use **RemoteCall**, you may want to refresh your page. The **Refresh** method, on a rowset object, reloads the rowset (scroll) using the current page keys. This causes the page to be redrawn. GetLevel0().Refresh() refreshes the entire page. If you only want a particular scroll to be redrawn, you can refresh just that part.



For more information, see Refresh rowset class method.

Returns

None.

Parameters

The parameters passed to RemoteCall can be broken into three parts: the RemoteCall Dispatcher Name, the standard Parameter Lists for the service, and the User Parameter List for the program being called on the service.

Dispatcher Name

The *dispatcher_name* parameter is a string value that specifies the type of RemoteCall performed. For PeopleTools 7 there is only one RemoteCall dispatcher delivered, PSRCCBL, which executes a COBOL program using the connect information of the current end-user, so the value you pass to this parameter should always be "PSRCCBL". Future versions of PeopleTools may provide support for Red Pepper, SQR, or customer supplied remote calls.

Parameter Lists

Both the standard parameter list and user parameter list have the same form. Think of the parameters passed to the service as being passed as pairs of variable names and values of input and output parameters:

variable_name, value

Where:

- *variable_name* is a string literal or string variable that contains the name of the input or output variable as referenced in the remote program. For example, if the remote program expects a variable named "USERNAME", then the PeopleCode should use "USERNAME" or &VARIABLE (which had been assigned the value "USERNAME").
- For input variables, *value* is the value to be passed to the remote program with the variable name. It can be either a variable or literal with a data type that corresponds to the *variable_name* variable. For output variables, *value* is the value returned to the PeopleCode program from the remote program. It must be a variable in this case, representing the buffer into which the value is returned.

An arbitrary number of parameters can be passed to the service. There is, however, a limitation on the number of variables that can be passed in PeopleCode, which is limited by the size of the PeopleCode parameter stack, currently 128.

In the case of the PSRCCBL dispatcher, there are three standard parameters, listed in the following table:

<i>Dispatcher</i>	<i>Parameter</i>	<i>Required</i>	<i>Description</i>
PSRCCBL	PSCOBOLPROG	Y	Name of the COBOL program to run.
PSRCCBL	PSRUNCTL	N	Run-control parameter to pass to the COBOL program.
PSRCCBL	INSTANCE	N	Process instance parameter to pass to the COBOL program.

User Parameter List

For PSRCCBL, the remote COBOL program must match the user parameters to the usage of its application. The names of the parameters are sent to the server and can be used by the COBOL program. The COBOL program returns any modified (output) parameters by name. Parameters which are not returned are not modified, and any extra returned parameters (that is, parameters beyond the number passed or of different names) are discarded with no effect.

Example

You could use the following PeopleCode to execute the program "CBLPROG1":

```

Rem Set the return code so we are sure it is sent back.
&Returncode = -1;
Rem Set the parameters that will be sent across.
&param1 = "John";
&param2 = "Smith";
Rem Set the standard parameters that indicate program name and run-control.
&RemoteCobolPgm = "CBLPROG1";
/* call the remote function */
RemoteCall ("PSRCCBL",
"PSCOBOLPROG", &RemoteCobolPgm,
"PSRUNCTL", workrec.runctl,
"FirstName", &param1,
"LastName", &param2,
"Returncode", &Returncode,
"MessageSet", &msgset,
"MessageID", &msgid,
"MessageText1", &msgtext1,
"MessageText2", &msgtext2);
if &Returncode <> 0
    WinMessage(MsgGet(&msgset, &msgid, "default message", &msgtext1,
&msgtext2));
end-if;

```

Related Topics

Exec, WinExec, and Using RemoteCall

Repeat

Syntax

```

Repeat
    statement_list
Until logical_expression

```

Description

The **Repeat** loop causes the statements in *statement_list* to be repeated until *logical_expression* is True. Any kind of statements are allowed in the loop, including other loops. A **Break** statement

inside the loop causes execution to continue with whatever follows the end of the loop. If the **Break** is in a nested loop, the **Break** does not apply to the outside loop.

Example

The following example repeats a sequence of statements until a complex Boolean condition is True:

```
Repeat
    &J = &J + 1;
    &ITEM = FetchValue(LOT_CONTROL_INV.INV_ITEM_ID, &J);
    &LOT = FetchValue(LOT_CONTROL_INV.INV_LOT_ID, &J);
Until (&ITEM = &INV_ITEM_ID And
    &LOT = &INV_LOT_ID) Or
    &J = &NUM_LOT_ROWS;
```

Replace

Syntax

```
Replace(oldtext, start, num_chars, newtext)
```

Description

Replace replaces a specified number of characters in a string.

Returns

Returns a String value in which specific characters in *oldtext* are replaced with *newtext*.

Parameters

<i>oldtext</i>	A String value, part of which is to be replaced.
<i>start</i>	A Number designating the position in <i>oldtext</i> from which to start replacing characters.
<i>num_chars</i>	A Number, specifying how many characters to replace in <i>oldtext</i> .
<i>newtext</i>	A String value that replaces <i>num_chars</i> characters.

Example

After the following statement &NEWDATESTSTR will equal "1997":

```
&NEWDATESTSTR = Replace("1996",3,2,"97");
```

If this example, where the number of characters in *newtext* is less than *num_chars*, &SHORTER will equal "txtx":

```
&SHORTER = Replace("txt123",4,3,"x");
```

In this example, where the number of characters in *newtext* is greater than *num_chars*, &LONGER will equal "txtxxx":

```
&LONGER = Replace("txt123",4,3,"xxx");
```

Related Topics

Substitute

Rept

Syntax

```
Rept(str, reps)
```

Description

Rept replicates a text string a specified number of times and combines the result into a single string.

Returns

Returns a String value equal to *str* repeated *reps* times.

Parameters

<i>str</i>	A String value to be replicated.
<i>reps</i>	A Number value specifying how many times to replicate <i>str</i> . If <i>reps</i> is 0, Rept returns an empty string. If <i>reps</i> is not a whole integer, it is truncated.

Example

This example sets &SOMESTARS to "*****".

```
&SOMESTARS = Rept(" ",10);
```

Return

Syntax

```
Return [expression]
```

Description

Return returns from the currently active function; the flow of execution continues from the point where the function was called. If the function returns a result, that is, if a return value is specified in the **Returns** clause of the function definition, *expression* specifies the value to pass back to the caller and must be included. If the function does not return a result, the *expression* is not allowed. If **Return** appears in a main program, it acts the same as **Exit**.

Example

In the example a Boolean return value is specified in the **Returns** clause of the **Function** statement. The **Return** statement returns a True or False value to the calling routine, based on the contents of &UPDATEOK.

```
function run_status_upd(&PROCESS_INSTANCE, &RUN_STATUS) returns boolean;
    &UPDATEOK = SQLExec( )("update PS_PRCs_RQST set run_status = :1 where
process_instance = :2", &RUN_STATUS, &PROCESS_INSTANCE);
    If &UPDATEOK Then
        Return True;
    Else
        Return False;
    End-if;
End-function;
```

Related Topics

Function, Exit

ReturnToServer

Syntax

```
ReturnToServer ({TRUE | FALSE | &NODE_ARRAY})
```

Description

The **ReturnToServer** function returns a value from a PeopleCode application messaging program to the publication or subscription server. You would use this in either your publication or subscription routing code, to either return an array of nodes that the message should be published to, or to do error processing (return False if entire message wasn't received.)

Generally the boolean value will only be used with subscription routing code, while the node of arrays will be used with publication routing code.

If TRUE is returned to the server, the publication will be saved to the database and subscription contracts will be created.

If FALSE is returned, nothing will be written to the database.

Parameters

TRUE | FALSE

Specify TRUE if you want the publication to be saved to the database.

| &NODE_ARRAY

Specify FALSE if you don't want anything written to the database.

Specify an object reference to an array of node names if you want to return a list of nodes to be published to.

Returns

None.

Example

The following is an example of a publication routing rule, which would be in the OnRoutePublication. It is used to create publication contracts.

```
local message &MSG;

local array &NODE_ARRAY;

&MSG = GetMessage();

&EMPLID = &MSG.GetRowset() (1).QA_INVEST_HDR.EMPLID.Value;

&SELECT_SQL = CreateSQL("select PUBNODE from PS_EMPLID_NODE where EMPLID = :1",
&EMPLID);

&NODE_ARRAY = CreateArray();

While &SELECT_SQL.Fetch(&PUBNODE)

    &NODE_ARRAY.Push(&PUBNODE);

End-While;

ReturnToServer(&NODE_ARRAY);
```

The following is an example of a subscription routing rule, which would be placed in the OnRouteSubscribe event:

```
local message &MSG;

&MSG = GetMessage();

&BUSINESS_UNIT = &MSG.GetRowset() (1).PO_HDR.BUSINESS_UNIT.Value;

SQLExec("Select BUSINESS_UNIT From PS_BUSINESS_UNIT where BUSINESS_UNIT =
:1", &BUSINESS_UNIT, &FOUND);

If all(&FOUND) Then

    ReturnToServer(True);

Else

    ReturnToServer(False);

End-if;
```

Related Topics

Defining Message Channels

RevalidatePassword

Syntax

```
RevalidatePassword()
```

Description

Use this function to validate the current end-user password.



Note. This function is being kept for backwards compatability only. Use the Operator method Validate instead. See Operator Class.

This function displays a window prompting the user for the same password that the user signed onto the database with.

The screenshot shows a small dialog box titled 'Revalidate Password'. Inside, it says 'User ID: PTDMO' next to a label. Below that is a text input field labeled 'Password'. At the bottom of the dialog are two buttons: 'OK' and 'Cancel'.

Revalidate Password

Restrictions on Use in PeopleCode Events

Control does not return to the line after **RevalidatePassword** until after the user has filled in a value or pressed enter. This interruption of processing makes **RevalidatePassword** a "think-time" function, (see Think-Time Functions), which means that it shouldn't be used in any of the following PeopleCode events:

- SavePreChange
- Workflow
- RowSelect
- SavePostChange
- Any PeopleCode event that fires as a result of a ScrollSelect, ScrollSelectNew, RowScrollSelect or RowScrollSelectNew function call.

Restrictions on Use in Signon PeopleCode

RevalidatePassword does *not* work in Signon PeopleCode. If you use this function in Signon PeopleCode, you will end up in an infinite loop.

Returns

Returns a numeric value or a constant: you can check for either.

Value	Constant	Meaning
0	%RevalPW_Valid	Password Validated
1	%RevalPW_Failed	Password Validation Check Failed
2	%RevalPW_Cancelled	Password Validation Cancelled

Example

RevalidatePassword is commonly used in the SaveEdit PeopleCode to verify that the user entering the data is the same as the one who signed onto the database.

```

&TESTOP = RevalidatePassword();

Evaluate &TESTOP

/* Password does not match the current value */

When 1

    Error MsgGet(48, 18, "Message Not Found");

    Break;

End-Evaluate;

```

Right

Syntax

```
Right(str [, num_chars])
```

Description

Right determines a specified number of characters on the right side of a string. The function is useful if, for example, you want to get the last set of characters in a zip code or other fixed-length identification string.

Returns

Returns a String value equal to the rightmost *num_chars* character(s) in *str*.

Parameters

<i>str</i>	A String value from which you want to get the rightmost character(s).
<i>num_chars</i>	A Number value, greater than or equal to zero. If <i>num_chars</i> is omitted it is assumed to be equal to 1.

Example

If &ZIP is equal to "90210-4455", the following example sets &SUFFIX to "4455":

```
&SUFFIX = Right (&ZIP, 4)
```

Related Topics

Left

Round**Syntax**

```
Round(dec, precision)
```

Description

Use **Round** to round a decimal number to a specified precision.

Returns

Returns a Number value equal to *dec* rounded up to *precision* decimal places.

Parameters

<i>dec</i>	A Number value to be rounded.
<i>precision</i>	A Number value specifying the decimal precision to which to round <i>dec</i> .

Example

The following examples set the value of &TMP to 2.2, 9, then 24.09:

```
&TMP = Round (2.15, 1) ;
&TMP = Round (8.789, 0) ;
&TMP = Round (24.09372, 2) ;
```

Related Topics

Int, Mod, Truncate

RoundCurrency**Syntax**

```
RoundCurrency(amt, currency_cd, effdt)
```

Description

Different currencies are represented at different decimal precessions. **RoundCurrency** is a rounding function that takes currency precision into account, using a value stored in the CURRENCY_CD_TBL PeopleTools table.



For more information about currencies, see Controlling Currency Display Format.

Returns

Returns a Number value equal to *amt* rounded to the currency precision for *currency_cd*.

Parameters

<i>amt</i>	The amount to be rounded.
<i>currency_cd</i>	The currency code.
<i>effdt</i>	The effective date of currency rounding.

Example

The following example rounds 12.567 to 12.57, using the appropriate currency precision for US Dollars ("USD"):

```
&RESULT = RoundCurrency(12.567, "USD", EFFDT);
```

RowFlush

Syntax

```
RowFlush(scrollpath, target_row)
```

Where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]  
RECORD.target_recname
```

To prevent ambiguous references, you can use **SCROLL.scrollname**, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation

Description

RowFlush is used to remove a specific row from a page scroll and from the component buffer. Rows that are flushed are not deleted from the database.



This function remains for backward compatibility only. Use the FlushRow Rowset method instead. See Data Buffer Access.

RowFlush is a specialized and rarely used function. In most situations, you will want to use **DeleteRow** to remove a row from the component buffer and remove it from the database as well when the component is saved.

Returns

None.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
<i>target_row</i>	The row number of the row to flush.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Example

The following example flushes a row in a view from the component buffer:

```
RowFlush(RECORD.BNK_RCN_DTL_VW, &ROW1);
```

Related Topics

ScrollFlush, DeleteRow

RowScrollSelect

Syntax

```
RowScrollSelect(levelnum, scrollpath,  
                RECORD.sel_recname  
                [, sqlstr [, bindvars]]  
                [, turbo])
```

Where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]  
RECORD.target_recname
```

and where *bindvars* is an arbitrary-length list of bind variables in the form:

```
bindvar1 [, bindvar2]...
```

To prevent ambiguous references, you can use **SCROLL.*scrollname***, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Description

RowScrollSelect is similar to ScrollSelect except that it reads data from the select record into a scroll under a specific parent row, rather than automatically distributing the selected rows under the correct parent rows throughout the buffer.



This function remains for backward compatibility only. Use the Select Rowset method instead. See Data Buffer Access.

You must use the WHERE clause in the SQL string to ensure that only rows that match the parent row are read into the scroll from the select record. Otherwise, all rows will be read in under the specified parent row.



For more information, see ScrollSelect.

Returns

The number of rows read (optional.) This counts only lines read into the specified scroll. It does not include any additional rows read into autoselect child scrolls of the scroll.

Parameters

<i>levelnum</i>	Specifies the scroll level of the scroll area into which selected rows will be read. It can be 1, 2, or 3.
<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.

RECORD . <i>sel_recname</i>	Specifies the select record. The <i>selname</i> record must be defined in Application Designer and SQL created as a table or a view, unless <i>sel_recname</i> and <i>target_recname</i> are the same. The <i>sel_recname</i> record can contain fewer fields than <i>target_recname</i> —although it must contain any key fields to maintain dependencies page records. This allows you to limit the amount of data read into the buffers on the client workstation.
<i>sqlstr</i>	Contains a WHERE clause to restrict the rows selected from <i>sel_recname</i> and/or an ORDER BY clause to sort the rows. The WHERE clause may contain the PeopleSoft SQL platform functions that are used for SQLExec processing, such as %DateIn or %Substring.
<i>bindvars</i>	A list of bind variables to be substituted in the WHERE clause. For further information, see SQLExec .
<i>turbo</i>	Setting this Boolean parameter to True turns on Turbo ScrollSelect.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Example

Here is an example of **RowScrollSelect** using bind variables:

```

If All (QTY_PICKED) Then
    &LEVEL1ROW = CurrentRowNumber(1);
    &LEVEL2ROW = CurrentRowNumber(2);
    &ORDER_INT_LINE_NO = FetchValue(RECORD.SHIP_SUM_INV_VW, &LEVEL1ROW,
ORDER_INT_LINE_NO, &LEVEL2ROW);
    &INV_ITEM_ID = FetchValue(RECORD.SHIP_SUM_INV_VW, &LEVEL1ROW, INV_ITEM_ID,
&LEVEL2ROW);
    &QTY = RowScrollSelect(3, RECORD.SHIP_SUM_INV_VW, CurrentRowNumber(1),
RECORD.SHIP_DTL_INV_VW, CurrentRowNumber(2), RECORD.DEMAND_LOC_INV,
RECORD.DEMAND_LOC_INV, "WHERE BUSINESS_UNIT = :1 AND ORDER_NO = :2 AND
DEMAND_SOURCE = :3 AND SOURCE_BUS_UNIT = :4 AND ORDER_INT_LINE_NO = :5 AND
SCHED_LINE_NO = :6 AND INV_ITEM_ID = :7 AND DEMAND_LINE_NO = :8",
SHIP_HDR_INV.BUSINESS_UNIT, ORDER_NO, DEMAND_SOURCE, SOURCE_BUS_UNIT,
ORDER_INT_LINE_NO, SCHED_LINE_NO, INV_ITEM_ID, DEMAND_LINE_NO, True);
End-If;

```

Related Topics

RowScrollSelectNew, ScrollSelect, ScrollSelectNew, ScrollFlush, **SQLExec**



For more information, see ScrollSelect and Using PeopleCode with PeopleSoft Internet Architecture.

RowScrollSelectNew

Syntax

```
RowScrollSelectNew(levelnum, scrollpath,  
  
                  RECORD.sel_recname,  
  
                  [sqlstr [, bindvars]]  
  
                  [, turbo])
```

Where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]  
RECORD.target_recname
```

where *bindvars* is an arbitrary-length list of bind variables in the form:

```
binvar1 [, binvar2]...
```

To prevent ambiguous references, you can use **SCROLL**.*scrollname*, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Description

RowScrollSelectNew is similar to RowScrollSelect, except that all rows read into the work scroll are marked *new* so they are automatically inserted into the database at Save time.



This function remains for backward compatibility only. Use the SelectNew Rowset method instead. See Data Buffer Access.

This capability can be used, for example, to insert new rows into the database by selecting data using a view of columns from another database tables.

Returns

The number of rows read (optional.) This counts only lines read into the specified scroll. It does not include any additional rows read into autoselect child scrolls of the scroll.

Parameters

<i>level</i>	Specifies the scroll level of the scroll area into which selected rows will be read. It can be 1, 2, or 3.
<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
RECORD , <i>sel_recname</i>	Specifies the select record. The <i>selname</i> record must be defined in the record definition and SQL created as a table or a view, unless <i>sel_recname</i> and <i>target_recname</i> are the same. The <i>sel_recname</i> record can contain fewer fields than <i>target_recname</i> —although it must contain any key fields to maintain dependencies with other page records. This allows you to limit the amount of data read into the buffers on the client workstation.
<i>sqlstr</i>	Contains a WHERE clause to restrict the rows selected from <i>sel_recname</i> and/or an ORDER BY clause to sort the rows. The WHERE clause may contain the PeopleSoft SQL platform functions that are used for SQLExec processing, such as %DateIn or %Substring.
<i>bindvars</i>	A list of bind variables to be substituted in the WHERE clause. The same restrictions that exist for SQLExec exist for these variables. See SQLExec for further information.
<i>turbo</i>	Setting this Boolean parameter to True turns on Turbo ScrollSelect. This will improve the performance of ScrollSelect verbs by as much as 300%, but should be implemented with caution on existing applications.



For more information on scroll path syntax, see [References Using Scroll Path Syntax and Dot Notation](#).

Example

The following example reads rows into the level 2 scroll and marks the rows as new:

```
&QTY = RowScrollSelectNew(2, RECORD.BI_LINE_VW, &ROW1, RECORD.BI_LINE_DST,
RECORD.BI_LINE_DST, "where business_unit = :1 and invoice = :2 and line_seq_num
= :3", BI_HDR.BUSINESS_UNIT, BI_HDR.INVOICE, &CURR_LINE_SEQ);
```

Related Topics

RowScrollSelect, ScrollSelect, ScrollSelectNew, ScrollFlush, SQLExec



For more information, see `ScrollSelect`, `RowScrollSelect` and the section `Using PeopleCode with PeopleSoft Internet Architecture`.

RTrim

Syntax

```
RTrim(source_str [, trim_str])
```

Description

Use the **RTrim** function to remove characters, usually trailing blanks, from the right of a string.

Returns

Returns a String formed by deleting, from the end of *source_str*, all occurrences of each character specified in *trim_str*.

Parameters

<i>source_str</i>	A String from which you want to remove trailing characters.
<i>trim_str</i>	A String consisting of a list of characters, all occurrences of which will be removed from the right of <i>source_str</i> . Characters in <i>trim_str</i> that occur in <i>source_str</i> to the left of any character not in <i>trim_str</i> will not be removed. If this parameter is not specified, " " is assumed.

Example

The following example removes trailing blanks from `&NAME` and places the results in `&TRIMMED`:

```
&TRIMMED = RTrim(&NAME);
```

The following example removes trailing punctuation marks from `REC.INP` and places the results in `&TRIMMED`:

```
&TRIMMED = RTrim(REC.INP, ".,:;!?" );
```

Related Topics

LTrim

ScheduleProcess

Syntax

```
ScheduleProcess(process_type, process_name
```

```
[, run_location] [, run_cntl_id] [, process_instance]
[, run_dttm] [, recurrence_name] [, server_name])
```

Description

ScheduleProcess schedules a process or job, writing a row of data to the Process Request table (PSPRCRQST).



This function remains for backward compatibility only. Use the `CreateProcessRequest` function instead. See `ProcessRequest Class`.

On the client, **ScheduleProcess** translates environment variables and expands Inline Bind Variables contained in strings passed in the function parameters. The Process Scheduler server program expands both server environment strings and predefined Metastrings.



For more information on scheduling processes, see `Process Scheduler`.

Returns

Returns a Number value. If the process is scheduled successfully, **ScheduleProcess** returns zero. A non-zero return code indicates an error condition and the process request is not written to the Process Request Table. The non-zero value is passed back from the scheduled process; and its value is determined by that process.

Parameters

The only required fields for **ScheduleProcess** are *process_type* and *process_name*.

<i>process_type</i>	String specifying process type. To run a job pass "PSJob" in this parameter. "PSJob" is case-sensitive.
<i>process_name</i>	String specifying process name.
<i>run_location</i>	String specifying where the process is run: "1" = client; "2" = server.



This property should only be used when you're scheduling processes. Jobs always run on the server. The value specified for this property will be used *only* if the run location for the pre-defined process is specified as *Both*. Otherwise, any value specified for this property will be ignored.

<i>run_cntl_id</i>	String specifying the run control id.
--------------------	---------------------------------------

<i>process_instance</i>	Pass a temporary variable in <i>process_instance</i> , which the function will fill.
<i>run_dttm</i>	DateTime value specifying the time to schedule the process.
<i>recurrence_name</i>	String specifying the recurrence name.
<i>server_name</i>	String specifying the server name.

Restrictions on Use in Three-Tier Mode and PeopleSoft Internet Architecture

The **ScheduleProcess** *run_location* parameter specifies whether the scheduled process is to be run on the client or on the server.

If **ScheduleProcess** is called in a program running on the application server, the process cannot run on the client; that is, the function's *run_location* parameter cannot be set to client (= "1"). This will cause a runtime error and the transaction will be canceled.

If the PeopleCode program where **ScheduleProcess** is called runs on the application server, then COBOL and SQR processes must be set to run on the server. If the PeopleCode program runs on the client (which could happen in either 2-tier or 3-tier mode), then COBOL or SQR processes can run on either the client or the server.

Inline Bind Variables

Strings passed in the parameter list can contain inline bind variables. Inline bind variables represent any *recordname.fieldname* used in the current component, and are preceded by a colon. For example, to pass the value of the server name contained in the record RPT_RQST_WRK in the field SERVER, you would use the following:

```
&PRCSNAME = "XRFWIN";

&PRCSTYPE = "SQR Report";

&RUNLOCATION = "2"; /* server */

&RUNCNTRLID = &RUN_CNTL_ID;

&RUNDTTM = %DateTime;

&RECURNAME = " ";

&SERVERNAME = :RPT_RQST_WRK.SERVER;

ScheduleProcess(&PRCSNAME, &PRCSTYPE, &RUNLOCATION, &RUNCNTRLID,

&PRCSINSTANCE, /*Process instance */

&RUNDTTM, &RECURNAME, &SERVERNAME);
```

Metastrings

The following predefined metastrings are defined for use with Process Scheduler definitions. These metastrings are used to replace variables at runtime.

<i>Predefined Metastring</i>	<i>Is Replaced With</i>
%%OPRID%%	User's Signon ID
%%OPRPSWD%%	User's Password
%%ACCESSID%%	Database Access ID
%%ACCESSPSWD%%	Database Access Password
%%DBNAME%%	Database Name
%%INSTANCE%%	Process Instance
%%OUTPUTDEST%%	Output Destination
%%RUNCNTLID%%	Run Control ID
%%DBNAMESRC%%	Upgrade Source Database Name
%%PRCSNAME%%	Process Name
%%SERVER%%	Server entered in Logon Dialog
%%OUTPUTTYPE%%	Output Type (from Process Request Dialog)
%%ACCESSIDSRC%%	Upgrade Source Database Access ID
%%ACCESSPSWDSRC%%	Upgrade Source Database Access Password
%%OPRACCT%%	User Account ID (Allbase Only)
%%ACCESSACCT%%	Owner Account ID (Allbase Only)
%%ACCESSACCTPSWD% %	Access Account Password (Allbase Only)
%%OPRACCTPSWD%%	User Account Password (Allbase Only)



One pair of percent signs is stripped out of predefined metastrings and server environment strings before the process request is created. As a result, all predefined metastrings and server environment strings are enclosed within single percent signs when the request is stored on the database.

Example

The following example schedules an SQR process:

```
If %Page = "DEPARTMENT_TBL" Then
    &ORGCHART = "1";
```

```

    Prompt(" ", "Enter SQR settings", RUN_CNTL_ID, "Run Control ID");
    &ORGCHART_MGR_ID = DEPT_TBL.MANAGER_ID;
    DEPT_TBL.MANAGER_ID = "X";
    DEPT_TBL.MANAGER_ID = &ORGCHART_MGR_ID;
    DoSave();
    &PROCESS_NAME = "ORGCHART";
    &PROCESS_TYPE = "SQR Process";
    &PROCESS_RC = ScheduleProcess(&PROCESS_TYPE, &PROCESS_NAME, "1",
    RUN_CNTL_ID);
    End-If;

```

Related Topics

ChDir, ChDrive, GetCwd, GetEnv, ExpandBindVar, ExpandEnvVar

ScrollFlush

Syntax

```
ScrollFlush(scrollpath)
```

Where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]
RECORD.target_recname
```

To prevent ambiguous references, you can use **SCROLL.scrollname**, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Description

ScrollFlush removes all rows inside the target scroll area and frees its associated buffer. Rows that are flushed are not deleted from the database. This function is often used to clear a work scroll before a call to ScrollSelect.



This function remains for backward compatibility only. Use the Flush Rowset method instead. See Data Buffer Access.

Returns

None.

Parameters

scrollpath A construction that specifies a scroll level in the component buffer.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Example

The following example clears the level-one scroll then fills the level-one and level-two scrolls.

```
/* Throw away all rows */
ScrollFlush(RECORD.DBFIELD_VW);
/* Fill in new values */
&FIELDSEL = "where FIELDNAME like '" | FIELDNAME | "%'";
&ORDERBY = " order by FIELDNAME";
ScrollSelect(1, RECORD.DBFIELD_VW, RECORD.DBFIELD_VW, &FIELDSEL | &ORDERBY);
ScrollSelect(2, RECORD.DBFIELD_VW, RECORD.DBFIELD_LANG_VW,
RECORD.DBFIELD_LANG_VW, &FIELDSEL | " and LANGUAGE_CD = :1" | &ORDERBY,
DBFIELD_SRCH.LANGUAGE_CD);
```

Related Topics

RowFlush, RowScrollSelect, RowScrollSelectNew, ScrollSelect, ScrollSelectNew

ScrollSelect

Syntax

```
ScrollSelect(levelnum,
            [RECORD.level1_recname, [RECORD.level2_recname,]]
            RECORD.target_recname, RECORD.sel_recname
            [, sqlstr [, bindvars]]
            [, turbo])
```

where *bindvars* is an arbitrary-length list of bind variables in the form:

```
bindvar1 [, bindvar2]...
```

Description

ScrollSelect, like the related ScrollSelect functions (ScrollSelectNew, RowScrollSelect, and RowScrollSelectNew) reads data from database tables or views into a scroll area on the active page.



This function remains for backward compatibility only. Use the Select Rowset method instead. See Data Buffer Access.

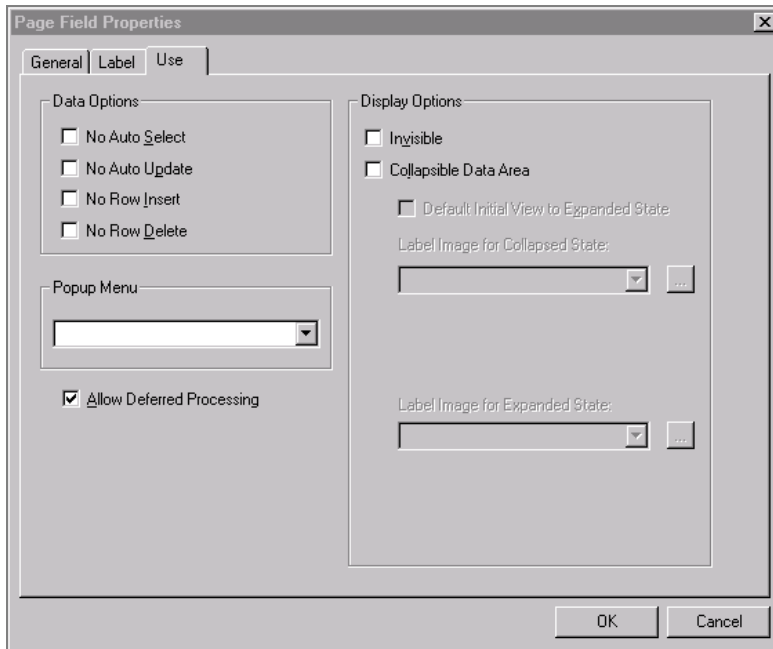
ScrollSelect automatically places child rows in the target scroll area under the correct parent row in the next highest scroll area. If it cannot match a child row to a parent row an error will occur.

ScrollSelect selects rows from a table or view and adds the rows to a scroll area of a page. Let's call the record definition of the table or view that it selects from the *select record*; and let's call the record definition normally referenced by the scroll area (as specified in the page definition) the *default scroll record*. The select record can be the same as the default scroll record, or it can be a different record definition that has compatible fields with the default scroll record. If you define a select record that differs from the default scroll record, you can restrict the number of fields that are loaded into the buffers on the client work station by including only the fields that you actually need.

ScrollSelect accepts an SQL string that can contains a WHERE clause restricting the number of rows selected into the scroll area. The SQL string can also contain an ORDER BY clause to sort the rows.

ScrollSelect functions generate a SQL SELECT statement at run time, based on the fields in the select record and WHERE clause passed to them in the function call. This gives ScrollSelect functions a significant advantage over SQLExec: they allow you to change the structure of the select record without affecting the PeopleCode, unless the field affected is referred to in the WHERE clause string. This can make the application easier to maintain.

Often, ScrollSelect is used to select rows into a *work scroll*, which is sometimes hidden using HideScroll. A work scroll is a scroll in which the **No Auto Select** option is selected so that PeopleTools does not automatically populate the scroll at page startup. You can right-click on the scroll area then select the scroll's **No Auto Select** attribute check box in the Page Field Properties dialog box:



Page Field Property Use Tab for Scroll Area

Depending on how you intend the scroll to be used by the end-user, you may also want to select **No Auto Update** to prevent database updates, and prevent row insertions or deletions in the scroll area by selecting **No Row Insert** or **No Row Update**.

To call **ScrollSelect** at page startup, place the function call in the RowInit event of a key field on the parent scroll record. For example, if you want to fill scroll level one, place the call to **ScrollSelect** in the RowInit event of a field on level zero.

Returns

The number of rows read (optional.) This counts only lines read into the specified scroll. It does not include any additional rows read into autoselect child scrolls of the scroll.

Parameters

<i>levelnum</i>	Specifies the level of the scroll level to be filled (the target scroll area. The value can be 1, 2, or 3.
<i>target_recname</i>	Specifies a record identifying the target scroll, into which the selected rows will be read. If <i>target_recname</i> is on scroll level 2, it must be preceded by a RECORD.level1_recname . If it is on level 3, you must specify both RECORD.level1_recname and RECORD.level2_recname .

RECORD . <i>sel_recname</i>	Specifies the select record. The <i>selname</i> record must be defined in Application Designer and SQL created (using Build, Project) as a table or a view, unless <i>sel_recname</i> and <i>target_recname</i> are the same. The <i>sel_recname</i> record can contain fewer fields <i>target_recname</i> —although it must contain any key fields to maintain dependencies with other page records. This allows you to limit the amount of data read into the buffers on the client workstation.
<i>sqlstr</i>	Contains a WHERE clause to restrict the rows selected from <i>sel_recname</i> and/or an ORDER BY clause to sort the rows. The WHERE clause can contain the PeopleSoft meta-SQL functions such as %Datein() or %CurrentDateIn. It can also contain inline bind variables.
<i>bindvars</i>	A list of bind variables to be substituted in the WHERE clause. The same restrictions that exist for SQLExec exist for these variables. See SQLExec for further information.
<i>turbo</i>	Setting this Boolean parameter to True turns on Turbo ScrollSelect. This will improve the performance of ScrollSelect verbs by as much as 300%, but should be implemented with caution on existing applications, which may depend on the behavior of ScrollSelect without Turbo.

Example

This example uses WHERE clauses to reset the rows in two scroll areas:

```
&FIELD_CNT = ActiveRowCount (DBFIELD_VW.FIELDNAME);
For &I = 1 to &FIELD_CNT;
  If RecordChanged(DBFIELD_VW.FIELDNAME, &I, DBFIELD_LANG_VW.FIELDNAME, 1)
  Then
    &FIELDNAME = FetchValue(DBFIELD_VW.FIELDNAME, &I);
    &RET = WinMessage("Descriptions for " | &FIELDNAME | " have been changed.
Discard changes?", 289, "DBField Changed!");
    If &RET = 2 Then
      /* Cancel selected */
      Exit;
    End-if;
  End-if;
End-for;
/* Now throw away all rows */
ScrollFlush(RECORD.DBFIELD_VW);
/* Fill in new values */
&FIELDSEL = "where FIELDNAME like '" | FIELDNAME | "%'";
&ORDERBY = " order by FIELDNAME";
&QTY1 = ScrollSelect(1, RECORD.DBFIELD_VW, RECORD.DBFIELD_VW, &FIELDSEL |
&ORDERBY);
&QTY2 = ScrollSelect(2, RECORD.DBFIELD_VW, RECORD.DBFIELD_LANG_VW,
```

```
RECORD.DBFIELD_LANG_VW, &FIELDSEL | " and LANGUAGE_CD = :1" | &ORDERBY,  
DBFIELD_SRCH.LANGUAGE_CD);
```

Related Topics

RowScrollSelect, RowScrollSelectNew, ScrollFlush, ScrollSelectNew, SQLExec



For more information, see Using PeopleCode with PeopleSoft Internet Architecture.

ScrollSelectNew

Syntax

```
ScrollSelectNew(levelnum,  
  
                [RECORD.level1_recname,  
  
                [RECORD.level2_recname, ]]  
  
                RECORD.target_recname,  
  
                RECORD.sel_recname [, sqlstr [, bindvars]]  
  
                [, turbo])
```

and where *bindvars* is an arbitrary-length list of bind variables in the form:

```
bindvar1 [, bindvar2]...
```

Description

ScrollSelectNew is similar to ScrollSelect, except that all rows read into the work scroll are marked *new* so they are automatically inserted into the database at Save time.



This function remains for backward compatibility only. Use the SelectNew Rowset class method instead. See Data Buffer Access.

This capability can be used, for example, to insert new rows into the database by selecting data using a view of columns from other database tables.

Returns

The number of rows read (optional.) This counts only lines read into the specified scroll. It does not include any additional rows read into autoselect child scrolls of the scroll.

Parameters

<i>levelnum</i>	Specifies the level of the scroll level to be filled (the target scroll area). The value can be 1, 2, or 3.
<i>target_recname</i>	Specifies a record identifying the target scroll, into which the selected rows will be read. If <i>target_recname</i> is on scroll level 2, it must be preceded by a RECORD.level1_recname . If it is on level 3, you must specify both RECORD.level1_recname and RECORD.level2_recname .
RECORD.sel_recname	Specifies the select record. The <i>selname</i> record must be defined in Application Designer and SQL created as a table or a view (using Build, Project), unless <i>sel_recname</i> and <i>target_recname</i> are the same. The <i>sel_recname</i> record can contain fewer fields <i>target_recname</i> —although it must contain any key fields to maintain dependencies with other records on the page. This allows you to limit the amount of data read into the buffers on the client workstation.
<i>sqlstr</i>	Contains a WHERE clause to restrict the rows selected from <i>sel_recname</i> and/or an ORDER BY clause to sort the rows. The WHERE clause may contain the PeopleSoft SQL platform functions that are used for SQLExec processing, such as %Datein or %Substring.
<i>bindvars</i>	A list of bind variables to be substituted in the WHERE clause. The same restrictions that exist for SQLExec exist for these variables. See SQLExec for further information.
<i>turbo</i>	Setting this Boolean parameter to True turns on Turbo ScrollSelect. This will improve the performance of ScrollSelect verbs by as much as 300%. But should be implemented with caution on existing applications.

Example

The following statement selects rows from DATA2 and reads them into scroll level one on the page. If the end-user saves the page, these rows will be inserted into DATA1:

```
&QTY = ScrollSelectNew(1, RECORD.DATA1, RECORD.DATA2,
    "Where SETID = :1 and CUST_ID = :2",
    CUSTOMER.SETID, CUSTOMER.CUST_ID);
```

Related Topics

RowScrollSelect, RowScrollSelectNew, ScrollSelect, ScrollFlush, SQLExec



For more information, see ScrollSelect and Using PeopleCode with PeopleSoft Internet Architecture.

Second

Syntax

`Second(timevalue)`

Description

Use the **Second** function to extract the seconds component of a Time value.

Returns

Returns a Number equal to the seconds part of *timevalue*.

Parameters

<i>timevalue</i>	A Time value from which to extract seconds.
------------------	---

Example

Assume that &TIMEOUT contains Time value of 16:48:21. The following would set &SECS to 21:

```
&SECS = Second(&TIMEOUT);
```

Related Topics

Hour, Minute

SendMail

Syntax

```
SendMail(flags, recipients, CCs, BCCs, subject, text,  
         [, attachment_filenames][, attachment_titles])
```

Description

You can use **SendMail** to send an email message from a PeopleSoft page. The **SendMail** function supports VIM and MAPI email subsystems for two-tier architecture, and SMTP for three-tier. The APIs that support these subsystems must be present on the system for the function to work.



The code that actually calls **SendMail** determines the run location (two-tier or three-tier) *not* the originating code. For example, if an online page (two-tier) calls a Component Interface (three-tier) that calls SendMail, the program is running in three-tier, not two-tier mode. The only exception to this is if SendMail is used in a program running on the batch server (for example, if an Application Engine PeopleCode program used SendMail.) In a batch server environment, SendMail *always* uses SMTP.

The function sends a message using standard mail options, including recipient, CC, BCC, subject, and the text of the note. The message can include attached files, for which you supply fully qualified file names (that is, file names with paths) and titles (which appear in place of the fully qualified filename in the message).

The *flags* parameter of this function and its return codes are platform dependent: the values are different depending on whether the platform is using the VIM or MAPI mail API (the SMTP mail API ignores any additional flags.) For this reason, **SendMail** should be used only in customizations that run on a known platform, if it is used at all. The preferred, platform-independent method for sending email from PeopleCode is to define an email routing as part of a business event, then trigger the business event using TriggerBusinessEvent.



For more information, see TriggerBusinessEvent and Defining Event Triggers.

Returns

Returns a Number value, which, if not one of the following general return codes, is platform-dependent.

General Return Codes

<i>Return Code</i>	<i>Description</i>
0	No Error
-1	No mail interface installed.

VIM Return Codes

<i>Return Code</i>	<i>Description</i>
0	SMISTS_SUCCESS
1	SMISTS_FAILURE
4	SMISTS_ATTACHMENT_NOT_FOUND
8	SMISTS_INSUFFICIENT_MEMORY
15	SMISTS_NAME_NOT_FOUND
16	SMISTS_NOT_SUPPORTED
22	SMISTS_OPEN_FAILURE

<i>Return Code</i>	<i>Description</i>
128	SMISTS_INVALID_ADDR_BOOK
129	SMISTS_TOO_MANY_FILES
130	SMISTS_TOO_MANY_RECIPIENTS
131	SMISTS_USER_CANCEL

MAPI Return Codes

<i>Return Code</i>	<i>Description</i>
1	MAPI_USER_ABORT
2	MAPI_E_FAILURE
3	MAPI_E_LOGIN_FAILURE
4	MAPI_E_DISK_FULL
5	MAPI_E_INSUFFICIENT_MEMORY
6	MAPI_E_ACCESS_DENIED
8	MAPI_E_TOO_MANY_SESSIONS
9	MAPI_E_TOO_MANY_FILES
10	MAPI_E_TOO_MANY_RECIPIENTS
11	MAPI_E_ATTACHMENT_NOT_FOUND
12	MAPI_E_ATTACHMENT_OPEN_FAILURE
13	MAPI_E_ATTACHMENT_WRITE_FAILURE
14	MAPI_E_UNKNOWN_RECIPIENT
15	MAPI_E_BAD_RECIPYTYPE
16	MAPI_E_NO_MESSAGES
17	MAPI_E_INVALID_MESSAGE
18	MAPI_E_TEXT_TOO_LARGE
19	MAPI_E_INVALID_SESSION
20	MAPI_E_TYPE_NOT_SUPPORTED
21	MAPI_E_AMBIGUOUS_RECIPIENT
22	MAPI_E_MESSAGE_IN_USE
23	MAPI_E_NETWORK_FAILURE
24	MAPI_E_INVALID_EDITFIELDS
25	MAPI_E_INVALID_RECIPS
26	MAPI_E_NOT_SUPPORTED



Additional VIM and MAPI error codes may be generated, depending on your email provider.

There are no special return codes for SMTP.

Parameters

flags

An integer value passed directly to the mail system API to control mail system options. The value passed in this parameter is platform-dependent. The SMTP mail API ignores this parameter.

The following values can be used with MAPI. You can combine different options by adding any of the following values and passing the sum. Pass zero to turn all of the options off.

Value	Description
1	Allows a logon interface if required.
2	Prevents Simple MAPI from using an existing shared session if one is present.
8	Displays a dialog box which allows the user to create or modify the message.

recipients

A string consisting of a semicolon-separated list of email addresses containing the names of the message's primary recipients.

CCs

A string consisting of a semicolon-separated list of email addresses that will be sent copies of the message.

BCCs

A string consisting of a semicolon-separated list of email addresses that will be sent copies of the message. These recipients won't appear on the message list.

subject

A string containing the text that will appear in the message's Subject field.

text

The text of the message.

attachment_filenames

A string consisting of a semicolon-separated list of fully qualified filenames, containing the complete path to the file and the filename itself.

attachment_titles

Another semicolon-separated list containing titles for each of the files provided in the *attachment_filenames* parameter. The titles will appear near the attachment icons in place of the fully qualified filename.

Example

The following example sets up several variables that are then used to construct an e-mail message that includes two attachments:

```
&MAIL_FLAGS = 0;
&MAIL_TO = "dduffield@peoplesoft.com;sweet_pea@peoplesoft.com";
&MAIL_CC = "";
&MAIL_BCC = "mom@aol.com";
&MAIL_SUBJECT = "Live long and prosper!";
&MAIL_TEXT = "Please read my attached CV. You will be amazed and hire me
forthwith.";
&MAIL_FILES = "c:\mydocs\resume.doc;c:\mydocs\coverlet.doc";
&MAIL_TITLES = "My CV;READ ME";
&RET = SendMail(&MAIL_FLAGS, &MAIL_TO, &MAIL_CC, &MAIL_BCC, &MAIL_SUBJECT,
&MAIL_TEXT, &MAIL_FILES, &MAIL_TITLES);
if not (&RET = 0) then
    WinMessage("Return status from mail = " | &RET);
end-if;
```

SetAuthenticationResult

Syntax

```
SetAuthenticationResult(AuthResult [, UserId] [, ResultDocument] [,
PasswordExpired])
```

Description

The **SetAuthenticationResult** function is used in signon PeopleCode to customize the authentication process. It allows the developer using Signon PeopleCode to implement additional authentication mechanisms beyond the basic PeopleSoft ID and Password authentication.

Parameters

AuthResult

Specify whether the authentication is successful. This parameter takes a boolean value. If True is used, the end-user of the UserId specified on the Signon page is allowed access to the system.

UserId

Specify the UserId of the user signing on. The default value is the UserId entered on the signon page. This parameter takes a string value.

<i>ResultDocument</i>	Specify an HTML document explaining the result of the authentication. This document is displayed to the user after either a successful or unsuccessful signon. The default value is null. This parameter takes a string value.
<i>PasswordExpired</i>	<p>Specify if the users password has expired. The valid values are:</p> <ul style="list-style-type: none"> • False (default) if the user's password hasn't expired. • True if the user's password has expired <p>If this value is specified as True, the user is allowed to log in, but is only able to access a limited portion of the system: just enough to change their expired password.</p>

Returns

A Boolean value: True if function completed successfully, False otherwise.

Example

```
If updateUserProfile(%SignonUserId, %SignonUserPswd, &array_attribs) Then
    SetAuthenticationResult(True, &SignonUserID, "", False);
End-If;
```

Related Topics

Understanding PeopleSoft Signon, %ResultDocument, %AuthenticationToken

SetChannelStatus

Syntax

```
SetChannelStatus(ChannelName, Status)
```

Description

The **SetChannelStatus** allows you set the status of the specified channel. You could use this function to restart a channel that had been paused, or pause a running channel.

Parameters

<i>ChannelName</i>	Specify the channel name.
--------------------	---------------------------

Status

Specify the status you want to set the channel to. The valid values are:

- 1 for Run
- 2 for Pause

Returns

A boolean value: True if the channel status was changed successfully. False otherwise.

Example

```
/* User has clicked on a channel to change its status */

If CHNL_STATUS = "1" Then
    rem running, so pause;
    &status = 2;
Else
    rem paused. So run;
    &status = 1;
End-If;

If SetChannelStatus(AMM_CHNL_SECVW.CHNLNAME, &status) Then
    CHNL_STATUS = String(&status);
Else
    MessageBox(0, MsgGetText(117, 1, ""), 117, 22, "");
End-If;
```

Related Topics

Message Class, Introduction to Application Messaging

SetControlValue
Syntax

```
SetControlValue(Value, PageName, PageFieldName [, RowNumber])
```

Description

The **SetControlValue** function enables you to set an override string on the current field so that it simulates an end-user entering data.

When a page is refreshed after a PeopleCode program completes, each field value gets set from the buffer. However, if you use this function to specify an override string for a field, the value you specify will be used *instead* of the value in the buffer. This value is inserted directly into the control on the page, as if the end-user typed it in. The field buffer remains unchanged. All validations, FieldEdit and FieldChange PeopleCode will run immediately.

This function can be used in the following scenario: Suppose you have a text field that has a menu pop-up associated with it. The end-user can use a secondary page to select an item to be used for the value. From the menu PeopleCode, you can verify that the value is valid, but the field doesn't turn red and the end-user can leave the field. This could potential mean saving the page with bad data. You can use this function after the secondary page is dismissed. This causes the same edits to be run as if the end-user had typed in the value.

This function will not work for radio button or check box controls.

Considerations with Field Verification

SetControlValue only sets the value of the field. If you specify an incorrect value, SetControlValue has an error at runtime.

For example, suppose you are setting a value like "1900-01-01" into a date field that is expecting the format 01/01/1900. If the end-user entered 1900-01-01 they would get an error, so SetControlValue causes an error with this value also. You may want to use a value in the format the end-user might type. You can get this value by using the FormattedValue method on a field. For example:

```
&DATE_IN_EFFECT = SF_PRDN_AREA_IT.DATE_IN_EFFECT.FormattedValue;

...

SetControlValue(&DATE_IN_EFFECT, %Page, "DATE_IN_EFFECT", &OCCURSNUM);
```

The FormattedValue function converts the field value from the PeopleSoft representation to the representation the end-user would see and enter.

Restrictions on use with Windows Client

The *PageName*, *PageFieldName* and *RowNumber* parameters are not required for Windows Client. If specified, they will be ignored.

Validations do *not* run immediately in Windows client. Only after a value is inserted into the control, and the user tries to tab out, will the validations be run.

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```

If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

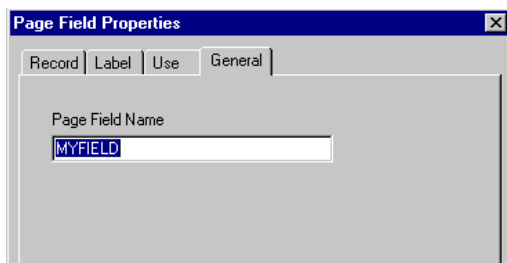
    . . .

End-if;

```

Parameters

<i>Value</i>	Specify an override value on the current field. This parameter takes a string value.
<i>pagename</i>	Specify the name of page where the field exists.
<i>pagefieldname</i>	Specify the page field name. This is <i>not</i> the name of the field. This is the name that is assigned to the field in Application Designer, on the page field properties.



Page Field Name dialog box

<i>RowNumber</i>	Specify the row number of the field. The default value is 1 if this parameter isn't set.
------------------	--

Returns

None.

Example

```

Declare Function item_seach PeopleCode FUNCLIB_ITEM.INV_ITEM_ID FieldFormula;

&SEARCHREC = "PS_" | RECORD.MG_ITEM_OWNI_VW;

item_seach("", SF_PRDN_AREA.BUSINESS_UNIT, "ITEM", &SEARCHREC, "", &INV_ITEM_ID,
"");

```

```
SetControlValue(&INV_ITEM_ID);
```

The following example is used in the PeopleSoft Internet Architecture:

```
Declare Function item_search PeopleCode FUNCLIB_ITEM.INV_ITEM_ID FieldFormula;
```

```
Component string &ITEM_ID_SEARCH;
```

```
&ITEMRECNAME = "PS_" | Record.MG_ITEM_PDO_VW;
```

```
item_serach("", EN_PDO_WRK.BUSINESS_UNIT, "ITEM", &ITEMRECNAME, "",  
&INV_ITEM_ID, "");
```

```
If All(&INV_ITEM_ID) Then
```

```
    Evaluate &ITEM_ID_SEARCH
```

```
    When "F"
```

```
        SetControlValue(&INV_ITEM_ID, Page.EN_PDO_COPY, "FROM_ITEMID")
```

```
    When "T"
```

```
        SetControlValue(&INV_ITEM_ID, Page.EN_PDO_COPY, "TO_ITEMID")
```

```
    End-Evaluate;
```

```
End-If;
```

SetCursorPos

Syntax

```
SetCursorPos(Page.pagename, scrollpath, target_row, [recordname.]fieldname)
```

where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]]  
RECORD.target_recname
```

To prevent ambiguous references, you can use **SCROLL**.*scrollname*, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Description

SetCursorPos places the focus in a specific field anywhere in the current component. To transfer to a page outside the current component, use Transfer.

You can use the **SetCursorPos** function in combination with an **Error** or **Warning** in SaveEdit to place the focus on the field that caused the error or warning condition. You need to call **SetCursorPos** *before* an **Error** statement, because **Error** in SaveEdit terminates all save processing, including the program from which it was called.

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```
If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;
```

Returns

None.

Parameters

<i>Pagename</i>	The name of the page specified in the page definition, preceded by the keyword Page . The <i>pagename</i> page must be in the current component. You can also pass the %page system variable in this parameter (without the Page reserved word).
<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
<i>[recordname.]fieldname</i>	Specify a field designating the record and field in the scroll where you want to place the cursor.
<i>target_row</i>	The row number of the row in which you want to place the cursor.

Example

The following example places the cursor in the appropriate field if a SaveEdit validation fails. Note the use of the %page system variable to get the page name. Note also that **SetCursorPos** is called before **Error**.

```
If None(&ITEM_FOUND) Then
    SetCursorPos(%Page, INV_ITEM_ID, CurrentLineNumber());
    Error (MsgGet(11100, 162, "Item is not valid in the order business unit.",
INV_ITEM_ID, CART_ATTRIB_INV.ORDER_BU));
End-If;
```

The following example is similar, but uses the **Page** reserved word and page name:

```
If %Component = COMPONENT.BUS_UNIT_TBL_GL Then
    SetCursorPos(PAGE.BUS_UNIT_TBL_GL1, DEFAULT_SETID, CurrentLineNumber());
End-If;
Error MsgGet(9000, 165, "Default TableSet ID is a required field.");
```

Related Topics

TransferPage

SetDefault

Syntax

```
SetDefault([recordname.]fieldname)
```

Description

SetDefault sets a field to a null value, so that the next time default processing occurs, it is set to its default value: either a default specified in its record field definition or one set programmatically by PeopleCode located in a FieldDefault event. If neither of these defaults exist, the Component Processor leaves the field blank.



This function remains for backward compatibility only. Use the SearchDefault Field class property instead. See Data Buffer Access.

Blank numbers correspond to zero on the database. Blank characters correspond to a space on the database. Blank dates and long characters correspond to NULL on the database. **SetDefault** gives each field data type its proper value.

Where to use SetDefault

If a PeopleCode program or function executes the SetDefault built-in on a field that does *not* exist in the component buffer, the remainder of the program or function is skipped. In the case of a function, execution of the calling program continues with the next statement after the call to the function. However, if the program containing the SetDefault call is at the "top level", meaning that it was called directly from the component processor or application engine runtime, it exits.

Therefore, if you want to control the behavior of `SetDefault`, you should encapsulate any calls to this built-in function inside your own functions. This enables your overall programs to continue, whether or not the `SetDefault` succeeds.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Parameters

<i>[recordname.]fieldname</i>	Specify a field designating the fields that you want to set to its default value.
-------------------------------	---

Example

This example resets the PROVIDER to its null value. This field will be reset to its default value when default processing is next performed:

```
If COVERAGE_SELECT = "W" Then
  SetDefault (PROVIDER) ;
End-if;
```



For more information about default processing, see [Default Processing](#).

Related Topics

`SetDefaultAll`, `SetDefaultNext`, `SetDefaultPrior`

SetDefaultAll

Syntax

```
SetDefaultAll ( [recordname.] fieldname )
```

Description

SetDefaultAll sets all occurrences of the specified *recordname.fieldname* within a scroll to a blank value, so that the next time default processing is run these fields will be set to their default value, as specified by the record definition, or one set programmatically by PeopleCode located in a `FieldDefault` event. If neither of these defaults exist, the Component Processor leaves the field blank.



This function remains for backward compatibility only. Use the `SetDefault Rowset` method instead. See [Data Buffer Access](#). See also [SearchDefault Field](#) method.

Example

The following example sets the fields TO_CUR and CUR_EXCHNG_RT to their default values on every row of the scroll area where the PeopleCode is run:

```
SetDefaultAll (TO_CUR) ;  
SetDefaultAll (CUR_EXCHNG_RT) ;
```

Related Topics

SetDefault, SetDefaultNext, SetDefaultNextRel, SetDefaultPrior, SetDefaultPriorRel

SetDefaultNext

Syntax

```
SetDefaultNext ([recordname.] fieldname)
```

Description

SetDefaultNext locates the next occurrence of the *recordname.fieldname* with the next effective date (and effective sequence number if specified) and sets the field to a blank value, so that the next time default processing is run this field will be set to its default value, as specified by the record definition, or one set programmatically by PeopleCode located in a FieldDefault event. If neither of these defaults exist, the Component Processor leaves the field blank.

SetDefaultNext is typically used to reset values within a scroll which are calculated within default PeopleCode based on a next value.

This function is only valid for effective dated records. If a next record does not exist, then the statement is skipped.

Related Topics

SetDefaultAll, SetDefaultNextRel, SetDefaultPrior, SetDefaultPriorRel

SetDefaultNextRel

Syntax

```
SetDefaultNextRel (search_field, default_field)
```

Description

SetDefaultNextRel locates the next occurrence of the *search_field* with the next effective date (and effective sequence number if the record contains an effective sequence number) and then sets the value of the specified *default_field* corresponding to the *search_field* to a blank value, so that the next time default processing is run this field will be set to its default value, as specified by the record definition, or one set programmatically by PeopleCode located in a FieldDefault event. If neither of these defaults exist, the Component Processor leaves the field blank.

This function is only valid for effective dated records. If a next record does not exist, then the statement is skipped.

Related Topics

SetDefault, SetDefaultAll, SetDefaultPrior, SetDefaultPriorRel

SetDefaultPrior

Syntax

```
SetDefaultPrior([recordname.]fieldname)
```

Description

SetDefaultPrior locates the prior occurrence of the *recordname.fieldname* with the prior effective date (and effective sequence number if specified) and sets the field to a blank value, so that the next time default processing is run this field will be set to its default value, as specified by the record definition, or one set programmatically by PeopleCode located in a FieldDefault event. If neither of these defaults exist, the Component Processor leaves the field blank.

SetDefaultPrior is typically used to reset values within a scroll which are calculated within FieldDefault PeopleCode based on a next value.

This function is only valid for effective dated records. If a prior record does not exist, then the statement is skipped.

Related Topics

SetDefault, SetDefaultAll, SetDefaultNext, SetDefaultNextRel, SetDefaultPriorRel

SetDefaultPriorRel

Syntax

```
SetDefaultPriorRel(search_field, default_field)
```

Description

SetDefaultPriorRel locates the prior occurrence of the *search_field* with the prior effective date (and effective sequence number if the record contains an effective sequence number) and then sets the specified *default_field* to a blank value, so that the next time default processing is run this field will be set to its default value, as specified by the record definition, or one set programmatically by PeopleCode located in a FieldDefault event. If neither of these defaults exist, the Component Processor leaves the field blank.

This function is only valid for effective dated records. If a next record does not exist, then the statement is skipped.

Related Topics

SetDefault, SetDefaultAll, SetDefaultNext, SetDefaultNextRel, SetDefaultPrior

SetDisplayFormat

Syntax

```
SetDisplayFormat(scrollpath, target_row,
                 [recordname.]fieldname, display_format_name)
```

where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]]
RECORD.target_recname
```

To prevent ambiguous references, you can use **SCROLL.scrollname**, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Description

Use **SetDisplayFormat** to change the display format of Custom Defined Fields at runtime. For instance, you may want to update a custom numeric display to reveal more decimal points.



This function remains for backward compatibility only. Use the DisplayFormat Field property instead. See Data Buffer Access.

Returns

Returns a Boolean value indicating whether the function executed successfully. The return value is not optional.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
<i>target_row</i>	The row number of the target row.
<i>[recordname.]fieldname</i>	The name of the field to change. The field can be on scroll level one, two, or three of the active page. The <i>recordname</i> prefix is required if the call to SetDisplayFormat is in a record definition other than <i>recordname</i> .
<i>display_format_name</i>	The name of the custom display format specified in the field definition.



For more information on scroll path syntax, see [References Using Scroll Path Syntax and Dot Notation](#).

Related Topics

GetStoredFormat

SetLabel

Syntax

```
SetLabel(scrollpath, target_row, [recordname.]fieldname, new_label_text)
```

Where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]]  
RECORD.target_recname
```

To prevent ambiguous references, you can use **SCROLL.scrollname**, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see [References Using Scroll Path Syntax and Dot Notation](#).

Description

SetLabel allows you to change the label text of a page field or grid column heading.



This function remains for backward compatibility only. Use the Label Field property instead. See [Data Buffer Access](#). See also [GetLongLabel](#), [GetShortLabel](#) Field methods.



You can't use this function to set labels longer than 100 characters. If you try to set a label of more than 100 characters, the label will be truncated to 100 characters.

Returns

Optionally returns a Boolean value indicating whether the function completed successfully.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
<i>target_row</i>	The row number of the target row.
[<i>recordname.</i>] <i>fieldname</i>	The name of the field with the associated label text. The field can be on scroll level one, two, or three of the active page. The <i>recordname</i> prefix is required if the function call is in a record definition other than <i>recordname</i> .
<i>new_label_text</i>	A String value specifying the new value for the field or grid column label.

Example

```
If training_loc = "HAW" then

    SetLabel(voucher_tbl.training_loc, "Hawaii Training Center");

End-if;
```

SetLanguage

Syntax

```
SetLanguage(language_code)
```

Description

SetLanguage sets the end-user's current language preference to the specified *language_code*. *language_code* must be a valid translate value for the field LANGUAGE_CD. **SetLanguage** returns True if it is successful, and it returns False if it fails or an invalid value was passed. The language preference is temporary, remaining in effect only until the end-user logs off, or until another call is made to **SetLanguage**.



SetLanguage does *not* work in Signon PeopleCode.

Considerations Using SetLanguage with %Language

The value of %Language depends on the type of application:

- For online applications, %Language is the language code that the current component is using.
- For non-online applications (such as batch or in a message subscription), %Language is the language code of the end-user.



%Language can also be changed using the Language menu (in the Windows client).

SetLanguage changes the default language *for the end-user*.

%Language will *not* reflect the value set using SetLanguage until a new component is started.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.
Returns False if an invalid language code is passed.

Parameters

<i>language_code</i>	A valid language code, stored in the Translate table for the LANGUAGE_CD field.
----------------------	---

Example

The following example switches the language code and displays a message indicating informing the end-user of the change:

```
If %Page = PAGE.INTERNATL_PREF Then

    If SetLanguage (LANGUAGE_CD) Then

        WinMessage (MsgGet (102, 5, "Language preference changed to ",
LANGUAGE_CD));

    Else

        WinMessage (MsgGet (102, 6, "Error in setting language. Language is
currently %1", %Language));

    End-if;

End-if;
```

Related Topics

%Language

SetNextPanel

Syntax

SetNextPanel (*panelname*)

Description

SetNextPanel specifies the panel name to which the user will be transferred when selecting the NextPanel (F6) function or specifying it with the PeopleCode TransferPage function.



The **SetNextPanel** function is supported for compatibility with previous releases of PeopleTools. Future applications should use SetNextPage instead.

SetNextPage

Syntax

SetNextPage (*pagename*)

Description

SetNextPage specifies the page name to which the user will be transferred when selecting the NextPage (ALT+6 and ENTER) function or specifying it with the PeopleCode TransferPage function.

SetNextPage validates that *pagename* is listed on current menu. This selection is cleared when the user transfers to a new page.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Parameters

<i>pagename</i>	A String equal to the name of the page as specified in the page definition.
-----------------	---

Example

The following example sets up a key list with AddKeyListItem, saves the current page and then transfers the user to a page named PAGE_2.

```
ClearKeyListItem( );
AddKeyListItem(OPRID, OPRID);
AddKeyListItem(REQUEST_ID, REQUEST_ID);
SetNextPage("PAGE_2");
DoSave( );
TransferPage( );
```

The following example sets up and transfers the user to page JOB_DATA.

```
If SetNextPage(PAGE.JOB_DATA) Then
    TransferPage( );
End-if;
```


Related Topics

TransferPage

SetPasswordExpired

Syntax

```
SetPasswordExpired(NewValue)
```

Description

The **SetPasswordExpired** function sets the password expired status for the current user. When the user's password expired flag is set to True, they can only access the page that allows them to change their password. The function returns the old value, that is, the value that represented the status of the flag before it was set to *NewValue*.



For more information, see Security.

Parameters

<i>NewValue</i>	Specify a new value for the user's password expired flag. This parameter takes a boolean value
-----------------	---

Returns

A Boolean value: True if you've set the password expire flag to False, False if you've set the password expire flag to True.

Example

```
If %PasswordExpired Then  
  
    &NewValue = SetPasswordExpired(True);  
  
End-If;
```

Related Topics

%PasswordExpired, SwitchUser, Security

SetReEdit

Syntax

```
SetReEdit(reedit_on)
```

Description

SetReEdit switches re-edit mode on and off. When re-edit mode is *on*, definitional edits (such as translate table and prompt table edits), as well as FieldEdit PeopleCode, are run on each editable field in the component when the component is saved. If an error is found, the component data is not saved. **SetReEdit** can be called at any time during the life of the component before the SaveEdit event fires, and would typically be called in RowInit when other page settings are being initialized. When a component is started, re-edit mode is *off* by default.

SetReEdit is used primarily in financial applications, where transactions are sometimes brought into the database by non-online processes. When re-edit mode is *on*, the values read in during these transactions can be validated by simply bringing them up in the page and saving. Any errors are then reported, as if the end-user had entered all of the data online.

Parameters

<i>reedit_on</i>	A Boolean value specifying whether to switch re-edit mode on or off. True turns re-edit mode on, False turns re-edit mode off.
------------------	--

Example

This example is used in RowInit PeopleCode to initialize component settings. Once re-edit mode is on, field-level edits will be re-applied when the component is saved.

```
SetReEdit (True) ;
```

SetSearchDefault

Syntax

```
SetSearchDefault ([recordname.]fieldname)
```

Description

SetSearchDefault enables system defaults (default values set in record field definitions) for the specified field on search dialogs. It does not cause the FieldDefault event to fire.



This function remains for backward compatibility only. Use the SearchDefault Field property instead. See Data Buffer Access.

The system default occurs only once, when the search dialog first starts, immediately after SearchInit PeopleCode. If the end-user subsequently blanks out a field, the field will not be reset to the default value. The related function **ClearSearchDefault** disables default processing for the specified field. **SetSearchDefault** is only effective when used in SearchInit PeopleCode programs.

Parameters

`[recordname.]fieldname` The name of the target field, a search or alternate search key that is about to appear in the search dialog. The *recordname* prefix is required if the call to **SetSearchDefault** is in a record definition other than *recordname*.

Example

This example, from SearchInit PeopleCode turns on edits and defaults for the SETID field in the search dialog:

```
SetSearchEdit (SETID) ;
SetSearchDefault (SETID) ;
```

Related Topics

ClearSearchDefault, ClearSearchEdit, SetSearchDialogBehavior, SetSearchEdit, and Search Processing in Update Modes

SetSearchDialogBehavior

Syntax

```
SetSearchDialogBehavior(force_or_skip)
```

Description

SetSearchDialogBehavior can be called in SearchInit PeopleCode to set the behavior of search and add dialogs before a page is displayed, overriding the default behavior. There are two dialog behavior settings: *skip if possible* (0) and *force display* (1).

Skip if possible means that the dialog will be skipped if all of the following are true:

- All required keys have been provided (either by system defaults or by PeopleCode).
- If this an Add dialog, then no "duplicate key error" results from the provided keys; if this error occurs, the processing resets to the default behavior.
- If this is a Search dialog, then at least one row is returned based on the provided keys.

Force display means that the dialog will be displayed even if all required keys have been provided.

The default behavior of the search and add dialogs is **force display**.



SetSearchDialogBehavior can only be used in SearchInit PeopleCode.

Returns

None.

Parameters

<i>force_or_skip</i>	A Number equal to one of the following values:
• 0	sets the dialog behavior to skip if possible .
• 1	sets the dialog behavior to force display .

Example

The following function call, which must occur in SearchInit PeopleCode, sets the dialog behavior to **skip if possible**.

```
SetSearchDialogBehavior(0);
```

Related Topics

SetSearchDefault, SetSearchEdit, ClearSearchEdit, ClearSearchDefault, IsUserInRole and Search Processing in Update Modes

SetSearchEdit

Syntax

```
SetSearchEdit([recordname.]fieldname)
```

Description

SetSearchEdit enables system edits (edits specified in the record field definition) for the specified *[recordname.]fieldname*, for the life of the search dialog, or until **ClearSearchEdit** is called with that same field.



This function remains for backward compatibility only. Use the SearchEdit Field property instead. See Data Buffer Access.

In the Add mode search dialog, the following edits are performed when the end-user clicks the **Add** button. In any other mode, the following edits are performed when the end-user clicks the **Search** button:

- Formatting
- Required Field
- Yes/No Table

- Translate Table
- Prompt Table

SetSearchEdit does not cause the FieldEdit, FieldChange, or SaveEdit PeopleCode events to fire during the search dialog.

You might use **SetSearchEdit** to control access to the system. For example, you can apply this function to the SETID field of a dialog box and require the end-user to enter a valid SETID.

Returns

Returns a Boolean value indicating whether the function executed successfully.

Parameters

<i>fieldname</i>	The name of the search dialog field on which to enable field edits.
------------------	---

Example

This example turns on edits and system defaults for the SETID field in the search dialog box:

```
SetSearchEdit (SETID) ;
SetSearchDefault (SETID) ;
```

Related Topics

ClearSearchEdit, ClearSearchDefault, SetSearchDefault, SetSearchDialogBehavior, and Search Processing in Update Modes

SetTempTableInstance

Syntax

```
SetTempTableInstance(instance_number)
```

Description

The **SetTempTableInstance** function sets the default temp table instance to the specified number for the processing of temporary tables. This default is used by all **%Table** meta-SQL references to temporary tables, and by all SQL operations. Generally, you will only use this function when you're trying to use any of the ScrollSelect functions, the Rowset class Select or SelectAll methods, the Record class SQL methods (SelectByKey, Insert, and so on.), or any of the meta-SQL statements that use %Table (%InsertSelect, %SelectAll, %Delete, and so on.) Generally, %Table should be used to override the default.

If you use this built-in within an Application Engine program, and the program wants to take advantage of a process-level instance on the request, the old instance value *must be* saved, then restored after you're finished using the new instance.

If you pass a zero for *instance_number*, the **Fill** method uses the physical table instance with no table append, for example, if the temporary table record is FI_INSTR_T, the physical table used is PS_FI_INSTR_T.

Parameters

instance_number Specify the instance number for the temporary tables.

Returns

Existing (or previous) instance number.

Example

To avoid interfering with other uses of temporary tables, you should only set the temporary table instance for your process, then set it back to the default. For example:

```
/* Set temp table instance */

&PrevInstNum = SetTempTableInstance(&NewInstNum);

/* use the temporary table */

. . .

/* Restore the temp table instance */

SetTempTableInstance(&PrevInstNum);
```

Related Topics

%Table

SetTracePC

Syntax

SetTracePC (*n*)

Description

SetTracePC allows you to control Trace PeopleCode settings programmatically. This is useful if you want to isolate and debug a single program or part of a program.



If you're using an API with the Session class, use the Trace Setting Class Properties object properties instead of this function. See About Trace Settings.

You can set options prior to starting a PeopleTools session using the Trace tab of Configuration Manager.



For more information, see Configuration Manager.

Trace PeopleCode creates and writes data to a trace file that it shares with Trace SQL; Trace SQL and Trace PeopleCode information are both output to the file in the order of execution. The trace file uses a file name and location specified in the Trace page of Configuration Manager. If no trace file is specified in Configuration Manager, the file defaults to DBG1.TMP in your Windows Temp directory. If you only specify a file name, and no directory is specified, the file will be written to the directory you're running Tools from. This file is cleared each time you log on and can be opened in a text editor while you are in a PeopleTools session, so if you want to save it, you'll need to print it or copy it from your text editor.

Returns

None.

Parameters

options

Either a Number or a constant value that sets trace options. Calculate *options* by either totaling the numbers associated with any of the following options or by adding constants together:

Option	Constants	Description
1	%TracePC_Functions	Provide a trace of the program as it is executed. This implies options 64, 128, and 256 described below.
2	%TracePC_List	Provide a listing of the entire program.
4	%TracePC_Assigns	Show the results of all assignments made to variables.
8	%TracePC_Fetches	Show the values fetched for all variables.
16	%TracePC_Stack	Show the contents of the internal machine stack. This option is normally used for debugging the PeopleCode language and not PeopleCode programs.
64	%TracePC_Starts	Provide a trace showing when each program starts.

128	%TracePC_ExtFuncs	Provide a trace showing the calls made to each external PeopleCode routine.
256	%TracePC_IntFuncs	Provide a trace showing the calls made to each internal PeopleCode routine.
512	%TracePC_ParamsIn	Show the values of the parameters to a function.
1024	%TracePC_ParamsOut	Show the values of the parameters as they exist at the return from a function.
2048		Show each statement as it's executed (and don't show statements on branches not taken.)

Example

The following example is part of a SavePreChange PeopleCode program that sets PeopleCode trace based on page field settings:

```

DEBUG_CODE = 0;
If DEBUG_TRACE_ALL = "Y" Then
    DEBUG_CODE = DEBUG_CODE + 1
End-if;
If DEBUG_LIST = "Y" Then
    DEBUG_CODE = DEBUG_CODE + 2
End-if;
If DEBUG_SHOW_ASSIGN = "Y" Then
    DEBUG_CODE = DEBUG_CODE + 4
End-if;
If DEBUG_SHOW_FETCH = "Y" Then
    DEBUG_CODE = DEBUG_CODE + 8
End-if;
If DEBUG_SHOW_STACK = "Y" Then
    DEBUG_CODE = DEBUG_CODE + 16
End-if;
If DEBUG_TRACE_START = "Y" Then
    DEBUG_CODE = DEBUG_CODE + 64
End-if;
If DEBUG_TRACE_EXT = "Y" Then
    DEBUG_CODE = DEBUG_CODE + 128
End-if;
If DEBUG_TRACE_INT = "Y" Then
    DEBUG_CODE = DEBUG_CODE + 256
End-if;
If DEBUG_SHOW_PARAMS = "Y" Then
    DEBUG_CODE = DEBUG_CODE + 512
End-if;
If DEBUG_SHOW_PARAMSRT = "Y" Then

```



```
    DEBUG_CODE = DEBUG_CODE + 1024
End-if;
SetTracePC (DEBUG_CODE) ;
```

The following example sets Trace PC to show a listing of all the calls made to external routines as well as calls made to internal routines:

```
SetTracePC (384) ;
```

The following is identical to the above:

```
SetTracePC (%TracePC_ExtFuncs + %TracePC_IntFuncs) ;
```

Related Topics

SetTraceSQL



For more information on using the PeopleCode Trace utility, see Performance Monitoring.

SetTraceSQL

Syntax

```
SetTraceSQL (options)
```

Description

SetTraceSQL gives you programmatic control over the Trace SQL utility, enabling you to control TraceSQL options during the course of program execution.



If you're using an API with the Session class, use the Trace Setting Class Properties object properties instead of this function. See About Trace Settings.

When you interact with PeopleTools, SQL statements transparently perform actions such as page construction. The Trace SQL utility creates and updates a file showing the SQL statements generated by PeopleTools.

You can set options prior to starting a PeopleTools session using the Trace tab of Configuration Manager.



For more information, see Configuration Manager.

Trace SQL creates and writes data to a trace file that it shares with Trace PeopleCode; Trace SQL and Trace PeopleCode information are both output to the file in the order of execution. The trace file uses a file name and location specified in the Trace page of Configuration Manager. If no

trace file is specified in Configuration Manager, the file defaults to DBG1.TMP in your Temp directory. If you only specify a file name, and no directory is specified, the file will be written to the directory you're running Tools from. This file is cleared each time you log on and can be opened in a text editor while you are in a PeopleTools session, so if you want to save it, you'll need to print it or copy it from your text editor.

Returns

None.

Parameters

options

Either a Number value or a constant that sets trace options. Calculate *options* by either totaling the numbers associated with any of the following options, or adding constants together:

Option	Constant	Description
0	%TraceSQL_None	No output
1	%TraceSQL_Statements	SQL statements
2	%TraceSQL_Variables	SQL statement variables (binds)
4	%TraceSQL_Connect	SQL connect, disconnect, commit and rollback
8	%TraceSQL_Fetch	Row Fetch (indicates that it occurred and the return code - not the data fetched.)
16	%TraceSQL_MostOthers	All other API calls except Set Select Buffers
32	%TraceSQL_SSB	Set Select Buffers(identifies the attributes of the columns to be selected)
64	%TraceSQL_DBSpecific	Database API-specific calls
128	%TraceSQL_Cobol	COBOL Statement Timings
256	%TraceSQL_SybBind	Sybase Bind Information
512	%TraceSQL_SybFetch	Sybase Fetch Information
1024	%TraceSQL_DB2400Server	Manager information for DB2/400 only



PeopleSoft recommends setting options to 3 to provide most essential information without performance degradation. Options 8 and 32 greatly increase the volume of the trace and will noticeably degrade performance.

Example

The following example switches off Trace SQL:

```
SetTraceSQL(0);
```

The following is identical to the above:

```
SetTraceSQL(%TraceSQL_None);
```

The following example sets Trace SQL to typical settings that won't degrade performance:

```
SetTraceSQL(3);
```

The following is identical to the above example:

```
SetTraceSQL(%TraceSQL_Statements + %TraceSQL_Variables);
```

Related Topics

SetTracePC



For more information on using the PeopleCode Trace utility, see Performance Monitoring.

Sign

Syntax

```
Sign(n)
```

Description

Sign determines the sign of a number.

Returns

Returns a Number value equal to:

- 1 if *n* is positive
- 0 if *n* is 0
- -1 if *n* is negative

Parameters

<i>n</i>	A Number value of which to determine the sign.
----------	--

Example

The example sets &NUMSIGN to 1:

```
&NUMSIGN = Sign(25);
```

Related Topics

Abs

Sin

Syntax

```
Sin(angle)
```

Description

Sin calculates the sine of the given angle (opposite / hypotenuse).

Parameters

<i>angle</i>	A value in radians.
--------------	---------------------

Returns

A real number between **-1.00** and **1.00**.

Example

The following example returns the sine of an angle measuring **1.2** radians:

```
&MY_RESULT = Sin(1.2);
```

Related Topics

Acos, Asin, Atan, Cos, Cot, Degrees, Radians, Tan

SinglePaymentPV

Syntax

```
SinglePaymentPV(int_rate, n_per)
```

Description

This function calculates the future value of a single monetary unit after a specified number of periods at a specified interest rate.

Returns

Returns a Number value equal to the value of the unit after *n_per* periods at an interest rate of *int_rate* per period.

Parameters

<i>int_rate</i>	A Number representing the interest rate at which value is accrued per period.
<i>n_per</i>	A Number specifying the number of periods on which to base the calculated value.

Example

The example calculates &PMT as .857338820301783265:

```
&PMT = SinglePaymentPV(8, 2);
```

Related Topics

UniformSeriesPV

SortScroll

Syntax

```
SortScroll(level, scrollpath, sort_fields)
```

Where *scrollpath* is:

```
[RECORD.level1_recname, [RECORD.level2_recname,] RECORD.target_recname
```

and where *sort_fields* is a list of field specifiers in the form:

```
[recordname.]field_1, order_1 [, [recordname.]field_2, order_2]...
```

Description

SortScroll programmatically sorts the rows in a scroll area on the active page. The rows can be sorted on one or more fields.



This function remains for backward compatibility only. Use the Sort Rowset method instead. See Data Buffer Access.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Parameters

<i>level</i>	Integer specifying the level of the scroll to sort. It can be 1, 2, or 3.
<i>scrollpath</i>	A construction that specifies a scroll area in the component buffer.
<i>sort_fields</i>	A list of field and order specifiers which act as sort keys. The rows in the scroll area will be sorted by the first field in either ascending or descending order, then by the second field in either ascending or descending order, and so on.
<i>[recordname.]field_n</i>	Specifies a sort key field in <i>target_recname</i> . The recordname prefix is required if the call to SortScroll is in a record other than <i>target_recname</i> .
<i>order_n</i>	A single-character string. "A" specifies ascending order; "D" specifies descending order.

Example

The first example repopulates a scroll in a page programmatically by first flushing its contents, selecting new contents using **ScrollSelect**, then sorting the rows in ascending order by `EXPORT_OBJECT_NAME`:

```
Function populate_scrolls;
  ScrollFlush(RECORD.EXPORT_OBJECT);
  ScrollSelect(1, RECORD.EXPORT_OBJECT, RECORD.EXPORT_OBJECT,
    "where export_type = :EXPORT_TYPE_VW.EXPORT_TYPE");
  SortScroll(1, RECORD.EXPORT_OBJECT, EXPORT_OBJECT.EXPORT_OBJECT_NAME, "A");
End-function;
```

The second example sorts the rows on scroll level one by primary and secondary key fields:

```
SortScroll(1, RECORD.EN_BOM_COMPS, EN_BOM_COMPS.SETID, "A",
  EN_BOM_CMOPS.INV_ITEM_ID, "A");
```

Related Topics

HideScroll, RowScrollSelect, RowScrollSelectNew, ScrollSelect, ScrollSelectNew, UnhideScroll

Split

Syntax

```
Split(string, separator)
```

Description

The **Split** function *converts* a **string** into an **array of strings** by looking for the string *separator* in the given string.



Split does not split an array.

If *separator* is omitted, a blank is used. If *separator* is a null string (""), the string is split into single characters.

Parameters

<i>string</i>	The string to be converted into an array
<i>separator</i>	The character used for dividing the string.

Returns

Returns a reference to the array.

Example

The following code produces in &AS the array ("This", "is", "a", "simple", "example.").

```
&STR = "This is a simple example.";

&AS = Split(&STR);
```

Related Topics

CreateArray, CreateArrayRept and Array Class

SQLExec

Syntax

The values for the parameters for SQLExec depend on whether the first parameter is a string (*sqlcmd*) or a reference to a SQL definition (SQL.*sqlname*). Version 1:

```
SQLExec(sqlcmd, bindvars, output)
```

where *bindvars* is a list of variable or record fields, one for each *:n* references within *sqlcmd*, in the form:

```
var_1 [, var_2]...
```

or

```
[recordname_1.]fieldname_1 [, [recordname_2.]fieldname_2]...
```

and where *output* is a list of variables or record fields, one for each column selected by *sqlcmd*, in the form:

```
var_1 [, var_2]...
```

or

```
[recordname_1.]fieldname_1 [, [recordname_2.]fieldname_2]...
```

Version 2:

```
SQLExec(SQL.sqlname, inval, outval)
```

inval and *outval* behave the same as the *bindvars* and *output* parameters in the above syntax, with these additions:

- The *inval* parameters can be values as well as variables.
- The *inval* and *outval* parameters can be record objects.

Description

The **SQLExec** function executes a SQL command from within a PeopleCode program by passing a SQL command string. The SQL command bypasses the Component Processor and interacts with the database server directly. If you want to delete, insert, or update a single record, use the corresponding PeopleCode record object method.

If you want to delete, insert, or update a series of records, all of the same type, use the CreateSQL or GetSQL functions then the **Execute** SQL object method.

Limitation of SQLExec SELECT Statement

SQLExec can only SELECT a single row of data. If your SQL statement (or your *SQL.sqlname* statement) retrieves more than one row of data, **SQLExec** outputs only the first row to its output variables. Any subsequent rows are discarded. This means if you only want to fetch a single row, **SQLExec** can perform better than the other SQL functions, because only a single row is fetched. If you need to SELECT multiple rows of data, use the CreateSQL or GetSQL functions and the Fetch method. You can also use ScrollSelect or one of the Select methods on a rowset object to read rows into a (usually hidden) work scroll.



The PeopleSoft record name specified in the SQL SELECT statement must be in uppercase.

Limitations of SQLExec UPDATE, DELETE, and INSERT Statements

SQLExec statements that result in a database update (specifically, UPDATE, INSERT, and DELETE) can only be issued in the following events:

- SavePreChange
- Workflow

- SavePostChange

Remember that **SQLExec** UPDATES, INSERTs and DELETEs go directly to the database server, not to the Component Processor (although **SQLExec** can *look at* data in the buffer via bind variables included in the SQL string). If a **SQLExec** assumes that the database has been updated based on changes made in the component, that **SQLExec** can be issued only in the SavePostChange event, because before SavePostChange none of the changes made to page data has actually been written back to the database.

Setting Data Fields to Null

SQLExec will *not* set Component Processor data buffer fields to NULL after a row not found fetching error. However, it does set fields that aren't part of the Component Processor data buffers to NULL. Work record fields are also reset to NULL.

Using Meta-SQL in SQLExec

Different DBMS platforms have slightly different formats for dates, times, and datetimes; and PeopleSoft has its own format for these data types as well. Normally the Component Processor performs any necessary conversions when platform-specific data types are read from the database into the buffer or written from the buffer back to the database.

When an SQLExec statement is executed, these automatic conversions don't take place. Instead, you need to use meta-SQL functions inside the SQL command string to perform the conversions. The basic types of meta-SQL functions are:

- General functions that expand at runtime to give you lists of fields, key fields, record fields, and so on. %List, %InsertSelect, or %KeyEqual are typical examples.
- In functions that expand at runtime into platform-specific SQL within the WHERE clause of a SELECT or UPDATE statement or in an INSERT statement. %DateIn is a typical example.
- Out functions that expand at runtime into platform-specific SQL in the main clause of SELECT statement. %DateOut is a typical example.

Following is an example of an SQL SELECT using both "in" and "out" metastring:

```
select emplid, %dateout(effdt) from PS_CAR_ALLOC a where car_id = '' |
&REGISTRATION_NO | '' and plan_type = '' | &PLAN_TYPE | '' and a.effdt = (select
max (b.effdt) from PS_CAR_ALLOC b where a.emplid=b.emplid and b.effdt <=
%currentdatein) and start_dt <= %currentdatein and (end_dt is null or end_dt >=
%currentdatein)";
```



For more information on how to use meta-SQL, see Example. For documentation of the meta-SQL functions, see Meta-SQL.

Bind Variables in SQLExec

Bind variables are references within the *sqlcmd* string to record fields listed in *bindvars*. Within the string, the bind variables are integers preceded by colons:

```
:1, :2,...
```

The integers must be in numerical order. Each of these *:n* integers represents a field specifier in the *bindvars* list, so that :1 refers to the first field reference in *bindvars*, :2 refers to the second field reference, and so on.

For example, in the following statement:

```
SQLExec("Select sum(posted_total_amt)
        from PS_LEDGER
        where deptid between :1 and :2", DEPTID_FROM, DEPTID_TO, &SUM);
```

:1 is replaced by the value contained in the record field DEPTID_FROM; :2 is replaced by the value contained in the record field DEPTID_TO.

Note the following points:

- Bind variables may not refer to long character (longchar) fields.
- Bind variables can be passed as parameters to meta-SQL functions, for example:

```
SQLExec("...%datein(:1)...", START_DT, &RESULT)
```

- If a bind variable *:n* is a Date field that contains a null value, **SQLExec** will replace all occurrences of " :n" located before the first WHERE clause with "NULL" and all occurrences of "= :n" located after the first WHERE to "IS NULL".

Inline Bind Variables in SQLExec

Inline bind variables are included directly in the SQL string in the form:

```
:recordname.fieldname
```

The following example shows the same **SQLExec** statement with standard bind variables, then with inline bind variables:

```
Rem without Inline Bind Variables;
SQLExec("Select sum(posted_total_amt)
        from PS_LEDGER
        where deptid between :1 and :2", deptid_from, deptid_to, &sum);

Rem with Inline Bind Variables;
SQLExec("Select sum(posted_total_amt)
        from PS_LEDGER
        where deptid between :LEDGER.DEPTID_FROM
        and :LEDGER.DEPTID_TO", &sum);
```

Inline bind variables, like all field and record references enclosed in strings, are considered by PeopleTools as a "black box". If you rename records and fields, PeopleTools will not update record and field names that are enclosed in strings as inline bind variables. For this reason, you should use standard bind variable in preference to inline bind variables wherever standard bind variables are available (as they are in **SQLExec**).

Prior to PeopleTools 8.0, PeopleCode replaced runtime parameter markers in SQL strings with the associated literal values. For databases that offer SQL statement caching, a match was never found in the cache so the SQL had to be re-parsed and re-assigned a query path. However, with PeopleTools 8.0, PeopleCode passes in bind variable parameter markers. For databases with SQL caching, this can offer significant performance improvements.

If you use inline bind variables, they will still be passed as literals to the database. However, if you convert them to bind variables, you may see significant performance improvements.

Output Variables in SQLExec

If you use **SQLExec** to SELECT a row of data, you will need to place the data into variables or record fields so that it can be processed. You list these variables or fields, separated by commas in the *output* part of the statement following the *bindvars* list. Supply one variable or field for each column in the row of data retrieved by **SQLExec**. They must be listed in the same order in which the columns will be selected.

The number of output variables cannot exceed 64.

SQLExec Maintenance Issues

SQLExec statements are powerful, but they can be difficult to upgrade and maintain. If you use a SQL string passed in the command, it's considered a "black box" by PeopleCode. If field names or table names change during an upgrade, table and field references within the SQL string are not updated automatically. For these reasons, you should use a SQL definition and the meta-SQL statements provided in PeopleTools 8.0, instead of typing in a SQL string.

Generally, you should use **SQLExec** only when you must interact directly with the database server and none of the ScrollSelect functions, or record object methods (which are somewhat easier to maintain) will serve your purpose effectively.

Be Careful How You Use It

SQLExec will perform any SQL statement the end-user has database privileges to perform. This normally includes SELECT, INSERT, UPDATE, and DELETE statements against application data tables. However, you can set up end-users with more privileges (and you have to on some platforms). In such cases, the end-user could alter the structure of tables using **SQLExec**, or even drop the database!



The PeopleSoft application will not stop the end-user from doing anything that the end-user has privileges to do on the database server, so be very careful what you write in an **SQLExec** statement.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Parameters

<i>sqlcmd</i> <i>SQL.sqlname</i>	Specify either a String containing the SQL command to be executed or a reference to an existing SQL definition. This string can include bind variables, inline bind variables, and meta-SQL.
<i>bindvars</i>	A list of references, each of which corresponds to a numeric (: <i>n</i>) bind variable reference in the SQL command string. If you've specified a SQL definition, these parameters can be values as well as variables. It can also be a reference to a record object.



For more information, see Bind Variables in SQLExec.

<i>output</i>	A list of PeopleCode variables or record fields to hold the results of a SQL SELECT. There must be one variable for each column specified in the SELECT statement. If you've specified a SQL definition, it can also be a reference to a record object.
---------------	---

Example

The following example, illustrates a SELECT statement in a SQLExec:

```
SQLExec("SELECT COUNT(*) FROM PS_AE_STMT_TBL WHERE AE_PRODUCT = :1 AND
AE_APPL_ID = :2 AND AE_ADJUST_STATUS = 'A' ", AE_APPL_TBL.AE_PRODUCT,
AE_APPL_TBL.AE_APPL_ID, AE_ADJ_AUTO_CNT);
```

Note the use of bind variables, where :1 and :2 correspond to AE_APPL_TBL.AE_PRODUCT and AE_APPL_TBL.AE_APPL_ID. AE_ADJ_AUTO_CNT is an output field to hold the result returned by the SELECT.

The next example is also a straightforward SELECT statement, but one which uses the %datein meta-SQL function, which expands to appropriate platform-specific SQL for the :5 bind variable:

```
SQLExec("SELECT 'X', AE_STMT_SEG FROM PS_AE_STMT_B_TBL where AE_PRODUCT = :1
AND AE_APPL_ID = :2 AND AE_SECTION = :3 AND DB_PLATFORM = :4 AND EFFDT =
%datein(:5) AND AE_STEP = :6 AND AE_STMT_TYPE = :7 AND AE_SEQ_NUM = :8",
AE_STMT_TBL.AE_PRODUCT, AE_STMT_TBL.AE_APPL_ID, AE_STMT_TBL.AE_SECTION,
AE_STMT_TBL.DB_PLATFORM, AE_STMT_TBL.EFFDT, AE_STMT_TBL.AE_STEP,
AE_STMT_TBL.AE_STMT_TYPE, &SEG, &EXIST, &STMT_SEG);
```

This last example (in SavePreChange PeopleCode) passes an INSERT INTO statement in the SQL command string. Note the use of a date string this time in the %datein meta-SQL, instead of a bind variable:

```
SQLExec("INSERT INTO PS_AE_SECTION_TBL ( AE_PRODUCT, AE_APPL_ID, AE_SECTION,
DB_PLATFORM, EFFDT, EFF_STATUS, DESCR, AE_STMT_CHUNK_SIZE, AE_AUTO_COMMIT,
AE_SECTION_TYPE ) VALUES ( :1, :2, :3, :4, %DATEIN('1900-01-01'), 'A', ' ', 200,
'N', 'P' )", AE_APPL_TBL.AE_PRODUCT, AE_APPL_TBL.AE_APPL_ID, AE_SECTION,
DB_PLATFORM);
```

In the following example, a SQLExec statement is used to select into a record object.

```
Local Record &DST;

&DST = CreateRecord(RECORD.DST_CODE_TBL);

&DST.SETID.Value = GetSetId(FIELD.BUSINESS_UNIT, DRAFT_BU, RECORD.DST_CODE_TYPE,
"");

&DST.DST_ID.Value = DST_ID_AR;

SQLExec("%SelectByKeyEffDt(:1,:2)", &DST, %Date, &DST);

/* do further processing using record methods and properties */
```

Related Topics

CreateSQL, FetchSQL, GetSQL, StoreSQL and SQL Class

Sqrt

Syntax

```
SQRT(n)
```

Description

Sqrt calculates the square root of a number.

Returns

Returns a Number equal to the positive square root of *n*. If *n* is a negative number, **Sqrt** displays an error.

Parameters

<i>n</i>	A Number of which you want to find the square root.
----------	---

Example

The examples return 15, 4, and 8.42615, respectively:

```

&NUM = Sqrt (225) ;
&NUM = Sqrt (16) ;
&NUM = Sqrt (71) ;

```

StopFetching

Syntax

`StopFetching()`

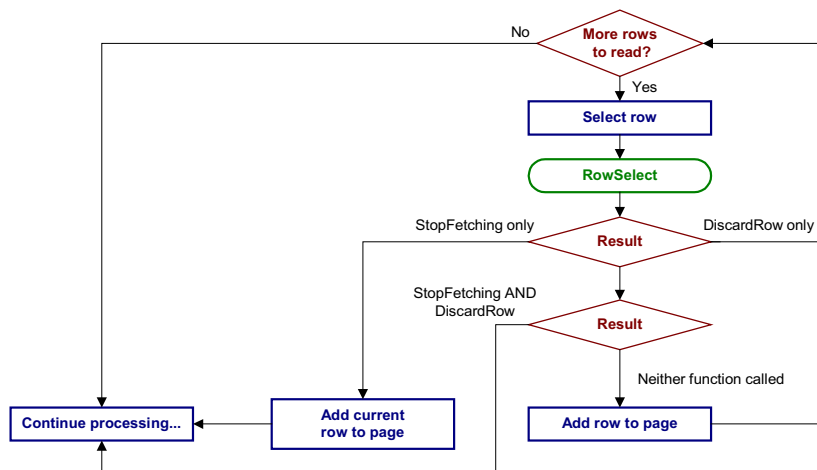
Description

StopFetching is called during Row Select processing, during which rows of data that have been selected down from the database can be filtered as they are added to the component. This function is valid only in RowSelect PeopleCode. If **StopFetching** is called without **DiscardRow**, it adds the current row to the component, then stops adding any more rows. If **StopFetching** is called with **DiscardRow**, the system skips the current row and stops adding rows to the component.

StopFetching has the same functionality as Error in the RowSelect event. The anomalous behavior of Error is supported for compatibility with previous releases of PeopleTools.



Row Select processing is used infrequently, because it is more efficient to filter out rows of data using a search view or an effective-dated record before the rows are selected down to the client from the database server.



Flow of Row Select Processing

Returns

None.

Related Topics

DiscardRow, and Row Select Processing

StoreSQL

Syntax

```
StoreSQL(sqlstring, [SQL.]sqlname[, dbtype[, effdt]])
```

Description

The **StoreSQL** function writes the given *sqlstring* value to a SQL definition, storing it under the name *sqlname*, with the database type *dbtype* and the effective date *effdt*. If *sqlname* is a literal name, it must be in the form SQL.*sqlname* or in quotes ("*sqlname*").

To specify a generic statement, that is, one that is overridden by any other matching statement, *dbtype* as **Default** and *effdt* as the null date (or Date(19000101)).

Parameters

sqlstring Specify the SQL string to be saved as the SQL definition. This parameter takes a string value.

sqlname Specify the name of the SQL definition to be created. This is either in the form SQL.*sqlname* or a string value giving the *sqlname*.

dbtype Specify the database type to be associated with the SQL definition. This parameter takes a string value. If *dbtype* isn't specified or is null (""), it defaults to the current database type (the value returned from the %DbName system variable.)

Valid values for *dbtype* are as follows. These values are not case sensitive:

- DEFAULT
- DB2ODBC
- DB2UNIX
- INFORMIX
- MICROSOFT
- ORACLE
- SYBASE



Database platforms are subject to change.

effdt

Specify the effective date to be associated with the SQL definition. If *effdt* isn't specified, it defaults to the current as of date, that is, the value returned from the %AsOfDate system variable.

Returns

None.

Example

The following code stores the select statement as a SQL definition under the name SELECT_BY_EMPLID, for the current database type and effective as of the current as of date:

```
StoreSQL("%Select(:1) where EMPLID = :2", SQL.SELECT_BY_EMPLID);
```

Related Topics

CreateSQL, DeleteSQL, FetchSQL, GetSQL, SQLExec, and SQL Class

String

Syntax

```
String(val)
```

Description

String converts any non-string data type (except Object) to a string.

Normally the Component Processor automatically handles data type conversions. However, for some operations, such as comparisons, you want to specify the data type explicitly. Assume, for example, that you have two fields FIELD_I and FIELD_J containing number values 5000 and 10000. As character fields, 10000 is less than 5000 (because the first character in 10000 is less than the first character in 5000). As numbers, however, 10000 is of course greater than 5000.

Returns

Returns a String value representing *val*.

Parameters

val

A value of any type other than object, to be converted to its String representation.

Example

To force the comparison of the two fields as strings, you could use:

```
if string(FIELD_1) > string(FIELD_2)...
```

You can use the String function with a field object as follows:

```
&DATE = GetRecord(RECORD.DERIVED_HR).GetField(FIELD.EFFDT);  
  
&STR = String(&DATE.Value);
```

Related Topics

Char, Exact, Find, Left, Substring

Substitute

Syntax

```
Substitute(source_text, old_text, new_text)
```

Description

Substitute replaces every occurrence of a substring found in a string with a new substring. To replace text that occurs in a specific location in a text string use Replace.

Returns

Returns a String resulting from replacing every occurrence of *old_text* found in *source_text* with *new_text*.

Parameters

<i>source_text</i>	A String in which you want to replace substrings.
<i>old_text</i>	A String equal to the substring of <i>source_text</i> you want to replace.
<i>new_text</i>	A String with which to replace occurrences of <i>old_text</i> in <i>source_text</i> .

Example

The following example changes "Second Annual Conference" to "Third Annual Conference":

```
&newstr = Substitute("Second Annual Conference", "Second", "Third");
```

The next example sets &newstr to "cdcdcd":

```
&newstr = Substitute("ababab", "ab", "cd");
```

Related Topics

Replace

Substring

Syntax

```
Substring(source_str, start_pos, length)
```

Description

Substring extracts a substring of a specified number of characters beginning at a specified location in a source string.

If you know the exact length of *source_str*, and that it is null terminated, you can set *length* to 1 plus the exact length of *source_str* to get everything from *start_pos* to the end.

Returns

Returns a String equal to a substring *length* characters long beginning at character *start* of *source_str*.

Parameters

<i>source_str</i>	A String from which to extract a substring.
<i>start_pos</i>	A Number representing the character position in <i>source_str</i> where the substring starts.
<i>length</i>	A Number specifying the number of characters in the substring.

Example

This example sets &PAGE_NAME to the first eight characters of the name of the current page:

```
&PAGE_NAME = Substring(%page,1,8);
```

Related Topics

Char, Exact, Find, Left, Right, String, Substringb

Substringb

Syntax

```
Substringb(source_str, start_pos, length)
```

Description

Substringb extracts a substring of a specified number of bytes beginning at a specified location in a source string. **Substringb** is byte-oriented, whereas **Substring** is character-oriented; this distinction is significant whenever the source string contains double-byte characters.

If you know the exact length of *source_str*, and that it is null terminated, you can set *length* to 1 plus the exact length of *source_str* to get everything from *start_pos* to the end.

Returns

Returns a String value equal to a substring *length* bytes long beginning at byte *start* of *source_str*.

Parameters

<i>source_str</i>	A String from which to extract a substring.
<i>start_pos</i>	A Number representing the byte position in <i>source_str</i> where the substring begins.
<i>length</i>	A Number specifying the number of bytes in the substring.

Example

This example sets &PAGE_NAME to the first eight bytes of the name of the current page:

```
&PAGE_NAME = Substringb(%page,1,8);
```

Related Topics

Char, Exact, Find, Left, Right, String, Substring

SwitchUser

Syntax

```
SwitchUser (UserID, Password, AuthToken , ExtAuthInfo)
```



Note. *Password* is **not** encrypted: it is passed as a string.

Description

The **SwitchUser** function change the user ID of the current user logged onto the PeopleSoft system.



SwitchUser only changes the user for the session of the local application server. It does *not* cascade back to the portal or to any other application servers in the environment.

The SwitchUser function might be used as following. Suppose there is a special user ID in the system called REGIST. REGIST **only** has access to the self-registration component. The self-registration component has logic that asks the user a list of questions and information based on data in the database. Are you a customer, vendor, or employee. Enter you customer name. Enter other information related to this customer account. (such as information only this customer knows or information this customer just received from a workflow email.) Once the program verifies the information, create a User ID for this customer. After the user ID is created, the program should take the user right into their transaction without having to logoff, by using SwitchUser.



You must never call SwitchUser from Signon PeopleCode. SwitchUser calls Signon PeopleCode, and so you are in an infinite loop.

Parameters

<i>UserID</i>	Specify the User ID to be started. This parameter takes a string value.
<i>Password</i>	Specify the Password for this User ID. This parameter takes a string value.



Note. *Password* is **not** encrypted: it is passed as a string.

<i>AuthToken</i>	Specify a single signon authentication token used to authenticate the user. If you are authenticating the user by Userid and password, specify a NULL value for this parameter, that is, two quotation marks with no blank space between them (""). If you specify a token, and the token is valid, SwitchUser switches to the User ID embedded in the token. All other parameters are ignored if a token is used. This parameter takes a string value.
<i>ExtAuthInfo</i>	Specify binary data (encoded as a base64 string) used as additional input to authenticate the user. If your application doesn't use external authentication information, specify a NULL value for this parameter, that is, two quotation marks with no blank space between them ("").

Returns

A boolean value: True if user ID is switched successfully, False otherwise.

Example

The most common use of SwitchUser only specifies a Userid and Password. If the SwitchUser function executes successfully, you should check to see if the password for the new user id has expired or not.

```
If Not SwitchUser("MYUSERID", "MYPASSWORD", "", "") Then

    /* switch failed, do error processing */

Else

    If %PasswordExpired Then

        /* application specific processing for expired passwords */

    End-If;

End-If;
```

Related Topics

%UserId, %PasswordExpired, SetPasswordExpired, Security

Tan

Syntax

Tan (angle)

Description

Tan calculates the tangent of the given angle (opposite / adjacent).

Parameters

<i>angle</i>	A value in radians.
--------------	---------------------



In theory, values of *angle* such that *angle mod pi* = *pi/2* are not valid for this function, because inputs approaching such values produce results that tend toward infinity. In practice, however, no computer system can represent such values exactly. Thus, for example, the statement **Tan(Radians(90))** will produce a number close to the largest value PeopleCode can represent, rather than an error.

Returns

A real number.

Example

The following example returns the tangent of an angle measuring **1.2** radians:

```
&MY_RESULT = Tan(1.2);
```

Related Topics

Acos, Asin, Atan, Cos, Cot, Degrees, Radians, Sin

Time

Syntax

```
Time(n)
```

Description

Time derives a Time value from a Number value. It can be used to assign values to Time fields and variables, since Time values cannot be directly represented as constants.

Returns

Returns a Time value based on the number *n*.

Parameters

<i>n</i>	A Number in the form HHMMSS[.SSSSSS], representing a time to a precision of up to .000001 second, based on a 24-hour clock.
----------	---

Example

The example sets &START_TIME to 12:34:56.123456:

```
&START_TIME = Time(123456.123456);
```

Related Topics

Date, DateTimeValue, Time3, TimeValue

Time3

Syntax

```
Time3(hours, mins, secs)
```

Description

Time3 derives a Time value from 3 supplied numbers. It can be used to assign values to Time fields and variables, since Time values cannot be directly represented as constants.

Returns

Returns a Time value based equal to the sum of the three input values representing hours, minutes, and seconds, to a precision of .000001 second.

Parameters

<i>hours</i>	A Number in the form HH between 00 and 23, representing hours on a 24-hour clock.
<i>mins</i>	A Number in the form MM between 00 and 59, representing minutes of the hour.
<i>secs</i>	A Number in the form SS[.SSSSSS], between 00 and 59.999999, representing seconds.

Example

The example sets &START_TIME to 11.14.09.300000:

```
&START_TIME = Time3(11,14,9.3);
```

Related Topics

Date3, DateTime6, Time, TimeValue

TimePart

Syntax

```
TimePart(datetime_val)
```

Description

TimePart derives the Time component of a DateTime value.

Returns

Returns a Time value equal to the time portion of *datetime_val*.

Parameters

<i>datetime_val</i>	A DateTime value from which to extract the Time component.
---------------------	--

Example

The example set &T to 15.34.35.000000:

```
&DT = DateTimeValue("12/13/93 3:34:35 PM");
&T = TimePart(&DT);
```

Related Topics

DatePart, Hour, Minute, Second

TimeToTimeZone

Syntax

```
TimeToTimeZone(OldTime, SourceTimeZone, DestinationTimeZone);
```

Description

TimeToTimeZone converts a time field from the time specified by *SourceTimeZone* to the time specified by *DestinationTimeZone*.

Considerations using this Function

This function should generally be used in PeopleCode, *not* for displaying time. If you take a time value, convert it from base time to client time, then try to display this time, depending on the user settings, when the time is displayed the system might try to do a *second* conversion on an already converted time. This function could be used as follows: suppose a user wanted to check to make sure a time was in a range of times on a certain day, in a certain timezone. If the times were between 12 AM and 12PM in EST, these resolve to 9 PM and 9AM PST, respectively. The start value is *after* the end value, which makes it difficult to make a comparison. This function could be used to do the conversion for the comparison, in temporary fields, and not displayed at all.

Parameters

<i>OldTime</i>	Specify the time value to be converted.
<i>SourceTimeZone</i>	Specify the time zone that <i>OldTime</i> is in. Valid values are: <i>timezone</i> - a time zone abbreviation or a field reference to be used for converting <i>OldTime</i> . Local - use the local time zone for converting <i>OldTime</i> . Base - use the base time zone for converting <i>OldTime</i> .



For more information about setting the base time, see PeopleTools Utilities.

DestinationTimeZone

Specify the time zone that you want to convert *OldTime* to. Valid values are:

timezone - a time zone abbreviation or a field reference to be used for converting *OldTime*.

Local - use the local time zone for converting *OldTime*.

Base - use the base time zone for converting *OldTime*.

Returns

A converted time value.

Example

The following example TESTTM is a time field with a value 01/01/99 10:00:00. This example converts TESTTM from Eastern Standard Time (EST) to Pacific Standard Time (PST).

```
&NEWTIME = TimeToTimeZone (TESTTM, "EST", "PST");
```

&NEWTIME is a time variable with a value of 7:00:00AM.

Related Topics

ConvertDatetimeToBase, ConvertTimeToBase, FormatDateTime, IsDaylightSavings, DateTimeToTimeZone, TimeZoneOffset

TimeValue
Syntax

```
TimeValue(time_str)
```

Description

TimeValue calculates a Time value based on an input string. This function can be used to assign a value to a Time variable or field using a string constant, since a Time value cannot be represented with a constant.

Returns

Returns a Time value based on *time_str*.

Parameters

time_str

A String representing a time in the form HH:MM:SS.SSSSSS, based on a 24-hour clock.

Example

The example sets &START_TIME to 12.13.00.000000:

```
&START_TIME = TimeValue("12:13:00 PM");
```

Related Topics

DateTimeValue, DateValue

TimeZoneOffset

Syntax

```
TimeZoneOffset(DateTime {[, timezone | "BASE" | "LOCAL"]})
```

Description

The **TimeZoneOffset** function generates a time offset for *datetime*. The offset represents the relative time difference to GMT. If no other parameters are specified with *datetime*, the server's base time zone is used.

Parameters

<i>datetime</i>	Specify the datetime value to be used for generating the offset.
<i>timezone</i> Local Base	Specify a value to be used with <i>datetime</i> . The valid values are: <i>timezone</i> - a time zone abbreviation or a field reference to be used with <i>datetime</i> . Local - use the local time zone with <i>datetime</i> . Base - use the base time zone with <i>datetime</i> .

Returns

An offset string of the following format:

```
S hh:mm
```

where

S	is + or -, with + meaning East of Greenwich
hh	is the hours of offset
mm	is the minutes of offset

Related Topics

ConvertDatetimeToBase, ConvertTimeToBase, FormatDateTime, IsDaylightSavings, DateTimeToTimeZone, TimeToTimeZone

TotalRowCount

Syntax

TotalRowCount (*scrollpath*)

Where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]
RECORD.target_recname
```

To prevent ambiguous references, you can use **SCROLL.scrollname**, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Description

TotalRowCount calculates the number of rows (including rows marked as deleted) in a specified scroll area of a page. Rows that have been marked as deleted remain accessible to PeopleCode until the database has been updated; that is, all the way through SavePostChange.



This function remains for backward compatibility only. Use the RowCount Rowset property instead. See Data Buffer Access.

TotalRowCount is used to calculate the upper limit of a For loop if you want the loop to go through rows in the scroll that have been marked as deleted. If the logic of the loop does not need to execute on deleted rows, use ActiveRowCount.

Returns

Returns a Number equal to the total rows (including rows marked as deleted) in the target scroll.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
-------------------	---



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Example

The example uses **TotalRowCount** to calculate the limiting value on a For loop, which loops through all the rows in the scroll area:

```
&ROW_COUNT = TotalRowCount(RECORD.BUS_EXPENSE_PER, CurrentRowNumber(1),
RECORD.BUS_EXPENSE_DTL);
for &I = 1 to &ROW_COUNT
  /* do something with row &I that has to be done to deleted as well as active
rows */
end-for;
```

Related Topics

ActiveRowCount, CopyRow, CurrentRowNumber, FetchValue, For

Transfer

Syntax

```
Transfer(new_instance,
          MENUNAME.menuname,
          BARNAME.barname,
          ITEMNAME.menu_itemname,
          PAGE.page_group_item_name,
          action [, keylist]);
```

where *keylist* is a list of field references in the form:

```
[recordname.]field1 [, [recordname.]field2]...
```

OR

```
&RecordObject1 [, &RecordObject2]. . .
```

Description

Transfer closes the current page and transfers the end-user to another page, either within the current component or in another component. **Transfer** can either start a new instance of the application and transfer to the new page there, or close the old page and transfer to the new one in the same instance of PeopleTools.

Transfer is more powerful than the simpler TransferPage, which permits a transfer only within the current component in the current instance of PeopleTools. However, any variables declared as Component do *not* remain defined after using the **Transfer** function, whether you're transferring within the same component or not.

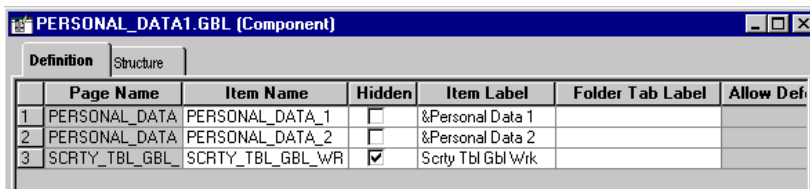
If you transfer from a primary page, the original page will be iconized. However, if you transfer from a secondary page, the transfer will *not* cause the original (that is, secondary) page to be iconized.

You can use **Transfer** from a secondary page (either with or without using a pop-up menu) *only* if you're transferring to a separate instance of a component. You can *not* use **Transfer** from a secondary page if you're not transferring to a separate instance of a component.

If you provide a valid search key for the new page in the optional *fieldlist*, the new page will open directly, using the same level-0 row as the page you are transferring from. If no key is provided, or if the key is invalid, the search dialog will appear, allowing the end-user to search for a row.

If *barname+itemname+page_group_item_name* is an invalid combination, the new menu comes up, but no page appears.

In the *page_group_item_name* parameter, make sure to pass the component item name for the page, not the page name. The component item name is specified in the Component Definition, in the Item Name column on the row corresponding to the specific page, as shown here:



	Page Name	Item Name	Hidden	Item Label	Folder Tab Label	Allow Def
1	PERSONAL_DATA	PERSONAL_DATA_1	<input type="checkbox"/>	&Personal Data 1		
2	PERSONAL_DATA	PERSONAL_DATA_2	<input type="checkbox"/>	&Personal Data 2		
3	SCRTY_TBL_GBL	SCRTY_TBL_GBL_WK	<input checked="" type="checkbox"/>	Scty Tbl Gbl Wk		

Component Item Name



For further information, see Creating Menu Definitions and Creating Component Definitions.

Restrictions on Use with PeopleSoft Internet Architecture

If you use the Transfer function to go from a page in an HTML template to a page in a Frame template, the target page may not be displayed. Instead, use the Response object RedirectURL method.

The following code example does *not* work:

```
&rec1 = CreateRecord(Record.EO_PE_TASK_DTL);

&rec1.OPRID.Value = %UserId;

Transfer( False, MenuName.PORTAL_COMPONENTS, BarName.HOMEPAGE,
ItemName.EO_PE_TASK_DTL, Page.EO_PE_TASK_DTL, "A", &rec1);
```

This should be changed to the following:

```
&qs = "ICType=Panel&Menu=" | MenuName.PORTAL_COMPONENTS |
"&Market=GBL&PanelGroupName=" | Page.EO_PE_TASK_DTL | "&Action=A";

&url = %Request.FullURI | "?" | &qs;

%Response.RedirectURL(&url);
```

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```

If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;

```

Returns

None.

Parameters

<i>new_instance</i>	Set this parameter to True to start a new application instance, or to False to use the current window and replace the page that was already there.
<i>menuname</i>	The name of the menu where the page is located prefixed with the reserved word MENUNAME .
<i>barname</i>	The name of the menu bar where the page is located, prefixed with the reserved word BARNAME .
<i>menu_itemname</i>	The name of the menu item where the page is located, prefixed with the reserved word ITEMNAME .
<i>page_group_item_name</i>	The component item name of the page to be displayed on top of the component when it displays. The component item name is specified in the component definition. This parameter must be prefixed with the keyword PAGE .
<i>action</i>	Uses a single-character code as in %Action. Valid actions are "A" (add), "U" (update), "L" (update/display all), "C" (correction), and "E" (data entry).

keylist

A list of field specifications used to select a unique row at level zero in the page you are transferring to, by matching keys in the page you are transferring from. It can also be an already instantiated record object. The keys in fieldlist must uniquely identify a row in the "to" page search record, and they must be shared by the "from" and "to" pages. If a keylist is needed but not specified, or if no unique row is identified, the search dialog appears. If a record object is specified, any field of that record object *that is also a field of the search record for the destination component* will be added to *keylist*.

Example

The example starts a new instance of PeopleTools and transfers to a new page in Update mode. The data in the new page is selected by matching the EMPLID field from the old page.

```
Transfer( true, MENUNAME.ADMINISTER_PERSONNEL, BARNAME.USE, ITEMNAME.
PERSONAL_DATA, PAGE.PERSONAL_DATA_1, "U" );
```

The following example is used with workflow.

```
Local Record &WF_WL_DEFN_VW, &MYREC, &PSSTEPDEFN;
```

```
If All (WF_WORKLIST_VW.BUSPROCNAME) Then
```

```
    &BPNAME = FetchValue(WF_WORKLIST_VW.BUSPROCNAME, CurrentRowNumber());
```

```
    &WLNAME = FetchValue(WF_WORKLIST_VW.WORKLISTNAME, CurrentRowNumber());
```

```
    &INSTANCEID = FetchValue(WF_WORKLIST_VW.INSTANCEID, CurrentRowNumber());
```

```
    &WF_WL_DEFN_VW = CreateRecord(RECORD.WF_WL_DEFN_VW);
```

```
    &PSSTEPDEFN = CreateRecord(RECORD.PSSTEPDEFN);
```

```
    SQLExec("select %List(SELECT_LIST, :1) from %Table(:1) where Busprocname = :2
and Worklistname = :3", &WF_WL_DEFN_VW, &BPNAME, &WLNAME, &WF_WL_DEFN_VW);
```

```
    SQLExec("select %List(SELECT_LIST, :1) from %Table(:1) where Activityname =
:2 and Stepno = 1 and Pathno = 1", &PSSTEPDEFN,
&WF_WL_DEFN_VW.ACTIVITYNAME.Value, &PSSTEPDEFN);
```

```

Evaluate &PSSTEPDEFN.DFLTACTION.Value

When = 0

    &ACTION = "A";

When = 1

    &ACTION = "U";

When-Other

    &ACTION = "U";

End-Evaluate;

&MYREC = CreateRecord(@"RECORD." | &WF_WL_DEFN_VW.WLRECNAME.Value));

SQLExec("Select %List(SELECT_LIST, :1) from %Table(:1) where Busprocname = :2
and Worklistname = :3 and Instanceid = :4", &MYREC, &BPNAME, &WLNAME,
&INSTANCEID, &MYREC);

Transfer( True, @"MENUNAME." | &PSSTEPDEFN.MENUNAME.Value), @"BARNAME." |
&PSSTEPDEFN.BARNAME.Value), @"ITEMNAME." | &PSSTEPDEFN.ITEMNAME.Value),
@"PAGE." | &PSSTEPDEFN.PAGEITEMNAME.Value), &ACTION, &MYREC);

End-if;

```

Related Topics

TransferPage, DoModalComponent

TransferPanel

Syntax

```
TransferPanel ( [PANEL.panel_name] )
```

Description

TransferPanel transfers control to the panel indicated by **PANEL.panel_name** within, or to the panel set with the SetNextPage function.



The **TransferPanel** function is supported for compatibility with previous releases of PeopleTools. Future applications should use TransferPage instead.

TransferPage

Syntax

```
TransferPage ( [PAGE.page_name] )
```

Description

TransferPage transfers control to the page indicated by **PAGE.page_name** within, or to the page set with the SetNextPage function. The page that you transfer to *must* be in the current component or menu. To transfer to a page outside the current component or menu, or to start a separate instance of PeopleTools prior to transfer into, use the Transfer function.



You can't use **TransferPage** from a secondary page.

TransferPage stops all further processing (including the program containing this function), unless you are in the middle of save processing. If **TransferPage** is called in SavePreChange or SavePostChange PeopleCode, the processing completes before the transfer is processed.

If **TransferPage** is called in a RowInit PeopleCode program, the PeopleCode program is terminated. *However*, the component processor continues with its RowInit processing, calling RowInit on the other fields. The actual transfer won't happen until after that completes. You may want to place any TransferPage functions in the Activate event for the page. **TransferPage** can be used without a *page_name* parameter, but the page must have been previously set with SetNextPage.

Any variable declared as a Component variable will still be defined after using a **TransferPage** function.

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```
If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;
```

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Parameters

Page.page_name A String equal to the name of the page you are transferring to, as set in the page definition. The page *must* be in the same component as the page you are transferring from.

Example

The following examples both perform the same function, which is to transfer to the JOB_DATA_4 page:

```
TransferPage(PAGE.JOB_DATA_4);
```

or

```
SetNextPage(PAGE.JOB_DATA_4);
TransferPage( );
```

Related Topics

DoModalComponent, SetNextPage, Transfer

TreeDetailInNode

Syntax

```
TreeDetailInNode(setID, tree, effdt, detail_value, node)
```

Description

TreeDetailInNode determines whether a specific record field value is a descendant of a specified node in a specified tree.



Dynamic Tree Controls are not the same as ActiveX Tree View Controls, and the functions used with the Dynamic Tree Controls can't be used with ActiveX Tree View Controls.

Restrictions on Use in Three-Tier Mode and Web Client

TreeDetailInNode is a client-only function, which limits its use in three-tier mode and in Web Client applications.

- In three-tier mode **TreeDetailInNode** can be used only in processing groups set to run on the client. If the **TreeDetailInNode** function is called in a processing group running on the application server, a runtime error will occur.
- In Web Client applications, where all PeopleCode runs on the application server,

TreeDetailInNode cannot be used.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```

If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;

```

Returns

Returns a Boolean value, True if *detail_value* is a descendant of *node* in *tree*.

Parameters

<i>setID</i>	SetID for the appropriate business unit. This parameter is required. If there is no SetID, you can pass a NULL string ("", not a blank) and a blank will be used.
<i>tree</i>	The tree name that contains the <i>detail_value</i> .
<i>effdt</i>	Effective date to be used in the search. You must use a valid date.
<i>detail_value</i>	The <i>recordname.fieldname</i> containing the value you are looking for.
<i>node</i>	The "owning" tree node name.

Example

This example sets the value of &APPR_RULE_SET to the value at the APPR_RULE_LN record and APPR_RULE_SET fieldname, on the tree ACCOUNT.

```

&APPR_RULE_SET = TreeDetailInNode("SALES", "ACCOUNT", %Date,
APPR_RULE_LN.APPR_RULE_SET, "test");

```

Related Topics

GetTreeNodeValue, GetTreeNodeRecordName, GetSubContractInstance, RefreshTree and Implementing Dynamic Tree Controls

TriggerBusinessEvent

Syntax

```
TriggerBusinessEvent (BUSPROCESS.bus_proc_name,
                     BUSACTIVITY.activity_name,
                     BUSEVENT.bus_event_name)
```

Description

TriggerBusinessEvent triggers a business event and the workflow routings associated with that event. This function should only be used in Workflow PeopleCode. You can edit Workflow PeopleCode via the Event Definition dialog while you are defining a workflow event.



For more information on defining business events, see Defining Events.

Returns

Returns a Boolean value: True if successful, false otherwise. The return value is not optional.



You must check the return from **TriggerBusinessEvent** to see if you have an error. If you have an error, all of the updates up to that **TriggerBusinessEvent** process will be rolled back. However, if you don't halt execution, even if you have an error, all updates *after* the **TriggerBusinessEvent** process *will* be committed. This could result in your database information being out of synch.

Parameters

<i>bus_proc_name</i>	A string consisting of the name of the business process, as defined in the Business Process Designer, prefixed with the reserved word BUSPROCESS .
<i>activity_name</i>	A string consisting of the name of the business activity, as defined in the Business Process Designer, prefixed with the reserved word BUSACTIVITY .
<i>bus_event_name</i>	A string consisting of the name of the business event, as defined in the Business Process Designer, prefixed with the reserved word BUSEVENT .

Example

The following example triggers the Deny Purchase Request event in the Manager Approval activity of the Purchase Requisition business process:

```
&SUCCESS = TriggerBusinessEvent(BUSPROCESS."Purchase Requisition",
    BUSACTIVITY."Manager Approval", BUSEVENT."Deny Purchase Request");
```

Related Topics

GetTreeNodeValue, MarkWLItemWorked



For more information, see Defining Event Triggers.

Truncate

Syntax

```
Truncate(dec, digits)
```

Description

Truncate truncates a decimal number *dec* to a specified precision.

Returns

Returns a Number value equal to *dec* truncated to a *digits* precision.

Parameters

<i>digits</i>	A Number value that sets the precision of the truncation (that is, the number of digits to leave on the right side of the decimal point).
---------------	---

Example

The example sets the value of &NUM to 9, 9.99, -9, then 0.

```
&NUM = Truncate(9.9999, 0);
&NUM = Truncate(9.9999, 2);
&NUM = Truncate(-9.9999, 0);
&NUM = Truncate(0.001, 0);
```

Related Topics

Int , Mod , Round

UnCheckMenuItem

Syntax

```
UnCheckMenuItem(BARNAME.menubar_name, ITEMNAME.menuitem_name)
```

Description

UnCheckMenuItem removes a check mark from the specified menu item. It is useful for menu items that act as a toggle. To apply this function to a pop-up menu, use the PrePopup Event of the field with which the pop-up menu is associated.

If you're using this function with a pop-up menu associated with a page (not a field), the earliest event you can use is the **PrePopup** event for the first "real" field on the page (that is, the first field listed in the **Order** view of the page in Application Designer.)

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```
If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;
```

Restrictions on Use in Three-Tier Mode and Web Client

UnCheckMenuItem is a client-only function, which limits its use in three-tier mode and in Web Client applications.

- In three-tier mode **UnCheckMenuItem** can be used only in processing groups set to run on the client. If the **UnCheckMenuItem** function is called in a processing group running on the application server, a runtime error will occur.
- In Web Client applications, where all PeopleCode runs on the application server, **UnCheckMenuItem** cannot be used at all.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

Returns

None.

Parameters

<i>menubar_name</i>	Name of the menu bar that owns the menuitem, or, in the case of pop-up menus, the name of the pop-up menu that owns the menuitem.
<i>menuitem_name</i>	Name of the menu item.

Example

```
UncheckMenuItem(BARNAME.MYPOPUP1, ITEMNAME.DO_JOB_TRANSFER) ;
```

Related Topics

CheckMenuItem, DisableMenuItem, EnableMenuItem, HideMenuItem

Unencode**Syntax**

```
Unencode (URLString)
```

Description

The **Unencode** function unencodes *URLString*, converting all character codes of the form %xx where xx is a hex number, to the character represented by that number.

Parameters

<i>URLString</i>	Specify the string you want unencoded. This parameter takes a string value.
------------------	---

Returns

An unencoded URL string.

Example

For the following example, the Target Content's URL is:

```
http://mrossi101899/PSPortal.jrun/?ICType=Panel&Menu=MANAGE_COMMITMENT_CONTROL&Market=GBL&PanelGroupName=KK_NOTIFY_GRP
```

The QueryString parameters are those values of the Target Content's URL that are specified by a &xxx. For example, if you wanted to know the value in the Target Content's URL for the parameter "Menu", then the following PeopleCode would return "MANAGE_COMMITMENT_CONTROL":

```
&MENU = Unencode(%Request.GetParameter("Menu"));
```

This method works for any querystring in the Target Content's URL.

If the link is constructed in a PeopleSoft Internet Architecture page, and the value of a link field, you should not call `EncodeURL` to encode the entire URL, as the PIA architecture does this for you. You will still need to unencode the parameter value when you retrieve it, however.

Related Topics

`EncodeURL`, `EncodeURLForQueryString`

Ungray

Syntax

```
Ungray(scrollpath, target_row, [recordname.]fieldname)
```

where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]]
RECORD.target_recname
```

To prevent ambiguous references, you can also use **SCROLL**.*scrollname*, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see [References Using Scroll Path Syntax and Dot Notation](#).

Generally, you will want to put the function on the same scroll level as the field that is being changed in `RowInit` (which executes on every row) or `FieldChange` (which executes on the current row). This simplifies the function's syntax to:

```
Ungray(fieldname)
```

A typical use of the more complex syntax is when looping through rows on a scroll on a lower level than the program.

Description

Ungray makes a gray (non-editable) page field editable, if the field was grayed with a call to the **Gray** function. If the page field is made display-only in the Page Field Properties dialog, then **Ungray** has no effect.



This function remains for backward compatibility only. Use the Enabled Field property instead. See [Data Buffer Access](#).

The Gray, **Ungray**, Hide, and Unhide functions usually appear in RowInit programs that set up the initial display of data, and FieldChange programs that change field display based on changes the end-user makes to a field.

Event Restrictions

This function shouldn't be used in any event prior to RowInit.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
<i>target_row</i>	The row number of the target row. If this parameter is omitted, the function assumes the row on which the PeopleCode program is executing.
<i>[recordname.]fieldname</i>	The name of the field to ungray. The field can be on scroll level one, two, or three of the active page. The <i>recordname</i> prefix is required if the call to Ungray is in a record definition other than <i>recordname</i> .



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Example

The following example checks to see if a person's emergency contact is the same as their home address and phone, then grays or ungrays the fields accordingly. In a typical case, this program would be in the FieldChange event.

```

If SAME_ADDRESS_EMPL = "Y" Then
    Gray(STREET1);
    Gray(STREET2);
    Gray(CITY);
    Gray(STATE);
    Gray(ZIP);
    Gray(COUNTRY);
    Gray(HOME_PHONE);
    STREET1 = PERSONAL_DATA.STREET1;
    STREET2 = PERSONAL_DATA.STREET2;
    CITY = PERSONAL_DATA.CITY;
    STATE = PERSONAL_DATA.STATE;
    ZIP = PERSONAL_DATA.ZIP;
    COUNTRY = PERSONAL_DATA.COUNTRY;

```

```

    HOME_PHONE = PERSONAL_DATA.HOME_PHONE;
Else
    Ungray(STREET1);
    Ungray(STREET2);
    Ungray(CITY);
    Ungray(STATE);
    Ungray(ZIP);
    Ungray(COUNTRY);
    Ungray(HOME_PHONE);
End-if;

```

Related Topics

Gray, Hide, Unhide

Unhide

Syntax

```
Unhide(scrollpath, target_row, [recordname.]fieldname)
```

where *scrollpath* is:

```

[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]]
RECORD.target_recname

```

To prevent ambiguous references, you can use **SCROLL.scrollname**, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Generally, you will want to put the function on the same scroll level as the field that is being changed in RowInit (which executes on every row) or FieldChange (which executes on the current row). This simplifies the function's syntax to:

```
unhide(fieldname)
```

A typical use of the more complex syntax is when looping through rows on a scroll on a lower level than the program.

Description

Unhide makes a field visible that was previously hidden with Hide. If the field was hidden by setting its Invisible property in Page Field Properties dialog box, then **Unhide** has no effect.



This function remains for backward compatibility only. Use the Visible Field property instead. See Data Buffer Access.

Event Restrictions

This function shouldn't be used in any event prior to RowInit.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
<i>target_row</i>	The row number of the target row. If this parameter is omitted, the function assumes the row on which the PeopleCode program is executing.
[<i>recordname.</i>] <i>fieldname</i>	The name of the field to unhide. The field can be on scroll level one, two, or three of the active page. The <i>recordname</i> prefix is required if the call to Unhide is in a record definition other than <i>recordname</i> .



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Example

The following example sets security for displaying a person's password:

```
If (&DISPLAY) Then
    Unhide(EMPLOYEE.PASSWORD);
Else
    Hide(EMPLOYEE.PASSWORD);
End-if;
```

Related Topics

Gray, Hide, Ungray

UnhideRow

Syntax

```
UnhideRow(scrollpath, target_row)
```

Where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]
RECORD.target_recname]
```

To prevent ambiguous references, you can use **SCROLL**.*scrollname*, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Description

UnhideRow programmatically unhides a row that has been hidden by **HideRow**. It unhides the specified row and any dependent rows at a lower scroll level.



This function remains for backward compatibility only. Use the Visible Row property instead. See Data Buffer Access.

UnhideRow works by putting the row that you unhide to the last non hidden row in the list. When **UnhideRow** is used in a loop, you have to process rows from low to high to achieve the correct results.



UnhideRow cannot be executed from the same scroll level where the insertion will take place, or from a lower scroll level. Place the PeopleCode in a higher scroll level record.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
<i>target_row</i>	An integer specifying which row in the scroll to unhide.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Example

This example uses a For loop to unhide rows previously hidden with the Hide function. Note that the loop counts backward to assure that renumbering of rows will not affect the loop:

```
AE_ROW_COUNT = ActiveRowCount (RECORD.AE_STMT_TBL) ;
for &ROW = ActiveRowCount (RECORD.AE_STMT_TBL) to 1 step - 1
```

```

UnhideRow(RECORD.AE_STMT_TBL, &ROW);
UpdateValue(RECORD.AE_STMT_TBL, &ROW, AE_ROW_NUM, &ROW);
end-for;

```

Related Topics

HideRow

UnhideScroll

Syntax

```
UnhideScroll(Scrollpath)
```

Where *scrollpath* is:

```

[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]
RECORD.target_recname

```

To prevent ambiguous references, you can use **SCROLL.scrollname**, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Description

UnhideScroll programmatically unhides a scroll area that has been hidden with HideScroll. It unhides the specified scroll and any associated scrolls at a lower level.



This function remains for backward compatibility only. Use the ShowAllRows Rowset method instead. See Data Buffer Access.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
-------------------	---



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Example

This example clears the contents of a level-one hidden scroll, then unhides it:

```
ScrollFlush(RECORD.ORDER_INQ_INV);
UnhideScroll(RECORD.ORDER_INQ_INV);
```

The following example hides or unhides a level-three scroll:

```
If APPR_QTY_SW = "N" Then
    HideScroll(RECORD.APPR_RULE_LN, CurrentLineNumber(1),
RECORD.APPR_RULE_DETL, CurrentLineNumber(2), RECORD.APPR_RULE_QTY);
Else
    UnhideScroll(RECORD.APPR_RULE_LN, CurrentLineNumber(1),
RECORD.APPR_RULE_DETL, CurrentLineNumber(2), RECORD.APPR_RULE_QTY);
End-If;
```

Related Topics

HideScroll, RowScrollSelect, RowScrollSelectNew, ScrollSelect, ScrollSelectNew, SortScroll

UniformSeriesPV

Syntax

```
UniformSeriesPV(int_rate,n_per)
```

Description

UniformSeriesPV calculates the present value of a single monetary unit after a uniform series of payments at a specified interest rate.

Returns

Returns a Number equal to the value of a single unit after *n_per* payments at an interest rate of *int_rate*.

Parameters

<i>int_rate</i>	A Number specifying the interest rate on the basis of which to calculate the return value.
<i>n_per</i>	A Number specifying the number of payments in the uniform series.

Example

The example sets &NUM to 3.790786769408448256:

```
&NUM = UniformSeriesPV(10,5);
```

Related Topics

SinglePaymentPV

UpdateSysVersion

Syntax

```
UpdateSysVersion( )
```

Description

UpdateSysVersion coordinates system changes and changes to system objects maintained by pages, such as messages and Set Tables. This function is not normally used in standard applications and should only be used in PeopleSoft-provided extensions of PeopleTools.

Returns

Returns the updated system version Number.

Example

The following example could be used to maintain the version number on MESSAGE_SET_TBL, which controls the refreshing of swap files for the message entries:

```
VERSION = UpdateSysVersion( );
```

UpdateValue

Syntax

```
UpdateValue(scrollpath, [recordname.]fieldname, target_row, value)
```

where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]]
```

To prevent ambiguous references, you can use **SCROLL.scrollname**, where *scrollname* is the same as the scroll level's primary record name.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Description

UpdateValue updates the value of a specified field with the *value* provided. The value must be of a data type compatible with the *field*.



This function remains for backward compatibility only. Use the Value Field property instead. See Data Buffer Access.

Returns

None.

Parameters

<i>scrollpath</i>	A construction that specifies a scroll level in the component buffer.
<i>target_row</i>	An integer specifying the row of the field to update.
[<i>recordname.</i>] <i>fieldname</i>	The name of the field that you want to update. The field can be on scroll level one, two, or three of the active page. The <i>recordname</i> prefix is required if the call to UpdateValue is in a record definition other than <i>recordname</i> .
<i>value</i>	The new value to put into the target field.



For more information on scroll path syntax, see References Using Scroll Path Syntax and Dot Notation.

Example

This example updates values in the level-one scroll:

```
For &I = 1 To &ROW_CNT
    UpdateValue(RECORD.ASGN_CMP_EFFDT, &I, ITEM_SELECTED, "Y");
End-For;
```

The next example loops through rows in the level-two scroll:

```
For &I = 1 To &CURRENT_L2
    UpdateValue(RECORD.ASGN_CMP_EFFDT, &CURRENT_L1, RECORED.SOME_L2_RECORD, &I,
    TO_CUR, &HOME_CUR);
End-For;
```

Related Topics

FetchValue, PriorValue

Upper

Syntax

Upper (*str*)

Description

Upper converts a text string to all upper case. This function can be used to perform a case-insensitive string comparison.

Returns

Returns a String value equal to *str* converted to all upper case.

Parameters

<i>str</i>	A String to convert to upper case.
------------	------------------------------------

Example

The following example converts the contents of two string variables to upper case before determining if they are equal to simulate a case-insensitive comparison:

```
If Upper(&STR1) = Upper(&STR2) Then
    /* do something */
End-If;
```

Related Topics

Lower, Proper

Value

Syntax

Value (*str*)

Description

Value converts a string representing a number to the number.

Returns

Returns the Number value represented by *str*.

Parameters

<i>str</i>	A String value representing a number.
------------	---------------------------------------

Example

The example sets &VAL1 to 5.25 and &VAL2 to 12500:

```
&VAL1 = Value("5.25");  
&VAL2 = Value("12,500");
```

ViewAttachment

Syntax

```
ViewAttachment(URLSource, DirAndFileName, UserFile)
```

where *URLSource* can have **one** of the following forms:

URL.URLName

OR a string URL, such as

```
ftp://worker:password@ftp.ps.com/
```

Description

The **ViewAttachment** function enables an end-user to view a file on a system. The file will actually be downloaded into the directory indicated by the TEMP environment variable.



.exe and .bat files can be launched instead of viewed.

Restrictions on Use in PeopleCode Events

ViewAttachment is a "think-time" function, which means that it shouldn't be used in any of the following PeopleCode events:

- SavePreChange
- SavePostChange
- Workflow
- RowSelect
- Any PeopleCode event that fires as a result of a Select or SelectNew method, or any of the ScrollSelect functions.



For more information, see Think-Time Functions.

File Name Considerations

When the file is transferred using `AddAttachment` or `PutAttachment`, the following characters are replaced with an underscore:

- space
- semi-colon
- plus sign
- percent sign
- ampersand
- apostrophe
- exclamation point
- @ sign
- pound sign
- dollar sign

Parameters

URLSource A reference to a URL. This can be either a URL name, in the form `URL.URLName`, or a string. This will be where the file will end up.

DirAndFileName A directory and filename. This is appended to the *URLSource* to make up the full URL of the file.



The *URLSource* requires "/" slashes. Because the *DirAndFileName* parameter is appended to the URL, it also requires only "/" slashes. "\" are NOT supported in anyway for either the *URLSource* the *DirAndFileName* parameter.

UserFile The name of the file on the source system.

Returns

An integer value. You can check for either the integer or the constant:

<i>Number</i>	<i>Constant</i>	<i>Description</i>
0	%Attachment_Success	File was attached successfully.

Number	Constant	Description
1	%Attachment_Failed	File was not successfully attached.
2	%Attachment_Cancelled	File attachment didn't complete because the operation was canceled by the user.
3	%Attachment_FileTransferFailed	File transfer didn't succeed.
5	%Attachment_NoDiskSpaceWebServ	No disk space on the web server.
7	%Attachment_DestSystNotFound	Cannot locate destination system for ftp.
8	%Attachment_DestSystFailedLogin	Unable to login to destination system for ftp.

Example

```

&retcode = ViewAttachment(URL.MYFTP, ATTACHSYSFILENAME, ATTACHUSERFILE);

If (&retcode = %Attachment_Failed) Then

    MessageBox(0, "File Attachment Status", 0, 0, "ViewAttachment failed");

End-If;

If (&retcode = %Attachment_Cancelled) Then

    MessageBox(0, "File Attachment Status", 0, 0, "ViewAttachment cancelled");

End-If;

If (&retcode = %Attachment_FileTransferFailed) Then

    MessageBox(0, "File Attachment Status", 0, 0, "ViewAttachment failed: File
Transfer did not succeed");

End-If;

/* following error message only in PeopleSoft Internet Architecture */

If (&retcode = %Attachment_NoDiskSpaceAppServ) Then

    MessageBox(0, "File Attachment Status", 0, 0, "ViewAttachment failed: No disk
space on the app server");

```

```

End-If;

/* following error message only in PeopleSoft Internet Architecture */

If (&retcode = %Attachment_NoDiskSpaceWebServ) Then

    MessageBox(0, "File Attachment Status", 0, 0, "ViewAttachment failed: No disk
space on the web server");

End-If;

If (&retcode = %Attachment_FileExceedsMaxSize) Then

    MessageBox(0, "File Attachment Status", 0, 0, "ViewAttachment failed: File
exceeds the max size");

End-If;

If (&retcode = %Attachment_DestSystNotFound) Then

    MessageBox(0, "File Attachment Status", 0, 0, "ViewAttachment failed: Cannot
locate destination system for ftp");

End-If;

If (&retcode = %Attachment_DestSysFailedLogin) Then

    MessageBox(0, "File Attachment Status", 0, 0, "ViewAttachment failed: Unable
to login into destination system for ftp");

End-If;

```

Related Topics

AddAttachment, DeleteAttachment, GetAttachment, PutAttachment, Using the Attachment Functions

ViewURL

Syntax

```
ViewURL(URL_str | URL.URL_ID [, ReplaceCurrentPage])
```

Description

Launches the default browser and navigates to the location specified by *URL_str* or *URL.URL_ID*. This is a deferred execution command: the browser is launched after any

executing PeopleCode has run to completion. You can also specify whether the new page will launch a new browser, or replace the current page in the browser.



The **ViewURL** function will not work if being run on a Window 95 operating system and Internet Explorer version 3.02 or greater has not been installed.



Portal applications should use the RedirectURL Response class method instead of ViewURL.

Restrictions on use with Windows Client

The *ReplaceCurrentPage* parameter is not available for Windows Client.

Parameters

<i>URL_str</i> URL.URL_ID	Specify the location to where you want to navigate. You can specify <i>either</i> a URL string, or , a URL saved in the URL table, by specifying the keyword URL followed by a dot and the URL Identifier.
------------------------------------	---



For more information about the URL table and URL identifiers, see URL Maintenance.

<i>ReplaceCurrentPage</i>	Specify whether the new page should replace the current page in the browser, or launch a new browser. This parameter takes a boolean value: True to replace existing page, False to launch a new browser. The default value is False.
---------------------------	---

Returns

None.

Example

```
If &MyPage Then
    ViewURL(URL.MYPAGE);
Else
    ViewURL("http://www.PeopleSoft.com");
End-If;
```

Related Topics

GetURL

Warning

Syntax

`Warning str`

Description

You typically use the **Warning** function in FieldEdit or SaveEdit PeopleCode to display a message alerting the end-user about a potentially incorrect data entry or change. It differs from **Error** in that it does not prevent the end-user from taking an action, and it does not stop processing in the PeopleCode program where it occurs.

Warning is also used in RowDelete and RowSelect PeopleCode, where its behavior is specialized. See Warnings in RowDelete and Warnings in RowSelect.

The text of the warning message (the *str* parameter), should always be stored in the Message Catalog and retrieved using the **MsgGet** or **MsgGetText** function. This makes it much easier to translate the text, and it also lets you include more detailed Explain text about the warning.

Returns

None.

Parameters

<i>str</i>	A string containing the text of the warning message. This string should always be stored in the Message Catalog and retrieved via the MsgGet or MsgGetText function. This makes translation much easier and also allows you to provide detailed Explain text about the warning.
------------	---

Warnings in FieldEdit and SaveEdit

The primary use of **Warning** is in FieldEdit and SaveEdit PeopleCode:

- In FieldEdit, **Warning** displays a message and highlights the relevant field.
- In SaveEdit, **Warning** displays a message, but does not highlight any field. You can move the cursor to a specific field using the SetCursorPos function.

Warnings in RowDelete

When the end-user attempts to delete a row of data, the system first prompts for confirmation. If the end-user confirms, the RowDelete event fires. A **Warning** in the RowDelete event displays a warning message with **OK** and **Cancel** buttons. If the end-user clicks **OK**, the row is deleted. If the end-user clicks **Cancel**, the row is not deleted.

Warnings in RowSelect

The behavior of **Warning** in RowSelect is totally anomalous and maintained for backward compatibility only. Use it to filter rows being added to a page scroll after the rows have been selected and brought down to the client. **Warning** causes the Component Processor to skip the current row (so that it is not added to the page scroll), then continue processing. No message is displayed.



Do not use **Warning** in this fashion. Use DiscardRow for replacement instead.

Warnings in Other Events

Do not use the **Warning** function in any of the remaining events, which include:

- FieldDefault
- FieldFormula
- RowInit
- FieldChange
- RowInsert
- SavePreChange
- SavePostChange

The end-user has no control over processing that occurs in these events. If the Component Processor encounters a condition that would warrant a warning message in these events, the end-user can't fix it. The Component Processor therefore requires the user to cancel the component to avoid unpredictable results, which results in loss of any changes that the end-user has made.

Example

The following example shows a warning that alerts an end-user to a possible error, but allows the end-user to accept the change:

```
if All(RETURN_DT, BEGIN_DT) and
    8 * (RETURN_DT - BEGIN_DT) < (DURATION_DAYS * 8 + DURATION_HOURS) then
    warning MsgGet(1000, 1, "Duration of absence exceeds standard hours for
number of days absent.");
end-if;
```

Related Topics

Error, MsgGet, MsgGetText, WinMessage

Weekday

Syntax

```
Weekday(dt)
```

Description

Weekday calculates the day of the week based on a date value.

Returns

Returns a Number value representing the day of the week. 1 is Sunday, 7 is Saturday.

Parameters

<i>dt</i>	A Date value. Weekday determines the day of the week that <i>dt</i> falls on.
-----------	--

Example

If &Date_HIRED equals October 30, 1996, a Monday, then the following statement sets &DAY_HIRED to 2:

```
&DAY_HIRED = Weekday(&Date_HIRED);
```

Related Topics

Date, Date3, DateValue, Day, Days360, Days365, Month, Year

While

Syntax

```
While  
    logical_expression  
    statement_list  
End-while
```

Description

The While loop causes the statements of the *statement_list* to be repeated until *logical_expression* is false. Statements of any kind are allowed in the loop, including other loops. A **Break** statement inside the loop causes execution to continue with whatever follows the end of the loop. If the **Break** is in a nested loop, the **Break** does not apply to the outside loop.

Example

The following example counts from 0 to 10:

```
&COUNTER = 1;  
while &COUNTER <= 10
```

```

    WinMessage(MsgGet(21000, 1, "Count is %1", &COUNTER));
    &COUNTER = &COUNTER + 1;
end-while;

```

Related Topics

Repeat

WinEscape

Syntax

```
WinEscape( )
```

Description

WinEscape simulates the behavior of the ESC key to cancel a page. This causes any changes to the page subsequent to the last successful save to be lost, and returns the Component Processor to the reset state (the state prior to starting a page or entering a search key).

Restrictions on Use with Internet Client Applications

This function isn't supported by Internet Client.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Example

The following example checks the user input to see if the action requested is valid and cancels the page if the action is invalid:

```

If Substring(%Page, 1, 8) = Substring(PAGE.JOB_DATA1, 1, 8) and
    %Mode = "A" Then
    If All(NAME) Then
        If %Page < PAGE.JOB_DATA1 Then
            WinMessage(MsgGet(1000, 81, "The selected Employee ID exists. It
cannot be added.));
            WinEscape( );
        End-if;
    Else
        If %Page = PAGE.JOB_DATA1 Then
            WinMessage(MsgGet(1000, 82, "The selected Employee ID does not exist.
The concurrent Job cannot be added.));
            WinEscape( );
        End-if;
    End-if;

```

```
End-if;
End-if;
```

Related Topics

EndModal, Error, Warning, DoCancel

WinExec

Syntax

```
WinExec(command_line, window_option [, synch_exec])
```

Description

WinExec executes a Windows executable file (.COM, .EXE, .BAT, or .PIF) specified in the *command_line* string. It is suitable for executing short batch-processing jobs that will complete execution and terminate, returning control to PeopleTools.

The function can make either a synchronous or asynchronous call. Synchronous execution will act as a modal or "think-time" function, suspending the PeopleSoft application until the called executable completes. This is appropriate if you want to force the end-user (or the PeopleCode program) to wait for the function to complete its work before continuing processing. Asynchronous processing, which is the default, launches the executable and immediately returns control to the calling PeopleSoft application.

Returns

WinExec returns a Number indicating whether the function completed successfully:

Error	Description
0	Completed successfully
non-zero	Error

Parameters

window_option Specifies the style of the newly created window. You can specify either a numeric value or a constant:

Option	Constant	Description
1	%WinExec_Normal	Normal window with focus
2	%WinExec_Minimized	Minimized window with focus
3	%WinExec_Maximized	Maximized window with focus
4	%WinExec_NormalNoFocus	Normal window without

		focus
7	%WinExec_MaximizedNoFocus	Maximized window without focus

command_line

The name of a .COM, .EXE, .BAT, or .PIF file. After the filename, additional command switches or parameters can be specified.

synch_exec

An optional Boolean value specifying whether to execute synchronously (True) or asynchronously (False). The default is asynchronous execution.

Restrictions on Use in PeopleCode Events

When **WinExec** is used to execute a program synchronously (that is, if its *synch_exec* parameter is set to True) it behaves as a think-time function, which means that it can't be used in any of the following PeopleCode events:

- SavePreChange
- SavePostChange
- Workflow
- RowSelect
- Any PeopleCode event that fires as a result of a ScrollSelect, ScrollSelectNew, RowScrollSelect or RowScrollSelectNew function call.

Restrictions on Use in Three-Tier Mode and PeopleSoft Internet Architecture

Because it is specific to Windows, **WinExec** is a client-only function, which limits its use in three-tier mode and in PeopleSoft Internet Architecture applications.

- In three-tier mode **WinExec** can be used only in processing groups set to run on the client. If the **WinExec** function is called in a processing group running on the application server, a runtime error will occur.
- In PeopleSoft Internet Architecture applications, where all PeopleCode runs on the application server, **WinExec** cannot be used at all.

In PeopleCode programs that run on the application server, you can use the `Exec_` function.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

Example

This example executes, synchronously, a program called control.exe:

```
If INTERNATIONAL_CHG = "Y" Then
    &TMP = WinExec("control.exe", 7, false);
End-if;
```

The following code sample is identical to the previous, but it uses a constant instead of a numeric value for the window option:

```
If INTERNATIONAL_CHG = "Y" Then
    &TMP = WinExec("control.exe", %WinExec_MaximizedNoFocus, false);
End-if;
```

Related Topics

Declare Function, Exec, RemoteCall, ScheduleProcess

WinMessage

Syntax

```
WinMessage(message [, style] [, title])
```

Description



The **WinMessage** function is supported for compatibility with previous releases of PeopleTools. Future applications should use MessageBox instead.

The **WinMessage** function displays a message in a message box.

WinMessage can be used for simple informational display, where the end-user reads the message, then clicks an **OK** button to dismiss the message box. **WinMessage** can also be used as a way of branching based on end-user choice, in which case the message box contains two or more buttons (such as **OK** and **Cancel** or **Yes**, **No**, and **Cancel**). The value returned by the function tells you which button the end-user clicked, and your code can branch based on that value.

If **WinMessage** displays more than one button, it causes processing to stop while it waits for user response. This makes it a "user think-time" function, restricting its use in certain PeopleCode events.

The contents of the message displayed by **WinMessage** can be passed to the function as a string, but unless you are using the function for testing purposes you should always retrieve the message from the Message Catalog using the **MsgGet** or **MsgGetText** function. This has the advantage of making the messages much easier to localize and maintain.

Restrictions on Use in PeopleCode Events

If the *style* parameter specifies more than one button, or if the *style* parameter is omitted, **WinMessage** returns a value based on user response and interrupts processing until the user has clicked one of the buttons. This makes it a "user think-time" function, subject to the same restrictions as other think-time functions (see Think-Time Functions), which means that it cannot be used in any of the following PeopleCode events:

- SavePreChange
- Workflow
- RowSelect
- SavePostChange
- Any PeopleCode event that fires as a result of a ScrollSelect, ScrollSelectNew, RowScrollSelect or RowScrollSelectNew function call.

If the *style* parameter specifies a single button (that is, the **OK** button), the function can be called in any PeopleCode event.



In Windows Client, MessageBox dialogs include an **Explain** button to display more detailed information stored in the Message Catalog. The presence of the **Explain** button has no bearing on whether a message box behaves as a think-time function.

The *style* parameter is optional in **WinMessage**. If *style* is omitted **WinMessage** displays **OK** and **Cancel** buttons, which causes the function to behave as a think-time function. To avoid unnecessary restrictions, you should always pass an appropriate value in the **WinMessage** *style* parameter.

Restrictions on Use with a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```

If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;
```

Message Box Icons

In the PeopleSoft Internet Architecture, you can't change the icon of a message box. You can change the number and type of buttons, as well as the default button, but the message always displays with the warning icon (a triangle with an exclamation mark in it.)

Returns

If the *style* parameter is provided, **WinMessage** optionally returns a Number value. If the *style* parameter is omitted, **WinMessage** optionally returns a Boolean value: True if the **OK** button was clicked, otherwise it returns False.

The return value is zero if there is not enough memory to create the message box.

If the *style* parameter is provided, **WinMessage** returns one of the following Number values:

<i>Value</i>	<i>Constant</i>	<i>Meaning</i>
-1	%MsgResult_Warning	Warning was generated.
1	%MsgResult_OK	OK button was selected.
2	%MsgResult_Cancel	Cancel button was selected.
3	%MsgResult_Abort	Abort button was selected.
4	%MsgResult_Retry	Retry button was selected.
5	%MsgResult_Ignore	Ignore button was selected.
6	%MsgResult_Yes	Yes button was selected.
7	%MsgResult_No	No button was selected.



In PeopleSoft Internet Architecture, pressing the ESC key has no effect. In Windows client, if a WinMessage dialog box has a **Cancel** button, the same value will be returned if either the ESC key is pressed or the **Cancel** button is selected. If the WinMessage dialog box has no **Cancel** button, pressing ESC has no effect.

Parameters

Message

Text displayed in message box. Normally you will want to use the **MsgGet** or **MsgGetText** function to retrieve the message from the Message Catalog.

Title

Title of message box.

Style

Either a numerical value or a constant specifying the contents and behavior of the dialog box. This parameter is calculated by cumulatively adding either a value or a constant from each category below:

Category	Value	Constant	Meaning
Buttons	0	%MsgStyle_OK	The message box contains one pushbutton: OK.
	1	%MsgStyle_OKCancel	The message box contains two pushbuttons: OK and Cancel.
	2	%MsgStyle_AbortRetryIgnore	The message box contains three pushbuttons: Abort, Retry, and Ignore.
	3	%MsgStyle_YesNoCancel	The message box contains three pushbuttons: Yes, No, and Cancel.
	4	%MsgStyle_YesNo	The message box contains two push buttons: Yes and No.
	5	%MsgStyle_RetryCancel	The message box contains two push buttons: Retry and Cancel.



In Windows client, if the *style* parameter is omitted, the message window displays an OK and Cancel button with an "i" (Information) icon.



Note. The following values for *style* can only be used in Windows Client. They have no affect in PeopleSoft Internet Architecture.

Category	Value	Constant	Meaning
Default Button	0	%MsgDefault_First	The first button is the default.
	256	%MsgDefault_Second	The second button is the default.
	512	%MsgDefault_Third	The third button is the default.
Icon	0	%MsgIcon_None	None
	16	%MsgIcon_Error	A stop-sign icon appears in the message box.
	32	%MsgIcon_Query	A question-mark icon appears in the message box.
	48	%MsgIcon_Warning	An exclamation-point icon appears in the message box.
	64	%MsgIcon_Info	An icon consisting of a lowercase letter "i" in a circle appears in the message box.

Related Topics

Date, Date3, DateValue, Day, Days360, Days365, Month, Weekday

CHAPTER 4

PeopleCode Classes

This section contains the reference material for all classes (and objects) used with PeopleTools.

Throughout this section, we use typographical conventions to distinguish between different elements of the PeopleCode language, such as bold to indicate function names, italics for arguments, and so on.



For more information, see [Typographical Conventions and Visual Cues](#).

Classes by Category

The following topics subdivide the PeopleCode classes by functional category and provide links from within each category to the reference entries.

API Classes

Component Interface Classes

PortalRegistry Classes

Query Classes

Search Classes

Session Class

Tree Classes

Application Engine Class

AESession Class

Data Buffer Access Classes

Field Class

Record Class

Row Class

Rowset Class

Internet Class

Internet Script Classes

PortalRegistry Classes

Page Display Classes

Field Class

GridColumn Class

Page Class

Integration Classes

File Class

Business Interlink Class

Internet Script Classes

Miscellaneous Classes

Array Class

Java Class

ProcessRequest Class

SQL Class

AESection Class

Before PeopleTools 8.0, users could perform SQL directly on the Application Engine tables, thereby changing their SQL and Application Engine "flow" in a dynamic manner, prior to

running their applications. Some applications, for example, let the user input their "rules" in a user-friendly application, then convert these rules, at save time, into Application Engine constructs.

With PeopleTools 8.0, Application Engine programs should not perform SQL directly on the Application Engine tables, as they will now be "system" tables, and will be cached. SQL on the Application Engine tables may not be accurately reflected when the applications are executed, because of the caching mechanism.

To overcome this problem, developers will get two basic operations that will let them modify their Application Engine programs from their online pages:

- Ability to modify SQL definitions, which will be referenced within Application Engine SQL using the PeopleCode SQL class and the meta-SQL function %SQL.



For more information, see SQL Class and %SQL.

- Ability to dynamically change the execution flow of a given Application Engine section.

In order to do this latter operation, the AERSection class is introduced. This section object is used to modify, using PeopleCode, the steps and SQL associated with a given section.

Before getting into the specifics of the AERSection class, the following are some general terms to understand:

- **base** section

The **base** section represents the Application Engine section you are working on. This is the section that you are changing in your PeopleCode. In other words, it's the target section.

- **template** section

The **template** section represents the Application Engine section that is the source, or "model" for the section that you are building. You will copy *from* the template section *to* the base section.

The template must exist in the database before you can use it.

If the base section you specify doesn't exist, a *new* base section will be created. This allows you to dynamically create sections as needed.

You can copy steps (and their attributes) from the template to the base. The only attribute of the step you can modify is the SQL statement that gets executed.



When you open or get an AERSection object, (that is, the base section) any existing steps in the section will be deleted. You must add a new step to the section before you can modify it.

The main assumption for this class is that your rules are dynamic primarily in the SQL that they execute, but that the structure of the rules are static, or at least defined well enough that a standard template can be applied.

Note that you cannot update the Application Engine *actions* that are not SQL-related (that is, PeopleCode, Call Section, or Log Message). In other words, you can't change the PeopleCode associated with a PeopleCode action within a step. You can add a step containing a PeopleCode action to your new section, but you cannot change the PeopleCode dynamically.



For more information about steps, see Steps.

How an AERSection is Accessed

When an AERSection is opened (or accessed), the system first looks to see if it exists with the given input parameters.

If the base section you specify doesn't exist, a *new* base section will be created. This allows you to dynamically create sections as needed.

If an effective date is specified (with *effdt*), but there is no match using that effective date, the AERSection is opened using the base effective date of '1900-01-01'.

The market defined for the current component is used for the section to be open. If no section with this market exists, the default GBL is used.

If you open an AERSection object from within a running Application Engine program, the market value is set to the value of the current process.

The database platform you're currently running is used as the database platform for the section to be opened. If no section with this database platform exists, the default "none" is used.

To find the right section to open, the precedence is:

1. Market
2. Database platform
3. Effective date

When a section is closed and its changes are saved, the market and database platform from the system that performed the changes will be used as the market and database platform for the modified section.

Let's take an example. Assume that you're running on DB2, and your current market is set to 'MKT'. Assume that you want to open a section called Sect1 in an Application Engine program called TestAppl, and that your data looks something like this:

Number	Application Engine Program	Application Engine Section	Market	Platform	Effective Date
1	TestAppl	Sect1	USA	DB2	1990-01-01
2	TestAppl	Sect1	USA	None	1900-01-01
3	TestAppl	Sect1	GBL	None	1900-01-01

If you use:

```
&SECTION.Open("TestAppl", "Sect1", "1998-12-14");
```

Then the third section above will be used, because the market takes highest precedence. When the section is saved, the values for market and platform will be updated to the current system values.



When you open or get an AESection object, (that is, any base section) any existing steps in the section will be deleted. You must add a new step to the section before you can modify it.

The other attributes of the section, however, are retained, with the exceptions noted above.

AESection Example

Assume that you have a template section called TEMPLATE in the Application Engine program called MY_APPL. The template looks like this:

Steps	Actions
NewStep1	DO When
	DO Select
	SQL
NewStep2	DO Select
	Call Section

Also, assume that you have a base section called DYN_SECT in the Application Engine program called RULES. When you start, this section looks like this:

Steps	Actions
Step1	DO Select
	Call Section
	DO Until
Step2	Call Section

Here's your PeopleCode:

```

Local AERSection &Section;

&Section = GetAERSection("RULES", "DYN_SECT");

    /* Open the base section */

&Section.SetTemplate("MY_APPL", "TEMPLATE");

    /* Set the template section */

&Section.AddStep("NewStep2");

    /* Insert NewStep2      */

    /* Do some SQL stuff here */

&Section.SetSQL("DO_SELECT", &MySql);

    /* Modify the SQL in the added step */

&Section.Save();

&Section.Close();

/* Save and close */

```

The base section looks like this after execution:

Steps	Actions
NewStep2	DO Select
	Call Section



The existing steps in the base section have been *overwritten* by the new step from the template section.

Declaring an AERSection Object

You should declare an AERSection object as type AERSection. For example:

```
Local AERSection &SECTION;
```

Scope of an AERSection Object

An AERSection object can only be instantiated from PeopleCode.

The AERSection Object is designed for use within online pages. Typically, dynamic sections should be constructed in response to an end-user action.



Do not call an AERSection object from an Application Engine PeopleCode Action. If you need to access another section, use Call Section Actions instead.

AERSection Class Built-in Function

GetAERSection

AERSection Class Methods

AddStep

Syntax

```
AddStep(ae_step_name [, NewStepName])
```

Description

The given step name from the template section is added as the next step into the base section, and named the existing step name.



When you open or get a section, all the existing steps are deleted. The first time you execute AddStep, you add the first step. The second time you execute AddStep, you add the second step, and so on.

All attributes of the step are copied, including all of its actions. The only changeable attribute of a step are its SQL statements, which can be modified using the SetSQL method. SetSQL is run on the *current* step, that is, the step most recently added.

You can also change the name of the step by using the optional parameter *NewStepName*.

If the step named does not exist in the template, an error will occur. Likewise, if you have not opened or set the template for the section, you will get an error.

Parameters

<i>ae_step_name</i>	Specify the step name from the template to be added as the <u>next</u> step in the base section. This parameter takes a string value.
<i>NewStepName</i>	Specify a new name for the step to be added. This parameter is optional, and takes a string value.

Returns

None.

Example

See AERSection Example.

Related Topics

SetSQL

Close

Syntax

```
Close()
```

Description

Close closes the AERSection object. Any unsaved changes to the section will be discarded.

Parameters

None.

Returns

None.

Example

See AERSection Example.

Related Topics

Save, Open, GetAERSection

Open

Syntax

```
Open(ae_applid, ae_section, [effdt])
```

Description

The **Open** method associates the AERSection object with the given Application Engine section, based on the *ae_applid* and *ae_section*. If the *effdt* is specified, this is also used to get the section object. In other words, the Open method sets the **base section**.



When you open or get an AERSection object, (that is, the base section) any existing steps in the section will be deleted.



If the base section you specify doesn't exist, a *new* base section will be created. This allows you to dynamically create sections as needed.

If the AERSection is still open when you issue the Open method, the previously opened object will be discarded, and none of the changes saved. To prevent accidentally discarding your changes, you can use the IsOpen property to verify if a section is already open.

The AERSection will be open based on the Market and database type of your current system.



For more information, see How an AERSection is Accessed.

Parameters

<i>ae_applid</i>	Specify the application ID of the section you want to access. This parameter takes a string value.
<i>ae_section</i>	Specify the section name of the section you want to access. This parameter takes a string value.
<i>effdt</i>	Specify the effective date of the section you want to access (optional). This parameter takes a string value.

Returns

An AERSection object.

Example

```
Local AERSection &SECTION;
```

```
&SECTION = GetAESecton("RULES1", "DYN_SECT");  
  
/* do some processing */  
  
&SECTION.Close();  
  
&SECTION.Open("RULES2", "DYN_SECT");  
  
/* do additional processing */
```

Related Topics

GetAESecton, How an AESecton is Accessed

Save

Syntax

```
Save()
```

Description

The **Save** method saves the section to the database. The AESecton object remains open after you use this method. To close the object, you must use the Close method.

Parameters

None.

Returns

None.

Example

See AESecton Example.

Related Topics

Close, Open

SetSQL

Syntax

```
SetSQL(action_type_string, string)
```

Description

The **SetSQL** method replaces the SQL associated with the given action type in the current step in the base section with the SQL in *string*. The *current* step is the latest step that was added (see `AddStep`.)

The valid action types are:

- DO_WHEN
- DO_WHILE
- DO_SELECT
- SQL
- DO_UNTIL



All action types need to be passed in as strings with quotation marks.

If the action specified does not exist in the current step, an error will occur.

You can use a SQL object as *string*.

```
&SECTION.SetSQL("SQL", &SQL);
```



For more information about SQL object, see SQL Class.

Parameters

<i>action_type</i>	Specifies the action type of the current step that should be changed. This parameter takes a string value.
<i>string</i>	Specifies the SQL to be used to replace the SQL in the current step.

Example

See ASection Example.

Returns

None.

Related Topics

`AddStep`

SetTemplate

Syntax

```
SetTemplate(ae_applid, ae_section)
```

Description

The **SetTemplate** method sets the **template** to be used with an AERSection object, as identified by *ae_applid* and *ae_section*.

The rules for assigning a template section are similar to the rules for selecting a base section. The selection of market and database platform are done exactly the same. However, the effective date of the template is always set based on the effective date of '1900-01-01'.

You must set the template before you can use any of the other methods.



For more information, see How an AERSection is Accessed.

Parameters

ae_applid

Specify the name of the Application Engine program that contains the section to be used as the template.

ae_section

Specify the name of the Application Engine section in the program to be used as the section template.

Returns

None.

Example

See AERSection Example.

Related Topics

Open, GetAERSection

AERSection Class Property

IsOpen

If this property is True, the section object is already open. If you try to open a section object that is already open, the open section object will be closed, and all the changes that haven't been saved are discarded before the object is re-opened.

Example

```
If Not (&MYSECTION.IsOpen) Then

    &MYSECTION.Open (MYAPPLID, SECTION2);

End-If;
```

Array Class

An array is a collection of data storage locations, each of which holds the same type of data. Each storage location is called an element of the array.

When you create an array, you don't have to declare the size of the array at declaration time. Arrays grow and shrink dynamically as you add (or remove) data. The size of an array is limited by the available memory. You can't access past the end of an array, but you can assign outside the existing boundaries, thereby growing the array.

Creating Arrays

Arrays are declared by using the Array type name, optionally followed by "of" and the type of the element(s). If the element type is omitted, it defaults to ANY.

```
Local Array of Number &MYARRAY;

Local Array &ARRAYANY;
```

Arrays can be composed of any valid PeopleCode data type, such as string, record, number, date, and so on.

PeopleSoft recommends you declare every object you use in PeopleCode. This will give you some syntax checking when you save your PeopleCode. It's always better to find out that you misspelled the name of a method or property at design time, rather than at runtime!

Arrays can be declared as Local, Global, or Component, just like any other PeopleTools object.

Arrays can be created with one or more dimensions. An array with more than one dimension is called an array of arrays. The **dimension** of an array is the number of Array type names in the declaration. This is also called the **depth** of an array. The maximum depth of a PeopleCode array is 15 dimensions.

In the following example, **&MYARRAY** has three dimensions, and **&MYA2** has two dimensions.

```
Local Array of Array of Array of Number &MYARRAY;

Local Array of Array &MYA2;
```

An array must always have a consistent dimension. This means that in a one-dimensional array *none* of the elements can be an array, in a two-dimensional array *all* of the elements must be one-dimensional arrays, and so forth.

After you declare your array, use one of the built-in array functions to instantiate it and return an object reference to it. For example, the following creates an array containing one element of type &TEMP, whatever data type &TEMP may be.

```
&MYARRAY = CreateArray(&TEMP);
```

Or you can use the **CreateArrayRept** function to instantiate an array. The **Rept** stands for *repeat*. CreateArrayRept creates an array that contains the number of copies you specify of a particular value. The following code will create an array with 3 copies of the string &MYSTRING. This does *not* create a 3-dimensional array, but rather creates an array that's already populated with 3 elements of data (**Len** = 3), each of which contain the same string (&MYSTRING).

```
&MYARRAY = CreateArrayRept(&MYSTRING, 3);
```

An array *object* can be assigned to an array *variable*. Array objects can be passed from and returned to any kind of PeopleCode function:

```
ANewFunc (&myarray);

MyFunc (&myarray);

&MyArray = YourFunc("something");
```

For example, the **ReturnToServer** function returns an array of nodes to which a message can be published.



For more information, see ReturnToServer.

Elements in an array are specified by providing a bracketed subscript after the array object reference:

```
&MyArray[1] = 123;

&temp = &memory[1][2][3];

&temp = &memory[1, 2, 3]; /* Same as preceding line. */

MyFunc(&MyArray[7]);

MyFunc(10)[15] = "a string";
```

To access data in a two-dimensional array, you must specify both indexes. The following accesses the second item in the first subarray:

```
&VALUE = &DOUBLE[1][2];
```

You will receive an error if you use a zero or negative index in an array. Accessing an array element whose index is larger than the last array element is also an error, but storing to such an index will extend the array. Any intervening elements between the former last element and the

new last element will be assigned a value based on the element type of the array. This will be the same value as an unassigned variable of that type.

An array is an object, which means that assignments to an array are the same as for any other object. An array variable can be assigned the distinguished value NULL, which indicates the absence of any array value.



For more information, see Object Assignment.

Array variables are supported for all scopes. This means that you can have local, global and Component array variables.

Populating an Array

There are several ways to populate an array. The example code following each of these methods creates the exact same array, with the same elements:

- Use **CreateArray** when you initially create the array:

```
Local Array of Number &MyArray;

&MyArray = CreateArray(100, 200, 300);
```

- Assign values to the elements of the array:

```
Local Array of Any &MYARRAY;

&MYARRAY = CreateArray();

&MYARRAY[1] = 100;

&MYARRAY[2] = 200;

&MYARRAY[3] = 300;
```



Using CreateArray without any parameters creates an array of Any.

- Use the **Push** method to add items to the end of the array:

```
Local Array of Number &MYARRAY;

Local Number &MYNUM;
```

```

&MYARRAY = CreateArrayRept (&MYNUM, 0);

/* this creates an empty array of number */

&MYARRAY.Push(100);

&MYARRAY.Push(200);

&MYARRAY.Push(300);

```

- Use the **Unshift** method to add items to the beginning of the array:

```

Local Array of Number &MYARRAY;

Local Number &MYNUM;

&MYARRAY = CreateArrayRept (&MYNUM, 0);

/* this creates an empty array of number */

&MYARRAY.Unshift(300);

&MYARRAY.Unshift(200);

&MYARRAY.Unshift(100);

```

You can also use **CreateArrayRept** (repeat) when you initially create the array. The following example creates an array with three elements: all three elements have the same data, that is, 100:

```

Local Array of Number &MYARRAY;

&MYARRAY = CreateArrayRept(100, 3);

```

Removing Items from an Array

You can remove elements from either the start or the end of the array:

- Use the **POP** method to select and remove an element from the end of an array

```

Local Array of Number &MYARRAY;

&MYARRAY = CreateArray();

&MYARRAY[1] = 100;

&MYARRAY[2] = 200;

&MYARRAY[3] = 300;

```

```
&ANSWER = &MYARRAY.Pop();
```

&ANSWER will equal 300.

- Use the SHIFT method to select and remove an element from the beginning of an array

```
Local Array of Number &MYARRAY;
```

```
&MYARRAY = CreateArray();
```

```
&MYARRAY[1] = 100;
```

```
&MYARRAY[2] = 200;
```

```
&MYARRAY[3] = 300;
```

```
&ANSWER = &MYARRAY.Shift();
```

&ANSWER will equal 100.

Creating Empty Arrays

If you create an array using **CreateArray** of any data type other than ANY, the new array will *not* be empty: it will actually contain one item. If you need to create a completely empty array that contains 0 elements, use one of the following:

```
Local Array of Number &AN;
```

```
Local Array of String &AS;
```

```
Local Array of Record &AR;
```

```
Local Array of Array of Number &AAN;
```

```
Local Record &REC;
```

```
&AN = CreateArrayRept(0,0); /* creates an empty array of number */
```

```
&AS = CreateArrayRept("", 0); /* creates an empty array of string */
```

```
&AR = CreateArrayRept(&REC, 0); /*create empty array of records */
```

```
&AAN = CreateArrayRept(&AN, 0); /*creates empty array of array of number */
```

```
&BOTH = CreateArray(CreateArrayRept("", 0), CreateArrayRept("", 0)); /* creates  
an empty array of array of string */
```

Creating and Populating Multi-Dimensional Arrays

You can create arrays with more than one dimension. Each element in a multi-dimensional array is itself an array. For example, a two dimensional array is really an array of arrays. Each subarray has to be created and populated as an array.

Each subarray in a multi-dimensional array must be of the same type. For example, you can't create a two dimensional array that has one subarray of type string and a second subarray of type number.

The following example creates an array of array of string, then reads two files, one into each "column" of the array. The **CreateArrayRept** function call creates an empty array of string (that is, the **Len** property is 0) but with two dimensions (that is, with two subarrays, **Dimension** is 2). The first **Push** method adds elements into the first subarray, so at the end of that WHILE loop in the example, **&BOTH** has **Len** larger than 0. The other **Push** methods add elements to the second subarray.

```

Local array of array of string &BOTH;

Local File &MYFILE;

Local string &HOLDER;

/* Create empty &BOTH array */

&BOTH = CreateArrayRept(CreateArrayRept("", 0), 0);

/* Read first file into first column */

&MYFILE = GetFile("names.txt", "R");

While &MYFILE.ReadLine(&HOLDER);

    &BOTH.Push(&HOLDER);

End-While;

/* read second file into second column */

&MYFILE = GetFile("numbers.txt", "R");

&LINENO = 1;

While &MYFILE.ReadLine(&HOLDER);

    If &LINENO > &BOTH.Len Then

```

```

/* more number lines than names, use a null name */

&BOTH.Push(CreateArray("", &HOLDER));

Else

    &BOTH[&LINENO].Push(&HOLDER);

End-If;

&LINENO = &LINENO + 1;

End-While;

/* if more names than numbers, add null numbers */

for &LINENO = &LINENO to &BOTH.Len

    &BOTH[&LINENO].Push("");

End-For;

```

Local Name	Local Value
&BOTH	Array
Dimension	2
Len	3
[1]	
Dimension	1
Len	2
[1]	Pete
[2]	100
[2]	
Dimension	1
Len	2
[1]	Sue
[2]	200
[3]	
Dimension	1
Len	2
[1]	Laszlo
[2]	400
&MYFILE	File
&HOLDER	400
&LINENO	4

&BOTH array expanded in PeopleCode debugger at program end

The following code reads from a two-dimensional array and writes the data from the each subarray into a separate file.

```

Local File &MYFILE1, &MYFILE2;

Local string &STRING1, &STRING2;

Local array of array of string &BOTH;

```

.

```

/* code to load data into array would be here */

.

/* open files to be written to */

&MYFILE1 = GetFile("names.txt", "A");
&MYFILE2 = GetFile("numbers.txt", "A");

/* loop through array and write to files */

For &I = 1 To &BOTH.Len

    &J = 1;

    &STRING1 = &BOTH[&I][&J];

    &MYFILE1.writeline(&STRING1);

    &J = &J + 1;

    &STRING2 = &BOTH[&I][&J];

    &MYFILE2.writeline(&STRING2);

End-For;

&MYFILE1.Close();

&MYFILE2.Close();

```

The following example populates a multi-dimensional string array using SQL. This could be used for reading small tables.

```

Component array of array of string &ArrRunStatus;

&ArrRunStatus = CreateArrayRept(CreateArrayRept("", 0), 0);

&ArrRunStatusDescr = CreateArrayRept("", 0);

&SQL = CreateSQL("SELECT FIELDVALUE, XLATSHORTNAME FROM XLATTABLE WHERE
FIELDNAME = 'RUNSTATUS'");

```

```

&LineNo = 1;

While &SQL.Fetch(&FieldValue, &XlatShortName)

    &ArrRunStatus.Push(&FieldValue);

    &ArrRunStatus[&LineNo].Push(&XlatShortName);

    &LineNo = &LineNo + 1;

End-While;

```

To search for a particular element in this array, use the following:

```

&iIndex = &ArrRunStatus.Find(&RunStatusToGet);

&RunStatusDescr = &ArrRunStatus[&iIndex][2];

```

Flattening and Promotion

Several of the functions and methods that support arrays in PeopleCode use **flattening** and **promotion** to convert their operands to the correct dimension for the array.

Flattening converts an array into its elements. For example, the **CreateArray** built-in function constructs an array from its parameters. If it is constructing a one-dimensional array and is given an array as a parameter, then it will flatten that array into its elements and add each of them to the array that it is building, rather than adding a reference to the array (which would be a dimension error) or reporting an error.

Likewise, for functions that operate on multiple-dimension arrays, if they are given a non-array parameter, they will use **promotion** to convert it into an array of suitable dimension. For example, the **Push** method appends elements onto the end of an array. If it is operating with a two-dimensional Array of Array of Number, and is given a numeric argument, it will convert the argument into a one-dimensional Array of Number with the given number as its only element, and then append that to the two-dimensional array.

An array value can only be assigned to an array variable if the value and variable have both the same dimension and base type. This means you cannot assign an Array of Any to an Array of Number variable or vice-versa. You can, however, assign an Array of Number to an Any variable, as long as you do not break the rule that the base element of an array cannot be an array reference value.

Declaring Array Objects

Arrays are declared by using the Array type name, optionally followed by "of" and the type of the element(s). If the element type is omitted, it defaults to ANY.

```

Local Array of Number &MYARRAY;

Local Array &ARRAYANY;

```

Arrays can be composed of any valid PeopleCode data type, such as string, record, number, date, and so on.

Scope of an Array Object

An array object can only be instantiated from PeopleCode. This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Component Interface PeopleCode, record field PeopleCode, and so on.

Array Class Built-in Functions

CreateArray

CreateArrayRept

Split

Array Class Methods

Clone

Syntax

```
Clone()
```

Description

Clone returns a reference to a new array, which is a copy of the given array. It copies all levels of the array, meaning, if the array contains elements which are themselves arrays (actually references to arrays), then the copy will contain references to copies of the subarrays. Furthermore, if the array contains elements that are references to the same subarray, then the copy will contain references to *different* subarrays (which will of course have the same value).

Parameters

None. The array object that the clone method is executed against is the array to be cloned. Assigning the result of this method assigns a reference to the new array.

Result

An array object copied from the original.

Example

In the following example, &AAN2 contains the three elements like &AAN, but they are distinct arrays. The last line changes only &AAN2[1][1], **not** &AAN[1][1].


```
Local Array of Array of String &AAN, &AAN2;
```

```
&AAN = CreateArray(CreateArray("A", "B"), CreateArray("C", "D"), "E");
```

```
&AAN2 = &AAN.Clone();
```

```
&AAN2 [1] [1] = "Z";
```

After the following example, &AAN contains three elements: two references to the subarray that was &AAN[2] (with elements C and D), and a reference to a subarray with element E.

```
&AAN[1] = &AAN[2];
```

After the following example, &AAN2 contains three elements: references to two different subarrays both with elements C and D, and a subarray with element E.

```
&AAN2 = &AAN.Clone();
```

Related Topics

Subarray

Find

Syntax

Find(*value*)

Description

For a one-dimensional array, **Find** returns the lowest index of an element in the array that is equal to the given value. If the value is not found in the array, it returns zero.

For a two-dimensional array, **Find** returns the lowest index of a subarray which has its first element equal to the given value. If such a subarray is not found in the array, it returns zero.



This method only works with arrays that have one or two dimensions. You'll receive a runtime error if you try to use this method with an array that has more than two dimensions.

Parameters

value

The string or subarray to search for.

Returns

An index of an element or zero.

Example

Given an array &AS containing (A, B, C, D, E), the following code will set &IND to the index of D, that is, &IND will have the value 4:

```
Local array of string &AS;

&AS = CreateArrayRept("", 0);

&AS.Push("A");

&AS.Push("B");

&AS.Push("C");

&AS.Push("D");

&AS.Push("E");

&IND = &AS.Find("D");
```

Given an array of array of string &AABYNAME containing (("John", "July"), ("Jane", "June"), ("Norm", "November")), the following code will set &IND to the index of the subarray starting with "Jane", that is, &IND will have the value 2:

```
&NAME = "Jane";
&IND = &AABYNAME.Find(&NAME);
```

Related Topics

Replace, Substitute

Get

Syntax

```
Get (index)
```

Description

Use the **Get** method to return the *index* element of an array. This method is used with the Java PeopleCode functions, instead of using subscripts (which aren't available in Java.)

Using this method is the same as using a subscript to return an item of an array. In the following example, the two lines of code are identical:

```
&Value = &MyArray[8];

&value = &MyArray.Get(8);
```

Parameters

index The array element to be accessed.

Returns

An element in an array.

Related Topics

Set, Internet Script Classes

Join

Syntax

```
Join([separator [, arraystart, arrayend ]])
```

Description

The **Join** function *converts* the array that is executing the method into a string by converting each element into a string and *joining* these strings together, separated by *separator*.



Join does *not* join two arrays together.

Each array or subarray to be joined is preceded by the string given by *arraystart* and followed by the string given by *arrayend*. *Separator* defaults to a comma (","), *arraystart* defaults to a left parenthesis ("("), and *arrayend* defaults to a right parenthesis (")"). If the given array is multi-dimensional, then (logically) each subarray is first joined, then the resulting strings are joined together.

Parameters

separator Specifies what the elements in the resulting string should be separated with in the resulting string. *Separator* defaults to a comma (",").

arraystart Specifies what each array or subarray to be joined should be preceded with in the resulting string. *arraystart* defaults to a left parenthesis ("(").

arrayend Specifies what each array or subarray to be joined should be followed by in the resulting string. *arrayend* defaults to a right parenthesis (")").

Returns

A string containing the converted elements of the array.

Example

The following example:

```
Local array of array of number &AAN;

&AAN = CreateArray(CreateArray(1, 2), CreateArray(3, 4), 5);

&STR = &AAN.Join(", ");
```

produces in &STR the string:

```
((1, 2), (3, 4), 5)
```

Related Topics

Split

Next

Syntax

```
Next (&index)
```

Description

The **Next** method increments the given index variable. It returns true if and only if the resulting index variable refers to an existing element of the array. **Next** is typically used in the condition of a **while** clause to process a series of array elements up to the end of the array.

&index must be a variable of type integer, or of type Any initialized to an integer, as **Next** will try to update it.

If you want to start from the first element of the array, start **Next** with an index variable with the value zero. The first thing **Next** will do is to increment the value by one.

Parameters

<i>&index</i>	The array element where processing should start. <i>&index</i> must be a variable of type integer, or of type Any initialized to an integer, as Next will try to update it.
-------------------	--

Returns

True if the resulting index refers to an existing element of the array, False otherwise.

Example

Next can be used in a **While** loop to iterate through an array in the following manner:

```
&INDEX = 0;
```

```

While &A.Next (&INDEX)

    /* Process &A [&INDEX] */

End-While;

```

In the following code example, &BOTH is a two-dimensional array. This example writes the data from each subarray in &BOTH into a different file.

```

&I = 0;

While &BOTH.Next (&I)

    &J = 1;

    &STRING1 = &BOTH [&I] [&J];

    &MYFILE1.writeline (&STRING1);

    &J = &J + 1;

    &STRING2 = &BOTH [&I] [&J];

    &MYFILE2.writeline (&STRING2);

End-While;

```

Related Topics

Len

Pop

Syntax

```
Pop ()
```

Description

Pop removes the last element from the array and returns its value.

Parameters

None.

Returns

The value of the last element of the array. If the last element is a subarray, the subarray is returned.

Example

Pop can be used with the **Push** method to use an array as a stack. To put values on the end of the array, use **Push**. To take the values back off the end of the array, use **Pop**.

Suppose we have a two dimensional array &SUBPARTS which gives the subparts of each part of some assemblies. Each row (subarray) of &SUBPARTS starts with the name of the part, and then has the names of the subparts. Assuming there are no "loops" in this data, the following code will put all the subparts, subsubparts, and so on, of the part given by &PNAME into the array &ALLSUBPARTS, in "depth first order" (that is, subpart1, subparts of subpart1, ..., subpart2, subparts of subpart2, ...). We will stack the indexes into &SUBPARTS when we want to go down to the subsubparts of the current subpart.

```

Local array of array of string &SUBPARTS;

Local array of string &ALLSUBPARTS;

Local array of array of number &STACK;

Local array of number &CUR;

Local string &SUBNAME;

/* Set the ALLSUBPARTS array to an empty array of string. */

&ALLSUBPARTS = CreateArrayRept("dummy", 0);

/* Start with the part name. */

&STACK = CreateArray(CreateArray(&SUBPARTS.Find(&PNAME), 2));

While &STACK.Len > 0

    &CUR = &STACK.Pop();

    If &CUR[1] <> 0 And

        &CUR[2] <= &SUBPARTS[&CUR[1]].Len Then

        /* There is a subpart here. Add it. */

        &SUBNAME = &SUBPARTS[&CUR[1], &CUR[2]];

        &ALLSUBPARTS.Push(&SUBNAME);

        /* Tour its fellow subparts later. */

        &STACK.Push(CreateArray(&CUR[1], &CUR[2] + 1));

```

```

/* Now tour its subsubparts. */

&STACK.Push(CreateArray(&SUBPARTS.Find(&SUBNAME), 2));

End-If;

End-While;

```

Related Topics

Push, Replace, Shift, Unshift

Push

Syntax

```
Push(paramlist)
```

Where *paramlist* is an arbitrary-length list of values in the form:

```
value1 [, value2] ...
```

Description

Push adds the values in *paramlist* onto the **end** of the array executing the method. If a value is not the correct dimension, it will be flattened or promoted to the correct dimension first, then the resulting value(s) are added to the end of the array.



For more information, see [Flattening and Promotion](#)

Parameters

<i>paramlist</i>	An arbitrary-length list of values, separated by commas.
------------------	--

Returns

None.

Example

The following example loads an array with data from a database table.

```

Local array of record &MYARRAY;

Local SQL &SQL;

&I = 1;

```

```

&SQL = CreateSQL("Select (:1) from %Table(:1) where EMPLID like '8%'", &REC);

While &SQL.Fetch(&REC);

    &MYARRAY.Push(CreateRecord(RECORD.PERSONAL_DATA));

    &I = &I + 1;

    &REC.CopyFieldsTo(&MYARRAY[&I]);

End-While;

```

Related Topics

Pop, Replace, Shift, Unshift

Replace

Syntax

```
Replace(start, length, paramlist)
```

Where *paramlist* is an arbitrary-length list of values in the form:

```
value1 [, value2] ...
```

Description

Replace replaces the *length* elements starting at *start* with the given values, if any. If *length* is zero, the insertion takes place *before* the element indicated by *start*. Otherwise, the replacement starts with *start* and continues up to and including *length*, replacing the existing values with *paramlist*.

If a negative number is used for *start*, it indicates the starting position relative to the last element in the array, such that -1 indicates the position just *after* the end of the array. To insert at the end of the array (equivalent to the **Push** method), use a *start* of -1 and a *length* of 0.

If a negative number is used for *length*, it indicates a length measuring downward to lower indexes. Both flattening and promotion can be applied to change the dimension of the supplied parameters to match the elements of the given array.



For more information, see Flattening and Promotion.

Similar to how the built-in function **Replace** is used to update a string, the Replace method is a general way to update an array, and can cause the array to grow or shrink.

Parameters

<i>start</i>	Specifies where to start replacing the given elements in the array. If a negative number is used for <i>start</i> , it indicates the starting position relative to the last element in the array.
<i>length</i>	Specifies the number of elements in the array to be replaced.
<i>paramlist</i>	Specifies values to be used to replace existing values in the array.







Returns

None.

Example

For example, given the following array:








```
Local array of string &AS;  
  
&AS = CreateArray("AA", "BB", "CC");
```

Local Name	Local Value
 &AS	Array
 Dimension	1
 Len	3
 [1]	AA
 [2]	BB
 [3]	CC

&AS expanded in PeopleCode debugger

After executing the next code, the array &AN will contain four elements, ZZ, YY, BB, CC:

```
&AS.Replace(1, 1, "ZZ", "YY");
```

Local Name	Local Value
 &AS	Array
 Dimension	1
 Len	4
 [1]	ZZ
 [2]	YY
 [3]	BB
 [4]	CC

&AS expanded in PeopleCode debugger

After executing the next code, the array &AN will contain three elements, ZZ, MM, CC:

```
&AS.Replace(2, 2, "MM");
```

Local Name	Local Value
▢ &AS	Array
▢ Dimension	1
▢ Len	3
▢ [1]	ZZ
▢ [2]	MM
▢ [3]	CC

&AS expanded in PeopleCode debugger

After executing the next code, the array &AN will contain three elements, ZZ, OO, CC.

```
&AS.Replace( - 2, - 1, "OO");
```

Local Name	Local Value
▢ &AS	Array
▢ Dimension	1
▢ Len	3
▢ [1]	ZZ
▢ [2]	OO
▢ [3]	CC

&AS expanded in PeopleCode debugger

Related Topics

Substitute, Find

Reverse

Syntax

```
Reverse()
```

Description

Reverse reverses the order of the elements in the array.

If the array is composed of subarrays, **Reverse** will only reverse the elements in the super-array, it doesn't reverse all the elements in the subarrays. For example, the following:

```
&AN = CreateArray(CreateArray(1, 2), CreateArray(3, 4), CreateArray(5, 6)).reverse();
```

results in &AN containing:

```
((5,6), (3,4), (1,2))
```

Parameters

None.

Returns

None.

Example

Suppose you had the following array.

```
Local Array of Sting &AS;  
  
&AS = CreateArray("R", "O", "S", "E");
```

If you executed the Reverse method on this array, the elements would be ESOR.

Related Topics

Sort

Set**Syntax**

```
Set(index)
```

Description

Use the **Set** method to set the value of the *index* element of an array. This method is used with the Java PeopleCode functions, instead of using subscripts (which aren't available in Java.)

Using this method is the same as using a subscript to reference an item of an array. In the following example, the two lines of code are identical:

```
&MyArray[8] = &MyValue;  
  
&MyArray.Set(8) = &MyValue;
```

Parameters

<i>index</i>	The array element to be accessed.
--------------	-----------------------------------

Returns

None.

Related Topics

Get, Internet Script Classes

Shift

Syntax

```
Shift()
```

Description

Shift removes the first element from the array and returns it. Any following elements are "shifted" to an index of one less than they had.

Parameters

None.

Returns

Returns the value of the first element of the array. If the first element is a subarray, the subarray is returned.

Example

```
For &I = 1 to &ARRAY.Len;  
  
    &ITEM = &ARRAY.Shift;  
  
    /* do processing */  
  
End-For;
```

Related Topics

Pop, Push, Replace, Unshift

Sort

Syntax

```
Sort([order])
```

Description

Sort rearranges the elements of the array executing the method into an order.

If the array is one dimensional, the elements are arranged in either ascending or descending order.

The type of the first element is used to determine the kind of comparison to be made. Any attempt to sort an array whose elements are not all of the same type results in an error.

If *order* is omitted or "A", the order is ascending; if it is "D", the order is descending. The comparison between elements is the same one as if done using the PeopleCode comparison operators (<, >, =, etc.)



If you execute this method on a server, the string sorting order is determined by the character set and localization of the server.

If the array is two dimensional, the subarrays are arranged in order by the first element of each subarray. Sorting an array whose subarrays have different types of first elements will result in an error. The comparison is done by using the PeopleCode comparison operators (<, >, =, and so on.)



This method only works with arrays that have one or two dimensions. You'll receive a runtime error if you try to use this method with an array that has more than two dimensions.

Parameters

order

Specifies whether the array should be sorted in ascending or descending order. If no value is specified for *order* the order is ascending. Valid values for *order* are:

- **A** Ascending
- **D** Descending

Returns

None.

Example

The following example changes the order of the elements in array &A to be ("Frank", "Harry", "John").

```
&A = CreateArray("John", "Frank", "Harry");  
  
&A.Sort();
```

The following example changes the order of the elements in array &A to be (("Frank", 1957), ("Harry", 1928), ("John", 1952)).

```
&A = CreateArray(CreateArray("John", 1952), CreateArray("Frank", 1957),  
CreateArray("Harry", 1928));
```

Local Name	Local Value
&A	Array
Dimension	2
Len	3
[1]	
Dimension	1
Len	2
[1]	John
[2]	1952
[2]	
Dimension	1
Len	2
[1]	Frank
[2]	1957
[3]	
Dimension	1
Len	2
[1]	Harry
[2]	1928

Array &A expanded in PeopleCode debugger

```
&A.Sort ("A" );
```

Local Name	Local Value
&A	Array
Dimension	2
Len	3
[1]	
Dimension	1
Len	2
[1]	Frank
[2]	1957
[2]	
Dimension	1
Len	2
[1]	Harry
[2]	1928
[3]	
Dimension	1
Len	2
[1]	John
[2]	1952

Array &A expanded in PeopleCode debugger, showing code results

Related Topics

Reverse

Subarray

Syntax

```
Subarray(start, length)
```

Description

Subarray creates a new array from an existing one, taking the elements from *start* for a total of *length*. If *length* is omitted, all elements from *start* to the end of the array are used.

If the array is multi-dimensional, the subarrays of the created array are **references** to the same subarrays from the existing array. This means if you make changes to the original subarrays, the referenced subarrays will also be changed. To make distinct subarrays, use the Clone method.

Parameters

<i>start</i>	Specifies where in the array to begin the subarray.
<i>length</i>	Specifies the number of elements in the array to be part of the subarray.

Returns

An array object.

Example

To make a distinct array from a multi-dimensional array, use the following:

```
&A = &AAN.Subarray(1, 2).Clone();
```

Related Topics

Clone

Substitute

Syntax

```
Substitute(old_val, new_val)
```

Description

Substitute replaces every occurrence of a value found in an array with a new value. To replace an element that occurs in a specific location in an array, use Replace.

If the array is one dimensional, **Substitute** replaces every occurrence of the *old_val* in the array with *new_val*.

If the array is two dimensional, **Substitute** replaces every subarray whose first element is equal to *old_val*, with the subarray given by *new_val*.



This method only works with arrays that have one or two dimensions. You'll receive a runtime error if you try to use this method with an array that has more than two dimensions.

Parameters

<i>old_val</i>	Specifies the existing value in the array to be replaced.
----------------	---

new_val Specifies the value with which to replace occurrences of *old_val*.

Returns

None

Example

The following example changes the array &A to be ("John", "Jane", "Hamilton").

```
&A = CreateArray();
&A[1] = "John";
&A[2] = "Jane";
&A[3] = "Henry";

&A.Substitute("Henry", "Hamilton");
```

The following example changes the array &A to be (("John", 1952), ("Jane", 1957), ("Hamilton", 1971), ("Frank", 1961)).

```
&A = CreateArray(CreateArray("John", 1952), CreateArray("Jane", 1957),
CreateArray("Henry", 1928), CreateArray("Frank", 1961));

&A.Substitute("Henry", CreateArray("Hamilton", 1971));
```

Related Topics

Find, Replace

Unshift

Syntax

```
Unshift(paramlist)
```

Where *paramlist* is an arbitrary-length list of values in the form:

```
value1 [, value2] ...
```

Description

Unshift adds the given elements to the **start** of the array. Any following elements are bumped up to indexes that are larger by the number of values moved. Flattening and Promotion are used to change the dimension of the supplied parameters to be one less than that of the given array.



For more information, see Flattening and Promotion.

Parameters

paramlist Specifies values to be added to the start of the array.

Returns

None.

Example

The following code changes &A to be ("x", "Y", "a", "B", "c").

```
&A = CreateArray("a", "B", "c");  
  
&A.Unshift("x", "Y");
```

Related Topics

Pop, Push, Replace, Shift

Array Class Properties

Dimension

Dimension is the number of "Array" type names from the declaration of the array, also called subarrays. This property returns a number.

This property is read-only.

Example

The following example sets &DIM to 2.

```
Local Array of Array of Number &AAN;  
  
&DIM = &AAN.Dimension;
```

Len

Len is the current number of elements in the array. This property can be updated. Setting it to a negative value results in an error.

If this property is set to a smaller (nonnegative) number than its current value, the array is truncated to that length, discarding any elements whose indexes are larger than the given new length.

If this property is set to a number larger than its current value, the array is extended to the new length. Any new elements will be set to a default value based on the element type of the array.

This property is read-write.

Example

The following is a test of whether an array is empty:

```
If &ARR.Len = 0 then  
  
/* &ARR is empty. */  
  
End-If;
```

Business Interlink Class

The PeopleSoft Business Interlink framework provides a gateway for PeopleSoft applications to the services of any external system. This framework provides the ability for any PeopleSoft component (that is, a page, an Application Engine program, and so on) to integrate with any external system in near real-time and batch modes.

This framework allows a PeopleCode program to map the input and outputs of a Business Interlink definition to PeopleCode variables and record fields, then, using an Interlink object, call the Interlink plug-in, which in turn calls the external system, passes the information to the external system, and returns values to the PeopleCode program.

This documentation only describes the Interlink object portion of the PeopleSoft Business Interlink framework.



For more information on the PeopleSoft Business Interlink Framework, see the PeopleSoft Business Interlink Application Developer Guide.

Each Interlink object is based on a Business Interlink definition, which is created in Application Designer. An Interlink object can only be based on a single Business Interlink definition.



For more information on how to create a Business Interlink definition, see the PeopleSoft Business Interlink Application Developer Guide.

Using the Interlink Object

After you instantiate an Interlink object, use the Interlink object to:

- Populate the input buffer
- Send the data to the Interlink plug-in (by using the **Execute** method)
- Fetch the outputs from the buffer
- Check for status and error messages

After the Business Interlink object is instantiated, you can assign values from constants, PeopleSoft variables, or record fields to the inputs of that Business Interlink Object.

When you execute the Business Interlink Object, it loads the appropriate Business Interlink Plug-in and passes itself to that Business Interlink Plug-in. The Business Interlink Plug-in processes the input data, passing the input values of the Business Interlink Object to the external system and then fills the output values of the Business Interlink Object (if there are outputs).

Deciding Which Methods to Use

After you create your Business Interlink definition, you must use PeopleCode to instantiate an Interlink object and execute the Business Interlink plug-in. This PeopleCode can be long and complex. Rather than write it directly, you can drag and drop the Business Interlink definition from Application Designer's Project View into an open PeopleCode edit pane. Application Designer analyzes the definition and generates initial PeopleCode as a template, which you can modify to suit your purpose.



For more information, see the PeopleSoft Business Interlink Application Developer Guide.

Executing the Business Interlink Object

In most cases, you must use the **Execute** method to execute the Business Interlink object. However, for bulk input, you can use the **BulkExecute** method instead.

The **Execute** and **BulkExecute** methods return a value you can use for status and error checking.

Supporting Batch Input and Output

The methods discussed in this section, except for BulkExecute, add input values to the Business Interlink Object one set, or row, at a time. The call to the Business Interlink Plug-in occurs only once. All the input data is passed with the single **Execute**. All output is returned as batch as well. The methods then get the output values one set, or row, at a time.

If you're sending a large amount of data to the input buffers, instead of adding one input row at a time, you might write the data to a staging table, then use the **BulkExecute** method. This method automatically executes; that is, you don't have to use the Execute method. It also automatically fills the output record specified with the method with all the output values in every row in the output buffer if you've specified an output record.

Supporting Rowsets

If your data is mapped into rowsets, you may want to use the **InputRowset** method. This method will take a standard rowset object to populate the inputs for the Business Interlink Object. You can use the **FetchIntoRowset** method to repopulate the rowset with new data.

Using the Flat Table Methods

If your data is in a flat table structure, you can use the flat table methods. **AddInputRow** adds rows of input to the Business Interlink Object; **FetchNextRow** fetches rows of output from the Business Interlink Object.

Supporting Dynamic Output

A Business Interlink can have dynamic output, meaning that the outputs for a Business Interlink object are changeable in data type or number of outputs.

Business Interlinks supplies a set of methods to support dynamic output. These methods enable you to interrogate the output buffer programmatically to determine the number of fields (columns), their types, and their values.

The following are the methods used to support dynamic output:

- **GetFieldCount**
- **GetFieldType**
- **GetFieldValue**
- **MoveFirst**
- **MoveNext**

You can use the **MoveFirst** method to move to the first column, first row of the output buffer, and within a loop, use the **MoveNext** method to move to each row on the output buffer.

Within a **MoveFirst** (or **MoveNext**) loop, you can use the **GetFieldCount** method to get the number of columns in the output buffer, which is also the number of outputs for this Business Interlink object. Then you can extract the outputs from the buffer in a loop.

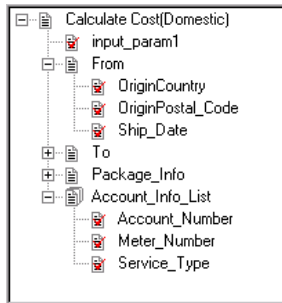
Within the **GetFieldCount** loop, you can use **GetFieldName** to get the name of the output, **GetFieldValue** to get the value of the output (which will be returned as a string), and **GetFieldType** to get the type of the output (if the output is not a string type, you will convert it to this type).

Using Hierarchical Data (BIDocs)

A Business Interlink can have hierarchical data. You can think of the structure as a tree, with a root doc, node docs, and values. The hierarchical data methods and objects are also referred to as *BIDocs*.

Input Structures

The following shows an example of an input structure:



Example Input Structure

The **root doc** is Calculate Cost(Domestic). A root doc can contain both values and node docs.

The **node docs** are From, To, Package_Info and Account_Info_List. Each node can contain both values and child nodes. Node docs can either be further described as the following:

- **Simple node docs** have only one set of values for a single instance of a root doc. From, To, Package_Info are all simple node docs.
- **List node docs** can contain more than one set of values for a single instance of a root doc. Account_Info_List is a list node doc.

The **values** in the From node are OriginCountry, OriginPostal_Code and Ship_Date. The value within the root node is input_param1. Notice that there are values both within the root doc and within node docs.

Business Interlinks support hierarchical input structures with the following methods:

- GetInputDocs
- AddDoc
- AddValue
- AddNextDoc

The **GetInputDocs** method returns a reference to the root doc of an input structure. From the above example, it returns a reference to Calculate Cost(Domestic).

Use the **AddDoc** method to access the node docs. From the above example, you would use AddDoc to access the From, To, Package_Info and Account_Info_List node docs. If any of these nodes contained nodes, you could use AddDoc to access those as well.

Use the **AddValue** method to set values. From the above example, you would use AddValue to set the value for input_param1, OriginCountry, OriginPostal_Code, and Ship_date. You must call AddDoc on a node before you can call AddValue for its values.

Use the **AddNextDoc** method to access the following:

- If a node doc is a list, that is, it can contain more than one set of values, use AddNextDoc to reference the next set of values.
- If you want to add another copy of the entire input structure, use AddNextDoc to return a

reference to the next root doc.

The following code example sets values for the node doc From, which is a simple node doc. It also sets the values for Account_Info_List, which is a list node doc.

```
&Calc_Input = &QE_FEDEX_COST.GetInputDocs("");

&FromDoc = &Calc_Input.AddDoc("From");

&ret = &FromDoc.AddValue("OriginCountry", "United States");

&ret = &FromDoc.AddValue("OriginPostal_Code", &ORIGIN);

&ret = &FromDoc.AddValue("Ship_Date", &SHIPDATE);

&Account_Doc = &Calc_Input.AddDoc("Account_Info_List");

&ret = &Account_Doc.AddValue("Account_Number", "CT-8001");

&ret = &Account_Doc.AddValue("Meter_Number", &METER);

&ret = &Account_Doc.AddValue("Service_Type", &MODE);

/* add next set of values in list */

&ret = &Calc_Input.AddNextDoc();

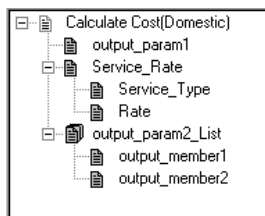
&ret = &Account_Doc.AddValue("Account_Number", "CT-8002");

&ret = &Account_Doc.AddValue("Meter_Number", &METER);

&ret = &Account_Doc.AddValue("Service_Type", &MODE);
```

Output Structures

The following shows an example of an output structure:



Example Output Structure

The **root doc** is Calculate Cost(Domestic).

The **node docs** are Service_Rate and output_param2_List. Each node can contain both values and child nodes. Node docs can further be described as follows:

- **Simple node docs** have only one set of values for a single instance of a root doc. Service_Rate is a simple node doc.
- **List node docs** can contain more than one set of values for a single instance of a root doc. output_param2_List is a list node doc.

The **values** in the Service_Rate node are Service_Type, Rate, and in the output_param2_List node, output_member1 and output_member2, and within the root node, output_param1.

Business Interlinks support hierarchical output structures with the following methods:

- GetOutputDocs
- GetDoc
- GetNextDoc
- GetPreviousDoc
- GetStatus
- GetValue
- GetCount
- MoveToDoc

The **GetOutputDocs** method returns a reference to the root doc of an output structure. From the above example, it returns a reference to Calculate Cost(Domestic).

Use the **GetDoc** method to access node docs. From the above example, you would use GetDoc to access the Service_Rate or output_param2_List node docs. If any of these nodes contain nodes, you use GetDoc to access those as well.

Use the **GetValue** method to retrieve the values. From the above example, you would use GetValue to retrieve the values for output_param1, and for the Service_Rate node, to get the values Service_Type, Rate, output_member1 and output_member2. You must call GetDoc on a node before you can call GetValue for its values.

Use the **GetNextDoc** (or **GetPreviousDoc**) method to access the following:

- A reference to the next (or previous, respectively) root doc.
- If a node doc is a list, that is, can contain more than one set of values, use GetNextDoc to reference the next node doc in the list (or GetPreviousDoc to access the previous node doc in the list.)

Use the **GetCount** method to return either the number of docs in a list node doc, or the number of root docs. In the example, you can count the number of Calculate Cost(Domestic) nodes or the number of output_param2_list nodes.

Use the **MoveToDoc** method to move to a particular doc at the root level or to a list node doc. In the example, you can move to a Calculate Cost(Domestic) node or to an output_param2_list_node.

The following code example gets values for the node docs Service Rate, which is a simple node doc. It also sets the values for output_param2_List, which is a list node doc.

```
&Calc_Input = &QE_FEDEX_COST.GetOutputDocs("");

&Service_Rate_Doc = &Calc_Input.GetDoc("Service_Rate");

&ret = &Service_Rate_Doc.GetValue("Service_Type", "Overnight");

&ret = &Service_Rate_Doc.GetValue("Rate", "50.00");

&Out_Param_Doc = &Calc_Input.GetDoc("output_param2_List");

&ret = &Out_Param_Doc.GetValue("output_member1", "value1");

&ret = &Out_Param_Doc.GetValue("output_member2", "value2");

/* get next set of values in list */

&Account_Doc = &Out_Param_Doc.AddNextDoc();

&ret = &Out_Param_Doc.GetValue("output_member1", "value3");

&ret = &Out_Param_Doc.GetValue("output_member2", "value4");
```

Using the Incoming Business Interlink Methods and Properties

The incoming Business Interlink methods allow you to parse an XML request and build an XML response.

The incoming Business Interlink uses a set of methods and functions.

The **GetContentBody** Request object method converts the request content into an XML string. You can then use this string with the **GetBiDoc** function to create a BiDocs structure from it that is the same shape and contains the same data as the XML document contained in the XML string.

Once you have the BiDocs structure containing the XML request, you can:

- Use the **GetNode** method to get one of the XML elements.
- Use the **NodeType** method to get the type of the node (element, processing instruction, comment).
- Use the **NodeName** property to get the name of the element.
- Use the **NodeValue** property to get the value of the element.

You can also use the standard BiDocs methods (such as **GetDoc**, **GetValue**) to retrieve information from this BiDocs object.

When an XML element contains attributes, the **AttributeCount** property gets the number of attributes, the **GetAttributeName** method gets the name of an attribute, and the **GetAttributeValue** method gets the value of an attribute.

To build an XML response, you can use the **GetBiDocs** function to create a blank BiDocs structure. To create the XML structure within that BiDocs, use **CreateElement** to create an XML tag, **AddComment** to add an XML comment, **AddAttribute** to add an attribute to an XML tag, and **AddProcessInstruction** to add a processing instruction (the first tag of the XML response).

Use the **GenXMLString** method to create an XML string from the BiDocs structure. Then you can use the Write Response method to write the response to the HTTP response string.



For more information, see the PeopleSoft Business Interlink Application Developer Guide.

State of an Interlink Object

PeopleSoft Business Interlink API that are run on the application server should be stateless, that is, if you want to save information from one call of the Interlink object to the next, you will have to do it yourself by writing the relevant information to the database. If you use the **Execute** method more than once within a single PeopleCode event (that is, if you have the **Execute** method in some sort of loop) the state will be preserved. Once you leave the event, any state associated with the Interlink object is lost.

You should only create one Business Interlink object, that is, you should only use the **GetInterlink** function once. After that, you can load it with data, pass the data to the Interlink plug-in (via **Execute**) and fetch output data as many times as you need.

Using Business Interlink with Application Engine

In a regular PeopleCode program, you can only declare a Business Interlink object as local. However, in an Application Engine program, you can declare a Business Interlink object as global. Instantiating a Business Interlink object once as a global saves on the significant overhead of reinstantiating a local object for every iteration of PeopleCode called in a loop.

- Global Business Interlink objects can only be used in Application Engine PeopleCode programs because PeopleCode that runs on an application server must be stateless.
- When a restartable Application Engine program abends, global Business Interlink objects that were instantiated before the last checkpoint are automatically reinstantiated at restart. So the object will be available, even though no call has been made to **GetInterlink** in the restarted process. However, the associated Business Interlink data buffers are *not* recovered, so the Application Engine program must be written such that these buffers are empty whenever a checkpoint is taken.
- Business Interlink objects should *not* be declared as global unless they are used in several PeopleCode actions, or in a PeopleCode action that is called in a loop. Only in these instances is the overhead of checkpointing them worthwhile.

Declaring a Business Interlink Object

You should declare a Business Interlink object using the data type Interlink. For example,

```
Local Interlink &MYBI;
```

Declare hierarchical data as type BIDocs. For example:

```
Local BIDocs &OutlistDoc;
```



BIDocs and Interlink objects used in PeopleCode programs run on the application server can only be declared as type Local. You can declare Interlinks as Global only in an Application Engine program.

Scope of a Business Interlink Object

A Business Interlink object can be instantiated from PeopleCode.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Application Engine PeopleCode, record field PeopleCode, and so on.

Business Interlink Class Built-in Function

GetBiDoc

GetInterlink

Business Interlink Class Methods

AddDoc

Syntax

```
AddDoc (docname)
```

Description

The **AddDoc** method adds a document to an input structure. The added document is an input parameter for a Business Interlink object that is not of simple type (such as integer or string). You must add the document to the input structure before you can add values to its members with **AddValue**.

Parameters

<i>docname</i>	The name of the document that AddDoc adds to the structure.
----------------	--

Returns

The document added to the BIDocs input document.

Example

In the following example, the input structure for Calculate Cost, or the root level document, is created with the **GetInputDocs** method. (If you wanted to create, or add, more input structures, use **AddNextDoc**.) The Calculate Cost input parameter From is a document, so the **AddDoc** method adds it to the input document.

```
Local Interlink &QE_FEDEX_COST;

Local BIDocs &CalcCostIn;

Local BIDocs &FromDoc, &ToDoc, &PackageDoc, &AccountDoc;

Local number &ret, &retinput;

&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);

&CalcCostIn = &QE_FEDEX_COST.GetInputDocs("");

&ret = &CalcCostIn.AddValue("input_param1","value");
```

```

&FromDoc = &CalcCostIn.AddDoc("From");

&ret = &FromDoc.AddValue("OriginCountry", "United States");

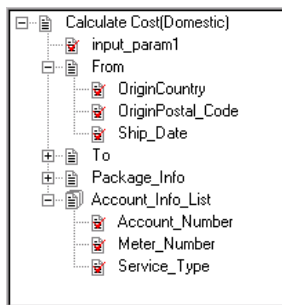
&ret = &FromDoc.AddValue("OriginPostal_Code", &ORIGIN);

&ret = &FromDoc.AddValue("Ship_Date", &SHIPDATE);

/* Call AddDoc and AddValue for To, Package_Info, and Account_Info_List (code
not shown) */

```

The following shows the input structure for this example. It contains five input parameters: `input_param1`, `From`, `To`, `Package_Info`, and `Account_Info_List`. This is a version of the Federal Express plug-in that was modified for this example (`input_param1` was added, `Account_Info` was modified to be a list).



Example Input Structure

Related Topics

GetInputDocs, AddValue, AddNextDoc

AddInputRow

Syntax

```
AddInputRow(inputname, value)
```

Description

where *inputname* and *value* are in matched pairs, in the form:

```
inputname1, value1 [, inputname2, value2] . . .
```

Description

The **AddInputRow** method adds a row of input data (*value*) from PeopleCode variables or record fields to the specified input names (*inputname*) for the Business Interlink object executing the method. These must be entered in matched pairs, that is, every input name must be followed by its matching value.



The input **name**, not the input path, of the interface definition is used for this method.

There must be an *inputname* for every input parameter defined in the interface definition used to instantiate the Business Interlink object.

If you specify a record field that is not part of the record the PeopleCode program is associated with, you must use *recname.fieldname* for that *value*.

You can specify default values for every input name in the interface definition (created in Application Designer.) These values will be used if you create a PeopleCode "template" by dragging the interface definition from the Project window in Application Designer to an open PeopleCode editor window.



For more information on generating a PeopleCode template, see Generating the PeopleCode Template.

Parameters

<i>inputname</i>	Specify the input name. There must be one <i>inputname</i> for every input name defined in the interface definition used to instantiate the Business Interlink object.
<i>value</i>	Specify the value for the input name. This can be a constant, a variable, or a record field. Each <i>value</i> must be paired with an <i>inputname</i> .

Returns

A Boolean value: True if the input values were successfully added. Otherwise, it returns False.

Example

In the following example, the Business Interlink object name is SRA_ALL_1, and the input name, such as ship_site_name, are being bound to record fields, such as QE_RP_SITENAME or VENDOR_INFO.QE_RP_PROMISEDDATE, to variables like &PARTNAME, or to literals, like 10 for quantity.

```
&SRA_ALL_1.AddInputRow("ship_site_name", QE_RP_SITENAME,
    "promise_date", VENDOR_INFO.QE_RP_PROMISEDDATE,
    "request_date", QE_RP_ORDERREQDATE,
    "subline_site_name", QE_RP_SITENAME,
    "quantity", 10,
    "part_name", &PARTNAME,
```

```
"site_name", INV_LOCATIONS.QE_RP_SITENAME);
```

Related Topics

Execute, GetInterlink

AddNextDoc

Syntax

```
AddNextDoc ()
```

Description

The **AddNextDoc** method adds a document to one of the following levels:

- The root level of the input structure for a Business Interlink object. You create a reference to this level with the **GetInputDocs** method.
- When a document within the input structure is a list, **AddNextDoc** adds another document to the list. If you use **AddNextDoc** on a document that is not a list, **AddNextDoc** fails and returns an error.

Parameters

None.

Returns

<i>Number</i>	<i>Enum value and Meaning</i>
0	eNoError. The method succeeded.
1	eNO_DOCUMENT. The document referenced by this method does not exist.
2	eDOCLIST_OUT_RANGE. You tried to access a document in a list, but the document list is out of range. For example, if a document list contains five documents, and then you call GetDoc/GetOutputDocs once, you can call GetNextDoc four times; the fifth time will result in this error.
3	eDOCUMENT_UNINITIALIZED. Internal error.
4	eNOT_DOCUMENTTYPE. You tried to perform an operation upon a parameter that is not a document type.
5	eNOT_LISTTYPE. You tried to perform a GetNextDoc or AddNextDoc upon a

	document that is not a list.
7	eNOT_BASICTYPE. You tried to perform a GetValue or AddValue upon a parameter that is not of basic type: integer, float, string, time, date, datetime.
8	eNO_DATA. You tried to retrieve data from a document that contained no data.

Example

In this example, the input structure for Calculate Cost, or the root level document, is accessed with the **GetInputDocs** method. The **AddNextDoc** method adds another BIDocs input document. The Calculate Cost input parameter Account_Info_List is a document, so the **AddDoc** method adds it to the BIDocs input document. Account_Info_List is also a document list, so **AddNextDoc** method adds another Account_Info_List document.

```

Local Interlink &QE_FEDEX_COST;

Local BIDocs &CalcCostIn;

Local BIDocs &FromDoc, &ToDoc, &PackageDoc, &AccountDoc;

Local number &ret, &retinput;

&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);

&CalcCostIn = &QE_FEDEX_COST.GetInputDocs("");

For &n = 1 to &number_of_input_sets
/* Get values for inputs, such as &ORIGIN (code not shown) */

    &ret = &CalcCostIn.AddValue("input_param1", "value");

    &FromDoc = &CalcCostIn.AddDoc("From");

    &ret = &FromDoc.AddValue("OriginCountry", "United States");

    &ret = &FromDoc.AddValue("OriginPostal_Code", &ORIGIN);

    &ret = &FromDoc.AddValue("Ship_Date", &SHIPDATE);

/* Call AddDoc and AddValue for To and Package_Info

```

```

        (code not shown) */

/* Call AddDoc and AddNextDoc for the AccountDoc document list */
&AccountDoc = &CalcCostIn.AddDoc("Account_Info_List");

For &m = 1 to &number_of_AccountDocs

    &ret = &AccountDoc.AddValue("Account_Number", &ACCOUNT);

    &ret = &AccountDoc.AddValue("Meter_Number", &METER);

    &ret = &AccountDoc.AddValue("Service_Type", &SVCTYPE);

    &retinput = &AccountDoc.AddNextDoc();

End-For;

&retinput = &CalcCostIn.AddNextDoc();

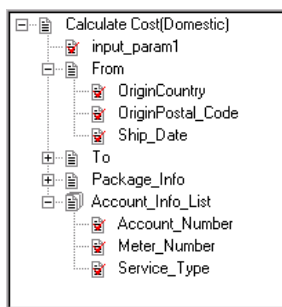
End-For;

```

The following shows the input structure for this example. It contains five input parameters: input_param1, From, To, Package_Info, and Account_Info_List.

From, To, and Package_Info are not lists, so if you try to use **AddNextDoc** with these parameters, such as the following line of code, **AddNextDoc** will fail and return an error message.

```
&retinput = &To.AddNextDoc();
```



Example Input Structure

Related Topics

GetInputDocs, AddDoc, AddValue

AddValue

Syntax

AddValue(*paramname*, *value*)

Description

The **AddValue** method adds a value to a member of a document within the input structure for a Business Interlink object.

Parameters

<i>paramname</i>	The name of the member of the document that is having a value added to it.
<i>value</i>	The value that is added. This can be a constant, a variable, or a record field. The data type can be string, integer, double, float, time, date, or datetime.

Returns

Number	Enum value and Meaning
0	eNoError. The method succeeded.
1	eNO_DOCUMENT. The document referenced by this method does not exist.
2	eDOCLIST_OUT_RANGE. You tried to access a document in a list, but the document list is out of range. For example, if a document list contains five documents, and then you call GetDoc/GetOutputDocs once, you can call GetNextDoc four times; the fifth time will result in this error.
3	eDOCUMENT_UNINITIALIZED. Internal error.
4	eNOT_DOCUMENTTYPE. You tried to perform an operation upon a parameter that is not a document type.
5	eNOT_LISTTYPE. You tried to perform a GetNextDoc or AddNextDoc upon a document that is not a list.
7	eNOT_BASICTYPE. You tried to perform a GetValue or AddValue upon a parameter that is not of basic type: integer, float, string, time, date, datetime.
8	eNO_DATA. You tried to retrieve data from

	a document that contained no data.
--	------------------------------------

Example

In the following example, the Business Interlink object name is QE_FEDEX_COST.

In the following example, the input structure for Calculate Cost, or the root level document, is accessed with the **GetInputDocs** method. (If you wanted to create, or add, more input structures, use **AddNextDoc**.) The Calculate Cost input parameter From is a document, so the **AddDoc** method adds it to the input structure. Then the **AddValue** method adds values to each of the From document members.

```
Local Interlink &QE_FEDEX_COST;

Local BIDocs &CalcCostIn;

Local BIDocs &FromDoc, &ToDoc, &PackageDoc, &AccountDoc;

Local number &ret;

&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);

/* Get some values for input, such as &ORIGIN (code not shown) */

&CalcCostIn = &QE_FEDEX_COST.GetInputDocs("");

&ret = &CalcCostIn.AddValue("input_param1","value");

&FromDoc = &CalcCostIn.AddDoc("From");

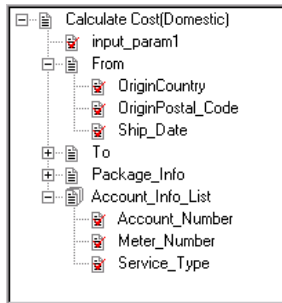
&ret = &FromDoc.AddValue("OriginCountry", "United States");

&ret = &FromDoc.AddValue("OriginPostal_Code", &ORIGIN);

&ret = &FromDoc.AddValue("Ship_Date", &SHIPDATE);

/* Call AddDoc, AddValue for Package, Account_Info_List, and also call
AddNextDoc for Account_Info_List (code not shown) */
```

The following shows the input structure for this example. It contains five input parameters: input_param1, From, To, Package_Info, and Account_Info_List.



Example Input Structure

Related Topics

GetInputDocs, AddDoc, AddNextDoc

BulkExecute

Syntax

```
BulkExecute(RECORD.inputrecname [, RECORD.outputrecname] [, {user_process_inst | user_operid}])
```

Description

The **BulkExecute** method uses the data in the specified record to populate the input buffer, copying **like-named** fields. Then the method executes, and, optionally, fills the record specified by *outputrecname* with data from the Business Interlink output buffer. **BulkExecute** will result in significantly faster performance for transactions that process large amounts of data. Instead of adding one input row at a time, then fetching the values one at a time, you might write the data to a staging table, use the **BulkExecute** method and then read the data from the output table. This would be especially effective in Application Engine programs that process sets of data rather than individual rows.

This method assumes that the names of the fields in the record match the names of the inputs (or outputs) defined in the Business Interlink Definition. If there is no field in the *inputrecname* for a Business Interlink input parameter, the parameter's default value is used. If no default is specified, an empty string is passed. It's up to the programmer to ensure that this default value is legitimate.

Fields in the output buffer are matched against the fields in the specified in the output record. You do not have to specify an output record. If you don't specify an output record, the output buffers will *not* be populated.

If you specify an output record, and if you have fields in the output record that are *not* specified as output parameters, they aren't populated with the BulkExecute method. In addition, they shouldn't be set as **NOT NULL**, otherwise inserts fail.



Before you use this method, you should flush the record used for output and remove any residual data that might exist in it.

If you specify an output record, and you want to use the output record for error checking, you must add the following fields to your output record:

- RETURN_STATUS
- RETURN_STATUS_MSG



The field names in the output record must match these names exactly.

Then you must mark one or more input parameters as "key fields" in the Business Interlink definition (using the BulkExecute ID checkbox.) The value of these key fields will be copied to the output record, and you can use these fields to match error messages with input (or output) rows.



For more information about the BulkExec ID checkbox, refer to PeopleSoft Business Interlink Application Developer Guide.

If you want to order your input (and output) rows, you must add the following column to both the input and output table:

- BI_SEQ_NUM



The field name in your input and output records must match this name exactly.

The input rows are read in the order of numbers in BI_SEQ_NUM, and the output rows are generated using the same order number.

This method automatically executes, so you don't need to use the **Execute** method with this method.

Whether this method halts on execution or not depends on the setting of the StopAtError configuration parameter. The default value is True, that is, stop if the error number returned is something other than a 1 or 2. This configuration parameter must be set *before* using the **BulkExecute** method.

Parameters

RECORD.*inputrecname*

Specify a record (SQL table) that contains the data you want to use to populate the input buffer of the Business Interlink.

RECORD.*outputrecname*

Specify a record (SQL table) that will hold the data that populates the output buffer of the Business Interlink. This is optional; when used, it is often used to error check the input.

user_process_inst |
user_operid

This is an optional parameter that allows **either** different Application Engine programs **or** different clients to populate the same **RECORD.outputrecname** at the same time.

For *user_process_inst*, the parameter takes an integer. **RECORD.outputrecname** must have a PROCESS_INSTANCE field. The PROCESS_INSTANCE field is used to identify the Application Engine program that is using this Business Interlink. You can use the %PROCESS_INSTANCE variable to populate *user_process_inst*.

For *user_operid*, the parameter takes a string. **RECORD.outputrecname** must have an OPERID field. The OPERID field is used to identify the client who is using this Business Interlink. You can use the %OPERID variable to populate *user_operid*.

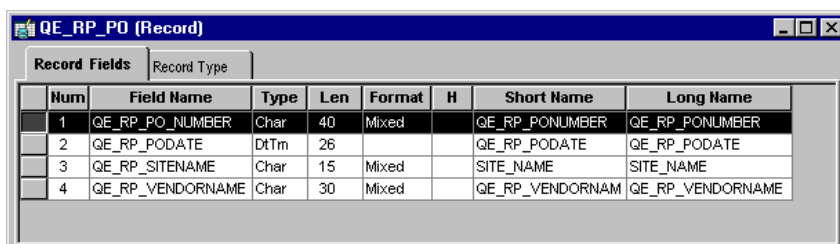
Returns

Number	Meaning
1	The Business Interlink Object executes successfully
2	The Business Interlink Object failed to execute
3	Transaction failed
4	Query failed
5	Missing criteria
6	Input mismatch
7	Output mismatch
8	No response from server
9	Missing parameter
10	Invalid username
11	Invalid password
12	Invalid server name
13	Connection error
14	Connection refused
15	Timeout reached
16	Unequal lists
17	No data for output

18	Output parameters empty
19	Driver not found
20	Internet connect error
21	XML parser error
22	XML deserialize

Example

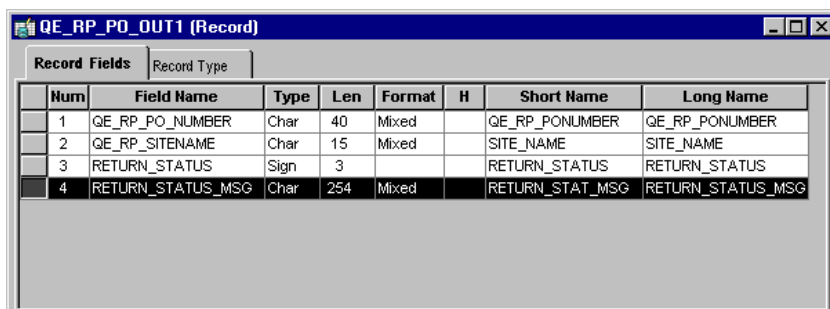
Here are two PeopleSoft records that could be used as input and output records for BulkExecute. The key fields in the input record, which need to have the BulkExecute ID checkbox checked in the Application Designer, are QE_RP_PO_NUMBER and QE_RP_SITENAME. These key fields are used both for input and output.



The screenshot shows a window titled "QE_RP_PO (Record)". It contains a table with the following data:

Num	Field Name	Type	Len	Format	H	Short Name	Long Name
1	QE_RP_PO_NUMBER	Char	40	Mixed		QE_RP_PONUMBER	QE_RP_PONUMBER
2	QE_RP_PODATE	DtTm	26			QE_RP_PODATE	QE_RP_PODATE
3	QE_RP_SITENAME	Char	15	Mixed		SITE_NAME	SITE_NAME
4	QE_RP_VENDORNAME	Char	30	Mixed		QE_RP_VENDORNAM	QE_RP_VENDORNAME

Example Input Record for BulkExecute



The screenshot shows a window titled "QE_RP_PO_OUT1 (Record)". It contains a table with the following data:

Num	Field Name	Type	Len	Format	H	Short Name	Long Name
1	QE_RP_PO_NUMBER	Char	40	Mixed		QE_RP_PONUMBER	QE_RP_PONUMBER
2	QE_RP_SITENAME	Char	15	Mixed		SITE_NAME	SITE_NAME
3	RETURN_STATUS	Sign	3			RETURN_STATUS	RETURN_STATUS
4	RETURN_STATUS_MSG	Char	254	Mixed		RETURN_STAT_MSG	RETURN_STATUS_MSG

Example Output Record for BulkExecute

Here are the corresponding inputs and outputs for a Business Interlink that has had its inputs and outputs renamed to match the field names in the records. The inputs and outputs have been renamed where necessary to match the record field names.

	Input Name	Input Path	Default	Data Type	Attribute	BulkExec Id	Path Ir
1	class_name	class_name	Purchase_Or	string	required	<input type="checkbox"/>	string
2	QE_RP_PODATE	order_date		string	required	<input type="checkbox"/>	string
3	QE_RP_VENDORNAME	vendor_name		string	required	<input type="checkbox"/>	string
4	QE_RP_PO_NUMBER	po_number		string	required	<input checked="" type="checkbox"/>	string
5	QE_RP_SITENAME	site_name		string	required	<input checked="" type="checkbox"/>	string

Example Business Interlink Inputs for BulkExecute

	Output Name	Output Path	Data Type	Attribute
1	return_status	return_status	int	required
2	return_status_msg	return_status_msg	string	required
3	QE_RP_VENDORNAME	purchase_order.vendor.display_name	string	
4	QE_RO_SITENAME	purchase_order.site.display_name	string	
5	QE_RP_PONUMBER	purchase_order.order_number	string	

Example Business Interlink Outputs for BulkExecute

The PeopleCode program using these records could contain the following code to run **BulkExecute**.

```

Local Interlink &CREATE_PO;

Local number &EXECSRSLT;

&CREATE_PO = GetInterlink(INTERLINK.QE_RP_CREATEPO1);

/* The next three lines set configuration parameters, which are not shown in the
previous examples. */

&CREATE_PO.SERVER_NAME = "bc1";

```

```

&CREATE_PO.RSERVER_HOST = "4.3.4.3";

&CREATE_PO.RSERVER_PORT = "2200";

&EXECRSLT = &CREATE_PO.BulkExecute(RECORD.QE_RP_PO, RECORD.QE_RP_PO_OUT1);

```

Related Topics

Execute

Clear

Syntax

```
Clear()
```

Description

The **Clear** method clears the input and output buffers. If you're using Business Interlinks in a batch program, every time after you use the **Execute** method, you'll want to use the **Clear** method to flush out the input and output buffers.

Parameters

None.

Returns

None.

Example

```

For &n = 1 to x

    &myinterlink.AddInputRow("input1", value1, "input2", value2);

    &myinterlink.Execute();

    &myinterlink.FetchNextRow("output1", value1, "output2", value2);

    /* do processing on data */

    &myinterlink.Clear();

    &n = &n + 1;

End-for;

```

Related Topics

Execute, BulkExecute

Execute

Syntax

Execute ()

Description

When an Interlink object executes the **Execute** method, the transaction associated with the Interlink object is executed.

If there is only one row, after appropriate substitution is made, the transaction is executed only once. Otherwise, the data is "batched up" and sent once. You only have to call **Execute** once to execute all the rows of the input buffer.

Generally, you would only use the **Execute** method after using the **AddInputRow** method. If you create a PeopleCode "template" by dragging the Business Interlink definition from the Project window in Application Designer to an open PeopleCode editor window, the usual order of the execution of methods is shown in the template.



For more information, see Generating the PeopleCode Template.

If you're using Business Interlinks in a batch program, every time after you use the **Execute** method, you want to use the Clear method to flush out the input and output buffers.

Whether this method halts on execution or not depends on the setting of the StopAtError configuration parameter. The default value is True, that is, stop if the error number returned is something other than a 1 or 2. This configuration parameter must be set *before* using the **Execute** method.

Parameters

None.

Returns

Number	Meaning
1	The Interlink object executes successfully
2	The Interlink object failed to execute
3	Transaction failed
4	Query failed
5	Missing criteria
6	Input mismatch
7	Output mismatch
8	No response from server

9	Missing parameter
10	Invalid username
11	Invalid password
12	Invalid server name
13	Connection error
14	Connection refused
15	Timeout reached
16	Unequal lists
17	No data for output
18	Output parameters empty
19	Driver not found
20	Internet connect error
21	XML parser error
22	XML deserialize

Example

```

Local number &EXECRESLT;

. . .

&EXECRESLT = &SRA_ALL_1.Execute();

If (&EXECRESLT <> 1) Then

    /* The instance failed to execute */

    /* Do Error Processing */

End-If;

```

Related Topics

BulkExecute, GetInterlink

FetchIntoRecord

Syntax

```
FetchIntoRecord(RECORD.recname, [, {user_process_inst | user_operid}}])
```

Description

The **FetchIntoRecord** method copies the data from the output buffer into the specified record (SQL table) copying **like-named** fields. This method assumes that the names of the fields in the record match the names of the outputs defined in the Business Interlink definition.

You can use the **FetchIntoRecord** method to perform a transaction on a large amount of data that you want to receive from your system. Instead of executing your Business Interlink and then fetching one output row at a time, you might execute your Business Interlink, then use the **FetchIntoRecord** method to write the data returned from the Business Interlink to a staging table.



Before you use this method, you should flush the record used for output and remove any residual data that might exist in it.

If your data is hierarchical, that is, in a rowset, and you want to preserve the hierarchy, use **FetchIntoRowset** instead of this method.

If you want to order your output rows, you must add the following column to the record:

- BI_SEQ_NUM

This column will be populated with a sequence number that corresponds to the order in which each row of the record was written to by the method.

Parameters

RECORD.*recname*

Specify a record (SQL table) that you want to populate with data from the output buffers.

user_process_inst |
user_operid

This is an optional parameter that allows *either* different Application Engine programs *or* different clients to populate the same **RECORD**.*outputrecname* at the same time.

For *user_process_inst*, the parameter takes an integer.

RECORD.*outputrecname* must have a PROCESS_INSTANCE field. The PROCESS_INSTANCE field is used to identify the Application Engine program that is using this Business Interlink. You can use the %PROCESS_INSTANCE variable to populate *user_process_inst*.

For *user_operid*, the parameter takes a string.

RECORD.*outputrecname* must have an OPERID field. The OPERID field is used to identify the client who is using this Business Interlink. You can use the %OPERID variable to populate *user_operid*.

Returns

Number of rows fetched if one or more non-empty rows are returned.

If an empty row is returned, that is, every if field is empty except the return_status and return_status_msg fields, this method returns one of the following error messages:

0	No error
1	NoInterfaceObject
2	ParamCountError
3	IncorrectParameterType
4	NoColumnForOutputParm
5	NoColumnForInputParm
6	BulkInsertStartFailed
7	BulkInsertStopFailed
8	CouldNotCreateSelectCursor
9	CouldNotCreateInsertCursor
10	CouldNotDestroySelectCursor
11	CouldNotDestroyInsertCursor
12	InputRecordDoesNotExist
13	OutputRecordDoesNotExist
14	ContainEmptyRow
15	SQLExecError
201	FieldTooLarge
202	IgnoreSignNumber
203	ConvertFloatToInt

Example

```

&RSLT = &QE_SM_CONCAT.FetchIntoRecord(RECORD.PERSONAL__VENDR_DATA) ;

If &RSLT = 0 Then

    /* no rows fetched */

Else

    /* do processing */

End-If;

```

Related Topics

BulkExecute

FetchIntoRowset

Syntax

```
FetchIntoRowset (&Rowset)
```

Description

The **FetchIntoRowset** method copies the data from the output buffer into the specified rowset, copying **like-named** fields. This method assumes that the names of the fields in the rowset match the names of the outputs defined in the Business Interlink definition, *and* that the structure is the same.



Before you use this method, you should flush the rowset used for output and remove any residual data that might exist in it.

Use this method only if you have a hierarchical data structure and you want to preserve the hierarchy. Otherwise, use **BulkExecute** or **FetchIntoRecord**.

Parameters

&Rowset

Specify rowset object that you want to populate with data from the output buffers. This must be an existing, instantiated rowset.

Returns

Number of rows fetched if one or more non-empty rows are returned.

If an empty row is returned, that is, every if field is empty except the return_status and return_status_msg fields, this method returns one of the following error messages:

Value	Description
0	No error
1	NoInterfaceObject
2	ParamCountError
3	IncorrectParameterType
4	NoColumnForOutputParm
5	NoColumnForInputParm
6	BulkInsertStartFailed
7	BulkInsertStopFailed

Value	Description
8	CouldNotCreateSelectCursor
9	CouldNotCreateInsertCursor
10	CouldNotDestroySelectCursor
11	CouldNotDestroyInsertCursor
12	InputRecordDoesNotExist
13	OutputRecordDoesNotExist
14	ContainEmptyRow
15	SQLExecError
201	FieldTooLarge
202	IgnoreSignNumber
203	ConvertFloatToInt

Example

The example uses the rowset on level 1 from the EMPLOYEE_CHECKLIST page. A PeopleCode program running in a field on Level 0 in that page can access the level 1 (child rowset).

	Lvl	Label	Type	Field	Record	Display Control	Related Field	Co
1	0	Frame	Frame					
2	0	Frame	Frame					
3	0	Frame	Frame					
4	0	Employee Name	Edit Box	NAME	PERSONAL_DAT			
5	0	ID	Edit Box	EMPLID	PERSONAL_DAT			
6	1	Checklist Item Tbl	Scroll Bar					
7	1	Checklist Sequen	Edit Box	CHECKLIST_SEQ	CHECKLIST_IT			
8	1	Scroll Bar 1	Scroll Bar					
9	1	Checklist Date	Edit Box	CHECKLIST_DT	EMPL_CHECKLIS			
10	1	derived_hr_effdt	Edit Box	EFFDT	DERIVED_HR			
11	1	Checklist	Edit Box	CHECKLIST_CD	EMPL_CHECKLIS			
12	1	Checklist Descripti	Edit Box	DESCR	CHECKLIST_TBL			11
13	1	Responsible ID	Edit Box	RESPONSIBLE_I	EMPL_CHECKLIS			
14	1	Responsible Nam	Edit Box	NAME	PERSONAL_DAT			13
15	1	Comment	Long Edit Box	COMMENTS	EMPL_CHECKLIS			
16	2	Scroll Bar 2	Scroll Bar					
17	2	Chklist Seq	Edit Box	CHECKLIST_SEQ	EMPL_CHKLSI_I			
18	2	Chklist Itm	Edit Box	CHKLSI_ITEM_C	EMPL_CHKLSI_I			
19	2	Briefing Descriptio	Edit Box	DESCR	CHKLSI_ITEM_T			18
20	2	Briefing Status	Drop Down List	BRIEFING_STAT	EMPL_CHKLSI_I			

EMPLOYEE_CHECKLIST Page Order

EMPL_CHECKLIST is the primary database record for the level 1 scrollbar on the EMPLOYEE_CHECKLIST page. The following PeopleCode accesses the level 1 rowset using EMPL_CHECKLIST. The Business Interlink Object name is QE_BI_EMPL_CHECKLIST. This Business Interlink Object uses the level 1 rowset as its input and its output.

The **InputRowset** method uses this rowset as input for QE_BI_EMPL_CHECKLIST. Then a blank duplicate of the rowset is created with **CreateRowset**, and then the output of QE_BI_EMPL_CHECKLIST is fetched into the blank rowset with **FetchIntoRowset**.

```

&MYROWSET = GetRowset (SCROLL.EMPL_CHECKLIST);

&ROWCOUNT = &QE_BI_EMPL_CHECKLIST.InputRowset (&MYROWSET);

&RSLT = &QE_BI_EMPL_CHECKLIST.Execute();

/* do some error processing */

&WorkRowset = CreateRowset (&MYROWSET);

&ROWCOUNT = &QE_BI_EMPL_CHECKLIST.FetchIntoRowset (&WorkRowset);

If &ROWCOUNT = 0 Then

    /* do some error processing */

Else

    /* Process the rowset from QE_BI_EMPL_CHECKLIST. Check it for errors. */

    For &I = 1 to &WorkRowset.RowCount

        For &K = 1 to &WorkRowset(&I).RecordCount

            &REC = &WorkRowset(&I).GetRecord(&K);

            &REC.ExecuteEdits();

            For &M = 1 to &REC.FieldCount

                If &REC.GetField(&M).EditError Then

                    /* there are errors */

                    /* do other processing */

                    End-If;

                End-For;

            End-For;

        End-For;

    End-for;

End-if;

```

Related Topics

InputRowset, Data Buffer Access, Rowset Class

FetchNextRow

Syntax

```
FetchNextRow(outputname, value)
```

where *outputname* and *value* are in matched pairs, in the form:

```
outputname1, value1 [, outputname2, value2] . . .
```

Description

After the Business Interlink object executes the method **Execute**, the **FetchNextRow** method can be used to retrieve a row of output and store the values of the output name (*outputname*) to PeopleCode variables or record fields (*value*). These must be entered in matched pairs, that is, every output name must be followed by its matching value.



The output **name**, not the output path of the interface definition is used for this method.

There must be an *outputname* for every output name defined in the interface definition used to instantiate the Business Interlink object.

If you specify a record field that is not part of the record the PeopleCode program is associated with, you must use *recname.fieldname* for that *value*.

You can specify default values for every output name in the interface definition (created in Application Designer.) These values will be used if you create a PeopleCode "template" by dragging the interface definition from the Project window in Application Designer to an open PeopleCode editor window.



For more information, see Generating the PeopleCode Template.

Parameters

outputname

Specify the output name. There must be one *outputname* for every output name defined in the interface definition used to instantiate the Business Interlink object.

value

Specify the value for the output name. This can be a constant, a variable, or a record field. Each *value* must be paired with an *outputname*.

Returns

A Boolean value: True if the row of output parameters was fetched. Otherwise, it returns False.

Example

In the following example, the Business Interlink object name is SRA_ALL_1, and the output names, such as **costs**, are being bound to record fields, such as STR_COST.

```
&RSLT = &SRA_ALL_1.FetchNextRow("costs", &STR_COST,
    "unit_costs", &STR_UNIT_COST,
    "customer_ship_dates", &STR_SHIP_DATE,
    "quantities", &STR_QUANTITY,
    "so_numbers", &STR_SO_NUM,
    "so_names", &STR_SO_NAME,
    "line_numbers", &STR_LINE_NUM,
    "ship_sets", &STR_SHIP_SET,
    "customer receipt dates", &STR_RECPT_DATE);
```

Related Topics

Execute, GetInterlink

GetCount

Syntax

GetCount (*docname*)

Description

The **GetCount** method returns the number of documents within a document list contained within an output structure for a Business Interlink object.

Parameters

<i>docname</i>	The name of the document list.
----------------	--------------------------------

Returns

The number of documents in the list.

Example

In the following example, the output structure for Calculate Cost, or the root level document, is accessed with the **GetOutputDocs** method. The Calculate Cost output parameter `output_param2_List` is a document, so the **GetDoc** method gets it from the output structure. Since `output_param2_List` is a document list, **GetCount** gets the number of documents in the list.

Local Interlink &QE FEDEX COST;

```

Local number &count;

Local BIDocs &CalcCostOut;

Local BIDocs &OutlistDoc;

Local number &ret;


&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);

// Get inputs, execute. (code not shown)


&CalcCostOut = &QE_FEDEX_COST.GetOutputDocs("");


/* Call GetValue for output_param1, call GetDoc, GetValue for Service_Rate (code
not shown) */


&OutlistDoc = &CalcCostOut.GetDoc("output_param2_List");

&count = &CalcCostOut.GetCount("output_param2_List");


&I = 0;

While (&I < &count)

    &ret = &OutlistDoc.GetValue("output_member1", &VALUE1);

    &ret = &OutlistDoc.GetValue("output_member2", &VALUE2);

    If &ret = 0 Then

        /* Process output values */

        &I = &I + 1;

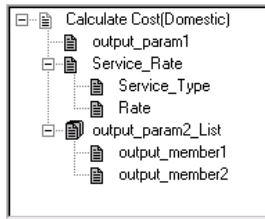
        &retoutput = &OutlistDoc.GetNextDoc();

    End-If;

End-While;

```

The following shows the output structure for this example. It contains three output parameters: output_param1, Service_Rate, and output_param2_List.



Example Output Structure

GetDoc

Syntax

GetDoc (*docname*)

Description

The **GetDoc** method gets a document from an output structure. The document is an output parameter for a Business Interlink object that is not of simple type (such as integer or string). You must get the document from the output structure before you can get values from its members with **GetValue**.

You can call **GetDoc** using dot notation, to access documents that are "nested" within other documents. For example, the following accesses a document nested three levels deep:

```
&Docs3 = &CalcCostOut.GetDoc ("Doc1.Doc2.Doc3") ;
```

Parameters

docname

The name of the document that **GetDoc** should get from the output structure.

Returns

A reference to the document received from the output structure.

Example

In the following example, the Output structure for Calculate Cost(Domestic), or the root level document, is created with the **GetOutputDocs** method. (If you wanted to create, or get, more Output structures, you would call **GetNextDoc**.) The Calculate Cost output parameter Service_Rate is a document, so the **GetDoc** method gets it from the Output structure.

```
Local Interlink &QE_FEDEX_COST;

Local number &count;

Local BIDocs &CalcCostOut;

Local BIDocs &ServiceRateDoc;

Local number &ret;
```

```

&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);

// Get inputs, execute. (code not shown)

&CalcCostOut = &QE_FEDEX_COST.GetOutputDocs("");

&ret = &CalcCostOut.GetValue("output_param1",&VALUE);

&ServiceRateDoc = &CalcCostOut.GetDoc("Service_Rate");

&ret = &ServiceRateDoc.GetValue("Service_Type", &SERVICE_TYPE);

&ret = &ServiceRateDoc.GetValue("Rate", &RATE);

/* Call GetDoc, GetValue, GetNextDoc for output_param2_List (code not shown) */

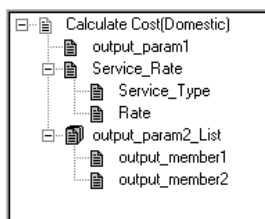
If &ret = 0 Then

    /* Process output values */

End-If;

```

The figure below shows the Output structure for this example. It contains three output parameters: output_param1, Service_Rate, and output_param2_List. This is a version of the Federal Express plug-in that was modified for this example (output_param1 and output_param2_List were added).



Example Output structure

In the following example, **GetDoc** is used to access a document that is nested more deeply. If you want to access a document that is more deeply nested, you can either call **GetDoc** for each nesting, or you can call **GetDoc** once using the nesting feature.

Calling **GetDoc** with the nesting feature:

```

Local Interlink &QE_4GETDOC;

Local BIDocs &CalcCostOut, &Docs3;

Local number &ret;

```

```

&QE_4GETDOC = GetInterlink(Interlink.QE_4GETDOC_EX);

// Get inputs, execute. (code not shown)

&CalcCostOut = &QE_4GETDOC.GetOutputDocs("");

&Docs3 = &CalcCostOut.GetDoc("Doc1.Doc2.Doc3");

&ret = &Docs3.GetValue("output_member3", &VALUE);

```

Calling **GetDoc** without the nesting feature:

```

Local Interlink &QE_4GETDOC;

Local BIDocs &CalcCostOut, &Docs1, &Docs2, &Docs3;

Local number &ret;

&QE_4GETDOC = GetInterlink(Interlink.QE_4GETDOC_EX);

// Get inputs, execute. (code not shown)

&CalcCostOut = &QE_4GETDOC.GetOutputDocs("");

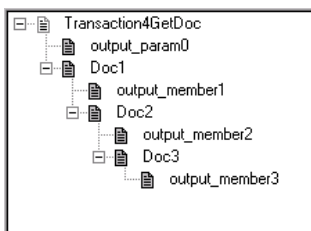
&Docs1 = &CalcCostOut.GetDoc("Doc1");

&Docs2 = &Docs1.GetDoc("Doc2");

&Docs3 = &Docs2.GetDoc("Doc3");

&ret = &Docs3.GetValue("output_member3", &VALUE);

```



Example Output Structure with Nested Documents

GetFieldCount

Syntax

```
GetFieldCount()
```

Description

Use the **GetFieldCount** method to support dynamic output. It returns the number of columns in the output buffer, which is the same as the number of outputs in each row of the output buffer.



This method can also be used with hierarchical doc data.

Parameters

None.

Returns

The number of columns in the output buffer, which is the same as the number of outputs in each row of the output buffer.

Example

The following example moves to the first column, first field of the output buffer. The **Repeat** loop goes through every row in the output buffer. The **For** loop processes every field in every row.

```
If (&MYBI.MoveFirst()) Then

    Repeat

        For &I = 1 to &MYBI.GetFieldCount

            &TYPE = &MYBI.GetFieldType(&I);

            Evaluate &TYPE

            Where = 1

            /* do processing */

            . . .

            End-Evaluate;

        End-For;

    Until Not (&MYBI.MoveNext());

Else

    /* do error processing - output buffer not returned */
```

```
End-If;
```

Related Topics

GetFieldType, GetFieldValue, MoveFirst, MoveNext, Using Hierarchical Data (BIDocs)

GetFieldType

Syntax

```
GetFieldType(index)
```

Description

Use the **GetFieldType** method to support dynamic output. It returns the type of field specified by *index*. You can only use this method after you have used the MoveFirst method: otherwise the system doesn't know where to start.



This method can also be used with hierarchical doc data.

Parameter

index

Specify the number of the field you want to find the type of.

Returns

A number indicating the type of the field. The valid values are:

<i>Value</i>	<i>Description</i>
1	String
2	Integer
3	Float
4	Boolean
5	Date
6	Time
7	DateTime
8	Binary
9	Object

Example

In the following example, the Business Interlink Object name is &MYBI. The example uses **MoveFirst** to move to the first row of the output buffer, and to the first column, or first field, in

that row. The **Repeat** loop uses **MoveNext** to go through every row in the output buffer. The **For** loop processes every field in every row, using the **GetFieldCount** method to get the number of fields, or outputs, in the row. Within the **For** loop, **GetFieldType** gets the type of the field data (string, integer, and so on.)

```
/* Add inputs to the Business Interlink Object, then call Execute to execute the
Business Interlink Object. You are then ready to get the outputs using the
following code. */
```

```
If (&MYBI.MoveFirst()) Then

  Repeat

    For &I = 1 to &MYBI.GetFieldCount

      &TYPE = &MYBI.GetFieldType(&I);

      Evaluate &TYPE

      Where = 1

      &STRING_VARIABLE = &MYBI.GetFieldValue(&I);

      /* test for and process other field types */

      End-Evaluate;

    End-For;

  Until Not (&MYBI.MoveNext());

Else

  /* Process error - no output buffer */

End-If;
```

Related Topics

GetFieldCount, GetFieldValue, MoveFirst, MoveNext

GetFieldValue

Syntax

```
GetFieldValue(index)
```

Description

Use the **GetFieldValue** method to support dynamic output. It returns the value of the field specified by *index*. You can only use this method after you have used the **MoveFirst** method: otherwise the system doesn't know where to start.



This method can also be used with hierarchical doc data.

Parameter

<i>index</i>	Specify the number of the field you want to find the value of.
--------------	--

Returns

A string containing the value of the field.

Example

In the following example, the Business Interlink Object name is &MYBI. The example uses **MoveFirst** to move to the first row of the output buffer, and to the first column, or first field, in that row. The **Repeat** loop uses **MoveNext** to go through every row in the output buffer. The **For** loop processes every field in every row, using the **GetFieldCount** method to get the number of fields, or outputs, in the row. Within the **For** loop, **GetFieldValue** gets the value of the field data.

```
/* Add inputs to the Business Interlink Object, then call Execute to execute the
Business Interlink Object. You are then ready to get the outputs using the
following code. */
```

```
If (&MYBI.MoveFirst()) Then

  Repeat

    For &I = 1 to &MYBI.GetFieldCount

      &TYPE = &MYBI.GetFieldType(&I);

      Evaluate &TYPE

      Where = 1

      &STRING_VARIABLE = &MYBI.GetFieldValue(&I);

      /* test for and process other field types */

      End-Evaluate;

    End-For;

  Until Not (&MYBI.MoveNext());

Else

  /* Process error - no output buffer */

End-If;
```

Related Topics

GetFieldCount, GetFieldType, MoveFirst, MoveNext

GetInputDocs

Syntax

```
GetInputDocs ( " " )
```

Description

The **GetInputDocs** method gets the top input document at the root level for a Business Interlink object. This is a hierarchical structure that contains the values for the inputs for this Business Interlink object. The methods that you use to put the input values into this document are the hierarchical data methods.



For more information about the hierarchical data methods, see Using Hierarchical Data (BIDocs).

Parameters

A null string.

Returns

An input document. This is the document at the top of the root level of the input document for a Business Interlink object.

Example

```
Local Interlink &QE_FEDEX_COST;  
  
Local BIDocs &CalcCostOut, &CalcCostIn;  
  
    &QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);  
  
    &CalcCostIn = &QE_FEDEX_COST.GetInputDocs ( " " );  
  
/* You can now insert the input values and execute the Business Interlink  
object. */
```

GetNextDoc

Syntax

```
GetNextDoc ( )
```

Description

The **GetNextDoc** method gets a document from one of the following levels:

- The root level of the Output structure for a Business Interlink object. This level was accessed with the **GetOutputDocs** method.
- When a document within the Output structure is a list, **GetNextDoc** gets another document from the list. If you use **GetNextDoc** on a document that is not a list, **GetNextDoc** fails and returns an error.

Parameters

None.

Returns

<i>Number</i>	<i>Enum value and Meaning</i>
0	eNoError. The method succeeded.
1	eNO_DOCUMENT. The document referenced by this method does not exist.
2	eDOCLIST_OUT_RANGE. You tried to access a document in a list, but the document list is out of range. For example, if a document list contains five documents, and then you call GetDoc/GetOutputDocs once, you can call GetNextDoc four times; the fifth time will result in this error.
3	eDOCUMENT_UNINITIALIZED. Internal error.
4	eNOT_DOCUMENTTYPE. You tried to perform an operation upon a parameter that is not a document type.
5	eNOT_LISTTYPE. You tried to perform a GetNextDoc or AddNextDoc upon a document that is not a list.
7	eNOT_BASICTYPE. You tried to perform a GetValue or AddValue upon a parameter that is not of basic type: integer, float, string, time, date, datetime.
8	eNO_DATA. You tried to retrieve data from a document that contained no data.

Example

In this example, the Output structure for Calculate Cost, or the root level document, is accessed with the **GetOutputDocs** method. The **GetNextDoc** method gets another Output structure,

assuming that there is more than one of them. The Calculate Cost output parameter `output_param2_List` is a document, so the **GetDoc** method gets it to the Output structure. `output_param2_List` is also a document list, so **GetNextDoc** method gets the next `output_param2_List` document.

```

Local Interlink &QE_FEDEX_COST;

Local number &count1, &count2;

Local BIDocs &CalcCostOut;

Local BIDocs &OutlistDoc;

Local number &ret1, &ret2;


&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);

// Get inputs, execute. (code not shown)


&CalcCostOut = &QE_FEDEX_COST.GetOutputDocs("");


&ret1 = 0;

While (&ret1)

    &ret1 = &CalcCostOut.GetValue("output_param1", &VALUE);

    &ServiceRateDoc = &CalcCostOut.GetDoc("Service_Rate");

    &ret1 = &ServiceRateDoc.GetValue("Service_Type", &SERVICE_TYPE);

    &ret1 = &ServiceRateDoc.GetValue("Rate", &RATE);

    /* Process output values */


    &OutlistDoc = &CalcCostOut.GetDoc("output_param2_List");

    &count2 = &CalcCostOut.GetCount("output_param2_List");

    While (&I < &count2)

        &ret2 = &OutlistDoc.GetValue("output_member1", &VALUE1);

        &ret2 = &OutlistDoc.GetValue("output_member2", &VALUE2);

        If &ret2 = 0 Then

            /* Process output values */

            &I = &I + 1;

```

```

        &ret2 = &OutlistDoc.GetNextDoc();

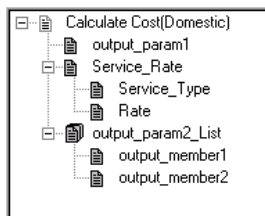
    End-If;

    &ret1 = &CalcCostOut.GetNextDoc();

End-While;

```

The following shows the Output structure for this example. It contains three output parameters: output_param1, Service_Rate, and output_param2_List.



Example Output structure

GetOutputDocs

Syntax

```
GetOutputDocs ( " " )
```

Description

The **GetOutputDocs** method gets the top output document at the root level for a Business Interlink object. This is a hierarchical structure that contains the values for the outputs for this Business Interlink object. The methods that you use to get output values from this document are the hierarchical data methods.



For more information about the hierarchical data methods, see [Using Hierarchical Data \(BIDocs\)](#).

Parameters

A null string.

Returns

An output document. This is the document at the top of the root level of the output document for a Business Interlink object.

Example

```

Local Interlink &QE_FEDEX_COST;

Local BIDocs &CalcCostIn, &CalcCostOut;

&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);

&CalcCostOut = &QE_FEDEX_COST.GetOutputDocs("");

/* You can now execute the Business Interlink object and get the output values.
*/

```

GetPreviousDoc

Syntax

```
GetPreviousDoc()
```

Description

The **GetPreviousDoc** method gets the previous document from either the root level of the Output structure for a Business Interlink Object, or from a document within the Output structure. When these documents are in a list, **GetPreviousDoc** gets the previous document in the list. You must get the document before you can get its values with **GetValue**.

Parameters

None.

Returns

<i>Number</i>	<i>Enum value and Meaning</i>
0	eNoError. The method succeeded.
1	eNO_DOCUMENT. The document referenced by this method does not exist.
2	eDOCLIST_OUT_RANGE. You tried to access a document in a list, but the document list is out of range. For example, if a document list contains five documents, and then you call GetDoc/GetOutputDocs once, you can call GetNextDoc four times; the fifth time will result in this error.
3	eDOCUMENT_UNINITIALIZED. Internal

	error.
4	eNOT_DOCUMENTTYPE. You tried to perform an operation upon a parameter that is not a document type.
5	eNOT_LISTTYPE. You tried to perform a GetNextDoc or AddNextDoc upon a document that is not a list.
7	eNOT_BASICTYPE. You tried to perform a GetValue or AddValue upon a parameter that is not of basic type: integer, float, string, time, date, datetime.
8	eNO_DATA. You tried to retrieve data from a document that contained no data.

Example

In this example, the Output structure for Calculate Cost, or the root level document, is accessed with the **GetOutputDocs** method. It contains one output parameter, output_param2_List, which is also a list. The **GetCount** and **MoveToDoc** methods point to the last output_param2_List document in the list. The **GetPreviousDoc** method is used in a loop to cycle through the output_param2_List list, starting with the last and ending with the first in the list, and get each output_param2_List document and its values.

```

Local Interlink &QE_FEDEX_COST;

Local number &count;

Local BIDocs &CalcCostOut;

Local BIDocs &OutlistDoc;

Local number &ret;

&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);

// Get inputs, execute. (code not shown)

&CalcCostOut = &QE_FEDEX_COST.GetOutputDocs("");

/* Call GetValue for output_param1, call GetDoc, GetValue for Service_Rate (code
not shown) */

&OutlistDoc = &CalcCostOut.GetDoc("output_param2_List");

&count = &CalcCostOut.GetCount("output_param2_List");

```

```

&ret = &OutlistDoc.MoveToDoc(&count-1);

&I = &count;

While (&I > 0)

    &ret = &OutlistDoc.GetValue("output_member1", &VALUE1);

    &ret = &OutlistDoc.GetValue("output_member2", &VALUE2);

    If &ret = 0 Then

        /* Process output values */

        &I = &I - 1;

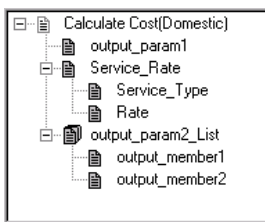
        &retoutput = &OutlistDoc.GetPreviousDoc("");

    End-If;

End-While;

```

The following shows the Output structure for this example. It contains three output parameters: output_param1, Service_Rate, and output_param2_List.



Example Output structure

GetStatus

Syntax

```
GetStatus()
```

Description

The **GetStatus** method tests the BIDocs document. To find the status for any BIDocs method that does not return a status value, call GetStatus just after you call that BIDocs method.

Parameters

None.

Return Value

Number	<i>Enum value and Meaning</i>
0	NoError. The method succeeded.
1	NO_DOCUMENT. The document referenced by this method does not exist.
2	LIST_OUT_RANGE. You tried to access a document in a list, but the document list is out of range. For example, if a document list contains five documents, and then you call GetDoc/GetOutputDocs once, you can call GetNextDoc four times; the fifth time will result in this error.
3	DOCUMENT_UNINITIALIZED. Internal error.
4	NOT_DOCUMENTTYPE. You tried to perform an operation upon a parameter that is not a document type.
5	NOT_DOCUMENTLISTTYPE. You tried to perform a GetNextDoc or AddNextDoc upon a document that is not a list.
6	NOT_LISTTYPE. You tried to perform a list operation using GetValue, AddValue, on a non-list.
7	NOT_SINGLEBASICTYPE. You tried to perform a GetValue or AddValue upon a list that does not use a single basic type: integer, float, string, time, date, datetime.
9	NO_DATA. You tried to retrieve data from a document that contained no data.
10	GENERIC_ERROR. There was an error with the transaction.

GetValue

Syntax

GetValue(paramname, value)

Description

The **GetValue** method gets a value from an output parameter within a document of the Output structure for a Business Interlink object.

Parameters

Paramname

The name of the member of the document that is having a value retrieved from it.

Value

The value that is retrieved. This can be a variable or a record field. The data type can be string, integer, double, float, time, date, or datetime.

Returns

Value	Meaning
string, <i>value</i>	The value of the output parameter.
Integer	When the syntax is integer <code>GetValue(name, value)</code> , integer is the return status of <code>GetValue</code> , listed in the table below.

Return Status for Integer

Number	Enum value and Meaning
0	NoError. The method succeeded.
1	NO_DOCUMENT. The document referenced by this method does not exist.
9	NO_DATA. You tried to retrieve data from a document that contained no data.
10	GENERIC_ERROR. There was an error with the transaction.

Example

In the following example, the Business Interlink Object name is QE_FEDEX_COST.

In the following example, the Output structure for Calculate Cost, or the root level document, is created with the **GetOutputDocs** method. (If you wanted to create, or get, more Output structures, you would call **GetNextDoc**.) The Calculate Cost output parameter `Service_Rate` is a document, so the **GetDoc** method gets it from the Output structure. Then the **GetValue** method gets values from each of the `Service_Rate` document members.

```
Local Interlink &QE_FEDEX_COST;

Local BIDocs &CalcCostOut;

Local BIDocs &ServiceRateDoc;

Local number &ret;

&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);
```

```
// Get inputs, execute. (code not shown)

&CalcCostOut = &QE_FEDEX_COST.GetOutputDocs("");

&ret = &CalcCostOut.GetValue("output_param1",&PARAM1);

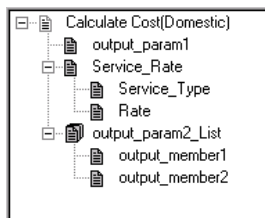
&ServiceRateDoc = &CalcCostOut.GetDoc("Service_Rate");

&ret = &ServiceRateDoc.GetValue("Service_Type", &SERVICE_TYPE);

&ret = &ServiceRateDoc.GetValue("Rate", &RATE);

/* Call GetDoc, GetValue, GetNextDoc for output_param2_List (code not shown) */
```

The figure below shows the Output structure for this example. It contains three output parameters: output_param1, Service_Rate, and output_param2_List. This is a version of the Federal Express plug-in that was modified for this example (output_param1 and output_param2_List were added).



Example Output structure

InputRowset

Syntax

```
InputRowset (&Rowset)
```

Description

The **InputRowset** method uses the data in the specified rowset to populate the input buffer, copying **like-named** fields in the appropriate structure. This method assumes that the names of the fields in the rowset match the names of the inputs defined in the Business Interlink definition, *and* that the structure of both rowsets are the same.

Use this method only if you have a hierarchical data structure and you want to preserve the hierarchy. Otherwise, use **BulkExecute** or **AddInputRow**.

Parameters

&Rowset Specify an existing, instantiated rowset object.

Returns

An optional value: the number of rows inserted into the output buffer.

Example

The example uses the rowset on level 1 from the EMPLOYEE_CHECKLIST page. A PeopleCode program running in a field on level 0 in that panel can access the child rowset (level 1), shown below from Scroll Bar 1 to Scroll Bar 2.

	Lv	Label	Type	Field	Record	Display Control	Related Field	Co
1	0	Frame	Frame					
2	0	Frame	Frame					
3	0	Frame	Frame					
4	0	Employee Name	Edit Box	NAME	PERSONAL_DAT			
5	0	ID	Edit Box	EMPLID	PERSONAL_DAT			
6	1	Checklist Item Tbl	Scroll Bar					
7	1	Checklist Sequen	Edit Box	CHECKLIST_SEQ	CHECKLIST_ITE			
8	1	Scroll Bar 1	Scroll Bar					
9	1	Checklist Date	Edit Box	CHECKLIST_DT	EMPL_CHECKLIS			
10	1	derived_hr_effdt	Edit Box	EFFDT	DERIVED_HR			
11	1	Checklist	Edit Box	CHECKLIST_CD	EMPL_CHECKLIS			
12	1	Checklist Descripti	Edit Box	DESCR	CHECKLIST_TBL			11
13	1	Responsible ID	Edit Box	RESPONSIBLE_I	EMPL_CHECKLIS			
14	1	Responsible Nam	Edit Box	NAME	PERSONAL_DAT			13
15	1	Comment	Long Edit Box	COMMENTS	EMPL_CHECKLIS			
16	2	Scroll Bar 2	Scroll Bar					
17	2	Chklist Seq	Edit Box	CHECKLIST_SEQ	EMPL_CHKLIST_I			
18	2	Chklist Itm	Edit Box	CHKLIST_ITEM_C	EMPL_CHKLIST_I			
19	2	Briefing Descriptio	Edit Box	DESCR	CHKLIST_ITEM_T			18
20	2	Briefing Status	Drop Down List	BRIEFING_STAT	EMPL_CHKLIST_I			

EMPLOYEE_CHECKLIST Page Structure

EMPL_CHECKLIST is the primary database record for the level 1 scrollbar on the EMPLOYEE_CHECKLIST page. The following PeopleCode access the level 1 rowset using EMPL_CHECKLIST. The Business Interlink Object name is QE_BI_EMPL_CHECKLIST. This Business Interlink Object uses the level 1 rowset as its input and its output.

The **InputRowset** method uses this rowset as input for QE_BI_EMPL_CHECKLIST. Then a blank duplicate of the rowset is created with **CreateRowset**, and then the output of QE_BI_EMPL_CHECKLIST is fetched into the blank rowset with **FetchIntoRowset**.

```
&MYROWSET = GetRowset (SCROLL.EMPL_CHECKLIST) ;

&ROWCOUNT = &QE_BI_EMPL_CHECKLIST.InputRowset (&MYROWSET) ;

&RSLT = &QE_BI_EMPL_CHECKLIST.Execute () ;
```

```

/* do some error processing */

&WorkRowset = CreateRowset (&MYROWSET);

&ROWCOUNT = &QE_BI_EMPL_CHECKLIST.FetchIntoRowset (&WorkRowset);

If &ROWCOUNT = 0 Then

    /* do some error processing */

Else

    /* Process the rowset from QE_BI_EMPL_CHECKLIST. Check it for errors. */

    For &I = 1 to &WorkRowset.RowCount

        For &K = 1 to &WorkRowset(&I).RecordCount

            &REC = &WorkRowset(&I).GetRecord(&K);

            &REC.ExecuteEdits();

            For &M = 1 to &REC.FieldCount

                If &REC.GetField(&M).EditError Then

                    /* there are errors */

                    /* do other processing */

                    End-If;

                End-For;

            End-For;

        End-For;

    End-for;

End-if;

```

Related Topics

FetchIntoRowset, Data Buffer Access, Rowset Class

MoveFirst

Syntax

```
MoveFirst()
```

Description

Use the **MoveFirst** method to support dynamic output. This method moves your cursor to the first column, first row of the output buffer. You must use this method *before* you try to access any data.

Parameters

None.

Returns

Boolean: True if successfully positioned cursor at the first column, first row of the output buffer, False otherwise.

Example

In the following example, the Business Interlink Object name is &MYBI. The example uses **MoveFirst** to move to the first row of the output buffer, and to the first column, or first field, in that row. The **Repeat** loop uses **MoveNext** to go through every row in the output buffer. The **For** loop processes every field in every row, using the **GetFieldCount** method to get the number of fields, or outputs, in the row.

```
/* Add inputs to the Business Interlink Object, then call Execute to execute the
Business Interlink Object. You are then ready to get the outputs using the
following code. */
```

```
If (&MYBI.MoveFirst()) Then

  Repeat

    For &I = 1 to &MYBI.GetFieldCount

      &TYPE = &MYBI.GetFieldType(&I);

      Evaluate &TYPE

      Where = 1

      &STRING_VARIABLE = &MYBI.GetFieldValue(&I);

      /* test for and process other field types */

      End-Evaluate;

    End-For;

  Until Not (&MYBI.MoveNext());

Else

  /* Process error - no output buffer */

End-If;
```

Related Topics

GetFieldCount, GetFieldType, GetFieldValue, MoveNext

MoveNext

Syntax

```
MoveNext()
```

Description

Use the **MoveNext** method to support dynamic output. This method moves your cursor to the first column of the next row of the output buffer. You can only use this method after you have used the MoveFirst method: otherwise, the system doesn't know where to start.

Parameters

None.

Returns

Boolean: True if successfully positioned cursor at the next row of the output buffer, False otherwise.

Example

In the following example, the Business Interlink Object name is &MYBI. The example uses **MoveFirst** to move to the first row of the output buffer, and to the first column, or first field, in that row. The **Repeat** loop uses **MoveNext** to go through every row in the output buffer. The **For** loop processes every field in every row, using the **GetFieldCount** method to get the number of fields, or outputs, in the row.

```
/* Add inputs to the Business Interlink Object, then call Execute to execute the
Business Interlink Object. You are then ready to get the outputs using the
following code. */
```

```
If (&MYBI.MoveFirst()) Then

  Repeat

    For &I = 1 to &MYBI.GetFieldCount

      &TYPE = &MYBI.GetFieldType(&I);

      Evaluate &TYPE

      Where = 1

      &STRING_VARIABLE = &MYBI.GetFieldValue(&I);

    /* test for and process other field types */
```

```

        End-Evaluate;

    End-For;

    Until Not (&MYBI.MoveNext());

Else

    /* Process error - no output buffer */

End-If;

```

Related Topics

GetFieldCount, GetFieldType, GetFieldValue, MoveFirst

MoveToDoc

Syntax

```
MoveToDoc(list_number)
```

Description

Within a list of documents in the Output structure, the **MoveToDoc** method moves to the documents given by the parameter *list_number*. After using **MoveToDoc**, the **GetValue** method gets the values of the document that is in the *list_number*+1 location in the list.

Parameters

list_number

The number indicating the document that **MoveToDoc** moves to. After using **MoveToDoc**, the **GetValue** method gets the values of the document that is in the *list_number*+1 location in the list. For example, if *list_number* is zero, then **MoveToDoc** moves to the first document in the list.

Returns

Number	Enum value and Meaning
0	eNoError. The method succeeded.
1	eNO_DOCUMENT. The document referenced by this method does not exist.
2	eDOCLIST_OUT_RANGE. You tried to access a document in a list, but the document list is out of range. For example, if a document list contains five documents, and then you call GetDoc/GetOutputDocs once, you can call GetNextDoc four times; the fifth time will result in this error.

3	eDOCUMENT_UNINITIALIZED. Internal error.
4	eNOT_DOCUMENTTYPE. You tried to perform an operation upon a parameter that is not a document type.
5	eNOT_LISTTYPE. You tried to perform a GetNextDoc or AddNextDoc upon a document that is not a list.
7	eNOT_BASICTYPE. You tried to perform a GetValue or AddValue upon a parameter that is not of basic type: integer, float, string, time, date, datetime.
8	eNO_DATA. You tried to retrieve data from a document that contained no data.

Example

The following example gets the values of the last document in the output_param2_List list. It uses **GetCount** to get the number of documents in the list, and then uses **MoveToDoc** to move to the last document in the list.

```

Local Interlink &QE_FEDEX_COST;

Local number &count;

Local BIDocs &CalcCostOut;

Local BIDocs &OutlistDoc;

Local number &ret;

&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);

// Get inputs, execute. (code not shown)

&CalcCostOut = &QE_FEDEX_COST.GetOutputDocs("");

&OutlistDoc = &CalcCostOut.GetDoc("output_param2_List");

&count = &CalcCostOut.GetCount("output_param2_List");

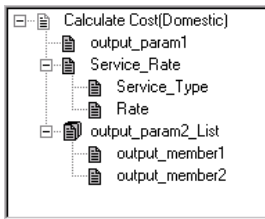
&ret = &OutlistDoc.MoveToDoc(&count-1);

&ret = &OutlistDoc.GetValue("output_member1", &VALUE1);

&ret = &OutlistDoc.GetValue("output_member2", &VALUE2);

```

The figure below shows the Output structure for this example. It contains three output parameters: `output_param1`, `Service_Rate`, and `output_param2_List`. This is a version of the Federal Express plug-in that was modified for this example (`output_param1` and `output_param2_List` were added).



Example Output structure

ResetCursor

Syntax

```
ResetCursor()
```

Description

The **ResetCursor** method resets the cursor in the Output structure for a Business Interlink object to the top. Once you call this method, the next time you call **GetValue**, you get an output value from the first document of the Output structures for a Business Interlink object.

Parameters

None.

Returns

None.

Example

The following code uses **ResetCursor** to reset the cursor in the Output structure to the top.

```

Local Interlink &QE_FEDEX_COST;

Local BIDocs &CalcCostOut;

&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);

// Get inputs, execute. (code not shown)

```

```

&CalcCostOut = &QE_FEDEX_COST.GetOutputDocs("");

// Perform actions on the output documents (code not shown)

&CalcCostOut.ResetCursor();

```

Business Interlink BIDocs Methods

This section describes the PeopleCode methods you use with Incoming Business Interlinks.



For more information, see PeopleSoft Business Interlink Application Developer Guide.

AddAttribute

Syntax

```
AddAttribute(attributename, attributevalue)
```

Description

The **AddAttribute** method adds an attribute name and its value to an XML element referenced by a BiDocs object.

Parameters

<i>attributename</i>	String. The name of the attribute.
<i>attributevalue</i>	String. The value of the attribute.

Return Value

Number. The return status. NoError, or 0, means the method succeeded.



For more information on the return status, see GetStatus.

Example

Here is a set of XML response code.

```

<?xml version="1.0"?>

<postreqresponse>

```

```

<candidate>

  <user>

    <location scenery="great" density="low" blank="eh?">

    </location>

  </user>

</candidate>

</postreqresponse>

```

Here is the PeopleCode that builds it.

```

Local BIDocs &rootDoc, &postreqresponse;

Local BIDocs &candidates, &location, &user;

Local number &ret;

&rootDoc = GetBiDoc("");

&ret = &rootDoc.AddProcessInstruction("<?xml version=\"1.0\"?>");

&postreqresponse = &rootDoc.CreateElement("postreqresponse");

&candidates = &postreqresponse.CreateElement("candidates");

&user = &candidates.CreateElement("user");

&location = &user.CreateElement("location");

&ret = &location.AddAttribute("scenery", "great");

&ret = &location.AddAttribute("density", "low");

&ret = &location.AddAttribute("blank", "eh?");

```

AddComment

Syntax

```
AddComment (comment)
```

Description

The **AddComment** method adds an XML comment after the beginning tag of an XML element referenced by a BiDoc object.

Parameters

comment String. The comment.

Return Value

Number. The return status. NoError, or 0, means the method succeeded.



For more information on the return status, see GetStatus.

Example

Here is a set of XML response code.

```
<?xml version="1.0"?>

<postreqresponse>

  <error>

    <!--this is a comment line-->

    <errorcode>1</errorcode>

    <errortext></errortext>

  </error>

</postreqresponse>
```

Here is the PeopleCode that builds it.

```
Local BIDocs &rootDoc, &postreqresponse, &error, &errorcode, &errortext;

Local string &blob;

Local number &ret;

&rootDoc = GetBiDoc("");

/* add a processing instruction*/

&ret = &rootDoc.AddProcessInstruction("<?xml version=""1.0""?>");

/* create an element and add text*/

&postreqresponse = &rootDoc.CreateElement("postreqresponse");

&error = &postreqresponse.CreateElement("error");

&ret = &error.AddComment("this is a comment line");
```



```
Local BIDocs &error, &errorcode, &errortext;

Local number &ret;

&rootDoc = GetBiDoc("");

/* add a processing instruction*/

&ret = &rootDoc.AddProcessInstruction("<?xml version=\"1.0\"?>");

/* create an element and add text*/

&postreqresponse = &rootDoc.CreateElement("postreqresponse");

&error = &postreqresponse.CreateElement("error");

&ret = &error.AddComment("this is a comment line");

&errorcode = &error.CreateElement("errorcode");

&ret = &errorcode.AddText("1");

&errortext = &error.CreateElement("errortext");
```

AddText

Syntax

AddText (*text*)

Description

The **AddText** method adds text to an XML element referenced by a BiDocs object.

Parameters

<i>text</i>	String. The text.
-------------	-------------------

Return Value

Number. The return status. NoError, or 0, means the method succeeded.



For more information on the return status, see [GetStatus](#).

Example

Here is a set of XML response code.

```
<?xml version="1.0"?>
```

```

<postreqresponse>

  <error>

    <!--this is a comment line-->

    <errorcode>1</errorcode>

    <errortext></errortext>

  </error>

</postreqresponse>

```

Here is the PeopleCode that builds it.

```

Local BIDocs &rootDoc, &postreqresponse;

Local BIDocs &error, &errorcode, &errortext;

Local number &ret;

&rootDoc = GetBiDoc("");

/* add a processing instruction*/

&ret = &rootDoc.AddProcessInstruction("<?xml version=\"1.0\"?>");

/* create an element and add text*/

&postreqresponse = &rootDoc.CreateElement("postreqresponse");

&error = &postreqresponse.CreateElement("error");

&ret = &error.AddComment("this is a comment line");

&errorcode = &error.CreateElement("errorcode");

&ret = &errorcode.AddText("1");

&errortext = &error.CreateElement("errortext");

```

CreateElement

Syntax

```
CreateElement(elementname)
```

Description

The **CreateElement** method creates an XML element with the given name within a BiDoc object.

Parameters

elementname String. The XML element name.

Return Value

BiDocs. The reference to the created element.

Example

Here is a set of XML response code.

```
<?xml version="1.0"?>

  <postreqresponse>

    <error>

      <errorcode>1</errorcode>

      <errortext></errortext>

    </error>

  </postreqresponse>
```

Here is the PeopleCode that builds it.

```
Local BiDocs &rootDoc, &postreqresponse;

Local BiDocs &error, &errorcode, &errortext;

Local number &ret;

&rootDoc = GetBiDoc("");

/* add a processing instruction*/
&ret = &rootDoc.AddProcessInstruction("<?xml version=\"1.0\"?>");

/* create an element and add text*/

&postreqresponse = &rootDoc.CreateElement("postreqresponse");

&error = &postreqresponse.CreateElement("error");

&errorcode = &error.CreateElement("errorcode");

&ret = &errorcode.AddText("1");

&errortext = &error.CreateElement("errortext");
```

GenXMLString

Syntax

```
GenXMLString()
```

Description

The **GenXMLString** method creates an XML string from a BiDocs object. The BiDocs object must contain the shape and data needed for an XML string. This is part of the Incoming Business Interlink functionality, which allow PeopleCode to receive an XML request and return an XML response.

Parameters

None.

Return Value

String. The XML string containing the shape and data of the BiDocs object. For example, you can use this method to create an XML string containing an XML response.

Example

The following example takes a BiDocs structure that contains an XML response and puts that into a text string. Once this is done, the %Response.Write function can send this as an XML response.

```
Local BiDocs &rootDoc;

Local string &xmlString;

/* Create a BiDoc structure containing the data and shape of your XML response
(code not shown) */

/* Generate the XML string containing the data and shape of your XML response
from the BiDoc structure */

&xmlString = &rootDoc.GenXMLString();

%Response.Write(&xmlString);
```

GetAttributeName

Syntax

```
GetAttributeName(attributenum)
```

Description

The **GetAttributeName** method gets the name of an attribute within an XML element referenced by a BiDocs object.

Parameters

attributenum Number. The index number of the attribute.

Return Value

String. The name of the attribute.

Example

Here is a set of XML request code.

```
<?xml version="1.0"?>

<postreq>

  <email>joe_blow@peoplesoft.com</email>

  <location scenery="great" density="low" blank="eh?">

    <city>San Rafael</city>

    <state>CA</state>

    <zip>94522</zip>

    <country>US</country>

  </location>

</postreq>
```

Here is the PeopleCode that gets the name of the second attribute in the location node. &attrName is density.

```
Local BiDocs &rootInDoc, postreqDoc, &locationDoc;

Local string &blob, &attrName;

&blob = %Request.GetContentBody();

&rootInDoc = GetBiDoc(&blob);

&postreqDoc = &rootInDoc.GetNode("postreq");

&locationDoc = &postreqDoc.GetNode("location");

&attrName = &locationDoc.GetAttributeName(2);
```

GetAttributeValue

Syntax

```
GetAttributeValue({attributename | attributenum})
```

Description

The **GetAttributeValue** method gets the value of an attribute within an XML element referenced by a BiDocs object.

Parameters

<i>attributenum</i> <i>attributename</i>	Specify the attribute you want to get the value for. You can specify either the attribute number (1 for the first attribute, 2 for the second, and so on) or the attribute name (an XML tag.)
---	---

Return Value

String. The value of the attribute.

Example

Here is a set of XML request code.

```
<?xml version="1.0"?>

<postreq>

  <email>joe_blow@peoplesoft.com</email>

  <location scenery="great" density="low" blank="eh?">

    <city>San Rafael</city>

    <state>CA</state>

    <zip>94522</zip>

    <country>US</country>

  </location>

</postreq>
```

Here is the PeopleCode that gets the value of the second attribute in the location node. &attrValue is low.

```
Local BiDocs &rootInDoc, &postreqDoc, &locationDoc;

Local string &blob, &attrValue;

&blob = %Request.GetContentBody();
```

```

&rootInDoc = GetBiDoc(&blob);

&postreqDoc = &rootInDoc.GetNode("postreq");

&locationDoc = &postreqDoc.GetNode("location");

&attrValue = &locationDoc.GetAttributeValue(2);

```

GetNode

Syntax

```
GetNode({nodename | nodenumber})
```

Description

The **GetNode** method returns a BiDocs reference to a child XML node (element or comment). Use the **GetNode** method upon a BiDocs reference to an XML element in order to access the child nodes for that element.

Parameters

<i>nodenumber</i> <i>nodename</i>	Specify the node you want to reference. You can specify either a node number (the first node is 1, the second 2, and so on) or a node name (that is, the XML tag.)
-------------------------------------	--

Return Value

BiDocs. The returned XML element within a BiDocs object.

Example

Here is a set of XML request code.

```

<?xml version="1.0"?>

  <postreq>

    <email>joe_blow@peoplesoft.com</email>

    <projtitle>

      <projsubtitle>first_subtitle</projsubtitle>

      <projsubtitle>second_subtitle</projsubtitle>

      <projsubtitle>third_subtitle</projsubtitle>

    </projtitle>

  </postreq>

```

Here is the PeopleCode that gets the postreqDoc element and the projtitle element.

```
Local BIDocs &rootInDoc, &postreqDoc, &projtitleDoc;

Local string &name, &blob;

&blob = %Request.GetContentBody();

&rootInDoc = GetBiDoc(&blob);

&postreqDoc = &rootInDoc.GetNode("postreq");

&projtitleDoc = &postreqDoc.GetNode("projtitle");
```

Here is the PeopleCode that gets the postreqDoc element, the projtitle element, and the third projsubtitle element.

```
Local BIDocs &rootInDoc, &postreqDoc, &projtitleDoc, &projsubtitleDoc;

Local string &name, &blob;

&blob = %Request.GetContentBody();

&rootInDoc = GetBiDoc(&blob);

&postreqDoc = &rootInDoc.GetNode(1);

&projtitleDoc = &postreqDoc.GetNode(2);

&projsubtitleDoc = &projtitleDoc.GetNode(3);
```

In order to parse a list of elements like <projsubtitle>, where elements have the same name, you must call **GetNode** using an index number rather than the element name.

ParseXMLString

Syntax

```
ParseXMLString (XMLstring)
```

Description

The **ParseXMLString** method fills an existing BiDocs object with the data and shape from an XML string. This is part of the Incoming Business Interlink functionality, which allows PeopleCode to receive an XML request and return an XML response.

Parameters

XMLstring	A string containing an XML document.
-----------	--------------------------------------

Return Values

Number. The return status. NoError, or 0, means the method succeeded.



For more information on the return status, see `GetStatus`.

Example

The following example gets an XML request, creates an empty BiDoc, and then fills the BiDoc with the data and shape contained in the XML string. Once this is done, the `GetDoc` method and the `GetValue` method can get the value of the skills XML element, which is contained within the `postreq` element in the XML request.

```
Local BiDocs &rootInDoc, &postreqDoc;

Local string &blob;

Local number &ret;

&blob = %Request.GetContentBody();

/* process the incoming xml(request)- Create a BiDoc and fill with the request*/

&rootInDoc = GetBiDoc("");

&ret = &rootInDoc.ParseXMLString(&blob);

&postreqDoc = &rootInDoc.GetDoc("postreq");

&ret = &postreqDoc.GetValue("skills", &skills);
```

Business Interlink BiDocs Properties

This section describes the PeopleCode properties you use with Incoming Business Interlinks.



For more information, see PeopleSoft Business Interlink Application Developer Guide.

AttributeCount

The **AttributeCount** property gets the number of attributes within an XML element referenced by a BiDocs object.

This property is read-only.

Example

Here is a set of XML request code.

```
<?xml version="1.0"?>

<postreq>

  <email>joe_blow@peoplesoft.com</email>

  <location scenery="great" density="low" blank="eh?">

    <city>San Rafael</city>

    <state>CA</state>

    <zip>94522</zip>

    <country>US</country>

  </location>

</postreq>
```

Here is the PeopleCode that gets the number of attributes in the location XML element. `&count` should be 3, for scenery, density, and blank.

```
Local BiDocs &rootInDoc, &postreqDoc, &locationDoc;

Local string &blob;

Local number &count;

&blob = %Request.GetContentBody();

&rootInDoc = GetBiDoc(&blob);

&postreqDoc = &rootInDoc.GetNode("postreq");

&locationDoc = &postreqDoc.GetNode("location");

&count = &locationDoc.AttributeCount;
```

ChildNodeCount

The **ChildNodeCount** property returns the number of XML child nodes within the element referenced by the BiDocs object. Child nodes include XML elements, comments, and processing instructions.

This property is read-only.

Example

Here is a set of XML request code.


```

<?xml version="1.0"?>

  <postreq>

    <email>joe_blow@peoplesoft.com</email>

    <projtitle>

      <!--this is a comment line-->

      <projsubtitle>first_subtitle</projsubtitle>

      <projsubtitle>second_subtitle</projsubtitle>

      <projsubtitle>third_subtitle</projsubtitle>

    </projtitle>

  </postreq>

```

Here is the XML code that gets the number of nodes within <postreq> and <projtitle>. &count1 is 2, for <email> and <projtitle>, and &count2 is 4, for the three <projsubtitle> nodes and the comment node.

```

Local BiDocs &rootInDoc, &projtitleDoc;

Local string &blob;

Local number &count1, &count2;

&blob = %Request.GetContentBody();

&rootInDoc = GetBiDoc(&blob);

&postreqDoc = &rootInDoc.GetNode("postreq");

&count1 = &postreqDoc.ChildNodeCount;

&projtitleDoc = &postreqDoc.GetNode("projtitle");

&count2 = &projtitleDoc.ChildNodeCount;

```

NodeName

The **NodeName** property gets the name of an XML element referenced by a BiDocs object. Use this to get the name of an XML element when you used GetNode with an index number to retrieve it (meaning that you did not have the name of the XML element when you used GetNode).

This property is read-only.

Example

Here is a set of XML request code.

```
<?xml version="1.0"?>

<postreq>

  <email>joe_blow@peoplesoft.com</email>

  <projtitle>

    <projsubtitle>first_subtitle</projsubtitle>

    <projsubtitle>second_subtitle</projsubtitle>

    <projsubtitle>third_subtitle</projsubtitle>

  </projtitle>

</postreq>
```

Here is the PeopleCode that gets the name of the <email> element, email.

```
Local BiDocs &rootInDoc, &postreqDoc, &emailDoc;

Local string &emailName, &blob;

&blob = %Request.GetContentBody();

&rootInDoc = GetBiDoc(&blob);

&postreqDoc = &rootInDoc.GetNode(1);

&emailDoc = &postreqDoc.GetNode(1);

&emailName = &emailDoc.NodeName;
```

NodeType

The **NodeType** property returns the type of an XML tag within a BiDocs object as an integer. The valid values are:

Value	Description
1	Element (a normal XML tag)
7	Processing instruction
8	Comment

This property is read-only.

Example

Here is a set of XML request code.

```
<?xml version="1.0"?>

<postreq>

    <email>joe_blow@peoplesoft.com</email>

    <!--this is a comment-->

    <projtitle>

        <projsubtitle>first_subtitle</projsubtitle>

        <projsubtitle>second_subtitle</projsubtitle>

        <projsubtitle>third_subtitle</projsubtitle>

    </projtitle>

</postreq>
```

Here is the PeopleCode that gets types: &xmlprocType is 7 for processing instruction, postreqDoc is 1 for element, and commentType is 8 for comment.

```
Local BIDocs &rootInDoc, &postreqDoc, &commentDoc;

Local number &xmlprocType, &postreqType, &commentDoc;

Local string &blob;

&blob = %Request.GetContentBody();

/* <?xml version="1.0"?> */

&rootInDoc = GetBiDoc(&blob);

&xmlprocType = &rootInDoc.NodeType;

/* <postreq> */

&postreqDoc = &rootInDoc.GetNode(1);

&postreqType = &postreqDoc.NodeType;

/* <!--this is a comment--> */

&commentDoc = &postreqDoc.GetNode(2);

&commentType = &commentDoc.NodeType;
```

NodeValue

The **NodeValue** property returns the value of a node within an XML document as a string.

This property is read-only.

Example

Here is a set of XML request code.

```
<?xml version="1.0"?>

<postreq>

  <email>joe_blow@peoplesoft.com</email>

  <projtitle>

    <projsubtitle>first_subtitle</projsubtitle>

    <projsubtitle>second_subtitle</projsubtitle>

    <projsubtitle>third_subtitle</projsubtitle>

  </projtitle>

</postreq>
```

Here is the PeopleCode that gets the value of the third `<projsubtitle>` element, `third_subtitle`.

```
Local BIDocs &rootInDoc, &postreqDoc, &projtitleDoc, &projsubtitleDoc;

Local string &name, &blob;

&blob = %Request.GetContentBody();

&rootInDoc = GetBiDoc(&blob);

&postreqDoc = &rootInDoc.GetNode(1);

&projtitleDoc = &postreqDoc.GetNode(2);

&projsubtitleDoc = &projtitleDoc.GetNode(3);

&projsubtitleName = &projsubtitleDoc.NodeValue;
```

Business Interlink Class Property

StopAtError

Specifies whether the execution of the PeopleCode program will stop when there's an error, or if the PeopleCode program will try to capture the errors. This property takes a Boolean value: True

to halt execution of your PeopleCode program at an error, False to continue executing. The default value is True.

This property is read-write.

Example

```
&QE_RP_SRAALL_1.StopAtError = False;
```

Configuration Parameters

There are two types of configuration parameters: the ones defined by the Interlink plug-in the Business Interlink definition is based on, and the ones that are standard, that is, defined for every Business Interlink object.c

All configuration parameters must be set before you add any data to the input buffers.

Interlink Plug-in Configuration Parameters

The configuration parameters defined by the Interlink plug-in are accessed as if they were properties on the Business Interlink object. That is, in PeopleCode, you assign the value of a configuration parameter by using the Business Interlink object followed by a dot and the configuration parameter, in this format:

```
&MYINTERLINK.parameter = value;
```

You can also return the value of a configuration parameter:

```
&MYVALUE = &MYINTERLINK.parameter;
```

Each Business Interlink plug-in has its own set of configuration parameters. For example, the email project uses configuration parameters of SMTP_MAIL_SERVER, POP3MAIL_SERVER, LOGIN_NAME, PASSWORD, SENDERS_EMAIL_ADDRESS and REPLY_EMAIL_ADDRESS, while the Red Pepper transaction driver uses SERVER_NAME, RSERVER_HOST, RSERVER_PORT and TIMEOUT.

You can specify default values for every configuration parameter in the Business Interlink definition (created in Application Designer.) These values will be used if you create a PeopleCode "template" by dragging the Business Interlink definition from the Project window in Application Designer to an open PeopleCode editor window.



For more information, see [Generating the PeopleCode Template](#).

In the following example, the Interlink object name is QE_RP_SRAALL_1, and the driver is the Red Pepper driver:

```
&QE_RP_SRAALL_1.SERVER_NAME = "server";
```

```

&QE_RP_SRAALL_1.RSERVER_HOST = "pt-sun02.peoplesoft.com";

&QE_RP_SRAALL_1.RSERVER_PORT = "9884";

&QE_RP_SRAALL_1.TIMEOUT = 999;

&QE_RP_SRAALL_1.URL = "HTTP://www.PeopleSoft.com";

&QE_RP_SRAALL_1.StopAtError = False;

```

URL Configuration Parameter

Specifies the location and name of the Business Interlink plug-in to be used to connect to the external system. This configuration parameter takes a string value.

If the plug-in is located on a web server, you have to specify the name of the web server.

If you specify a file, you can specify either a relative or an absolute path:

- If you specify an absolute path, you must specify the drive letter:

```
&MYBI.URL = "File://D:\TEMP.MYDLL";
```

- If you specify a relative path, just use the file name:

```
&MYBI.URL = "File://TEMP.MYDLL";
```

If you specify a relative path, the system will first look for the file in the Location directory (specified by the user when the Business Interlink was first created), then it will look in the directory where PeopleTools is installed, in the PSTOOLS/Interface Drivers directory.

BIDocValidate Configuration Parameter

Specifies whether the system should verify whether the hierarchical data object (BIDoc) exists before adding or getting values from it. This configuration parameter takes a Boolean value: True if the system should verify before accessing the object, False otherwise.

The default value is True.

If this configuration parameter is specified as True, and the object specified doesn't exist, the PeopleCode program halts execution and an error is displayed.

Component Interface Classes

Component Interfaces are an evolution of the Message Agent, and are the focal points for externalizing access to existing PeopleSoft components. They provide realtime synchronous access to the PeopleSoft business rules and data associated with a component outside the PeopleSoft online system. Component Interfaces can be viewed as "black boxes" that encapsulate PeopleSoft data and business processes, and hide the details of the structure and implementation of the underlying page and data.

Component Interfaces are one of the many APIs that PeopleSoft provides for allowing integration with other systems.

A Component Interface maps to one, and only one, PeopleSoft component. The **Component Interface object**, instantiated from a session object, is created at runtime as a way to access the data specified by the Component Interface.

When you instantiate a Component Interface object:

- all the PeopleCode programs associated with the record fields, pages, component, and so on, **AND**
- the runtime component processor still perform all of the work they do in the online environment.

(The exceptions are any GUI manipulation found in a PeopleCode program, and search dialog specific processing.)



For more information about the exceptions, see Reusing Existing Code.

Component Interfaces are programmable through an OLE/COM or C/C++ interface, and through PeopleCode. Application Engine programs, application message subscription programs, or any other PeopleCode programs will be able to use Component Interfaces.

Like a component, you create the structure of a Component Interface in Application Designer, then at runtime, you populate the structure with data. This document is concerned with the runtime portion of a Component Interface.



For more information about creating a Component Interface at design time, see PeopleSoft Component Interface.

When you populate a Component Interface with data, the first thing you fill out are its keys, as you would in a component. These can be keys for getting an existing instance of the data or for creating a new instance of the data.

In addition to keys, a Component Interface is composed of *properties* and *methods*.

- Component Interface *properties* provide access to the data in a component buffer.
- Component Interface *methods* are functions that can be called to perform operations on a Component Interface.

There are two types of both methods and properties: *standard* and *user defined*. Standard properties and methods are provided automatically when you create a Component Interface. They perform operations common to all Component Interfaces, such as indicating what mode to operate the Component Interface, saving or creating a Component Interface. User defined properties are the specific record fields that an application developer has chosen to expose to an external system with the Component Interface. User-defined methods are PeopleCode programs

that an application developer can write to perform operations on a Component Interface. Each is specific to that Component Interface.

You can only instantiate a Component Interface object from a session object. Through the session object you can control access to the Component Interface, check for errors, control the runtime environment, and so on.

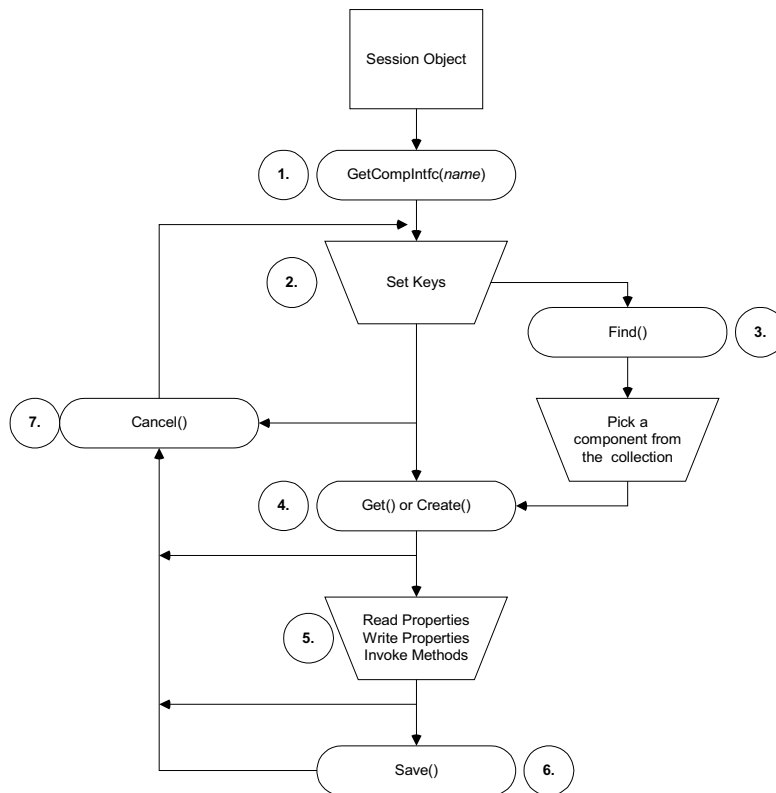


For more information, see Session Class.

Life Cycle of a Component Interface

At runtime, there are certain things you want to do with a Component Interface, like getting an instance of it, populating it with data, and so on. The following is an overview of this process. These steps are expanded in other sections.

1. Execute the GetCompIntfc method on the PeopleSoft session object to get a Component Interface.
2. Set the key values for the Component Interface object. If the keys you specify don't uniquely describe a component (partial keys), proceed to step 3. If the keys uniquely describe a component, skip to step 4.
3. Do *one* of the following:
 - Execute the Find standard method on the Component Interface. This returns a collection of Component Interfaces with their key values filled out. The user can then pick the unique Component Interface they want.
 - Execute either the Get or Create standard method to instantiate the Component Interface (populate it with data.)
4. Get property values, set property values, or execute user-defined methods. Setting a property value will fire the standard PeopleSoft business rules for the field associated with the property (any PeopleCode program associated with FieldChange, RowInsert, and so on.)
5. Execute the Save standard method to fire the standard PeopleSoft save business rules (any PeopleCode programs associated with SavePreChange, WorkFlow, and so on.) and commit any changes to the database.
6. At any point, the standard method Cancel can be executed to reset the Component Interface to its state in step 1.

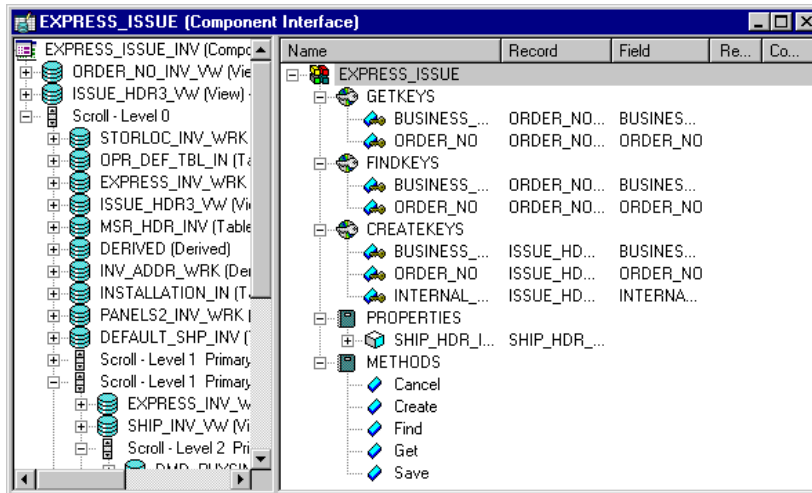


Life cycle of a Component Interface

Setting Component Interface Keys

The keys for a Component Interface are based on the key fields of the underlying component. There can be different types of keys for a Component Interface.

- **CREATEKEYS:** A list of the primary key fields of the search record specified to be used in Add mode for the component. This list is automatically generated.
- **GETKEYS:** A list of the primary (required) key fields on the search record. This list is automatically generated.
- **FINDKEYS:** A list of primary and alternate key fields on the primary search record for the component.



GETKEYS, FINDKEYS and CREATEKEYS for a Component Interface

If the Component Interface has CREATEKEYS, these are the keys you must set before you execute the Create() method to create a new instance of the data. If the Component Interface doesn't have CREATEKEYS, use the GETKEYS to specify a new instance of the data.

Use either the GETKEYS or FINDKEYS to specify an existing instance of the data.

To set key values, use the field names listed under GETKEYS, CREATEKEYS or FINDKEYS like properties on the Component Interface object. The following example sets the CREATEKEYS values to create a new instance of the data.

```
&MYCI = GetCompIntfc(CompIntfc.EXPRESS_ISSUE);

&MYCI.BUSINESS_UNIT = "H01B";

&MYCI.INTERNAL_FLG = "Y";

&MYCI.ORDER_NO = "NEXT";

&MYCI.Create();
```

Standard and User Defined Component Interface Methods

Every Component Interface comes with a standard set of methods:

- Cancel()
- Create()
- Find()
- Get()
- Save()

Use these methods during runtime to affect the data of the Component Interface.



For more information about each method, see Component Interface Class Methods.

The application developer can, at design time, disable any of these methods for the Component Interface.



For more information, see other PeopleSoft Component Interface.

In addition, an application developer can write their own methods. These methods are written as Functions using Component Interface PeopleCode. For example, suppose you wanted to be able to copy an instance of Component Interface data. You might write your own Clone method.



Note. User-defined method names must *not* be named **GetPropertyName**. The C header for Component Interfaces creates functions with that name so you can access each property. If you create your own **GetPropertyName** functions, you'll receive errors at runtime.

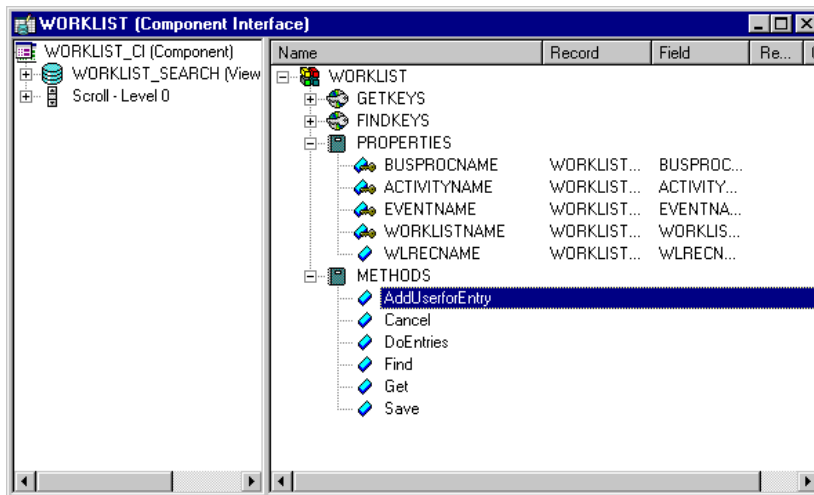


User-defined methods can only take simple types of arguments (such as number, character, and so on) because user-defined methods can be called from C/C++ or COM as well as from PeopleCode. PeopleCode can use more complex types (like rowset, array, record, and so on), but these types of arguments are unknown to C/C++ or COM.



For more information about writing your own methods, see PeopleSoft Component Interface.

In the following example, user-defined methods AddUserforEntry and DoEntries have been added to the standard methods.



Component Interface definition with user-defined methods

Accessing Component Interface Standard Properties

Every Component Interface comes with a standard set of properties. These properties can be divided as follows:

- Properties that affect how the Component Interface is executed
 - GetHistoryItems
 - InteractiveMode

These properties must be set before the Component Interface is populated with data (that is, before you use the **Get** or **Create** method.)

GetHistoryItems lets you access the data in the Component Interface in a similar manner as if you were accessing a component in correction mode. If you don't set this property as True, it's as if you were accessing a component in update/display mode.

InteractiveMode causes the Component Interface to emulate an online component. For example, if you set a value for a field in a Component Interface and you have set InteractiveMode to True, then any FieldChange PeopleCode programs associated with that field will fire as soon as you set that value.

- Properties that return information about the structure of the Component Interface
 - CreateKeyInfoCollection
 - FindKeyInfoCollection
 - GetKeyInfoCollection
 - PropertyInfoCollection

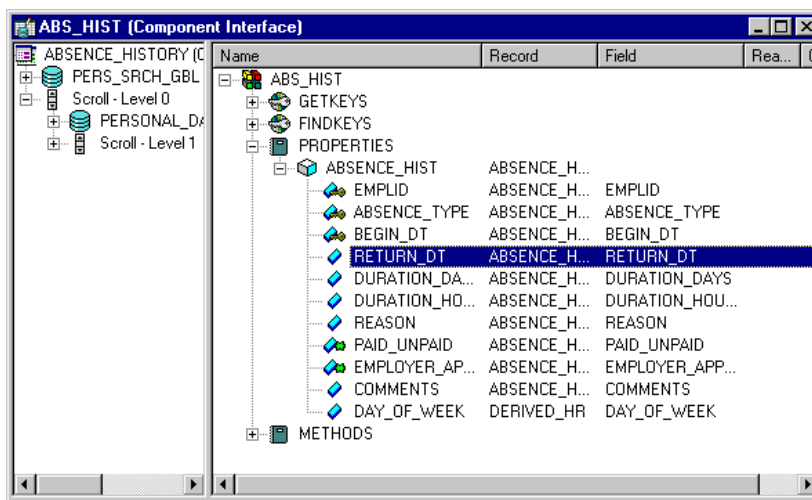


For more information, see Component Interface Class Properties.

Accessing User Defined Component Interface Properties

Every user defined property in a Component Interface **definition** can be used like a property on the **object** instantiated from that Component Interface at runtime.

For example, the following Component Interface definition has RETURN_DT defined as one of its properties.



RETURN_DT Component Interface property highlighted

At runtime, you can use PeopleCode to assign a value to that property (field), or to access the value of that field.

```
&MYCI.RETURN_DT = "05/05/2000";
```

```
/* OR */
```

```
&DATE = &MYCI.RETURN_DT;
```

Declaring a Component Interface Object

Component Interfaces are declared as type ApiObject. For example,

```
Local ApiObject &MYCI;
```



Component Interface objects can only be declared as Local.

Scope of a Component Interface Object

A Component Interface can be instantiated from PeopleCode, from a Visual Basic program, from COM and C/C++.



For more information, see PeopleSoft Component Interface.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Application Engine PeopleCode, record field PeopleCode, and so on.

If you're instantiating a Component Interface from an online page, after you finish working with the component, you may want to refresh your page. The **Refresh** method, on a rowset object, reloads the rowset (scroll) using the current page keys. This causes the page to be redrawn. `GetLevel0().Refresh()` refreshes the entire page. If you only want a particular scroll to be redrawn, you can refresh just that part.



For more information, see Refresh method.

Considerations using Global and Component Variables with Component Interfaces

There are some important differences in how variables declared as Component or Global work with Component Interfaces.

- A variable declared as Component is only valid for a particular Component Interface. If you Cancel a Component Interface or Get a new one, the values for variables declared as Component will be reinitialized. Component variables are only valid for the Component Interface, *not* the session.
- Variables declared as Global in a Component Interface work the same as variables declared as Component. That is, Global variables are only global to the Component Interface, *not* to the session.

Implementing a Component Interface

After you create your Component Interface definition, you can use PeopleCode to access it. This PeopleCode can be long and complex. Rather than write it directly, you can drag and drop the Component Interface definition from Application Designer's Project View into an open PeopleCode edit pane. Application Designer analyzes the definition and generates initial PeopleCode as a template, which you can modify to suit your purpose.

You can also access your Component Interface using COM. You can automatically generate a Visual Basic template, similar to the PeopleCode template, to get you started.



For more information, see Component Interface.

The following are the usual actions that you perform with a Component Interface:

- Create a new instance of data
- Get an existing instance of data
- Retrieve a list of instances of data

The following procedures cover each of these actions in more detail.

Another standard action is inserting or deleting a row of data (an item). This involves traversing a Component Interface (going from level 0 to level 1, from level 1 to level 2, and so on) and accessing data collections.



For more information, see Traversing a Component Interface and Using Data Collections

You may want to work with effective dated information. There are several properties you can set to allow you to do this.



For more information, see Working with Effective Dated Data.

In addition to these standard actions, you can also look at the structure of a Component Interface.



For more information, see Accessing the Structure of a Component Interface.

To create a new instance of data:

In this example, you are creating a new instance of data for the EXPRESS Component Interface, which is based on the EXPRESS_ISSUE_INV component. The following is the complete code sample: the steps explain each line.

```
Local ApiObject &MYSESSION;

Local ApiObject &MYCI;

&MYSESSION = GetSession();

&MYSESSION.Connect(1, "EXISTING", "", "", 0);
```

```

&MYCI = &MYSESSION.GetCompIntfc (COMPINTFC.EXPRESS) ;

&MYCI.BUSINESS_UNIT = "H01B";

&MYCI.INTERNAL_FLG = "Y";

&MYCI.ORDER_NO = "NEXT";

&MYCI.Create();

&MYCI.CUSTOMER = "John's Chicken Shack";

&MYCI.LOCATION = "H10B6987";

.
.
.

If NOT(&MYCI.Save()) Then

    /* save didn't complete */

    &COLL = &MYSESSION.PSMessages;

    For &I = 1 to &COLL.Count

        &ERROR = &COLL.Item(&I);

        &TEXT = &ERROR.Text;

        /* do error processing */

    End-For;

    &COLL.DeleteAll();

End-if;

```

1. Get a session object.

Before you can get a Component Interface, you have to get a session object. The session controls access to the Component Interface, provides error tracing, allows you to set the runtime environment, and so on.

```

&MYSESSION = GetSession();

&MYSESSION.Connect(1, "EXISTING", "", "", 0);

```

2. Get a Component Interface.

Use the `GetCompIntfc` method with a session object to get the Component Interface. You must specify a Component Interface definition that has already been created. You will receive a runtime error if you specify a Component Interface that doesn't exist.

```

&MYCI = &MYSESSION.GetCompIntfc (COMPINTFC.EXPRESS) ;

```


After you execute the `GetCompIntfc` method, you only have the *structure* of the Component Interface. You haven't populated the Component Interface with data yet.

3. Set the CREATEKEYS.

These key values are required to open a new instance of the data. If the values you specify aren't unique, that is, if an instance of the data already exists in the database with those key values, you will receive a runtime error.

```
&MYCI.BUSINESS_UNIT = "H01B";

&MYCI.INTERNAL_FLG = "Y";

&MYCI.ORDER_NO = "NEXT";
```

4. Create the instance of data for the Component Interface.

After you set the key values, you have to use the `Create` method to populate the Component Interface with the key values you set.

```
&MYCI.Create();
```

This will create an instance of this data. However, it hasn't been saved to the database. You must use the `Save` method before the instance of data is committed to the database.

5. Set the rest of the fields.

Assign values to the other fields.

```
&MYCI.CUSTOMER = "John's Chicken Shack";

&MYCI.LOCATION = "H10B6987";

.

.

.
```

If you have specified `InteractiveMode` as `True`, every time you assign a value or use the `InsertItem` or `DeleteItem` methods, any PeopleCode programs associated with that field (either with record field or the component record field) will fire (`FieldEdit`, `FieldChange`, `RowInsert`, and so on.)

6. Save the data.

When you execute the `Save` method, the new instance of the data will be saved to the database.

```
If NOT (&MYCI.Save()) Then
```

The **Save** method returns a Boolean value: `True` if the save was successful, `False` otherwise. You can use this value to do error checking.

The standard PeopleSoft save business rules (that is, any PeopleCode programs associated with SaveEdit, SavePreChange, WorkFlow, etc.) will be executed. If you didn't specify the Component Interface to run in interactive mode, FieldEdit, FieldChange, and so on, will also run at this time for all fields that had values set.



If you're running a Component Interface from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

7. Check Errors

You can check if there were any errors using the PSMessages property on the session object.

```
If NOT (&MYCI.Save ()) Then

    /* save didn't complete */

    &COLL = &MYSESSION.PSMessages;

    For &I = 1 to &COLL.Count

        &ERROR = &COLL.Item (&I);

        &TEXT = &ERROR.Text;

        /* do error processing */

    End-For;

    &COLL.DeleteAll ();

End-if;
```

If there are multiple errors, all errors will be logged to the PSMessages collection, not just the first occurrence of an error. As you correct each error, you will want to delete it from the PSMessages collection.

The **Text** property of the PSMessage returns the text of the error message. At the end of this text is a contextual string that contains the name of the field that generated the error. The contextual string has the following syntax:

```
{BusinessComponentName.[CollectionName(Row)].[CollectionName(Row)].[CollectionName(Row)]}.PropertyName}
```

For example, if you specified the incorrect format for a date field of the Component Interface named ABS_HIST, the **Text** property would contain the following string:

```
Invalid Date {ABS_HIST.BEGIN_DT} (90), (1)
```

The contextual string (by itself) is available using the **Source** property of the PSMessage.



For more information, see About Error Handling and Source property.



If you've called your Component Interface from an Application Engine program, all errors are also logged in the Application Engine error log tables.

To get an existing instance of data:

In this example, you are getting an existing instance of data for the EMPL_CHKLIST_BC Component Interface, which is based on the EMPLOYEE_CHECKLIST component. The following is the complete code sample: the steps explain each line.

```

Local ApiObject &MYSESSION;

Local ApiObject &MYCI;


&MYSESSION = GetSession();

&MYSESSION.Connect(1, "EXISTING", "", "", 0);

&MYCI = &MYSESSION.GetCompIntfc(COMPINTFC.EMPL_CHKLIST_BC);

&MYCI.EMPLID= "8001";

&MYCI.Get();


/* Get checklist Code */


&CHECKLIST_CD = &MYCI.CHECKLIST_CD;


/* Set Effective date */


&MYCI.EFFDT = "05-01-1990";

.

.

.

If NOT(&MYCI.Save()) Then

```

```

/* save didn't complete */

&COLL = &MYSESSION.PSMessages;

For &I = 1 to &COLL.Count

    &ERROR = &COLL.Item(&I);

    &TEXT = &ERROR.Text;

    /* do error processing */

End-For;

&COLL.DeleteAll();

End-if;

```

8. Get a session object.

Before you can get a Component Interface, you have to get a session object. The session controls access to the Component Interface, provides error tracing, allows you to set the runtime environment, etc.

```

&MYSESSION = GetSession();

&MYSESSION.Connect(1, "EXISTING", "", "", 0);

```

9. Get a Component Interface.

Use the `GetCompIntfc` method with a session object to get the Component Interface. You must specify a Component Interface definition that has already been created. You will receive a runtime error if you specify a Component Interface that doesn't exist.

```

&MYCI = &MYSESSION.GetCompIntfc(COMPINTFC.EMPL_CHKLIST_BC);

```

After you execute the `GetCompIntfc` method, you only have the **structure** of the Component Interface. You haven't populated the Component Interface with data yet.

10. Set the GETKEYS

These are the key values required to return a unique instance of existing data. If the keys you specify allow for more than one instance of the data to be returned, or if no instance of the data matching the key values is found, you will receive a runtime error.

```

&MYCI.EMPLID = "8001";

```

11. Get the instance of data for the Component Interface.

After you set the key values, you have to use the `Get` method.

```

&MYCI.Get();

```

This will populate the Component Interface with data, based on the key values you set.

12. Get field values or set field values

At this point, you can either get values or set values, depending on what you want to do.

```
&CHECKLIST_CD = &MYCI.CHECKLIST_CD;
```

```
/* OR */
```

```
&MYCI.EFFDT = "05-01-1990";
```

If you have specified `InteractiveMode` as `True`, every time you assign a value, any PeopleCode programs associated with that field (either with record field or the component record field) will fire (`FieldEdit`, `FieldChange`, and so on.)

13. Save or Cancel the Component Interface, as appropriate

If you've changed values and you want to save your changes to the database, you must use the **Save** method.

```
If NOT (&MYCI.Save ()) Then
```

The **Save** method returns a boolean value: `True` if the save was successful, `False` otherwise. You can use this value to do error checking.

The standard PeopleSoft save business rules (that is, any PeopleCode programs associated with `SaveEdit`, `SavePreChange`, `WorkFlow`, etc.) will be executed. If you didn't specify the Component Interface to run in interactive mode, `FieldEdit`, `FieldChange`, and so on, will also run at this time.



If you're running a Component Interface from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a `COMMIT`.

If you don't want to save any changes to the data, you can use the **Cancel** method. The Component Interface will be reset to the state it was in just after you used the `GetCompIntfc` method.

```
&MYCI.Cancel ();
```

This is similar to a user pressing `ESC` while in a component, and choosing to not save any of their changes.

14. Check Errors

You can check if there were any errors using the `PSMessages` property on the session object.

```
If NOT (&MYCI.Save ()) Then
```

```
/* save didn't complete */
```

```

&COLL = &MYSESSION.PSMessages;

For &I = 1 to &COLL.Count

    &ERROR = &COLL.Item(&I);

    &TEXT = &ERROR.Text;

    /* do error processing */

End-For;

&COLL.DeleteAll();

End-if;

```

If there are multiple errors, all errors will be logged to the PSMessages collection, not just the first occurrence of an error. As you correct each error, you will want to delete it from the PSMessages collection.

The **Text** property of the PSMMessage returns the text of the error message. At the end of this text is a contextual string that contains the name of the field that generated the error. The contextual string has the following syntax:

```
{BusinessComponentName.[CollectionName(Row).[CollectionName(Row).[CollectionName(Row)]]].PropertyName}
```

For example, if you specified the incorrect format for a date field of the Component Interface named ABS_HIST, the **Text** property would contain the following string:

```
Invalid Date {ABS_HIST.BEGIN_DT} (90), (1)
```

The contextual string (by itself) is available using the **Source** property of the PSMMessage.



For more information, see About Error Handling and Source property.



If you've called your Component Interface from an Application Engine program, all errors are also logged in the Application Engine error log tables.

To retrieve a list of instances of data:

In this example, you are getting a list of existing instances of data for the EMPL_CHKLIST_CI Component Interface, which is based on the EMPLOYEE_CHECKLIST component. The following is the complete code sample: the steps break it up and explain each line.

```

Local ApiObject &MYSESSION;

Local ApiObject &MYCI;

```

```

Local ApiObject &MYNEWCI;

&MYSESSION = GetSession();

&MYSESSION.Connect(1, "EXISTING", "", "", 0);

&MYCI = &MYSESSION.GetCompIntfc(COMPINTFC.EMPL_CHKLIST_CI);

&MYCI.EMPLID= "8";

&MYCI.LAST_NAME_SRCH = "S";

&MYLIST = &MYCI.Find();

For &I = 1 to &MYLIST.Count;

    /* note: do not reuse the CI used to create the list, or the list will be
    destroyed */

    &MYNEWCI = &MYLIST.Item(&I);

    /* CI from list still must be instantiated to use it */

    &MYNEWCI.Get();

    /* do some processing */

End-For;

```

15. Get a session object.

Before you can get a Component Interface, you have to get a session object. The session controls access to the Component Interface, provides error tracing, allows you to set the runtime environment, and so on.

```

&MYSESSION = GetSession();

&MYSESSION.Connect(1, "EXISTING", "", "", 0);

```

16. Get a Component Interface.

Use the `GetCompIntfc` method with a session object to get the Component Interface. You must specify a Component Interface definition that has already been created. You will receive a runtime error if you specify a Component Interface that doesn't exist.

```
&MYCI = &MYSESSION.GetCompIntfc (COMPINTFC.EMPL_CHKLIST_CI);
```

After you execute the `GetCompIntfc` method, you only have the **structure** of the Component Interface. You haven't populated the Component Interface with data yet.

17. Set the FINDKEYS values

These can be alternate key values or partial key values. If no instance of the data matching the key values is found, you will receive a runtime error.

```
&MYCI.EMPLID = "8";
```

```
&MYCI.LAST_NAME_SRCH = "S";
```

18. Get a list of data instances for the Component Interface

After you set the alternate or partial key values, use the `Find` method to return a list of data instances for the Component Interface.

```
&MYLIST = &MYCI.Find();
```

This is analogous to the user filling in a partial key value in a search dialog box and pressing ENTER:

Employee Checklist

Find an Existing Value

Search By:

EmpID:

[Advanced Search](#)

Search Results

View All First 1-8 of 8 Last

EmpID	Name	Last Name
8101	Penrose, Steven	PENROSE
8102	Sullivan, Theresa	SULLIVAN
8105	DeHaven, Joanne	DEHAVEN
8113	Frumman, Wolfgang	FRUMMAN
8120	Jones, Theresa	JONES
8121	Gregory, Jan	GREGORY
8146	Inman, Joanne	INMAN
8154	Peck, Jan	PECK

Search results with partial key value

19. Select an instance of the data

To select an instance of the data, you can use any of the following standard data collection methods:

- First()
- Item(*number*)
- Next()



For more information see Data Collection Methods

For example, you could loop through every instance of the data to do some processing:

```
For &I = 1 to &MYLIST.Count;

    /* note: do not reuse the Component Interface used to */

    /* create the list here, or the list will be destroyed */

    &MYNEWCI = &MYLIST.Item(&I);

    /* CI from list must still be instantiated to use it */

    &MYNEWCI.Get();

    /* do some processing */

End-For;
```

After you have a specific instance of the data, you can get values, set values, and so on.

Component Interface Methods and Timeouts

The file **pstools.properties** controls when a Component Interface method timeouts. If you're having problems with methods timing out, you may want to change the values in this file. This file is located in the default directory for the application you're running. If this file isn't in this directory, copy it into the directory before you make your changes to it.

Traversing a Component Interface and Using Data Collections

The data in a Component Interface can be contained in a hierarchy: like the page it's built on, there may be data at level 0, level 1, level 2, and so on.

Each level of data in a Component Interface is known as a collection, such as:

```
Level 0

    -- Level 1 (Collection)

        -- Level 2 (Collection)
```

A *collection* is a set of similar things, like a group of already existing Component Interfaces. Most collections have the same standard properties and methods, with some additional ones specific to that collection. For example, all collections have the property **Count**, which tells you how many items are in that collection, but only a data collection has the method **ItemByKey**.

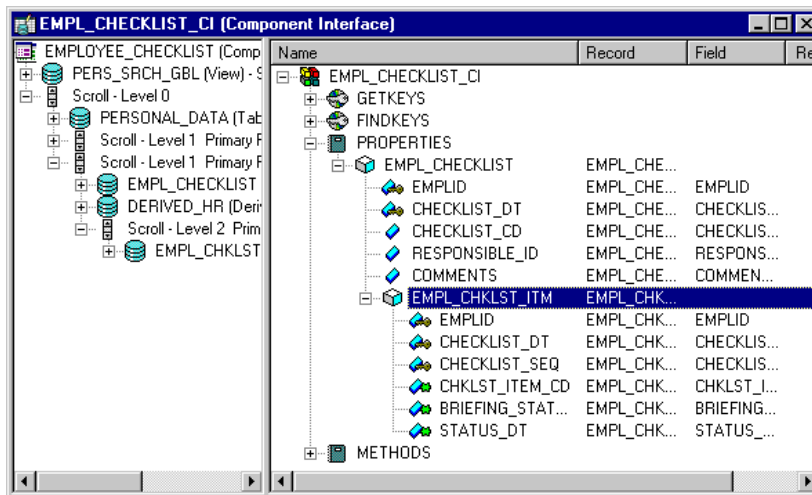
A *data collection* is the collection of data, available at runtime or during test mode, associated with a particular scroll (or record.) The data collection object returns information about every **row of data** (item) that is returned for that record at runtime.

To access the level 2 collection, in general, you could use the following:

```
&Level_1 = &CI.Level_1;

&Level_1_Item = &Level_1.Item(ItemNumber);

&Level_2 = &Level_1_Item.Level_2;
```



Sample Component Interface with Collection Highlighted

The above example shows a Component Interface with a two-level hierarchy, that is, *two* data collections: EMPL_CHECKLIST and EMPL_CHKLIST_ITM. To get a data collection for the first level, use the following code:

```
&Level1 = &MYCI.EMPL_CHECKLIST;
```

To access the secondary scroll (EMPL_CHKLIST_ITM) you have to get the appropriate row (item) of the upper level scroll first:

```
&Level1 = &MYCI.EMPL_CHECKLIST;

&Item = &Level1.Item(1);

&Level2 = &Item.EMPL_CHKLIST_ITM;
```



Scrolls represent a hierarchical order. You must get the first level row that contains the secondary scroll *before* you can access the secondary scroll. For more information see Data Buffer Access.

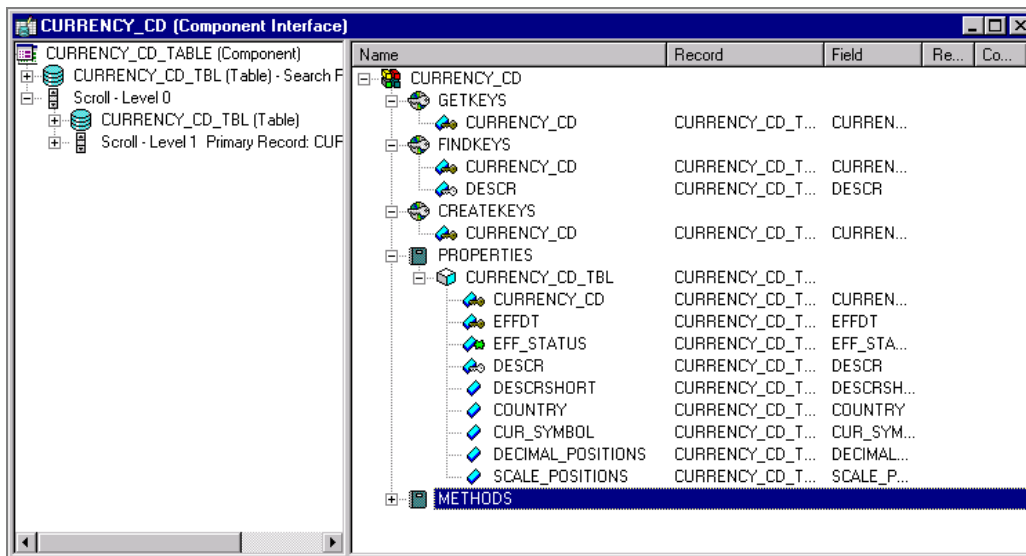
Every data collection has a set of methods and standard properties.



For more information, see Data Collection Methods and Data Collection Properties.

You can rename a collection in a Component Interface. For example, suppose you had the same record at level 0 and at level 1. You may want to rename the level 1 data collection to reflect this. The data in a data collection is associated with the primary database record of a scroll, *not* with the name you supply.

Here is an example of using a Component Interface that has the same record at level 0 and level 1. The Component Interface is based on the CURRENCY_CD_TBL component.



Example of CI with same record at level 0 and level 1

The following code example is based on this Component Interface and does the following:

1. Gets an existing Component Interface.
2. Finds the current effective dated item index.
3. Inserts a new row before the current effective dated item using the InsertItem method.

```
Local ApiObject &Session;

Local ApiObject &CURRENCY_CD;
```

```
Local ApiObject &CURRENCY_CD_TBLCol;

Local ApiObject &CURRENCY_CD_TBLItm;

Local ApiObject &PSMessages;

Local string &ErrorText, &ErrorType;

Local number &ErrorCount;

Local boolean &Error;

Function CheckErrorCodes()

    &PSMessages = &Session.PSMessages;

    &ErrorCount = &PSMessages.Count;

    For &i = 1 To &ErrorCount

        &ErrorText = &PSMessages.Item(&i).Text;

        &ErrorType = &PSMessages.Item(&i).Type;

    End-For;

End-Function;

/* Initialize Flags */

&Error = False;

/* Get Session and Connect */

&Session = GetSession();

&Connect = &Session.Connect(1, "EXISTING", "", "", 0);

If Not &Connect Then

    CheckErrorCodes();

    /* Application Specific Error Processing */
```

Else

```
&CURRENCY_CD = &Session.GetCompIntfc(CompIntfc.CURRENCY_CD);
```

```
If &CURRENCY_CD = Null Then
```

```
    CheckErrorCodes();
```

```
    /* Application Specific Error Processing */
```

Else

```
    /* Set Component Interface Get Keys */
```

```
&CURRENCY_CD.CURRENCY_CD = "USD";
```

```
If Not &CURRENCY_CD.Get() Then
```

```
    CheckErrorCodes();
```

```
    /* Application Specific Error Processing */
```

```
    &Error = True;
```

```
End-If;
```

```
If Not &Error Then
```

```
    &CURRENCY_CD_TBLCol = &CURRENCY_CD.CURRENCY_CD_TBL;
```

```
    &CURRENCY_CD_TBLItm =  
&CURRENCY_CD_TBLCol.InsertItem(&CURRENCY_CD_TBLCol.CurrentItemNum());
```

```
    &CURRENCY_CD_TBLItm.EFFDT = %Date;
```

```
    &CURRENCY_CD_TBLItm.EFF_STATUS = "A";
```

```

&CURRENCY_CD_TBLItm.DESCR = "NewCurrencyCode";

&CURRENCY_CD_TBLItm.DESCRSHORT = "New";

&CURRENCY_CD_TBLItm.COUNTRY = "USA";

&CURRENCY_CD_TBLItm.CUR_SYMBOL = "?";

&CURRENCY_CD_TBLItm.DECIMAL_POSITIONS = 4;

&CURRENCY_CD_TBLItm.SCALE_POSITIONS = 0;

/* Save Instance of Component Interface */

If Not &CURRENCY_CD.Save() Then

    CheckErrorCodes();

    /* Application Specific Error Processing */

End-If;

End-If;

/* End: Set Component Interface Properties */

/* Cancel Instance of Component Interface */

&CURRENCY_CD.Cancel();

End-If;

End-If;

```

Here's a code example in Visual Basic that does the same thing as the code example above.

```

Private Sub CURRENCY_CD()

    On Error GoTo eMessage

```

```
'***** Set Object References *****

Dim oCISession As Object

Dim oCURRENCY_CD As Object

Dim oCURRENCY_CD_TBL As Object

Dim oCURRENCY_CD_TBItem As Object

'***** Set Connect Parameters *****

strAppSeverPath = "//PSOFT101999:9000"

strOperatorID = "PTDMO"

strPassword = "PTDMO"

'***** Create PeopleSoft Session Object *****

Set oCISession = CreateObject("PeopleSoft.Session")

'***** Connect to the App Sever *****

oCISession.Connect 1, strAppSeverPath, strOperatorID, strPassword, 0

'***** Get the Component *****

Set oCURRENCY_CD = oCISession.GetCompIntfc("CURRENCY_CD")

'***** Set the Component Interface Mode *****

oCURRENCY_CD.InteractiveMode = False

oCURRENCY_CD.GetHistoryItems = True

'***** Set Component Get/Create Keys *****

oCURRENCY_CD.CURRENCY_CD = "USD"

'***** Execute Create *****

oCURRENCY_CD.Get
```

```

        'Set CURRENCY_CD_TBL Collection Field Properties -- Parent: PS_ROOT
Collection

        Set oCURRENCY_CD_TBL = oCURRENCY_CD.CURRENCY_CD_TBL

        Set oCURRENCY_CD_TBLItem =
oCURRENCY_CD_TBL.InsertItem(oCURRENCY_CD_TBL.CurrentItemNum())

        oCURRENCY_CD_TBLItem.EFFDT = Date

        oCURRENCY_CD_TBLItem.EFF_STATUS = "A"

        oCURRENCY_CD_TBLItem.DESCR = "NewCurrencyCode"

        oCURRENCY_CD_TBLItem.DESCRSHORT = "New"

        oCURRENCY_CD_TBLItem.COUNTRY = "USA"

        oCURRENCY_CD_TBLItem.CUR_SYMBOL = "?"

        oCURRENCY_CD_TBLItem.DECIMAL_POSITIONS = 4

        oCURRENCY_CD_TBLItem.SCALE_POSITIONS = 0

        '***** Save Component Interface *****

        oCURRENCY_CD.Save

        oCURRENCY_CD.Cancel

Exit Sub

eMessage:

        '***** Display VB Runtime Errors *****

        MsgBox Err.Description

        '***** Display PeopleSoft Error Messages *****

        If oCISession.PSMessages.Count > 0 Then

                For i = 1 To oCISession.PSMessages.Count

                        MsgBox oCISession.PSMessages.Item(i).Text

```



```

                Next i

            End If

```

```

End Sub

```

```

Sub MAIN()

    CURRENCY_CD

End Sub

```

A data collection represents a row of data. One of the common actions you'll want to do is to insert or delete a row of data.

To insert or delete a row of data:

In this example, you are getting a existing instance of data for the BUS_EXP Component Interface, which is based on the BUSINESS_EXPENSES component, then inserting (or deleting) a row of data in the second level scroll. The following is the complete code sample: the steps explain each line.

```

Local ApiObject &MYSESSION;

Local ApiObject &MYCI;

&MYSESSION = GetSession();

&MYSESSION.Connect(1, "EXISTING", "", "", 0);

&MYCI = &MYSESSION.GetCompIntfc(COMPINTFC.BUS_EXP);

&MYCI.EMPLID= "8001";

&MYCI.Get();

&MYLEVEL1 = &MYCI.BUS_EXPENSE_PER;

/* get appropriate item in lvl1 collection */

For &I = 1 to &MYLEVEL1.Count

    &ITEM = &MYLEVEL1.Item(&I);

    &MYLEVEL2 = &ITEM.BUS_EXPENSE_DTL;

    &COUNT = &MYLEVEL2.Count

/* get appropriate item in lvl2 collection */

```

```
For &J = 1 to &COUNT

    &LVL2ITEM = &MYLEVEL2.Item(&J);

    &CIDATE = &LVL2ITEM.CHARGE_DT;

    If &CIDATE <> %Date Then

        /* insert row */

        &NEWITEM = &MYLEVEL2.InsertItem(&COUNT);

        /* set values for &NEWITEM */

        Else

            /* delete last row */

            &MYLEVEL2.DeleteItem(&COUNT);

            End-If;

        End-For;

    End-For;

If NOT(&MYCI.Save()) Then

    /* save didn't complete */

    &COLL = &MYSESSION.PSMessages;

    For &I = 1 to &COLL.Count

        &ERROR = &COLL.Item(&I);

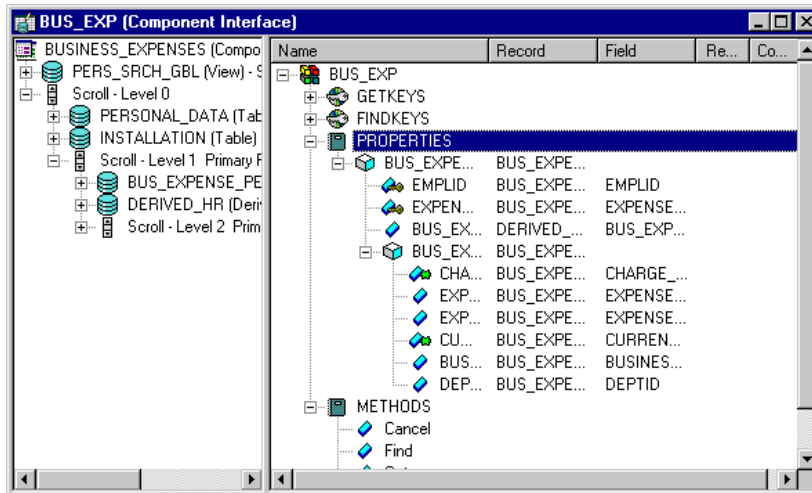
        &TEXT = &ERROR.Text;

        /* do error processing */

    End-For;

    &COLL.DeleteAll();

End-if;
```



BUS_EXP Component Interface definition

1. Get a session object.

Before you can get a Component Interface, you have to get a session object. The session controls access to the Component Interface, provides error tracing, allows you to set the runtime environment, and so on.

```
&MYSESSION = GetSession();

&MYSESSION.Connect(1, "EXISTING", "", "", 0);
```

2. Get a Component Interface.

Use the GetCompIntfc method with a session object to get the Component Interface. You must specify a Component Interface definition that has already been created. You will receive a runtime error if you specify a Component Interface that doesn't exist.

```
&MYCI = &MYSESSION.GetCompIntfc(COMPINTFC.BUS_EXP);
```

After you execute the GetCompIntfc method, you only have the **structure** of the Component Interface. You haven't populated the Component Interface with data yet.

3. Set the GETKEYS

These are the key values required to return a unique instance of existing data. If the keys you specify allow for more than one instance of the data to be returned, or if no instance of the data matching the key values is found, you will receive a runtime error.

```
&MYCI.EMPLID = "8001";
```

4. Get the instance of data for the Component Interface.

After you set the key values, you have to use the Get method.

```
&MYCI.Get();
```

This will populate the Component Interface with data, based on the key values you set.

5. Get the level 1 scroll

The name of the scroll can be treated like a property. It returns a data collection that contains all the level 1 data.

```
&MYLEVEL1 = &MYCI.BUS_EXPENSE_PER
```

6. Get the appropriate item in the level 1 data collection

Remember, scroll data is hierarchical. You have to get the appropriate level 1 row before you can access the level 2 data.

```
For &I = 1 to &MYLEVEL1.Count
    &ITEM = &MYLEVEL1.Item(&I);
```

7. Get the level 2 scroll

This is done the same way as you accessed the level 1 scroll, by using the scroll name as a property.

```
&MYLEVEL2 = &ITEM.BUS_EXPENSE_DTL;
```

8. Get the appropriate item in the level 2 data collection

A data collection is made up of a series of items (rows of data.) You have to access the appropriate item (row) in this level as well.

```
&COUNT = &MYLEVEL2.Count

/* get appropriate item in lvl2 collection */

For &J = 1 to &COUNT
    &LVL2ITEM = &MYLEVEL2.Item(&J);
```

9. Insert or delete a row of data

You can insert or delete a row of data from a data collection. The following example finds the last item (row of data) in the second level scroll. If it matches the value of %Date, the last item is deleted. If it doesn't match, a new row is inserted.

```
&CIDATE = &LVL2ITEM.CHARGE_DT;

If &CIDATE <> %Date Then

    /* insert row */

    &NEWITEM = &MYLEVEL2.InsertItem(&COUNT);

    /* set values for &NEWITEM */

Else
```

```

/* delete last row */

&MYLEVEL2.DeleteItem(&COUNT);

End-If;

```

10. Save the data.

When you execute the Save method, the new instance of the data will be saved to the database.

```
If NOT (&MYCI.Save()) Then
```

The **Save** method returns a boolean value: True if the save was successful, False otherwise. You can use this value to do error checking.

The standard PeopleSoft save business rules (that is, any PeopleCode programs associated with SaveEdit, SavePreChange, WorkFlow, and so on) will be executed. If you did not specify the Component Interface to run in interactive mode, FieldEdit, FieldChange, and so on, will also run at this time.



If you're running a Component Interface from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

11. Check Errors

You can check if there were any errors using the PSMessages property on the session object.

```

If NOT (&MYCI.Save()) Then

/* save didn't complete */

&COLL = &SESSION.PSMessages;

For &I = 1 to &COLL.Count

    &ERROR = &COLL.Item(&I);

    &TEXT = &ERROR.Text;

/* do error processing */

End-For;

&COLL.DeleteAll();

End-if;

```

If there are multiple errors, all errors will be logged to the PSMessages collection, not just the first occurrence of an error. As you correct each error, you will want to delete it from the PSMessages collection.

The **Text** property of the PSMMessage returns the text of the error message. At the end of this text is a contextual string that contains the name of the field that generated the error. The contextual string has the following syntax:

```
{BusinessComponentName.[CollectionName(Row).[CollectionName(Row).[CollectionName(Row)]]].PropertyName}
```

For example, if you specified the incorrect format for a date field of the Component Interface named ABS_HIST, the **Text** property would contain the following string:

```
Invalid Date {ABS_HIST.BEGIN_DT} (90), (1)
```

The contextual string (by itself) is available using the **Source** property of the PSMMessage.



For more information, see About Error Handling and Source property.



If you've called your Component Interface from an Application Engine program, all errors are also logged in the Application Engine error log tables.

Working with Effective Dated Data

When you work with effective dated data, you will want to use some combination of the following methods:

- CurrentItem
- GetEffectiveItem

Both these methods return an item, however, there are some differences:

- The CurrentItem method takes no parameters, so it bases the item it returns on the **current system date**.
- The GetEffectiveItem method bases the item it returns on a **user-provided date**. They're all doing the same logic, just using a different date.



In order to show all history for an effective dated collection, you must set GetHistoryItems to *True* before you populate the Component Interface.

How a Developer would Use GetEffectiveItem

If a user is making an update to an effective-dated record, they don't always want to insert a row at the end.

Suppose the database contained the following data:

EMPLID	EFFDT	SEQNO
8000	3/1/99	0
8000	7/1/99	0
8000	9/1/99	0
8000	12/1/99	0

Now suppose your user wants to enter info with EFFDT of 11/1/99. If they were looking at a PeopleSoft component, they would visually scan to see where that date falls and then press ALT+7 and ENTER at the row that they want to insert after.

The `GetEffectiveItem` gives you the ability to pass in the correct effective date, instead of having to loop through every item in the collection doing a comparison, looking for the correct item.

Why can't it just go at the end?

The `InsertItem` method simulates pressing ALT+7 and ENTER online to insert a row in a scroll. Part of the logic that occurs in the Component Processor is that if the scroll is effective dated, the ALT+7 and ENTER carries the values forward from the previous row. This functionality is still there if you use `InsertItem()` at the end of the collection, but the values may be incorrect.

Reusing Existing Code

One of the advantages of using a Component Interface is that it allows you to reuse existing PeopleCode and business logic. However, a Component Interface isn't exactly equivalent to a component, which means there are a few key areas in which you should expect differences in behavior between a Component Interface and the component it's based on. These differences are discussed below.

Differences in Search Dialog Processing

When you run a Component Interface, the **SearchInit**, **SearchSave**, and **RowSelect** events don't fire. This means that any PeopleCode associated with these events won't run. The first event to run is **RowInit**.

Differences in PeopleCode Event and Function Behavior

PeopleCode events and functions that relate exclusively to GUI and online processing can't be used by Component Interfaces. These include:

- Menu PeopleCode and pop-up menus. The **ItemSelected** and **PrePopup** PeopleCode events are not supported. In addition, the **CheckMenuItem**, **DisableMenuItem**, **EnableMenuItem**, **HideMenuItem**, and **UnCheckMenuItem** functions aren't supported.
- Transfers between components, including modal transfers. The **DoModal**, **EndModal**,

IsModal, **Transfer**, **TransferPage**, **DoModalComponent**, and **IsModalComponent** functions cannot be used.

- Dynamic tree controls. Functions related to this control, such as **GetSelectedTreeNode**, **GetTreeNodeParent**, **GetTreeRecordName**, **RefreshTree** and **TreeDetailInNode** cannot be used.
- ActiveX controls. The **PSControlInit** and **PSLostFocus** events aren't supported, and the **GetControl** and **GetControlOccurance** functions cannot be used.
- Cursor position. **SetControlValue** and **SetCursorPos** cannot be used.
- **WinMessage** cannot be used.
- Save in the middle of a transaction. **DoSave** can't be used.

For the unsupported functions, you should put a condition around them, testing whether there's an existing Component Interface or not.

```

If %ComponentName Then

    /* process is being called from a Component Interface */

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;

```

Component Interface Reference

The following sections go into more detail of the properties and methods that can be used with a Component Interface.

Session Object Methods

Component Interfaces don't have any built-in functions. They are instantiated from a session object.

FindCompIntfc

Syntax

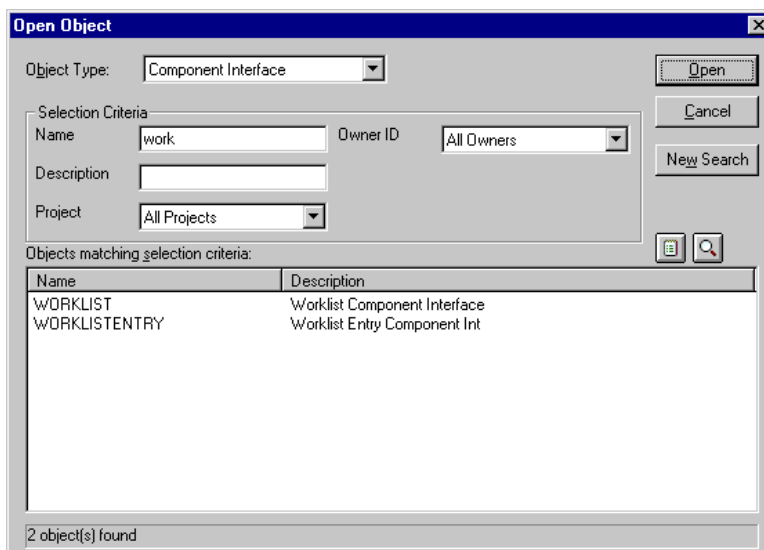
```
FindCompIntfc ( [ComponentName] )
```


Description

The **FindCompIntfc** method, used with the session object, returns a reference to a Component Interface collection, filled with zero or more Component Interfaces. The *ComponentName* parameter takes a string value. You can use a partial value to limit the set of Component Interfaces returned. You can also specify a null value, that is, two quotation marks with no space between them, (""), to return the entire list of Component Interfaces available.

Using the **FindCompIntfc** method is equivalent to using **File, Open**, and selecting **Component Interface** in Application Designer.

In the following example, a partial key was used to open all components starting with "WORK". The collection that is returned has two items in it: WORKLIST and WORKLISTENTRY.



Open dialog using partial keys

Example

In the following example, a partial key was used to open all components starting with "APP".

```
Local ApiObject &MYSESSION;

Local ApiObject &MYCI;

&MYSESSION = GetSession();

&MYSESSION.Connect(1, "EXISTING", "", "", 0);

&MYCICOLL = &MYSESSION.FindCompIntfc("WORK");

&MYCI = &MYCICOLL.First();
```

GetCompIntfc

Syntax

```
GetCompIntfc ( [COMPINTFC.] ComponentName)
```

Description

The **GetCompIntfc** method, used with the session object, returns a reference to a Component Interface. *ComponentName* when used by itself takes a string value. If you use `COMPINTFC.componentname` it isn't a string value: it's a constant that automatically is renamed in your code if the Component Interface definition is ever renamed. You must specify an existing Component Interface, otherwise you will receive a runtime error.

Example

```
Local ApiObject &MYSESSION;

Local ApiObject &MYCI;

&MYSESSION = GetSession();

&MYSESSION.Connect(1, "EXISTING", "", "", 0);

&MYCI = &MYSESSION.GetCompIntfc(COMPINTFC.PERSONAL_DATA_BC);
```

Component Interface Collection Methods

A Component Interface collection is a list of the available Component Interfaces. An equivalent list is generated by starting Application Designer, selecting **File, Open, Component Interface**.

You get a Component Interface collection by using the `FindCompIntfc` method with a session object.

First

Syntax

```
First()
```

Description

The **First** method returns the first Component Interface in the Component Interface collection object executing the method. This returns the structure of the Component Interface with only the key fields have been filled in. The rest of the data is not present. To populate the Component Interface with data, you must set the key values and use the **Get** method.

Example

```
&MYCI = &CICOLLECTION.First();
```

Item

Syntax

```
Item(number)
```

Description

The **Item** returns the Component Interface that exists at the *number* position in the Component Interface collection executing the method. This returns the structure of the Component Interface with only the key fields filled in. The rest of the data is not present. To populate the Component Interface with data, you must set the key values and use the **Get** method. *number* takes a number value.

Example

```
For &I = 1 to &COLLECTION.Count;  
  
    &MYCI = &COLLECTION.Item(&I);  
  
    /* do processing */  
  
End-For;
```

Next

Syntax

```
Next()
```

Description

The **Next** method returns the next Component Interface in the Component Interface collection object executing the method. You can only use this method after you have used either the **First** or **Item** methods; otherwise the system doesn't know where to start. This returns the structure of the Component Interface with only the key fields filled in. To populate the Component Interface with data, you must set the key values and use the **Get** method.

Example

```
&MYCI = &COLLECTION.Next();
```

Component Interface Collection Property

Count

This property returns the number of Component Interfaces in the Component Interface collection, as a number.

This property is read-only.

Example

```
&COUNT = &MYCI_COLLECTION.Count;
```

Component Interface Class Methods

Cancel

Syntax

```
Cancel()
```

Description

The **Cancel** method cancels the instance of the Component Interface object executing the method, rolling back any changes that were made. This will set the Component Interface state to the same state it was in immediately after it was created by **GetCompIntfc**. This closes the Component Interface.

Parameters

None.

Returns

A Boolean value: True if component was successfully cancelled, False otherwise.

CopyRowset

Syntax

```
CopyRowset (&Rowset [, InitialRow] [, record_list]);
```

Where *record_list* is a list of record names in the form:

```
[RECORD.source_rename1, RECORD.target_rename1  
[, RECORD.source_rename2, RECORD.target_rename2]] . . .
```

Description

The **CopyRowset** method copies the specified rowset object to the Component Interface executing the method, copying *like-named* record fields and data collections (child rowsets) at corresponding levels. If pairs of source and destination record names are given, these are used to pair up the records and data collections *before* checking for like-named record fields.

CopyRowset uses the Page Field order when copying properties. This helps ensure that dependent fields are set in the required order.



This method works in PeopleCode only.



This method uses the names of the records in the collection, *not* the name you may give the collection when you create the Component Interface.

If there are blanks in *source* rowset or record, they will only be copied over to the Component Interface if the field's **IsChanged** property is set to True. Otherwise blanks are *not* copied.

You could use this method when you were using a Component Interface to verify the data in your application message.



The structure of the rowset you're copying data from must exactly match the existing rowset structure, with the same records at level 0, 1, 2, and so on.



CopyRowset is intended to be used with a subscription process, that is, with message data. As all subscriptions only work in two-tier mode, **CopyRowset** only works in two-tier mode.

Parameters

&Rowset

Specify an existing, instantiated rowset object that contains data.

InitialRow

Specify the initial transaction row to begin with. This parameter provides a quick way to loop through a rowset that has multiple level 0 rows. The default value for this parameter is 1.

record_list

Specify a list of source and target record names. All *source_renames* are records being copied *from*, in the rowset object being copied from. All *target_renames* are records being copied *to*, in the Component Interface being copied to.

Returns

None.

Example

The following example would be in your Subscription PeopleCode. The Exit(1) causes all changes to be rolled back, and the message will be marked with the status ERROR so you can correct it.

```

Local message &MSG;

Local ApiObject &SESSION;

Local ApiObject &PO;

Local rowset &ROWSET;


&MSG = GetMessage();

&ROWSET = &MSG.GetRowset();


&SESSION = GetSession();

&RSLT = &SESSION.Connect(1, "EXISTING", "", "", 0);


If &RSLT Then

    /* Session connected correctly */

    /* Set key values to create component */

    &PO = &SESSION.GetCompIntfc (COMPINTFC.PO);

    &PO.BU = &MSG(&I).PO_HDR.BU.Value;

    &PO.PO_ID = &MSG(&I).PO_HDR.PO_ID.Value;

    &PO.Create();


    &PO.CopyRowset (&ROWSET);

    If NOT (&PO.Save()) Then

        Exit(1);

    End-if;

Else
```

```

        /* do error processing */

End-If;

```

The following example loops through all the transactions of a message rowset.

```

Local message &MSG;

Local ApiObject &SESSION;

Local ApiObject &CI;

Local rowset &ROWSET;

&MSG = GetMessage();

&ROWSET = &MSG.GetRowset();

&SESSION = GetSession();

If (&SESSION.Connect(1, "EXISTING", "", "", 0)) Then

    &CI = &SESSION.GetCompIntfc(CompIntfc.VOL_ORG);

    &I = 0;

    While (&I < &ROWSET.RowCount)

        &I = &I + 1;

        &CI.VOLUNTEER_ORG = &ROWSET.GetRow(&I).VOLNTER_ORG_2.
VOLUNTEER_ORG.Value;

        If &CI.Create() Then

            If &CI.CopyRowset(&ROWSET, &I, Record.VOLNTER_ORG_TBL,
Record.VOLNTER_ORG_TBL) Then

                /* App specific processing */

                If Not &CI.Save() Then

                    Winmessage("Save Failed");

                    /* Other app specific processing */

                End-If;
            End-If;
        End-If;
    End-While;
End-If;

```

```

                End-If;

            End-If;

            &CI.Cancel();

        End-While;

    End-If;

```

CopyRowsetDelta

Syntax

```
CopyRowsetDelta(&Rowset [, InitialRow] [, record_list]);
```

Where *record_list* is a list of record names in the form:

```

[RECORD.source_recname1, RECORD.target_recname1
  [, RECORD.source_recname2, RECORD.target_recname2]] . . .

```

Description

The **CopyRowsetDelta** method copies the *changed* rows in the specified rowset object to the Component Interface executing the method, copying *like-named* record fields and data collections (child rowsets) at corresponding levels. If pairs of source and destination record names are given, these are used to pair up the records and data collections *before* checking for like-named record fields.



This method works in PeopleCode only.



This method uses the names of the records in the collection, *not* the name you give the collection when you create the Component Interface.

If there are blanks in *source* rowset or record, they will only be copied over to the Component Interface if the field's **IsChanged** property is set to True. Otherwise blanks are *not* copied.

You will generally use this method with a Component Interface to verify data from a message.



CopyRowsetDelta is intended to be used with a subscription process, that is, with message data. As all subscriptions only work in two-tier mode, **CopyRowsetDelta** only works in two-tier mode.

If the rowset you're copying from is a message rowset, the **CopyRowsetDelta** method uses the AUDIT_ACTN field in the PSCAMA table in the message to know whether the row is to be inserted, updated, or deleted inside the Component Interface.



For more information, see PSCAMA Table.

If the rowset you're copying from *is not* a message rowset, that rowset must have the same structure as a message, that is, it must have a PSCAMA record with an AUDIT_ACTN field on every level of the rowset.

Warning! CopyRowsetDelta uses a record's keys to locate the target row to change for all audit actions other than **Add**. CopyRowsetDelta actions (other than Add) therefore only work on rowsets that have keys *that uniquely identify all rows in the rowset*. Rowsets that do **not** distinguish between rows using a key field will be updated in an unpredictable fashion.

Parameters

<i>&Rowset</i>	Specify an existing, instantiated rowset object that contains data.
<i>InitialRow</i>	Specify the initial transaction row to begin with. This parameter provides a quick way to loop through a rowset that has multiple level 0 rows. The default value for this parameter is 1.
<i>record_list</i>	Specify a list of source and target record names. All <i>source_renames</i> are records being copied <i>from</i> , in the rowset object being copied from. All <i>target_renames</i> are records being copied <i>to</i> , in the Component Interface being copied to.

Returns

None.

Example

The following PeopleCode would exist in your subscription process. The Exit(1) causes all changes to be rolled back, and the message will be marked with the status ERROR so you can correct it.

```
Local Message &MSG;

Local ApiObject &SESSION;

Local ApiObject &PO;

Local Rowset &ROWSET;
```

```

&MSG = GetMessage();

&ROWSET = &MSG.GetRowset();

&SESSION = GetSession();

&RSLT = &SESSION.Connect(1, "EXISTING", "", "", 0);

If &RSLT Then

    /* Session connected correctly */

    /* Set key values to get component */

    &PO = &SESSION.GetCompIntfc(COMPINTFC.PO);

    &PO.BU = &MSG(&I).PO_HDR.BU.Value;

    &PO.PO_ID = &MSG(&I).PO_HDR.PO_ID.Value;

    &PO.Get();

    &PO.CopyRowsetDelta(&ROWSET);

    If NOT (&PO.Save()) Then

        Exit(1);

    End-if;

Else

    /* do error processing */

End-If;

```

CopySetupRowset

Syntax

```
CopySetupRowset(&Rowset [, InitialRow] [, record_list]);
```

Where *record_list* is a list of record names in the form:

```
RECORD.source_recname, RECORD.target_recname
```

Description

The **CopySetupRowset** method is used to copy a setup table application message to a Component Interface. A setup table has the same record at level 0 and level 1, while a

Component Interface has only a single copy of a record. This method copies the contents of the message (at level 0) to the first level collection (level 1) of the Component Interface.

The **CopySetupRowset** method copies **like-named** record fields. If a pair of source and destination record names are given, these are used to pair up the records and data collections *before* checking for like-named record fields.



This method works in PeopleCode only.



This method uses the names of the records in the collection, *not* the name you may give the collection when you create the Component Interface.

If there are blanks in *source* rowset or record, they will only be copied over to the Component Interface if the field's **IsChanged** property is set to True. Otherwise blanks are *not* copied.



CopySetupRowset is intended to be used with a subscription process, that is, with message data. As all subscriptions only work in two-tier mode, **CopySetupRowset** only works in two-tier mode.

Parameters

&Rowset

Specify an existing, instantiated rowset object that contains data.

InitialRow

Specify the initial transaction row to begin with. This parameter provides a quick way to loop through a rowset that has multiple level 0 rows. The default value for this parameter is 1.

record_list

Specify source and target record names. The *source_recname* is the record being copied *from*, in the rowset object being copied from. The *target_recname* is the record being copied *to*, in the Component Interface being copied to.

Returns

None.

Example

The following example would be in your Subscription PeopleCode.

```
Local ApiObject &SESSION;

Local ApiObject &PSMESSAGES;
```

```
Local ApiObject &CI;

Local Message &MSG;

Local Rowset &RS;

&SESSION = GetSession();

&SESSION.connect(1, "Existing", "", "", 0);

&PSMESSAGES = &SESSION.psmessages;

&MSG = GetMessage();

&RS = &MSG.GetRowset();

&CI = &SESSION.getcomponent(Component.VOL);

/** Set Business Component Properties */
&CI.gethistoryitems = True;

/*&CI.InteractiveMode = True;*/

/*&CI.stoponfirsterror = True;*/

For &TRANSACTION = 1 To &RS.RowCount

    &CI.VOLUNTEER_ORG = &RS(&TRANSACTION).VOLNTER_ORG_TBL.VOLUNTEER_ORG.Value;

    /* note: You will achieve much better performance if you add code here to
    check if the keys are the same L0 keys as the last time through the loop and,
    if so, skip the Get/Create section. */

    If Not &CI.create() Then

        &PSMESSAGES.DeleteAll();
```

```
If Not &CI.get() Then

    /** Check Error Messages **/

    For &I = 1 To &PSMESSAGES.count

        &TYPE = &PSMESSAGES.item(&I).type;

        &TEXT = &PSMESSAGES.item(&I).text;

    End-For;

    Exit (1);

End-If;

End-If;


If Not &CI.CopySetupRowset(&RS, &TRANSACTION) Then

    /** Check Error Messages **/

    For &I = 1 To &PSMESSAGES.count

        &TYPE = &PSMESSAGES.item(&I).type;

        &TEXT = &PSMESSAGES.item(&I).text;

    End-For;

    Exit (1);

End-If;


If Not &CI.Save() Then

    /** Check Error Messages **/

    For &I = 1 To &PSMESSAGES.count

        &TYPE = &PSMESSAGES.item(&I).type;

        &TEXT = &PSMESSAGES.item(&I).text;

    End-For;

    Exit (1);

End-If;
```

```

        &CI.Cancel();

    End-For;

```

Related Topics

CopySetupRowsetDelta, CopyRowset

CopySetupRowsetDelta

Syntax

```
CopySetupRowsetDelta(&Rowset [, InitialRow] [, record_list]);
```

Where *record_list* is a list of record names in the form:

```
RECORD.source_recname, RECORD.target_recname
```

Description

The **CopySetupRowsetDelta** method is used to copy a setup table with *changed* rows in an application message to a Component Interface. A setup table has the same record at level 0 and level 1, while a Component Interface has only a single copy of a record. This method copies the contents of the message (at level 0) to the first level collection (level 1) of the Component Interface.



This method works in PeopleCode only.



CopySetupRowsetDelta copies all the like-named fields from the changed *row* into the message. It is *not* copying just the changed fields.

The **CopySetupRowsetDelta** method copies *like-named* record fields, in those rows where the **IsChanged** property is True. If a pair of source and destination record names are given, these are used to pair up the records and data collections *before* checking for like-named record fields.



Note. This method uses the names of the records in the collection, *not* the name you may give the collection when you create the Component Interface.

If there are blanks in *source* rowset or record, they will only be copied over to the Component Interface if the field's **IsChanged** property is set to True. Otherwise blanks are *not* copied.



CopySetupRowsetDelta is intended to be used with a subscription process, that is, with message data. As all subscriptions only work in two-tier mode, **CopySetupRowsetDelta** only works in two-tier mode.

Warning! CopySetupRowsetDelta uses a record's keys to locate the target row to change for all audit actions other than **Add**. CopySetupRowsetDelta actions (other than Add) therefore only work on rowsets that have keys *that uniquely identify all rows in the rowset*. Rowsets that do **not** distinguish between rows using a key field will be updated in an unpredictable fashion.

Parameters

<i>&Rowset</i>	Specify an existing, instantiated rowset object that contains data.
<i>InitialRow</i>	Specify the initial transaction row to begin with. This parameter provides a quick way to loop through a rowset that has multiple level 0 rows. The default value for this parameter is 1.
<i>record_list</i>	Specify source and target record names. The <i>source_recname</i> is the record being copied <i>from</i> , in the rowset object being copied from. The <i>target_recname</i> is the record being copied <i>to</i> , in the Component Interface being copied to.

Returns

None.

Example

```

Local ApiObject &SESSION;

Local ApiObject &PSMESSAGES;


Local ApiObject &CI;


Local Message &MSG;

Local Rowset &RS;


&SESSION = GetSession();

&SESSION.connect(1, "Existing", "", "", 0);

&PSMESSAGES = &SESSION.psmessages;


&MSG = GetMessage();

```

```

&RS = &MSG.GetRowset();

&CI = &SESSION.getcomponent(Component.VOL);

/** Set Business Component Properties **/
&CI.gethistoryitems = True;
/*&CI.InteractiveMode = True;*/
/*&CI.stoponfirsterror = True;*/

For &TRANSACTION = 1 To &RS.RowCount

    &CI.VOLUNTEER_ORG = &RS(&TRANSACTION).VOLNTER_ORG_TBL.VOLUNTEER_ORG.Value;

    /* note: You will achieve much better performance if you add code here to
    check if the keys are the same L0 keys as the last time through the loop and,
    if so, skip the Get/Create section. */

    If Not &CI.create() Then

        &PSMESSAGES.DeleteAll();

        If Not &CI.get() Then

            /** Check Error Messages **/

            For &I = 1 To &PSMESSAGES.count

                &TYPE = &PSMESSAGES.item(&I).type;

                &TEXT = &PSMESSAGES.item(&I).text;

            End-For;

            Exit (1);

        End-If;

    End-If;

```



```

If Not &CI.CopySetupRowsetDelta(&RS, &TRANSACTION) Then

    /** Check Error Messages **/

    For &I = 1 To &PSMESSAGES.count

        &TYPE = &PSMESSAGES.item(&I).type;

        &TEXT = &PSMESSAGES.item(&I).text;

    End-For;

    Exit (1);

End-If;

If Not &CI.Save() Then

    /** Check Error Messages **/

    For &I = 1 To &PSMESSAGES.count

        &TYPE = &PSMESSAGES.item(&I).type;

        &TEXT = &PSMESSAGES.item(&I).text;

    End-For;

    Exit (1);

End-If;

&CI.Cancel();

```

Related Topics

CopySetupRowset, CopyRowsetDelta, IsDelta Message class property

Create

Syntax

```
Create()
```

Description

The **Create** method associates the Component Interface object executing the method with a new, open Component Interface that matches the key values that were set prior to using the Create method. If there are CREATEKEYS values associated with the Component Interface, these are the values you must set. If there are no CREATEKEYS values, you must set the required GETKEYS values instead. All keys required for creating a new Component Interface must be set

before using the Create method, otherwise you will receive a runtime error. If you do not use unique key values, (that is, you try to set the keys to values that already exist in the database) you will receive a runtime error.

Setting the key values prior to using the Create method is analogous to filling in the key values in the Add dialog for a component when you access it in add mode:

Action Concurrent Jon Add dialog box

Parameters

None.

Returns

A Boolean value: True if component was successfully created, False otherwise.

Example

```
&MYCI = &MYSESSION.GetCompIntfc (COMPINTFC.ACTION_REASON);

&MYCI.ACTION = "Additional Job";

&MYCI.ACTION_REASON = "0001";

&MYCI.Create();
```

Find

Syntax

```
Find()
```

Description

The **Find** method returns a reference to a data collection matching the key values that were set prior to using the Find method. This data collection may have zero elements in it if no items matching the key values you set were found. You do not have to set values for all the key values. You can use the same wildcards in your Find that you can use in the search dialog, that is, % for one or more characters in a search pattern, and _ (underscore) for exactly one character. In addition, you can use partial values. For example, the following code will find all the data items where the employee ID starts with an "8":

```
&MYCI.Emplid = "8";
```

```
&MYDC = &MYCI.Find();
```

This is analogous to using a partial key from the search dialog, opening a component.

EmpID	Name	Last Name
8101	Penrose,Steven	PENROSE
8102	Sullivan,Theresa	SULLIVAN
8105	DeHaven,Joanne	DEHAVEN
8113	Frumman,Wolfgang	FRUMMAN
8120	Jones,Theresa	JONES
8121	Gregory,Jan	GREGORY
8146	Inman,Joanne	INMAN
8154	Peck,Jan	PECK

Using a partial key to open a component

After you have a data collection, you can use one of the data collection methods to open the Component Interface.



For more information, see Data Collection Methods.

Parameters

None.

Returns

A collection of Component Interfaces.

Get

Syntax

```
Get ()
```

Description

The **Get** method associates the Component Interface object executing the method with an open Component Interface that matches the key values that were set prior to using the Get method. The key values you must set are the required GETKEYS values for the Component Interface.



If you want to retrieve all the history data for a Component Interface, you must specify the `GetHistoryItems` property as `True` *before* you use the **Get** method. If you want any PeopleCode programs associated with the fields to fire immediately after a value is changed, you must set the `InteractiveMode` property as `True` *before* you use the **Get** method.

After any execution of **Get**, you should check if there are any errors pending on the session object. In some special circumstances (involving failure of previously cached operations failing after the `Get` has executed) `Get` will return `True` even though the component wasn't retrieved.

Parameters

None.

Returns

A Boolean value: `True` if component was successfully retrieved, `False` otherwise.

Example

```
&MYCI.EMPLID = "8001";

&MYCI.Get();

If %Session.ErrorPending Then

    /* Get Unsuccessful, do error processing */

Else

    /* do regular processing */

End-if;
```

Save

Syntax

```
Save ()
```

Description

Saves any changes that have been made to the data of the Component Interface executing the method. You must save any new Component Interfaces you create before they will be added to the database.

The standard PeopleSoft save business rules (that is, any PeopleCode programs associated with `SaveEdit`, `SavePreChange`, `WorkFlow`, etc.) will be executed after you execute this method. If you didn't specify the Component Interface to run in interactive mode, `FieldEdit`, `FieldChange`, and so on, will also run at this time.

If there are multiple errors, all errors will be logged to the PSMessages collection, not just the first occurrence of an error. As you correct each error, you will want to delete it from the PSMessages collection.



If you're running a Component Interface from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

Parameters

None.

Returns

A Boolean value: True if component was successfully saved, False otherwise.

Example

```
&MYCI.Emplid = "8001";  
  
&MYCI.Get();  
  
&MYCI.CHECKLIST_CD = "00001";  
  
&MYCI.Save;
```

Component Interface Class Properties

CreateKeyInfoCollection

This property returns a CompIntfPropInfoCollection collection that contains a CompIntfPropInfoCollection object for every key in CREATEKEYS.



For more information, see [Accessing the Structure of a Component Interface](#).

This property is read-only.

Example

```
&CREATEKEYS = &CI.CreateKeyInfoCollection;
```

FindKeyInfoCollection

This property returns a CompIntfPropInfoCollection collection that contains a CompIntfPropInfoCollection object for every key in FINDKEYS.



For more information, see [Accessing the Structure of a Component Interface](#).

This property is read-only.

GetHistoryItems

When this property is set to True, and you use the **Get** method, the Component Interface object will return all history data. You will be able to change this history data, the same as if you accessed a component in correction mode.

When this property is set to False, the Component Interface will run in "update/display" mode, that is, none of the history will be accessible or editable.

You must set this property to True *before* you execute the **Get** method.

This property is read-write.

Example

The following example checks the current status of the mode, then sets the GetHistoryItems property to True if the mode is Correction mode.

```
If %Mode = "C" Then  
    &CI.GetHistoryItems = True;  
End-if;
```

GetKeyInfoCollection

This property returns a CompIntfPropInfoCollection collection that contains a CompIntfPropInfoCollection object for every key in GETKEYS.



For more information, see [Accessing the Structure of a Component Interface](#).

This property is read-only.

InteractiveMode

When this property is set as True, the Component Interface will run the same as a component: that is, any PeopleCode programs associated with FieldChange, RowInsert, and so on, will run immediately after you make a change. If this property is set to False, these programs won't run until you execute the **Save** method.

You must set this property to True *before* you execute the **Get** method.

This property is read-write.

PropertyInfoCollection

This property returns a `CompIntfPropInfoCollection` object for every property that isn't a collection (that is, a scroll.) If the property is a collection (scroll), use the **PropertyInfoCollection** property to get another collection.



For more information, see [Accessing the Structure of a Component Interface](#).

This property is read-only.

StopOnFirstError

When this property is set as `True`, the first error generated by the Component Interface will halt execution of the program. The default value is `False`.

This property is read-write.

Data Collection Methods

A *data collection* is the collection of data, available at runtime or during test mode, associated with a particular scroll (or record.) The data collection object returns information about every **row of data** (item) that is returned for that record at runtime.

To access a data collection, use the name of the record (scroll) as a property on a Component Interface.



For more information, see [Traversing a Component Interface and Using Data Collections](#).

CurrentItem

Syntax

```
CurrentItem()
```

Description

If the component associated with the Component Interface is effective dated, **CurrentItem** will return a reference to the current effective dated item (row of data). To get a specific item based on a date, use **GetEffectiveItem**.

Example

```
&MYCD = &MYCI.EMPL_CHKLIST_ITM;  
  
&ITEM = &MYDC.CurrentItem();
```

DeleteItem

Syntax

```
DeleteItem(number)
```

Description

The **DeleteItem** method deletes the item (row of data) at the position specified by *number*. This parameter takes a number value.

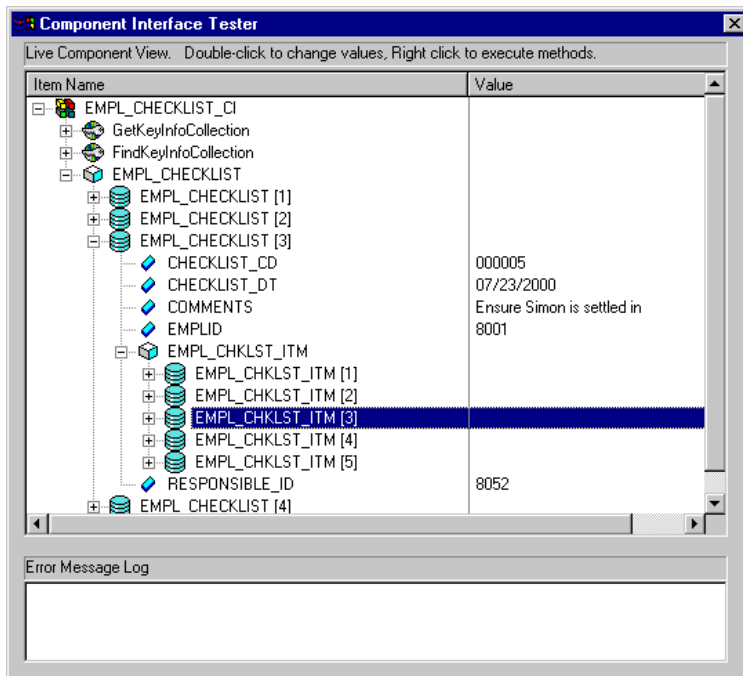
When the DeleteItem method is executed, if there are any RowDelete PeopleCode programs associated with any of the fields, they will fire as well, as if the user pressed ALT+8 and ENTER or clicked the DeleteRow icon. However, the program(s) will be executed as if turbo mode was selected. (In turbo mode default processing is performed *only* for the row being deleted.)

If you set the **InteractiveMode** property to True, any RowDelete PeopleCode will run immediately after you execute DeleteItem. If this property is set to False, any RowDelete PeopleCode will run after you execute the **Save** method.

The deleted item will not actually be deleted from the database until after you use the **Save** method.

Example

For example, suppose your Component Interface had two scrolls: EMPL_CHECKLIST and EMPL_CHKLIST_ITM. The data collection EMPL_CHECKLIST has four items (rows of data.) The data collection EMPL_CHKLIST_ITM (under the third item) has five items (rows of data.) If you run this component in the Component Interface Tester, it would look as follows:



EMPL_CHK_BC in Component Interface Tester

If you wanted to delete the third row in the third item, you could use the following:

```
Local ApiObject &MYSESSION;

Local ApiObject &MYCI;

&MYSESSION = GetSession();

&MYSESSION.Connect(1, "EXISTING", "", "", 0);

&MYCI = &MYSESSION.GetCompIntfc(COMPINTFC.EMPL_CHK_BC);

&MYLVL1 = &MYCI.EMPL_CHECKLIST;

&ITEM2 = &MYLVL1.Item(3);

&MYLVL2 = &ITEM2.EMPL_CHKLIST_ITM;

&MYLVL2.DeleteItem(3);

&MYCI.Save();
```

GetEffectiveItem

Syntax

```
GetEffectiveItem(DateString, SeqNo)
```

Description

If the component associated with the Component Interface is effective dated, **GetEffectiveItem** will return a reference to the closest effective dated item (row of data) that is less than or equal to the date specified by the *DateString*. To get an item based on the current effective date, use **CurrentItem**.



DateString only takes a string value. You must convert a date variable into a string before you can use it for *DateString*. You can use the String function to do this.

Parameters

<i>DateString</i>	Specify the year, month and day of the effective date you want to look for. This parameter takes a string value. You can specify the date either as YYYY-MM-DD or MM/DD/YY.
<i>SeqNo</i>	Specify the sequence number of the effective date you want to look for. This parameter takes a number value.

Returns

A reference to an effective dated item.

Example

```
&MYCD = &MYCI.EMPL_CHKLIST_ITM;

&DSTRING = String(&MYDATE);

&ITEM = &MYDC.GetEffectiveItem(&DSTRING, 1);
```

GetEffectiveItemNum

Syntax

```
GetEffectiveItemNum(DateString, SeqNo)
```

Description

If the component associated with the Component Interface is effective dated, **GetEffectiveItemNum** returns a reference to the number of the closest effective dated item (row of data) that is less than or equal to the date specified by the *DateString*. To get an item number based on the current effective date, use **CurrentItemNum**.



DateString only takes a string value. You must convert a date variable into a string before you can use it for *DateString*. You can use the String function to do this.

Considerations for Returning Rows

GetEffectiveItemNum returns a valid row number only when $EFFDT \leq DateString$. If the value you use for *DateString* pre-dates all the rows in the data collection, this method returns a zero and logs a message in the PSMessages collection.

For example, if 12/01/1990 was the earliest date in the collection, the following would return zero:

```
&NUM = &MYDC.GetEffectiveItemNum("01/01/1900", 1);
```

Parameters

DateString Specify the year, month and day of the effective date you want to look for. This parameter takes a string value. You can specify the date either as YYYY-MM-DD or MM/DD/YYYY.

SeqNo Specify the sequence number of the effective date you want to look for. This parameter takes a number value.

Returns

A number. This method returns 0 if no row matching the criteria is found.

Example

```
&MYCD = &MYCI.EMPL_CHKLIST_ITM;

&DSTRING = String(&MYDATE);

&ITEMNUM = &MYDC.GetEffectiveItemNum(&DSTRING, 1);
```

InsertItem

Syntax

```
InsertItem(number)
```

Description

The **InsertItem** method inserts the item (row of data) at the position specified by *number*. This parameter takes a number value. You can only insert items below the zero level scroll. If you need to add a new data item, you should use the **Create** method instead.

InsertItem adds the new row *after* the current row. If the row has an effective date (EFFDT) or an effective sequence (EFFSEQ), these values will be copied into the new row.

If you specify -1 for *number*, InsertItem will insert a new item (row) after the **last** row.

The **InsertItem** method returns a reference to the newly created item (row of data).

When the `InsertItem` method is executed, if there are any `RowInsert` PeopleCode programs associated with any of the fields, they will fire as well, as if the user pressed ALT+7 and ENTER or clicked the `InsertRow` icon. However, the program(s) will be executed as if turbo mode was selected. (In turbo mode default processing is performed *only* for the row being inserted.)

If you set the **InteractiveMode** property to True, any `RowInsert` PeopleCode will run immediately after you execute `InsertItem`. If this property is set to False, any `RowInsert` PeopleCode will run after you execute the **Save** method.

The inserted item will not be added to the database until after you use the **Save** method.

Example

In the following example a new item (row of data) is added at the *end* of the current collection.

```
&MYDC = &MYCI.EMPL_CHECKLIST;

&COUNT = &MYDC.Count;

&ITEM = &MYDC.InsertItem(&COUNT);

&ITEM.CHECKLIST_CD = "00001";

&ITEM.RESPONSIBLE_ID = "6609";

&RSLT = &MYCI.Save();
```

Item

Syntax

Item(*number*)

Description

The **Item** returns the item (row of data) that exists at the *number* position in the data collection executing the method. The parameter takes a number value.

ItemByKey

Syntax

ItemByKey(*key_values*)

Description

The **ItemByKey** method returns the item specified by the parameters. The number and type of keys are unique to each specific collection. Each key must be separated by a comma.

The collection reference must be the name of the Component Interface, followed by the record name. This method won't work on a collection reference (that is, `&CI.RECNAME.ItemByKey`, not `&MYCOLLECTION.ItemByKey`).

After you've returned an item, you can use the ItemNum property to determine the number of the item.

The keys must be in the *exact* order as in the Component Interface. A second level data collection will also contain the keys of the parent data collection.

An easy way to determine the keys and their order in PeopleCode is to open the Component Interface in Application Designer, and use the Test Component. To determine the keys in Visual Basic, use the Object Browser.



For more information about using the Test Component, see PeopleSoft Component Interface.

To see the signature for ItemByKeys:

1. Open the Component Interface in Application Designer.
2. Start the Component Interface Tester.

Select the open Component Interface, then right click, and select Test Component Interface from the pop-up menu.

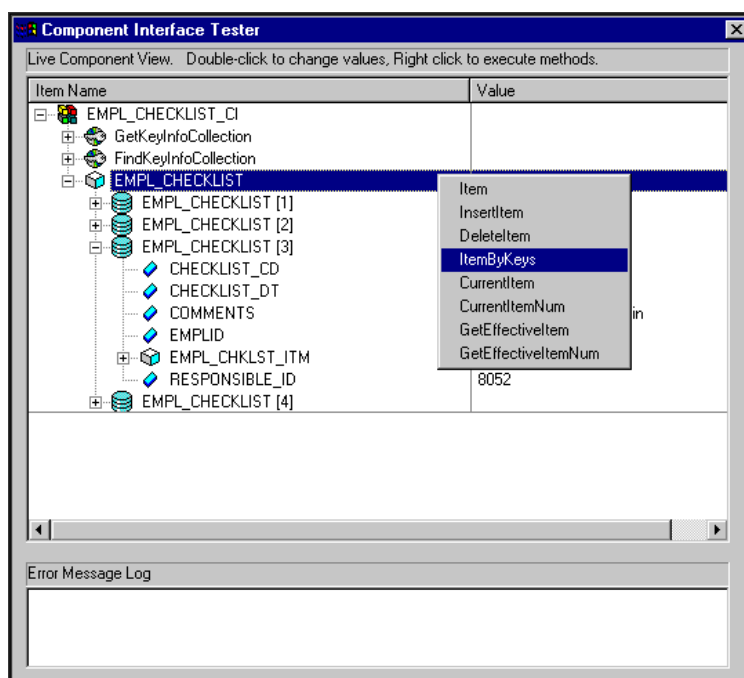


For more information on using the Test Component, see PeopleSoft Component Interface.

3. Instantiate an object.

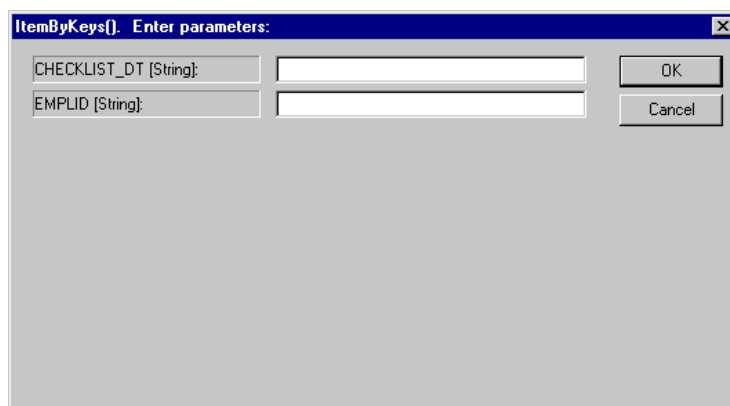
Add data to the Get or Create keys and click on **Get Existing** or **Create New**, respectively.

4. Expand the instantiated component until you find the collection in which you're interested.
5. Right-click on the collection and choose ItemByKeys from the resulting pop-up menu.



Component Interface in Component Tester with pop-up menu

- The dialog that follows shows you the specific parameters, types, and order in which you should call ItemByKeys.



ItemByKeys dialog box

Returns

An item (row) of data from a data collection.

Example

```
Local ApiObject &MYSESSION;

Local ApiObject &CI;

Local ApiObject &CI_COLLECTION;
```

```
Local ApiObject &CI_ITEM;

&MYSESSION = GetSession();

&MYSESSION.Connect(1, "Existing", "", "", 0);

&CI = &MYSESSION.GetCompIntfc(COMPINTFC.CM_EVAL);

&CI.EMPLID = "8001";

If &CI.Get() <> 1 Then

    Exit;

End-If;

&CI_COLLECTION = &CI.CM_EVALUATIONS;

&COUNT = &CI_COLLECTION.Count;

&CI_ITEM = &CI.CM_EVALUATIONS.itembykeys("01");

&CI_ITEM.DESCR50 = "TEST";

If &CI.Save() <> 1 Then

    Exit;

End-If;
```

Data Collection Properties

Count

This property returns the number of data items (rows of data) in the data collection.

This property is read-only.

Example

```
&CI = &MYSESSION.GetCompIntfc(COMPINTFC.CM_EVAL_BC);

&CI.EMPLID = "8001";

&CI.Get()
```

```

&CI_COLLECTION = &CI.CM_EVALUATIONS;

&COUNT = &CI_COLLECTION.Count;

```

CurrentItemNumber

If the component associated with the Component Interface is effective dated, this property returns the item number for the current effective dated item (row of data).

This property is read-only.

Data Item Class Property

ItemNum

This property returns the number of the data item (row) in the collection. For example, many of the data collection methods takes a number as a parameter. You can use this property to determine the item number (row number) of an item in a collection, then use that number in another method.

This property is read-only.

Example

```

Evaluate USER_ACTION

. . .

When = "D"

    &CI_ITEM = &CI_LVL1_NAMES.ItemByKey(&NAME_TYPE, &NAME_PART);

    If &CI_ITEM <> Null then

        &I = &CI_LVL1_NAMES.ItemNum;

        &CI_LVL1_NAMES.DeleteItem(&I);

    End-if;

. . .

End-Evaluate;

```

Accessing the Structure of a Component Interface

The structure of a Component Interface can be accessed using a `CompIntfPropInfoCollection` object. You access a `CompIntfPropInfoCollection` object from a `CompIntfPropInfoCollection` collection. There is more than one way to instantiate a `CompIntfPropInfoCollection` collection.



You don't have to populate a Component Interface with data before you access the structure. You can access the structure of a Component Interface immediately after you use the `GetCompIntfc` method with a session object. Accessing the structure of a Component Interface before you populate it with data may increase your performance.

CompIntfPropInfoCollection Collection

There are two types of `CompIntfPropInfoCollection` object properties: *field* properties and *collection* properties.

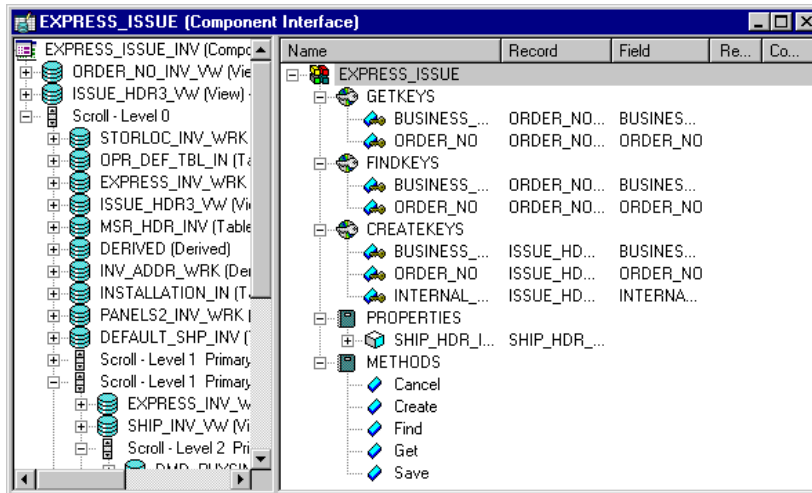
A *field* property maps to a specific record field. You can access structural information about the field via a `CompIntfPropInfoCollection` object. This information includes the name of the field, whether it's required, is it based on a prompt table, and so on.

A *collection* property is just that, a collection of properties. And before you can access a `CompIntfPropInfoCollection` object, you must first get a `CompIntfPropInfoCollection` **collection**. The following are the valid types of `CompIntfPropInfoCollection` collections:

- CREATEKEYS, GETKEYS and FINDKEYS

When you create a component, you must specify the search record to be used with that component. You can also specify an alternate search record to be used when the component is accessed in Add mode. The key fields from those records make up GETKEYS, FINDKEYS and CREATEKEYS collections.

- CREATEKEYS: A collection containing the key fields of the search record specified to be used in Add mode. Use the **CreateKeyInfoCollection** property to instantiate the `CompIntfPropInfoCollection` collection.
- GETKEYS: A collection containing the primary required key fields from the primary search record. Use the **GetKeyInfoCollection** property to instantiate the `BCPropertInfo` collection.
- FINDKEYS: A collection containing the key fields and the alternate key fields from the primary search record. Use the **FindKeyInfoCollection** property to instantiate the `BCPropertInfo` collection.



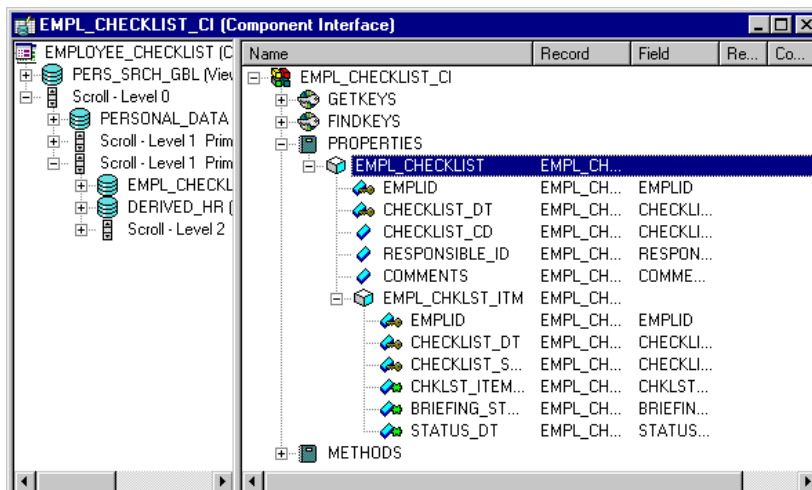
EXPRESS_ISSUE with GETKEYS, FINDKEYS and CREATEKEYS

- A page scroll

The fields associated with a page scroll are in this type of collection. This may or may not be all the fields associated with the record. Use the **PropertyInfoCollection** property to instantiate this kind of collection.

If the page the Component Interface is based on contains a secondary scroll, you can check the **Type** property to determine if the object is a **CompIntfPropInfoCollection** object (that is, a field), or a scroll. Then, if you want to get the properties of the fields associated with that secondary scroll, you can use the **PropertyInfoCollection** property on the **CompIntfPropInfoCollection** object.

For example, the following Component Interface has a level 0 and level 1 scroll.



Component Interface with Secondary Scroll

The level 0 scroll is made up of three fields: CHECKLIST_CD, RESPONSIBLE_ID, COMMENTS, and a level 1 scroll, EMPL_CHKLIST_ITM. The **CompIntfPropInfoCollection**

collection for this Component Interface would have four items in it. Three would be `CompIntfPropInfoCollection` objects (the three fields.) The last item, `EMPL_CHKLIST_ITM` would *not* be a valid `CompIntfPropInfoCollection` object. You can use the **IsCollection** property to verify if an item in a collection is itself a collection or a valid `CompIntfPropInfoCollection` object.

If you want to access the fields in a lower level scroll, you need to use the **PropertyInfoCollection** property first, to return a collection of those fields.

The following example loops through a Component Interface. It pulls out the names of the properties in the first three levels of a Component Interface. If the property is a nested collection, it prefixes the ancestor collection name to the property name.

```
&MYSESSION = GetSession();

&MYSESSION.Connect(1, "EXISTING", "", "", 0);

&CI = &MYSESSION.GetCompIntfc(COMPINTFC.VOLNEW);

&PROPINFO_0 = &CI.PropertyInfoCollection;

For &I = 1 To &PROPINFO_0.Count;

    &PROPITEM_0 = &PROPINFO_0.Item(&I);

    Warning (&PROPITEM_0.Name);

    If (&PROPITEM_0.IsCollection) Then

        &PROPINFO_1 = &PROPITEM_0.PropertyInfoCollection;

        For &J = 1 To &PROPINFO_1.Count;

            &PROPITEM_1 = &PROPINFO_1.Item(&J);

            &S1 = &PROPITEM_0.Name | "." | &PROPITEM_1.Name;

            Warning (&S1);

            If (&PROPITEM_1.IsCollection) Then

                &PROPINFO_2 = &PROPITEM_1.PropertyInfoCollection;

                For &K = 1 To &PROPINFO_2.Count;

                    &PROPITEM_2 = &PROPINFO_2.Item(&K);

                    &S1 = &PROPITEM_0.Name | "." | &PROPITEM_1.Name | "." |
&PROPITEM_2.Name;

                    Warning (&S1);

                End-For;

            End-If

        End-For;

    End-For;
```

```
End-If;

End-For;
```

CompIntfPropInfoCollection Collection Methods

First

Syntax

```
First()
```

Description

The **First** method returns the first CompIntfPropInfoCollection object in the CompIntfPropInfoCollection collection object executing the method.

Example

```
&CIINFO = &CIPOPCOLL.First();
```

Item

Syntax

```
Item(number)
```

Description

The **Item** method returns a CompIntfPropInfoCollection object that exists at the *number* position in the CompIntfPropInfoCollection collection executing the method

If the specified item is *itself* a collection, the CompIntfPropInfoCollection object that gets returned is invalid (set to NULL). You can use the **PropertyInfoCollection** property to return a collection of CompIntfPropInfoCollection objects for the collection.



For more information, see CompIntfPropInfoCollection Collection.

Example

```
&SCROLL1 = &MYCI.PropertyInfoCollection;

For &I = 1 to &SCROLL1.Count;

    &PROPERTY = &SCROLL1.Item(&I);

    If &PROPERTY.IsCollection Then
```

```

        &SCROLL2 = &PROPERTY.PropertyInfoCollection;

        /*do scroll 2 processing */

    Else

        /* do scroll 1 processing */

    End-If;

End-For;

```

Next

Syntax

Next ()

Description

The **Next** method returns the next **CompIntfPropInfoCollection** object in the **CompIntfPropInfoCollection** collection object executing the method. You can only use this method after you have used either the **First** or **Item** methods: otherwise the system doesn't know where to start.

CompIntfPropInfoCollection Collection Properties

Count

This property returns the number of **CompIntfPropInfoCollection** objects in the **CompIntfPropInfoCollection** collection object executing the method.

This property is read-only.

Example

```
&COUNT = &MYCIPROPINFOC.Count;
```

CompIntfPropInfoCollection Object Properties

Format

This property returns the field format for the object executing the property (that is, name, phone, zip, SSN, and so on) as a number. The valid values are:

<i>Value Returned</i>	<i>Description</i>
0	No Format

1	Name
2	Phone
3	Zip
4	SSN
5	Routine
6	Mixed Case
7	Raw Binary
8	Number only
9	SIN
10	Phone International
11	Zip International
12	Seconds
13	Microseconds
14	Custom



For more information, see Custom Field Formats.

This property is read-only.

IsCollection

This property returns True if the object executing the property is a data collection, False otherwise. If IsCollection is True, other field-oriented properties like Required, Type, Xlat, YesNo, Prompt and Format are undefined. If IsCollection is False, the object represents a field and all the above properties are defined as described.

This property is read-only.

Key

This property returns True if the object executing the property is a key, False otherwise.

This property is read-only.

LabelLong

This property returns the record field LongName value as a string. If there is a component override for this value, it will *not* be included.

This property is read-only.

LabelShort

This property returns the record field ShortName value as a string. If there is a component override for this value, it will *not* be included.

This property is read-only.

Name

This property returns the name of the object executing the property as a string.

This property is read-only.

Prompt

This property returns True if the object executing the property is associated with a prompt table, False otherwise.

This property is read-only.

PropertyInfoCollection

This property returns another CompIntfPropInfoCollection collection if the object executing the property is a collection.

This property is read-only.



For more information, see CompIntfPropInfoCollection Collection.

Example

```
&SCROLL1 = &MYCI.PropertyInfoCollection;

For &I = 1 to &SCROLL1.Count;

    &PROPERTY = &SCROLL1.Item(&I);

    If &PROPERTY.IsCollection Then

        &SCROLL2 = &PROPERTY.PropertyInfoCollection;

        /*do scroll 2 processing */

    Else

        /* do scroll 1 processing */
```

```
End-If;

End-For;
```

Required

This property returns True if the object executing the property is a required property, False otherwise.

This property is read-only.

Type

This property returns the field type, as a number, of the object. The valid values are:

<i>Value Returned</i>	<i>Description</i>
0	Character
1	Long Character
2	Number
3	Signed Number
4	Date
5	Time
6	DateTime
7	SubRecord (Not supported with Component Interfaces)
8	Image (Limited support with Component Interfaces)
9	ImageReference (Not supported with Component Interfaces)

This property is read-only.

Xlat

This property returns True if the object executing the property is associated with an XLAT table, False otherwise.

This property is read-only.

YesNo

This property returns True if the object executing the property is associated with the Yes/No table, False otherwise.

This property is read-only.

CHAPTER 5

Field Class

A *field* object, instantiated from the Field class, is a single instance of data within a record object, and is based on a field definition.

Accessing or changing the value of a field using the **Value** property is a common action.

SetDefault is a commonly used method. **Name**, **Enabled** and **Type** are several commonly used field properties. If you are working with message objects, **EditError** is also a commonly used field property.

The Field class is one of the data buffer access classes.



For more information, see Data Buffer Access.

Note

An expression of the form

```
FIELD.fieldname.property
```

or

```
FIELD.fieldname.method(...)
```

is converted to an object expression by using **GetField**(FIELD.fieldname). An expression of the form

```
recname.fieldname.property
```

or

```
recname.fieldname.method(. . .)
```

is converted to an object expression by using **GetField**(recname.fieldname).

Declaring a Field Object

Field objects are declared as type Field. For example,

```
Local Field &MYFIELD;
```

Scope of a Field Object

A field can only be instantiated from PeopleCode.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Application Engine PeopleCode, record field PeopleCode, and so on.

Field Class Built-in Function

GetField

Field Class Methods

GetLongLabel

Syntax

```
GetLongLabel (LabelID)
```

Description

The **GetLongLabel** method returns the long name for a field given a label ID. If the given label ID isn't found, a null string is returned. *LabelID* takes a string value.



If a hyperlink or button has an associated record field, and a custom label has been set for that field using PeopleCode, this label value will be used for the tooltip pop-up in place of the default long record field text.

Returns

A text string containing the long name of the field for the specified label ID.

Example

The following code will set the label for a field to one of two different texts, based on the page.

```
Local Field &FIELD;  
  
&FIELD = GetField(RECORD.MYFIELD);  
  
If %Page = PAGE.PAGE1 Then  
    &FIELD.Label = &FIELD.GetLongLabel ("LABEL1");  
  
Else If %Page = PAGE.PAGE2 Then
```

```

&FIELD.Label = &FIELD.GetLongLabel ("LABEL2") ;

End-If;

```

If the Label ID is the same as the name of the field, you could use the following:

```

&LABELID = &FIELD.Name;

&FIELD.Label = &FIELD.GetLongLabel (&LABELID) ;

```

Related Topics

GetShortLabel method, Label property, SetLabel function

GetRelated

Syntax

```
GetRelated(recname.fieldname)
```

Description

The **GetRelated** method returns a field object for a related field *recname.fieldname* that has the field executing the method as its control field.

This method is similar to the **GetRelField** built-in function, however, the built-in only works in the current context, while this method can be applied to a field from any position in the buffer structure.

Using GetRelated with a Control Field

PeopleCode events on the Control Field can be triggered by the Related Edit field. When this happens, there can different behavior than with other types of fields:

- If the events are called from FieldEdit of the Control Field, and that FieldEdit is triggered by a change in the Related Edit field, the functions return the previous value.
- If the events are called from FieldChange of the Control Field, and that FieldChange is triggered by a change in the Related Edit field, the functions return the value entered into the Related Edit. This may be a partial value, that will subsequently be expanded to a complete value when the processing is complete.

Parameters

<i>recname.fieldname</i>	Specifies a field in the same row as the current field object, that has the field executing the method as its control field.
--------------------------	--

Returns

The field object for the field with the specified name that is related to the field executing the method.

Example

In the following example, the field object is instantiated, then the related display field object. The **Value** property for the related display is changed, it is disabled, and another variable is assigned its value.

```
Local Field &FIELD, &REL_FIELD;

&FIELD = GetField(OPC_9A2FIELDS.COMPANY);

/* control field object */

&REL_FIELD = &FIELD.GetRelated(COMPANY_TBL.DESCR);

/* related display object*/

&REL_FIELD.Value = "Change";

&REL_FIELD.Enabled = False;

&TMP = &REL_FIELD.Name;

&REL_FIELD.Value = &TMP;
```

If you were not going to use the &FIELD variable later, the first two lines of code in the above example could be combined:

```
&REL_FIELD = GetField(OPC_9A2FIELDS.COMPANY).GetRelated(COMPANY_TBL.DESCR);
```

Suppose you had two control fields, EMPLID and MANAGER_ID, and both use the NAME field on the PERSONAL_DATA table as their related display. If you need to access both related display fields, you could do the following:

```
&NAME_EMPLID = GetField(PERSONAL_DATA.EMPLID).GetRelated(PERSONAL_DATA.NAME);

&NAME_MANAGER =
GetField(PERSONAL_DATA.MANAGER_ID).GetRelated(PERSONAL_DATA.NAME);
```

Related Topics

GetField

GetShortLabel

Syntax

```
GetShortLabel (LabelID)
```

Description

The **GetShortLabel** method returns the short name for a field given a label ID. If the given label ID isn't found, a null string is returned. *LabelID* takes a string value.



If a hyperlink or button has an associated record field, and a custom label has been set for that field using PeopleCode, this label value will be used for the tooltip pop-up in place of the default long record field text.

Returns

A text string containing the short name of the field for the specified label ID.

Example

The following code will set the label for a field to one of two different texts, based on the page.

```
Local Field &FIELD;

&FIELD = GetField(RECORD.MYFIELD);

If %Page = PAGE.PAGE1 Then

    &FIELD.Label = &FIELD.GetShortLabel("LABEL1");

Else If %Page = PAGE.PAGE2 Then

    &FIELD.Label = &FIELD.GetShortLabel("LABEL2");

End-If;
```

If the Label ID is the same as the name of the field, you could use the following:

```
&LABELID = &FIELD.Name;

&FIELD.Label = &FIELD.GetShortLabel(&LABELID);
```

Related Topics

GetLongLabel method, Label property, SetLabel function

SetCursorPos

Syntax

```
SetCursorPos(PAGE.pagename | %Page)
```

Description

The **SetCursorPos** allows you to set the focus to the page field corresponding to the field object (on the specified page). The current page may be specified as %Page.

Restrictions on Use with a Component Interface

This method can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface or not.

```

If %CompIntfcName Then

    /* process is being called from a Component Interface */

    /* do BC specific processing */

Else

    /* do regular processing */

    . . .

End-if;

```

Returns

None.

Parameters

<i>pagename</i>	The name of the page, preceded by the keyword Page . The <i>pagename</i> page must be in the current component. You can also pass the %page system variable in this parameter (without the Page reserved word).
-----------------	---

Returns

None.

Example

The following pseudo-code enables you to set the focus to a related field:

```

GetField(ControlRec.ControlField).GetRelated(RelatedRec.RelatedField).SetCursorPos(PAGE.pagename);

```

Related Topics

SetCursorPos built-in function, SetDefault field method

SetDefault

Syntax

```
SetDefault()
```

Description

SetDefault sets the value of the field to a null value, or to a default, depending on the type of field.

- If this method is used against data from the Component buffers, the next time default processing occurs, it is set to its default value: either a default specified in its record field definition or one set programmatically by PeopleCode located in a FieldDefault event. If neither of these defaults exist, the Component Processor leaves the field blank.
- If this method is used with a field that isn't part of the data buffer (for example, a field in a record object instantiated with CreateRecord) the field is automatically set to its default value if one is set for the **field**, not for the record field. Any FieldDefault PeopleCode will *not* be run on these types of fields. If you want to set the default values for all the fields in a record, use the **SetDefault** record class method.



For more information, see the SetDefault record class method.

Blank numbers correspond to zero on the database. Blank characters correspond to a space on the database. Blank dates and long characters correspond to NULL on the database. **SetDefault** gives each field data type its proper value.

Parameters

None.

Returns

None.

Example

```
&CHARACTER.SetDefault();
```

Related Topics

Default Processing, SetDefault Record class method

Field Class Properties

DisplayFormat

Use this property to specify a custom format to use for the field.

This property is read-write.

The custom format for a field is specified in the field definition. This property allows you to switch between display formats that are defined as part of a custom format. For example, suppose your field used the PHONE-INTL custom format:

PHONE-INTL Custom Format

Both the LONG and the SHORT stored formats have two display formats: STANDARD and UNFORMATTED.

Using this property, you could select either of the display formats. For example:

```
If %Component = PAGE.INTERNATIONAL
    &CHAR.DisplayFormat = "STANDARD";
Else
    &CHAR.DisplayFormat = "UNFORMATTED";
End-If;
```



You can *not* change the Stored format, but you can find its value using the StoredFormat property.

Example

```
&CHARACTER.DisplayFormat = "STANDARD";

/* this assumes that &CHARACTER is a custom formatted field, and a display
format option is STANDARD */
```


DisplayOnly

This property is set to True if the field is set to DisplayOnly. May be set to False to change how the field displays.



Note. This property *overwrites* whatever value is set in Application Designer.

This property is read-write.

EditError

This property is True if an error for this field has been found after executing the **ExecuteEdits** method with either a message object or a record object. This property can be used with the **MessageSetNumber** and **MessageNumber** properties to find the error message set number and error message number.



For more information, see ExecuteEdits record class method, ExecuteEdits message class method, MessageNumber and MessageSetNumber.

This property is read-write. After you have fixed the errors, you need to set this property to False before running **ExecuteEdits** again.

Example

The following is an example showing how **EditError**, along with the **ExecuteEdits** method could be used:

```
&REC.ExecuteEdits();

If &REC.IsEditError Then

    For &I = 1 to &REC.FieldCount

        If &REC.GetField(&I).EditError Then

            LOG_ERROR(); /* application specific pgm */

        End-If;

    End-For;

End-If;
```

Enabled

This property is True if this field is enabled. May be set False to disable any control that displays this field.

This property is read-write.

Example

```
&CHARACTER.Enabled = True;  
  
&CHARACTER.Enabled = False;
```

FormattedValue

This property returns the value of a field as a string, formatted exactly as it would be displayed on an edit field on a page. This is useful when you're using a prompt field for the label of another field.

Because a record field can be bound to more than one page field, FormattedValue takes the first associated page field, looking first on the current page, then through all the pages in the component. This property returns a null string ("") if used with an Image or ContentReference field. It also returns a null string if no associated page field is found.

Example

This property could be used to set up a hyperlink labeled by data generated on a page. To do this, you have to do the following:

1. Define a work field.
2. Associate the work field with the hyperlink page field.
3. Define a hidden page field with the formatting that you want.
4. Associate the hidden page field with the data field.
5. Write PeopleCode (probably for RowInit) to set the label on the work field to be the FormattedValue of the data field.

For example, suppose you wanted a hyperlink labeled by the QE_POSITION_ID field on the QE_PLYR_POSITN record. This field is a prompt table field, edited by table QE_POSITION.

You'd like to display the DESCRSHORT field from the prompt table, instead of the QE_POSITION_ID code value. Create a work field, POS_LINK on the WORK_REC record, and create a hyperlink page field associated with the DESCRSHORT field. For the label, create an invisible drop down list page field, associated with QE_PLYR_POSITN.QE_POSITION_ID. Set its Prompt Table Field to DESCRSHORT. Then put a PeopleCode program on the component RowInit for either the QE_PLYR_POSITN record or the WORK_REC.

```
WORKREC.POS_LINK.Label = QE_PLYR_POSITN.QE_POSITION_ID.FormattedValue;
```

IsAltKey

This property is True if this field references a field definition that is defined as an Alternate Search key in the associated record definition.

This property is read-only.

IsAuditFieldAdd

This property is True if this field references a field definition that is defined as an audit field in the associated record definition, and the field is audited whenever new data is added.

This property is read-only.

IsAuditFieldChg

This property is True if this field references a field definition that is defined as an audit field in the associated record definition, and the field is audited whenever data is changed.

This property is read-only.

IsAuditFieldDel

This property is True if this field references a field definition that is defined as an audit field in the associated record definition, and the field is audited whenever data is deleted.

This property is read-only.

IsAutoUpdate

This property is True if this field references a field definition that is defined as an Auto-Update field in the associated record definition.

This property is read-only.

IsChanged

This property returns True if the value for the field has been changed.



This property is only meant to be used with the primary database record. It will not return valid results if used with a work record.

This property is read-only.

Example

```
If &CHARACTER.IsChanged Then  
  
    Warning ("The character field has been changed");  
  
End-If;
```

IsDateRangeEdit

This property is True if this field references a field definition that is defined as requiring a Reasonable Date in the associated record definition.

This property is read-only.

IsDescKey

This property is True if this field references a field definition that is defined as a descending key in the associated record definition.

This property is read-only.

IsDuplKey

This property is True if this field references a field definition that is defined as a duplicate key in the associated record definition.

This property is read-only.

IsEditTable

This property is True if this field references a field definition that has a table edit type of a Prompt Table with Edit.

This property is read-only.

IsEditXlat

This property is True if this field references a field definition that has a table edit type of a Translate Table Edit.

This property is read-only.

IsFromSearchField

This property is True if this field references a field definition that is defined as a From Search Field in the associated record definition.

This property is read-only.

IsInBuf

This property returns True if the data of the field is present in the data buffers. For example, all of the fields of a derived record are not present in the data buffer.



For more information on what data is accessible, see [What Record Fields Are in the Component Buffer?](#)

This property is read-only.

Example

The following example iterates over all the fields in a record. The code verifies that the data for the field is accessible before it tries to assign the value to a variable.

```
For &I = 1 to &REC.FieldCount
    &FIELD = &REC.GetField(&I);
    If &FIELD.IsInBuf Then
        &VALUE = &FIELD.Value;
        /* do other processing */
    End-If;
End-For;
```

IsKey

This property is True if this field references a field definition that is defined as a primary key of the associated record definition.

This property is read-only.

Example

```
If &CHARACTER.IsKey Then
    Warning ("The field " | &Character.Name | " is a key");
End-If;
```

IsListItem

This property is True if this field references a field definition that is defined as a List Box item of the associated record definition.

This property is read-only.

IsRequired

This property is True if this field references a field definition that is defined as Required in the associated record definition.

This property is read-only.

IsSearchItem

This property is True if this field references a field definition that is defined as a Search key in the associated record definition.

This property is read-only.

IsSystem

This property is True if this field references a field definition that is defined as System Maintained field in the associated record definition.

This property is read-only.

IsThroughSearchField

This property is True if this field references a field definition that is defined as a Through Search Field in the associated record definition.

This property is read-only.

IsUseDefaultLabel

This property is True if this field references a field definition that has its label defined as Use Default Label in the associated record definition.

This property is read-only.

IsYesNo

This property is True if this field references a field definition that has a table edit type of a Yes/No Table Edit.

This property is read-only.

Label

This property returns a reference to the label you have set for the field. You can use this property to change the label text for the field when it's displayed on a page.

This property returns a blank string if you use it before you've set the label with either the SetLabel function or with this property.

To return the text of a label before you set it, use the `GetShortLabel` or `GetLongLabel` methods.



You can't use this property to set labels longer than 100 characters. If you try to set a label of more than 100 characters, the label will be truncated to 100 characters.



If a hyperlink or button has an associated record field, and a custom label has been set for that field using PeopleCode, this label value will be used for the tooltip pop-up in place of the default long record field text.

This property is read-write.

Example

The following code sample changes all the field labels for a record to the short label. It assumes that there is a label with the same name as the name of the field for all fields in the record.

```
Local Field &FLD;

Local Record &REC;

If CHECK_FIELD Then

    &REC = GetRecord();

    For &I = 1 to &REC.FieldCount

        &FLD = &REC.GetField(&I);

        &LABELID = &FLD.Name;

        &FLD.Label = &FLD.GetShortLabel (&LABELID);

    End-For;

End-If;
```

LongTranslateValue

This property returns a string that contains the Long translate (XLAT) value of the field if the field is based on an XLATABLE.

If the field has a null value, a null string is returned. If the field isn't based on a translate table, or the value isn't in the translate table, the field's current value is returned. Since the current value can be of any type, this property has a type of Any.



If you're accessing a field based on the XLATTABLE, the Value property will only return the one or two letter XLAT value.

Use the ShortTranslateValue property to return the short translate value of a field.

This property is read-only.

Example

```
Local Any &VALUE;

Local Field &MYFIELD;

&MYFIELD = GetField();

&VALUE = &MYFIELD.LongTranslateValue;

If ALL(&VALUE) Then

    /* do processing */

End-if;
```

MessageNumber

This property returns the error message number (as a number) if an error for this field is found after executing **ExecuteEdits** method. You can use this property in conjunction with the **EditError** property (which can be used to determine whether there are any errors) and the **MessageSetNumber** property (which contains the error message set number if an error is found.)

This property is read-only.

Example

```
For &I = 1 to &RECORD.FieldCount

    &MYFIELD = &RECORD.GetField(&I);

    If &MYFIELD.EditError Then

        &MSGNUM = &MYFIELD.MessageNumber;

        &MSGSET = &MYFIELD.MessageSetNumber;

        /* Do processing */

    End-If;

End-For;
```


MessageSetNumber

This property returns the error message set number (as a number) if an error for this field is found after executing **ExecuteEdits** method. You can use this property in conjunction with the **EditError** property (which can be used to determine whether there are any errors) and the **MessageNumber** property (which contains the error message number if an error is found.)

This property is read-only.

Example

```
For &I = 1 to &RECORD.FieldCount

    &MYFIELD = &RECORD.GetField(&I);

    If &MYFIELD.EditError Then

        &MSGNUM = &MYFIELD.MessageNumber;

        &MSGSET = &MYFIELD.MessageSetNumber;

        /* Do processing */

    End-If;

End-For;
```

Name

This property returns the name of the field definition that the field object is based on as a string value.

This property is read-only.

Example

```
WinMessage("The character field's name is : " | &CHARACTER.Name);
```

OriginalValue

This property returns the value of a field, that is, the value that from the database. If the value hasn't been changed and saved by the user, it will be the original value from the database. If the value has been changed and saved by the user, it will be the existing value in the database.

This property is read-only.

Example

```
&Orig = &MyField.OriginalValue;

If &Orig = &Date Then

    /* do current day processing */
```

```

Else
    /* do other processing */
End-If;

```

ParentRecord

This property returns a reference to the record object for the record containing the field.

This property is read-only.

Example

```

&NUMBER_OF_FIELDS = &CHARACTER.ParentRecord.Fieldcount;

/* note that FieldCount is a property of the Record class */

```

SearchDefault

If this property is set to True, system defaults (the default values set in the record field definitions) are enabled on search dialogs for the field. Setting this property to True *does not* cause the FieldDefault event to fire.

The system default is done only once, when the search dialog first starts, immediately after any SearchInit PeopleCode. If the user subsequently blanks out a field, the field isn't reset to the default value. Setting **SeachDefault** to False disables default processing for the field executing the property.

SearchDefault is only effective when used in SearchInit PeopleCode programs.

Example

```

&CHARACTER.SearchDefault = True;

/* assuming &CHARACTER is a search key, and has a default value */

```

This example turns on edits and system defaults for the SETID field in the search dialog:

```

&SETID = GetField(INV_ITEMS.SETID);

&SETID.SeachDefault = True;

&SETID.SearchEdit = True;

```

SearchEdit

If this property is set to True, **SearchEdit** enables system edits (edits specified in the record field definition) for the field, for the life of the search dialog. Setting SeachDefault to False disables system edits. In the Add mode search dialog, the following edits are performed when the end-user

clicks the **Add** button. In any other mode, the following edits are performed when the end-user clicks the **Search** button:

- Formatting
- Required Field
- Yes/No Table
- Translate Table
- Prompt Table

SearchEdit does not cause the FieldEdit, FieldChange, or SaveEdit PeopleCode events to fire during the search dialog.

You might use SearchEdit to control access to the system. For example, you can apply this function to the SETID field of a dialog box and require the user to enter a valid SETID before they are able to OK the search dialog.

Example

```
&CHARACTER.SearchEdit = True;  
  
/* assuming &CHARACTER is a search key, and contains an edit table such as  
translate table defined in its field properties. */
```

This example turns on edits and system defaults for the SETID field in the search dialog:

```
&SETID = GetField(INV_ITEMS.SETID);  
  
&SETID.SeachDefault = True;  
  
&SETID.SearchEdit = True;
```

ShortTranslateValue

This property returns a string that contains the Short translate (XLAT) value of the field if the field is based on an XLATTABLE.

If the field has a null value, a null string is returned. If the field isn't based on a translate table, or the value isn't in the translate table, the field's current value is returned. Since the current value can be of any type, this property has a type of Any.



If you're accessing a field based on the XLATTABLE, the Value property will only return the one or two letter XLAT value.

Use the LongTranslateValue property to return the long translate value of a field.

This property is read-only.

Example

```

Local Any &VALUE;

Local Field &MYFIELD;

&MYFIELD = GetField();

&VALUE = &MYFIELD.ShortTranslateValue;

If ALL(&VALUE) Then

    /* do processing */

End-if;

```

ShowRequiredFieldCue

With PeopleTools 8, an asterisk (*) is displayed on pages beside fields that are defined as Required in Application Designer. You can use this property to specify whether this asterisk, also called the **required field cue**, is displayed for a particular field.

For example, many fields are made required or non-required either procedurally or through PeopleCode. This means they aren't defined as Required in Application Designer, and your end-user may be confused. For these fields, you can use this property.



This property only affects fields where a required field cue is otherwise permissible. That is, regardless of the setting of the property, no cue is ever shown on a pushbutton, a display-only field, and so on.

This property is read-write.

StoredFormat

This property returns the custom character format (as a string) for the field executing the property.

If the field doesn't have a custom format associated with it, the user will receive a runtime error message.

If the field has a display format associated with it, you can change that using the DisplayFormat property.

This property is read-only.

Example

```

WinMessage("The character field's custom stored format is : " |
&CHARACTER.StoredFormat);

```

```
/* this assumes that &CHARACTER is a custom formatted field */
```

Style

If a page has a style sheet associated with it, this property can be used to specify a different style class with a field. This property will only *override* the existing style. It will not *change* it. The next time the page is accessed, the original style will be used.



This property is only valid for fields on pages that were used with Internet Client, that is, with pages created using **View, Internet Options**, and that have a style sheet associated with them.

This property takes a string value.

This property is read-write.

Example

The following example associates a test field first with a style class depending on the value of the field.

```
Local Field &field;
```

```
&field = GetField();
```

```
If TESTFIELD1 = 1 Then;
```

```
    &field.Style = "PSHYPERLINK";
```

```
End-If;
```

```
If TESTFIELD1 = 2 Then;
```

```
    &field.Style = "PSIMAGE";
```

```
End-If;
```

```
TESTFIELD1: 1
```

Field with PSHYPERLINK style

```
TESTFIELD1: 2
```

Field with PSIMAGE style

Type

This property returns the type of field. The values can be one of the following strings:

- CHAR
- CONTENTREFERENCE
- DATE
- DATETIME
- IMAGE
- LONGCHAR
- NUMBER
- SIGNEDNUMBER
- TIME

This property is read-only.

Example

```
If &CHARACTER.Type = "NUMBER" Then  
    /* perform processing */  
Else  
    /* error processing */  
End-If;
```

Value

This property contains the current value of the field, converted to an appropriate PeopleCode data type.

In most contexts, the Value property can be used to assign a new value to the field. However, there are some restrictions:

- You cannot assign the Value property in any RowSelect PeopleCode program.
- You cannot assign the current record field value to the Value property in any FieldEdit PeopleCode program.



If you're accessing a field based on the XLATTABLE, Value will only return the one letter XLAT value. To get the full XLAT value, use the LongTranslateValue or ShortTranslateValue properties.

Considerations when Checking for Values

If you have an edit-box field, and if the end-user selects the value in it and deletes the value, leaving the field empty, the value of the field in PeopleCode is *not* an empty (zero-length) string. Instead, it is a string with one space character in it. The following is the valid way to check for this:

```
If (fieldname.Value = " ") Then
```

There is no way to distinguish between cases where the user entered nothing or they entered one space character.

Example

```
&CHARACTER.Value = "Hello";
```

Visible

This property is True if this field is visible in the page displaying it. Setting this property to False will hide the field. Because every field is implicitly associated with a rowset, row, and record, setting the Visible property for a field on the first page of a component will only hide that field. If that field is repeated on other pages in the component, the other occurrences of the field will not be hidden.

Example

The following code hides the field &ROUND_OPTION based on the value of AVG_DFLT_OPTION:

```
Local field &ROUND_OPTION;

&ROUND_OPTION = GetField();

If AVG_DFLT_OPTION = "Y" Then

    &ROUND_OPTION.SetDefault();

    &ROUND_OPTION.Visible = True;

Else

    &ROUND_OPTION.Visible = False;

End-If;
```

File Class

The File class provides methods and properties for reading from and writing to external files.

Declaring a File Object

The File Object is an instance of the File class. A file object is declared using the File data type. For example,

```
Local File &MYFILE;
```

This creates an object &MYFILE of the class File.

Scope of a File Object

A file object can only be instantiated from PeopleCode. This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Component Interface PeopleCode, record field PeopleCode, and so on.

A file object is passed to and returned from PeopleCode functions and methods as a reference, so the file object is never copied; rather, alternate identifiers are used to refer to the same object. Any change made to a file using one identifier has the same effect as it would with any other identifier for that file. In the following code example, two variables, &F1 and F2, are declared by using the data type File. The file is opened using the GetFile built-in function. Next, &F1 is assigned to &F2. This does not copy &F1 to &F2. Instead, &F1 and &F2 both refer to the same object. Therefore, in the last step when &F2 is closed, &F1 is closed too.

```
Local File &F1, &F2;  
  
&F1 = GetFile("somefile.txt", "R");&F2 = &F1;  
  
&F2.Close(); /* Now &F1 is also closed. */
```



For more information, see Object Assignment and GetFile function.

File Layout

PeopleTools supports reading and writing to plain text files, as well as to files that have a format based on a **File Layout** that has been created in the Application Designer.

- If the file is a plain text file, data is read or written using text strings.
- If the file is based on a File Layout, you can use text strings, rowset or record objects.

This greatly simplifies reading, writing and manipulating hierarchical transaction data with PeopleCode.

File layout methods and properties are noted as such in their descriptions.



For more information, see [Using Plain Text Files](#) and [Using File Layouts](#). For more information about creating a file layout, see [File Layout](#).

File Security Considerations

When you're using file objects in PeopleCode that might run on a server, you must be aware of some security concerns. The underlying system doesn't provide security checks on access to the files. Instead, the PeopleCode programs use whatever authority the server process has, and *not* that of the user. This means you must write your PeopleCode program to prevent the user from reading or writing files that they should not. In particular, be especially wary of opening files where any part of the filename is derived from user input.

Recovering from File Access Interruptions

You can use the `GetPosition` or `SetPosition` methods to minimize the loss of work in the event of a system failure during file access. In order to use them and recover from access interruptions, you must access the file in **Update** mode using the `GetFile` built-in function or the `Open` method, specifying the mode parameter **"U"**. You need to use this mode in anticipation of possible interruptions if you want to recover from them later.

To start reading or writing from the beginning of a file, use `SetPosition` to set a read/write position of **0**, immediately after you open the file in Update mode.



In Update mode, any write operation will clear the file of all data that follows the position you set.

When reading from or writing to a file, use `GetPosition` periodically to determine your current byte position in the file. This establishes a checkpoint to which you can return after a failure. You must save the checkpoint value in a separate file or a database so you can retrieve it later. How often you save a checkpoint, depends on your requirements: the less work you want to redo, the more frequent your checkpoints should be. You may also want to save information identifying the data you're reading or writing at the time.

After a failure interrupts your file access, reopen the file in Update mode. You can then retrieve the last checkpoint value you saved, and use `SetPosition` to apply that value. Your read/write position in the file will be the position of the last checkpoint, and you can continue reading or writing from there.

Use the Update mode with `GetPosition` and `SetPosition` only for recovering from interruptions as described here, or for starting at the beginning of the file again.



The effect of the Update mode is not well defined for Unicode files. Use the Update mode only on files stored with the ANSI character set.



For more information, see the Open, SetPosition and GetPosition methods, and the GetFile function.

Using Plain Text Files

To read and write from plain text files involves reading and writing strings of data. These text strings can be manipulated with built-in string functions, like RTrim, Find, Replace, and so on.



If your data is hierarchical in nature, or based on existing PeopleSoft records or pages, you will want to use a File Layout definition for reading and writing your data, rather than doing it line by line (or field by field.) See Using File Layouts.

The following example creates an array of array of string, then reads in two files, one into each "column" of the array. The **Names** file contains names; the **Numbers** file contains employee numbers. The **ReadLine** method will read each successive line in a file, until it reaches the end of the file. Notice that the first file is opened using **GetFile**. The second file is not opened using **GetFile**, but rather with **Open**. After the data is read into the array, you can do processing on the data. The end of the program writes the changes back to the files, using the **WriteLine** method, which includes a system end of line character at the end of every line.

```

Local array of array of string &BOTH;

Local File &MYFILE;

Local string &HOLDER;

/* Create empty &BOTH array */

&BOTH = CreateArrayRept(CreateArrayRept("", 0), 0);

/* Read first file into first column */

&MYFILE = GetFile("names.txt", "R");

While &MYFILE.ReadLine(&HOLDER);

    &BOTH.Push(&HOLDER);

```

```
End-While;

/* read second file into second column */

&MYFILE.Open("numbers.txt", "R");

&LINENO = 1;

While &MYFILE.ReadLine(&HOLDER);

    If &LINENO > &BOTH.Len Then

        /* more number lines than names, use a null name */

        &BOTH.Push(CreateArray("", &HOLDER));

    Else

        &BOTH[&LINENO].Push(&HOLDER);

    End-If;

    &LINENO = &LINENO + 1;

End-While;

/* if more names than numbers, add null numbers */

For &LINENO = &LINENO to &BOTH.Len

    &BOTH[&LINENO].Push("");

End-For;

&MYFILE.Close();

/* do processing with array */

/* write data back to files */

&MYFILE1 = GetFile("names.txt", "A");

&MYFILE2 = GetFile("numbers.txt", "A");

/* loop through array and write to files */
```

```
For &I = 1 To &BOTH.Len  
    &STRING1 = &BOTH[&I][1];  
    &MYFILE1.writeline(&STRING1);  
    &STRING2 = &BOTH[&I][2];  
    &MYFILE2.writeline(&STRING2);  
End-For;  
  
&MYFILE1.Close();  
  
&MYFILE2.Close();
```

Using File Layouts

If your data is hierarchical in nature, or based on existing PeopleSoft records or pages, you will want to use a File Layout definition for reading and writing your data, rather than doing it line by line (or field by field.)



For more information about creating a file layout, see File Layout.

For example, suppose you wanted to write all the information from a record to a file. You can use the **WriteRecord** method to write all the data from the record, instead of having to loop through every field, find the value, and write it to the file.

In addition, you could write all the information from a transaction (or several transactions) from a page to a file. Each transaction can be considered a **rowset**. A rowset can contain more than one record and is generally composed in a hierarchical structure. You could create a File Layout definition that has the same structure as the page (or component), and use the **WriteRowset** method. If you have a file that contains data in the correct format, you can use the **ReadRowset** method to read the data from the file to the page.



For more information about rowsets, see Data Buffer Access.

Each file layout is associated with a **format**. This format specifies the type of data in the files. You specify the format as part of the File Layout Properties. You can only specify one format for a file layout. The following is a list of the available formats:

- FIXED (default)
- CSV

- XML

The file layout methods and properties use this information to handle each file type in a transparent manner. For the most part, you don't need to do anything different based on file type. Any exceptions are noted in the documentation.



Unlike other PeopleTools definitions, records and field are *copied* to a File Layout definition. There are no pointers. This means if you change a record definition (add or remove a field) you will have to change the File Layout definition as well. The changes are not automatically propagated. This is why the documentation refers to these elements as file records, file fields, and so on, to show that they are no longer part of the original definition they were created from.



For more examples writing from and reading to files using File Layout and standalone rowsets, see Using Standalone Rowsets.

WriteRecord Example

In the following example, the File Layout definition is based on the record ABSENCE_HISTORY, and looks like this:



Example File Layout Definition (ABS_HIST)

You should note the following about the using **WriteRecord**:

- Not all the fields in the File Layout definition and the record have to match. The **WriteRecord** method, like all File Layout methods, only pays attention to the *like-named* fields. If there are additional fields in the record or in the File Layout definition, they are ignored.
- The **WriteRecord** method only writes to **like-named** records. If you rename a record after

you use it to create a File Layout definition, you will have to rename it to the exact same name in your File Layout. Because **WriteRecord** writes like-named records, the same file layout definition can contain more than one record.

- The **WriteRecord** method takes a *record object* as its parameter. A populated record object references a *single row* of data in the SQL table. This is why a SQL Fetch statement is used in a condition around the **WriteRecord** method. This fetches every row of data from the SQL table, then writes it to the file.

The following code writes the ABSENCE_HIST record to the file record.txt.

```

Local Record &RecLine;

Local File &MYFILE;

Local SQL &SQL2;

&MYFILE = GetFile("record.txt", "A");

If &MYFILE.IsOpen Then

    If &MYFILE.SetFileLayout(FileLayout.ABS_HIST) Then

        &RecLine = CreateRecord(RECORD.ABSENCE_HIST);

        &SQL2 = CreateSQL("%Selectall(:1)", &RecLine);

        While &SQL2.Fetch(&RecLine)

            &MYFILE.WriteRecord(&RecLine);

        End-While;

    Else

        /* do error processing - filelayout not correct */

    End-If;

Else

    /* do error processing - file not open */

End-If;

&MYFILE.Close();

```

If you only wanted to write changed records to the file, you could add the following code **in bold**:

```

While &SQL2.Fetch(&RecLine)

```

```

If &RecLine.IsChanged Then

    &MYFILE.WriteRecord(&RecLine);

End-If;

End-While;

```

The following is the first part of a sample data file created by the above code. The field format is FIXED:

8001	VAC	09/12/1981	09/26/1981	14	0		P	Y
8001	VAC	03/02/1983	03/07/1983	5	0		P	Y
8001	VAC	08/26/1983	09/10/1983	13	0		P	Y
8105	CNF	02/02/1995	??/??/		0	0	U	N
8516	MAT	06/06/1986	08/01/1986	56	0		P	Y
8516	SCK	08/06/1988	08/07/1988	1	0		P	Y
8516	VAC	07/14/1987	07/28/1987	14	0		P	Y
8553	JUR	12/12/1990	12/17/1990	5	0	Local Jury Duty	P	N
8553	MAT	02/20/1992	10/01/1992	224	0	Maternity Leave	U	N
8553	MAT	08/19/1994	03/01/1995	194	0	Maternity-2nd child	U	Y
8553	PER	04/15/1993	04/19/1993	4	0		U	N
						Personal Day required		
8553	SCK	01/28/1987	01/30/1987	2	0	Hong Kong Flu	P	N
8553	SCK	08/02/1988	08/03/1988	1	0	Sick	P	N
8553	SCK	09/12/1995	09/13/1995	1	0		P	N
8641	VAC	06/01/1988	06/15/1988	14	0		P	Y
8641	VAC	07/01/1989	07/15/1989	14	0		P	Y
G001	MAT	07/02/1991	09/28/1991	88	0	3-month Maternity leave	P	Y
						Maternity will be paid as 80% of Claudia's current salary.		

If a record in the File Layout definition has a File Record ID, each line in the file referencing this record will be prefaced with this number. The File Record ID is not a field in the data itself. It is also referred to as the **rowid**. File Record IDs can be useful when the file you're producing refers to more than one record.

File Record IDs can only be used with File Layout definitions that have a type of FIXED. If a File Layout definition has only one level then File Record IDs can be omitted. But for File Layout definitions with more than one level, you *must* use File Record IDs.

The following is sample file, produced with the same code, but with a File Record ID of 101 added:

101	8001	VAC	09/12/1981	09/26/1981	14	0		P	Y
101	8001	VAC	03/02/1983	03/07/1983	5	0		P	Y
101	8001	VAC	08/26/1983	09/10/1983	13	0		P	Y
101	8105	CNF	02/02/1995	??/??/		0	0	U	N
101	8516	MAT	06/06/1986	08/01/1986	56	0		P	Y
101	8516	SCK	08/06/1988	08/07/1988	1	0		P	Y
101	8516	VAC	07/14/1987	07/28/1987	14	0		P	Y
101	8553	JUR	12/12/1990	12/17/1990	5	0	Local Jury Duty	P	N
101	8553	MAT	02/20/1992	10/01/1992	224	0	Maternity Leave	U	N
101	8553	MAT	08/19/1994	03/01/1995	194	0	Maternity-2nd child	U	Y
101	8553	PER	04/15/1993	04/19/1993	4	0		U	N
			Personal Day required						
101	8553	SCK	01/28/1987	01/30/1987	2	0	Hong Kong Flu	P	N
101	8553	SCK	08/02/1988	08/03/1988	1	0	Sick	P	N
101	8553	SCK	09/12/1995	09/13/1995	1	0		P	N
101	8641	VAC	06/01/1988	06/15/1988	14	0		P	Y
101	8641	VAC	07/01/1989	07/15/1989	14	0		P	Y
101	G001	MAT	07/02/1991	09/28/1991	88	0	3-month Maternity leave	P	Y
			Maternity will be paid as 80% of Claudia's current salary.						



File positions start at 1, not 0. When you specify a File Record ID, make sure that the starting position of the Record ID is greater than 0.

ReadRecord Example

This following example uses the same File Layout definition as the above example. The file format is CSV. The fields are separated by commas and delimited by double-quotes.

```
"8001","VAC","09/12/1981","09/26/1981","14","0","","P","Y",""
"8001","VAC","03/02/1983","03/07/1983","5","0","","P","Y",""
"8001","VAC","08/26/1983","09/10/1983","13","0","","P","Y",""
```



```

"8105","CNF","02/02/1995","??/?/?","0","0","","U","N",""

"8516","MAT","06/06/1986","08/01/1986","56","0","","P","Y",""

"8516","SCK","08/06/1988","08/07/1988","1","0","","P","Y",""

"8516","VAC","07/14/1987","07/28/1987","14","0","","P","Y",""

"8553","JUR","12/12/1990","12/17/1990","5","0","Local Jury Duty","P","N",""

"8553","MAT","02/20/1992","10/01/1992","224","0","Maternity Leave","U","N",""

"8553","MAT","08/19/1994","03/01/1995","194","0","Maternity-2nd
child","U","Y",""

"8553","PER","04/15/1993","04/19/1993","4","0","","U","N","Personal Day
required"

"8553","SCK","01/28/1987","01/30/1987","2","0","Hong Kong Flu","P","N",""

"8553","SCK","08/02/1988","08/03/1988","1","0","Sick","P","N",""

"8553","SCK","09/12/1995","09/13/1995","1","0","","P","N",""

"8641","VAC","06/01/1988","06/15/1988","14","0","","P","Y",""

"8641","VAC","07/01/1989","07/15/1989","14","0","","P","Y",""

"G001","MAT","07/02/1991","09/28/1991","88","0","3-month Maternity
leave","P","Y","Maternity will be paid as 80% of Claudia's current salary."

```

To read in the above CSV file we use the following PeopleCode. It reads the file into a temporary record. First each line of the file is read into a string. The string is split into an array, with the value of each field in the array becoming an element in the array. The value of each field in the record is assigned a value from the array. After additional processing (for example, converting strings into dates or numbers, verifying data, and so on) the record can be inserted into the database. In order to insert the final data into the database, this code must be associated with a PeopleCode event that allows database updates, that is, SavePreChange, WorkFlow, SavePostChange, and so on. This code could also be used as part of an Application Engine program.

```

Local File &MYFILE;

Local Record &REC;

Local array of string &ARRAY;

&MYFILE = GetFile("c:\temp\vendor.txt", "R", %FilePath_Absolute);

&REC = CreateRecord(RECORD.ABS_HIST_TEST);

&ARRAY = CreateArrayRept("", 0);

```

```
If &MYFILE.IsOpen Then

    If &MYFILE.SetFileLayout(FILELAYOUT.ABS_HIST) Then

        While &MYFILE.ReadLine(&STRING);

            &ARRAY = Split(&STRING, ",");

            For &I = 1 To &REC.FieldCount

                &REC.GetField(&I).Value = &ARRAY[&I];

            End-For;

            /* do additional processing here for converting values */

            &REC.Insert();

        End-While;

    Else

        /* do error processing - filelayout not correct */

    End-If;

Else

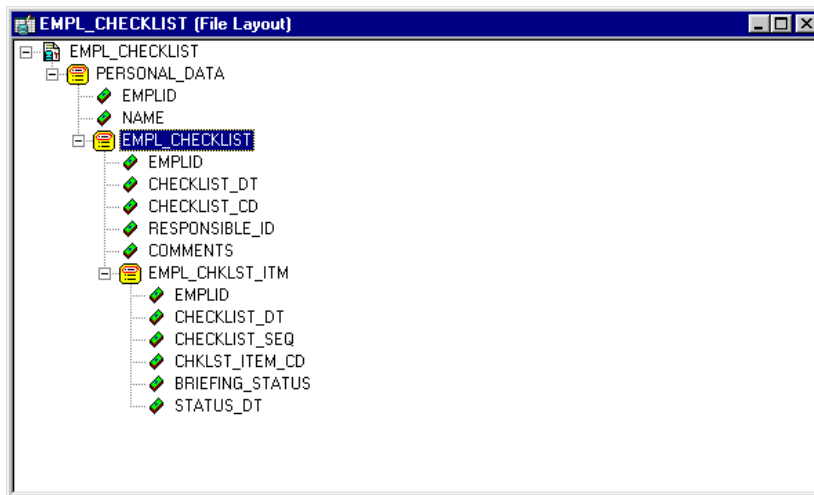
    /* do error processing - file not open */

End-If;

&MYFILE.Close();
```

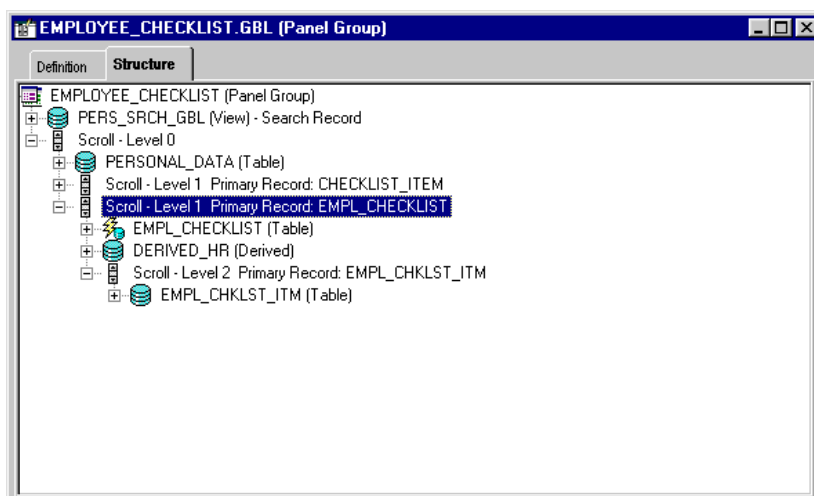
WriteRowset Example

In the following example, the File Layout definition is based on the component EMPL_CHECKLIST, and looks like this:



Example File Layout Definition (EMPL_CHECKLIST)

Here's the structure of the component EMPLOYEE_CHECKLIST:



EMPLOYEE_CHECKLIST Component Structure

It's important to note the following:

- Every **field** in the two structures don't have to match (that is, every field or record that's in the file layout doesn't have to be in the component, and vice versa.)
- The two **structures** must be the same. That is, if the component has PERSONAL_DATA at Level 0, and EMPL_CHECKLIST at Level 1, the file layout must have the same hierarchy.

The following example uses the above File Layout definition to copy data from the EMPL_CHECKLIST page into a file.

The **CreateRowset** function creates an empty rowset that has the structure of the file layout definition. The **GetRowset** function is used to get all the data from the component and copy it into the rowset. The **GetLevel0** function copies all **like-named** fields to **like-named** records.

The **WriteRowset** method writes all the component data to the file. Because this code is associated with a SavePreChange event, it will run on the server so an absolute file path is used.

```

Local File &MYFILE;

Local Rowset &FILEROWSET;

&MYFILE = GetFile("c:\temp\EMP_CKLS.txt", "A", %FilePath_Absolute);

If &MYFILE.IsOpen Then

    If &MYFILE.SetFileLayout(FILELAYOUT.EMPL_CHECKLIST) Then

        &FILEROWSET = &MYFILE.CreateRowset();

        &FILEROWSET = GetLevel0();

        &MYFILE.WriteRowset(&FILEROWSET);

    Else

        /* file layout not found, do error processing */

    End-If;

Else

    /* file not opened, do error processing */

End-if;

&MYFILE.Close();

```

The following is a sample data file created by the above code:

```

8113      Frumman,Wolfgang

          08/06/1999 000001      8219      Going to London office

          100      000015 I 08/06/1999

          200      000030 I 08/06/1999

          300      000009 I 08/06/1999

          400      000001 I 08/06/1999

          500      000011 I 08/06/1999

          600      000002 I 08/06/1999

          700      000021 I 08/06/1999

          800      000024 I 08/06/1999

```

```

          900      000004 I 08/06/1999

          1000     000006 I 08/06/1999

09/06/1999 000004      7707      What to do after he arrives

          100      000022 I 08/06/1999

          200      000008 I 08/06/1999

          300      000018 I 08/06/1999

          400      000019 I 08/06/1999

8101      Penrose, Steven

          07/06/1999 000006      8229      New hire

          1          000033 I 08/06/1999

          2          000034 I 08/06/1999

          3          000035 I 08/06/1999

          4          000036 I 08/06/1999

          5          000037 I 08/06/1999

          6          000038 I 08/06/1999

          7          000039 I 08/06/1999

          8          000040 I 08/06/1999

          9          000041 I 08/06/1999

         10          000042 I 08/06/1999

```

When you create the File Layout definition, you can choose for each field whether to inherit a value from a higher level record field or not. If a value is inherited, it is only written *once* to the file, the first time it's encountered.

In the above data example, there are three records: PERSONAL_DATA, EMPL_CHECKLIST, and EMPL_CHKLST_ITM. The field EMPLID is on all three records, but is only written to the file the first time a new value is encountered. So, the value for EMPLID is inherited by EMPL_CHECKLIST, and the value for EMPL_CHKLST_ITM is inherited from PERSONAL_DATA. The field CHECKLIST_DT is on EMPL_CHECKLIST and EMPL_CHKLST_ITM. However, the field value only appears once, in the parent record, and not in the child record.

If a record in the File Layout definition has a File Record ID, each line in the file referencing this record will be prefaced with this number. The File Record ID is not a field in the data itself. It is also referred to as the *rowid*. File Record IDs can be useful when the file being created refers to more than one record. File Record IDs can only be used with File Layout definitions that have a type of FIXED.

The following is sample file, produced with the same code, but with a File Record IDs added to all the records. 001 was added to the first level, 002 to the second, and 003 to the third.

001	8113	Frumman, Wolfgang	
002	08/06/1999	000001 8219	Going to London office
003		100	000015 I 10/13/1999
003		200	000030 I 10/13/1999
003		300	000009 I 10/13/1999
003		400	000001 I 10/13/1999
003		500	000011 I 10/13/1999
003		600	000002 I 10/13/1999
003		700	000021 I 10/13/1999
003		800	000024 I 10/13/1999
003		900	000004 I 10/13/1999
003		1000	000006 I 10/13/1999
002	09/06/1999	000004 7707	What to do after he arrives
003		100	000022 I 10/13/1999
003		200	000008 I 10/13/1999
003		300	000018 I 10/13/1999
003		400	000019 I 10/13/1999
001	8101	Penrose, Steven	
002	10/13/1999	000006 8229	New hire
003		1	000033 I 10/13/1999
003		2	000034 I 10/13/1999
003		3	000035 I 10/13/1999
003		4	000036 I 10/13/1999
003		5	000037 I 10/13/1999
003		6	000038 I 10/13/1999
003		7	000039 I 10/13/1999
003		8	000040 I 10/13/1999
003		9	000041 I 10/13/1999

003

10

000042 I 10/13/1999

ReadRowset Example

The following program reads all the rowsets from a file and updates a work scroll with that data. The work scroll isn't hidden on the page: it's created in the Component buffer from existing records using CreateRowset. The structure of the work scroll and the file layout are identical: that is, they're composed of two records, and the names of the records in the file layout are exactly the same as the names of the record definitions.

```

Local File &CHARTINPUT_F;

Local Rowset &INPUT_ROWSET, &TEMP_RS, &WORK_DATA;

Local Record &WRK_DATA;

&filename = "c:\temp\test.txt";

If FileExists(&filename, %FilePath_Absolute) Then

    &CHARTINPUT_F = GetFile(&filename, "R", "A", %FilePath_Absolute);

Else

    Exit;

End-If;

&CHARTINPUT_F.SetFileLayout(FileLayout.CHART_INFO);

/* Create rowset to be read into
NOTE that you have to start at LOWEST level of rowset */

&TEMP_RS = CreateRowset(RECORD.CHART_ITEM);
&WORK_DATA = CreateRowset(RECORD.CHART_DATA, &TEMP_RS);
&INPUT_ROWSET = CreateRowset(RECORD.CHART, &WORK_DATA);

While &INPUT_ROWSET <> Null

    &INPUT_ROWSET = &CHARTINPUT_F.ReadRowset();

    &INPUT_ROWSET.CopyTo(&WORK_DATA);

```

```

/* do processing -- Though file may contain more than one level 0
Component processor only allows one level 0 at a time */

End-While;

```

File Rowset Considerations

- Although you can create a File Layout definition with more than four levels of hierarchy, a rowset created from Component buffer data can only contain four levels (level 0 through 3). Any additional levels of data will be ignored.
- **ReadRowset** populates the rowset with *one* transaction from the file. (A transaction is considered to be one instance of level zero data contained in a file record, plus all of its subordinate data.) **WriteRowset** writes one transaction to a file.

Handling Multiple File Layouts

In the above examples, the input file contained rowsets based on a single File Layout definition. However, PeopleTools provides the functionality to process input files containing rowsets that require several *different* File Layouts.



You can only use FIXED format files to implement multiple file layouts.

Reading Multiple File Layouts

If your input file contains data based on more than one File Layout, it must contain an indicator, called a **FileId** that specifies:

- when a different File Layout definition should be used
- which File Layout definition should be used

The FileId must be specified on a separate line and must precede *every* rowset that requires a layout different from the previous rowset. It isn't considered part of the rowset.

In the following example, the file contains two FileId lines; they use a file record ID that distinguishes them from the rowset data—in this case, "999".

```

999 PRODUCT /* The following rowset uses the PRODUCT layout */

001 /* Level 0 record data */

101 /* Level 1 record data */

201 /* Level 2 record data */

201 /* Level 2 record data */

```



```

999 ORDER  /* The following two rowsets use the ORDER layout */

001  /* Level 0 record data */

111  /* Level 1 record data */

111  /* Level 1 record data */

001  /* Level 0 record data */

111  /* Level 1 record data */

111  /* Level 1 record data */

```

The FileId can contain any information you want that indicates which file layout to use; the "PRODUCT" and "ORDER" fields shown are just examples.

6. To read this file, you should do the following in your program:
7. Use the SetFileId method to specify the file record ID value.
8. Use the ReadRowset method to read the data.
9. Check if the rowset is NULL.

NULL indicates you've reached either the end of the file or a new rowset.

10. Use the IsNewFileId property to check if the next line is a **FileId** file record (line).

- If IsNewFileId is False, you've reached the end of the file.
- If IsNewFileId is True, use the **CurrentRecord** property to determine which File Layout to use next.



For more information about the details of handling multiple file layouts, see the SetFileId method.

The following example reads rowsets from a file. When it finds a new rowset (indicated by &IsNewFileId returning True) the value of &CurrentRecord is passed to a function that reads and evaluates the line, then returns the name of the new file layout. (The code for the function FindFileId is included at the start of the example.)

```

Local File &MYFILE;

Local Rowset &rsFile;

Local Record &rSomeRec1, &rSomeRec2;

Local SQL &SQL1;

Function FindFileID(&CurrentRecord As string) Returns string ;

```

```
Evaluate RTrim(Substring(&CurrentRecord, 5, 50))

When "SOME_REC1"

    &FILELAYOUT = "SOME_REC1";

When "SOME_REC2"

    &FILELAYOUT = "SOME_REC2";

End-Evaluate;

Return &FILELAYOUT;

End-Function;


&rSomeRec1 = CreateRecord(Record.SOME_REC1);

&rSomeRec2 = CreateRecord(Record.SOME_REC2);

&SQL1 = CreateSQL("%Insert(:1)");

&MYFILE = GetFile("c:\temp\MULTI_FILE.out", "R", %FilePath_Absolute);


rem Set temporary first file layout;

&MYFILE.SetFileLayout(FileLayout.SOME_REC1);

&MYFILE.SetFileId("999", 1);


rem Read rowset to find actual first file ID;

&rsFile = &MYFILE.ReadRowset();

&CurrentRecord = &MYFILE.CurrentRecord;

&IsNewFileID = &MYFILE.IsNewFileId;

If &MYFILE.IsNewFileId Then

    &FILELAYOUT = FindFileID(&CurrentRecord);

    &MYFILE.SetFileLayout(@"FileLayout." | &FILELAYOUT);

    &MYFILE.SetFileId("999", 1);

End-If;


rem Read first 'real' rowset;
```

```

&rsFile = &MYFILE.ReadRowset();

&CurrentRecord = &MYFILE.CurrentRecord;

&IsNewFileID = &MYFILE.IsNewFileId;

While &rsFile <> Null Or

    &IsNewFileID

    If &MYFILE.IsNewFileId Then

        &FILELAYOUT = FindFileID(&CurrentRecord);

        &MYFILE.SetFileLayout(@"FileLayout." | &FILELAYOUT);

        &MYFILE.SetFileId("999", 1);

        If &IsNewFileID Then

            &rsFile = &MYFILE.ReadRowset();

            &CurrentRecord = &MYFILE.CurrentRecord;

            &IsNewFileID = &MYFILE.IsNewFileId;

        End-If;

    End-If;

End-If;

Evaluate &FILELAYOUT

When "SOME_REC1"

    &rsFile(1).SOME_REC1.CopyFieldsTo(&rSomeRec1);

    &rSomeRec1.ExecuteEdits(%Edit_Required);

    If Not &rSomeRec1.IsEditError Then

        &SQL1.Execute(&rSomeRec1);

    End-If;

    Break;

When "SOME_REC2"

    &rsFile(1).SOME_REC2.CopyFieldsTo(&rSomeRec2);

    &rSomeRec2.ExecuteEdits(%Edit_Required);

    If Not &rSomeRec2.IsEditError Then

```

```

        &SQL1.Execute (&rSomeRec2);

    End-If;

End-Evaluate;


&rsFile = &MYFILE.ReadRowset();

&CurrentRecord = &MYFILE.CurrentRecord;

&IsNewFileID = &MYFILE.IsNewFileId;

End-While;


&MYFILE.Close();

```

Writing Multiple File Layouts

If you're writing files that contain data based on more than one File Layout definition, you need to consider the following:

- If the file is going to a third-party vendor, you should work with the third-party to determine what their requirements are for specifying the different data formats.
- If the file is going to be used by another PeopleSoft system, you must add the **FileId** between each rowset that requires a different layout. FileId file records are *not* part of any rowset. They should be designed so they won't be mistaken for part of a rowset. You can create and write them to the file in many ways. The following are suggestions:
 - Build each line as a string, using any of the built-in string manipulation functions, then write them to the file using the File class WriteLine or WriteString methods.
 - Design a file layout consisting of a single file record definition for the FileId file records, then build the records using ProcessRequest Class methods and functions, and write them to the file using the WriteRecord method.

The following code example writes each record from the level 1 scroll on a page to the file using a different File Layout. Between each WriteRowset the File ID file record is written to the file, describing the new File Layout being used.

```

Local File &MYFILE;

Local Rowset &FILEROWSET;

Local Record &REC1, &REC2;

Local SQL &SQL;


&MYFILE = GetFile("c:\temp\Records.txt", "W", %FilePath_Absolute);

```

```

If &MYFILE.IsOpen Then

    If &MYFILE.SetFileLayout(FileLayout.TREE_LEVEL) Then

        &REC1 = CreateRecord(Record.PSTREELEVEL);

        &FILEROWSET = &MYFILE.CreateRowset();

        &SQL = CreateSQL("%Selectall(:1)", &REC1);

        /* write first File ID to file */

        &MYFILE.WriteLine("999 FILE LAYOUT 1");

        While &SQL.Fetch(&REC1)

            &REC1.CopyFieldsTo(&FILEROWSET.GetRow(1).PSTREELEVEL);

            &MYFILE.WriteRowset(&FILEROWSET);

        End-While;

    Else

        /* file layout not found, do error processing */

    End-If;

    If &MYFILE.SetFileLayout(FileLayout.TREE_USERLEVEL) Then

        &REC2 = CreateRecord(Record.TREE_LEVEL_TBL);

        &FILEROWSET = &MYFILE.CreateRowset();

        &SQL = CreateSQL("%Selectall(:1)", &REC2);

        /* write second File ID to file */

        &MYFILE.WriteLine("999 FILE LAYOUT 2");

        While &SQL.Fetch(&REC2)

            &REC2.CopyFieldsTo(&FILEROWSET.GetRow(1).TREE_LEVEL_TBL);

            &MYFILE.WriteRowset(&FILEROWSET);

        End-While;

    Else

        /* file layout not found, do error processing */

    End-If;

Else

```

```

        /* file not opened, do error processing */

End-If;

&MYFILE.Close();

```

Application Engine Example

You can also use PeopleCode in an Application Engine program to either write to or read from files. This example isn't a proper Application Engine program: it only contains the PeopleCode for opening and reading from a file. However, it's included here as starting point for your own Application Engine programs.



For more information, see *Introducing Application Engine*.

Here is the Application Engine program:

The screenshot shows the 'Definition' tab of the 'FILE_TO_TBL (App Engine Program)' window. It displays a hierarchical structure with a 'MAIN' section containing a 'Step01' step, which in turn contains a 'PeopleCode' object. The 'Step01' step has fields for 'Commit After', 'Frequency', 'On Error', and 'Default', with 'Active' checked. The 'PeopleCode' object has fields for 'On Return' and 'Skip Step'.

Section	Step	Action
MAIN	Step01	PeopleCode

Application Engine example program

Here is the PeopleCode in the step 1.

```

Local File &FILE;

Local Record &REC;

Local Rowset &FRS;

&FILE = GetFile("TEST.txt", "R");

&REC = CreateRecord(Record.QEPC_FILE_REC);

&SQL = CreateSQL("%Insert(:1)");

If Not &FILE.IsOpen Then

```

```

        Error ("TEST: failed file open");
    Else
        If Not &FILE.SetFileLayout(FileLayout.QEPC_FILE_REC) Then
            Error ("TEST: failed SetFilelayout");
        Else
            &FRS = &FILE.ReadRowset();

            While &FRS <> Null

                &FRS.GetRow(1).QEPC_FILE_REC.CopyFieldsTo(&REC);

                &SQL.execute(&REC);

                &FRS = &FILE.ReadRowset();

            End-While;

        End-If;

        &FILE.Close();

    End-If;

```

The example Application Engine program reads the following CSV file:

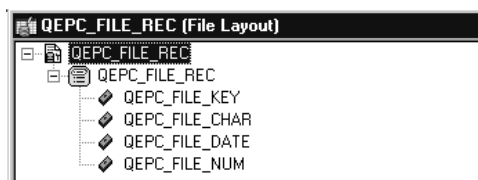
```

"TEST2","1ST","01/01/1901",10
"TEST2","2ND","01/01/1902",20
"TEST2","3RD","01/01/1903",30
"TEST2","4TH","01/01/1904",40

```

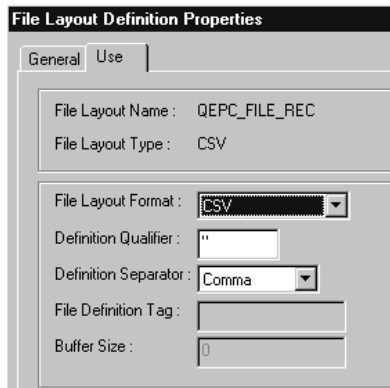
Note that the last field has no qualifier.

The File Layout used to read this record has the following form:



QEPC_FILE_REC File Layout

The properties for the QEPC_FILE_REC File Layout are as follows:



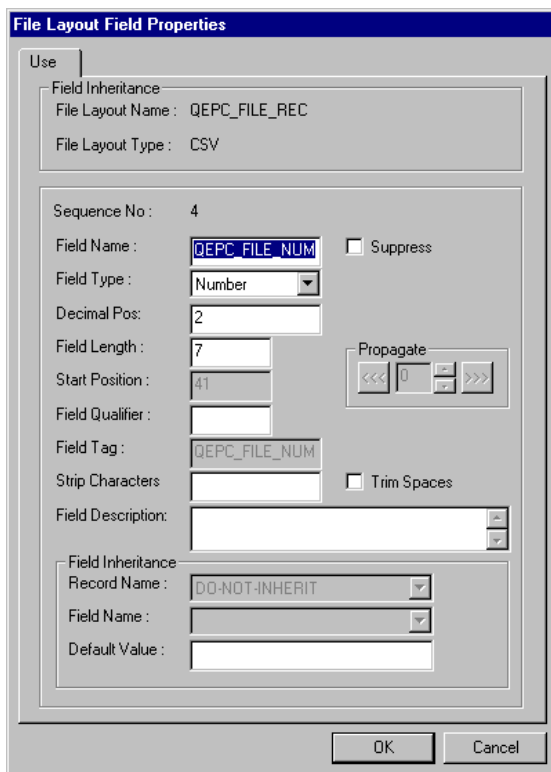
The **File Layout Definition Properties** dialog box has two tabs: **General** and **Use**. The **General** tab is active. It contains the following fields:

- File Layout Name :** QEPC_FILE_REC
- File Layout Type :** CSV
- File Layout Format :** CSV (dropdown menu)
- Definition Qualifier :** " (text field)
- Definition Separator :** Comma (dropdown menu)
- File Definition Tag :** (text field)
- Buffer Size :** 0 (text field)

File Layout Definition properties

Note that the Definition Qualifier is double-quotes ("), while the separator is a comma.

Remember, the last field didn't have a qualifier. To account for that, the field properties for this field need to be edited, and a blank needs to be put in the **Field Qualifier** property.



The **File Layout Field Properties** dialog box has a **Use** tab. It contains the following fields and controls:

- Field Inheritance** section:
 - File Layout Name :** QEPC_FILE_REC
 - File Layout Type :** CSV
- Sequence No :** 4
- Field Name :** QEPC_FILE_NUM (text field)
- Field Type :** Number (dropdown menu)
- Decimal Pos :** 2 (text field)
- Field Length :** 7 (text field)
- Start Position :** 41 (text field)
- Field Qualifier :** (text field)
- Field Tag :** QEPC_FILE_NUM (text field)
- Strip Characters :** (text field)
- Field Description :** (text area)
- Field Inheritance** section:
 - Record Name :** DO-NOT-INHERIT (dropdown menu)
 - Field Name :** (dropdown menu)
 - Default Value :** (text field)
- Controls:**
 - ☐ Suppress
 - Propagate:** <<< 0 >>> (buttons)
 - ☐ Trim Spaces

At the bottom are **OK** and **Cancel** buttons.

File layout Field Properties

Handling File Layout Errors

If an error occurs on any field in any record of a rowset object populated with the ReadRowset method, the rowset object's IsEditError property will return True. For example, you can use the

method `ExecuteEdits` on a record, to verify that the data in the record is valid (has the correct format, the right data type, and so on.) This type of error is indicated by the `IsEditError`.

In some instances, however, the rowset object won't receive the error; in that case the file object's `IsError` property will return `True`. To discover all field errors, check both properties after executing `ReadRowset`.

To determine which field has the error, you must examine the `EditError` property of every field in the rowset to find the one returning `True`. You can then examine that field's `MessageSetNumber` and `MessageNumber` properties to determine the relevant error message. The following example shows how this might be done:

```
&MYFILE.Open(&SOMENAME, "R");

&MYFILE.SetFileLayout(FILELAYOUT.SOMELAYOUT);

&MYROWSET = &MYFILE.ReadRowset();

If &MYFILE.IsError Then

  For &I = 1 to &MYROWSET.ActiveRowCount

    If &MYROWSET.GetRow(&I).IsEditError Then

      &ROW = &MYROWSET.GetRow(&I);

      For &J = 1 to &ROW.RecordCount

        If &ROW.GetRecord(&J).IsEditError Then

          &REC = &ROW.GetRecord(&J);

          For &K = 1 to &REC.FieldCount

            If &REC.GetField(&K).EditError Then

              /* Examine the field's
                 MessageSetNumber and MessageNumber properties,
                 and respond accordingly */

              End-If;

            End-For;

          End-If;

        End-For;

      End-If;

    End-For;

  End-If;

End-For;

End-If;
```

```
&MYFILE.Close();
```



Only field errors will set the `IsError`, `IsEditError` or `EditError` properties. All other errors triggered by File class methods will terminate your PeopleCode program.



For more information, see `ExecuteEdits` record class method, `ExecuteEdits` message class method, `MessageNumber` and `MessageSetNumber` field class properties, as well as `ReadRowset` and `IgnoreInvalidId`.

File Class Built-In Functions

`FileExists`

`FindFiles`

`GetFile`

File Class Methods

Close

Syntax

```
Close()
```

Description

The **Close** method discards any changes that haven't been written to the external file, disassociates the file object from the external file, and releases all resources connected with the file object and goes out of scope. You cannot use any methods or properties on the object once it is closed. You must get another file object (using **GetFile**) and instantiate another file object after you use **Close**.

Parameters

None.

Returns

None.

Example

```
&MYFILE.Open("somefile.txt", "W", %FilePath_Relative);
```

```
&MYFILE.WriteLine("Some text.");  
  
&MYFILE.Close();
```

Related Topics

Open, IsOpen property, and GetFile function

CreateRowset

Syntax

```
CreateRowset()
```

Description

The **CreateRowset** method is a file layout method. It instantiates a PeopleCode rowset object containing one unpopulated row, based on the file layout definition specified with SetFileLayout.



You must specify a file layout using the SetFileLayout method before using **CreateRowset**, otherwise it will return NULL.

Once the empty rowset object has been created, you can use Rowset class methods such as Select or Fill, and built-in functions such as GetLevel0 or GetRowset, to populate the rowset with data. After the rowset contains data, you can use the method WriteRowset to write the data to a file.



For more information about Rowset class methods, see Rowset Class.

Don't use **CreateRowset** when reading from a file. Instead, use the ReadRowset method to instantiate and populate rowsets with data from the file.

Parameters

None.

Returns

An empty rowset object.

Example

```
&SOMEROWSET = &MYFILE.CreateRowset();
```

The following rowset is created from a file object, then populated with data using the GetLevel0 function:

```
&FILEROWSET = &MYFILE.CreateRowset();
```

```
&FILEROWSET = GetLevel0();  
  
&MYFILE.WriteRowset (&FILEROWSET);
```

The following rowset is created from a file object, then populated with data using Fill method:

```
&FILEROWSET = &MYFILE.CreateRowset();  
  
&NUM_READ = &FILEROWSET.Fill("where MYRECORD = :1", &UVAL);
```

Related Topics

ReadRowset, SetFileLayout, WriteRowset

GetPosition

Syntax

```
GetPosition()
```

Description

The **GetPosition** method returns the current read or write position in the external file. GetPosition will only work with a file that was opened in Update mode. This method is designed to be used in combination with the SetPosition method to establish checkpoints for restarting during file access.



For more information about the appropriate use of GetPosition, see Recovering from File Access Interruptions.



The effect of the Update mode is not well defined for Unicode files. Use the Update mode and **GetPosition** only on files stored with the ANSI character set.

Parameters

None.

Returns

A number representing the current read or write position in the file.



When you use ReadRowset to read from a file in Update mode, the CurrentRecord property returns, the entire record just read. The current read/write position will be at the *end* of the CurrentRecord, that is, just past the end of the rowset.

Example

The following example opens a file in Update mode, and saves the current position after each read operation:

```
&MYFILE.Open(&SOMENAME, "U");

If &MYFILE.IsOpen Then

    while &MYFILE.ReadLine(&SOMESTRING)

        &CURPOS = &MYFILE.GetPosition();

        /* Save the value of &CURPOS after each read,

           and process the contents of each &SOMESTRING */

    End-While;

End-If;

&MYFILE.Close();
```

Related Topics

Open, SetPosition, and GetFile built-in function

Open

Syntax

```
Open(filespec, mode [, charset] [, pathtype])
```

Description

The **Open** method associates the file object with an external file for input or output.

You can use the GetFile built-in function to access an external file, but each execution of GetFile instantiates a new file object. If you plan to access only one file at a time, you only need one file object. Use GetFile to instantiate a file object for the first external file you access, then use **Open** to associate the same file object with as many different external files as you want. You'll need to instantiate multiple file objects with GetFile only if you expect to have multiple files open at the same time.

If the file object is currently associated with a file, that file will first be closed.

Parameters

<i>filespec</i>	Specify the name, and optionally, the path, of the file you want to open.
-----------------	---

mode

A string indicating the manner in which you want to access the file. The mode can be one of the following:

"R" Read mode: opens the file for reading, starting at the beginning.

"W" Write mode: opens the file for writing.



When you specify Write mode, any existing content in the file is discarded and will be overwritten.

"A" Append mode: opens the file for writing, starting at the end. Any existing content is retained.

"U" Update mode: opens the file for reading or writing, starting at the end. Any existing content is retained. Use this mode and the `GetPosition` and `SetPosition` methods to maintain checkpoints of the current read/write position in the file.



In Update mode, any write operation will clear the file of all data that follows the position you set.



Currently, the effect of the Update mode and the `GetPosition` and `SetPosition` methods is not well defined for Unicode files. Use the Update mode only on files stored with the ANSI character set.



For more information, about using Update mode, see [Recovering from File Access Interruptions](#).

"E" Conditional "exist" read mode: opens the file for reading only if it exists, starting at the beginning. If it doesn't exist, **Open** has no effect. Before attempting to read from the file, use the `IsOpen` property to confirm that it's open.

"N" Conditional "new" write mode: opens the file for writing, only if it doesn't already exist. If a file by the same name already exists, **Open** has no effect. Before attempting to write to the file, use the `IsOpen` property to confirm that it's open.

You can insert an asterisk (`*`) in the filename to ensure that a new file is created. The system replaces the asterisk with numbers starting at **1** and incrementing by 1, and checks for the existence of a file by each resulting name in turn. It uses the first name for which a file *doesn't* exist. In this way you can generate a set of automatically numbered files. If you insert more than one asterisk, all but the first one are discarded.

charset

A string indicating the character set you expect when you read the file, or the character set you want to use when you write to the file. You can abbreviate ANSI to "A" and Unicode to "U". All other character sets must be spelled out in full, for example, ASCII, THAI, or UTF8. If you don't specify a character set, the default value is ANSI. You can also use a record field value to specify the character set (for example, `RECORD.CHARSET`.)



If you attempt to read data from a file using a different character set than was used to write that data to the file, the methods used will generate a runtime error or the data returned will be unusable.

pathtype

If you have prepended a path to the file name, use this parameter to specify whether the path is an absolute or relative path. The valid values for this parameter are:

- `%FilePath_Relative` (default)
- `%FilePath_Absolute`

If you don't specify *pathtype* the default is `%FilePath_Relative`.

If you don't specify a path, the file is assumed to be in one of two locations, depending on where your PeopleCode program is executing.

- When your program is executing on the client, the location is the directory specified by the **TEMP** environment variable.
- When your program is executing on the server, the location is the "**files**" directory under the directory specified by the PeopleSoft **PS_SERVDIR** environment variable.



Your system security should verify if a user has the correct permissions before allowing access to a drive (for example, if a user changes their **TEMP** environmental variable to a network drive they don't normally have access to, your system security should detect this.)

If you specify a relative path, that path will be appended to the TEMP or PS_SERVDIR variable (depending on where your PeopleCode is executing.) You must make sure to not include a drive letter when using a relative path.

If the path is an absolute path, whatever path you specify is used verbatim. You must specify a drive letter as well as the complete path. You can't use any wildcards when specifying a path.

The Component Processor automatically converts platform-specific separator characters to the appropriate form for where your PeopleCode program is executing. On a WIN32 system, UNIX "/" separators are converted to "\", and on a UNIX system, WIN32 "\" separators are converted to "/".



The syntax of the file path does *not* depend on the file system of the platform where the file is actually stored; it depends only on the platform where your PeopleCode is executing.



For more information about setting a program's processing location, see PeopleCode and PeopleSoft Internet Architecture..

Returns

None.

Example

The following example opens an existing Unicode file for reading:

```
&MYFILE.Open(&SOMENAME, "E", "U");

If &MYFILE.IsOpen Then

    while &MYFILE.ReadLine(&SOMESTRING)

        /* Process the contents of each &SOMESTRING */

    End-While;

    &MYFILE.Close();

End-If;
```

The following example opens a numbered file for writing in ANSI format, without overwriting any existing files:

```
&MYFILE.Open("C:\temp\item*.txt", "N", %FilePath_Absolute);

If &MYFILE.IsOpen Then

    &MYFILE.WriteLine("Some text.");

    &MYFILE.Close();

End-If;
```

Related Topics

Close, IsOpen property, and GetFile function

ReadLine

Syntax

```
ReadLine(string)
```

Description

The **ReadLine** method reads one line of text from the external file. The line includes the newline character used on the platform where the file is stored, but **ReadLine** strips out the newline character and inserts the result into the string variable *string*.

When **ReadLine** is executed, it moves the starting point for the next read operation to the end of the text just retrieved, so each line in the file can be read in turn by subsequent **ReadLine** operations. When no more data remains to be read from the file, **ReadLine** returns False, and clears the string variable is of any content.

Parameters

<i>string</i>	A string variable that will receive the input text.
---------------	---

Returns

A boolean value: True if the method succeeds, False otherwise. The return value is **not** optional, it is required.

Example

The following example reads a file called `&MYFILE` and puts each line as a separate element in an array.

```
Local File &MYFILE;  
  
Local array of string &MYARRAY;  
  
Local string &TEXT;  
  
  
&MYFILE = GetFile("names.txt", "R");  
  
&MYARRAY = CreateArrayRept("", 0);  
  
While &MYFILE.ReadLine(&TEXT);  
  
    &MYARRAY.Push(&TEXT);  
  
End-While;  
  
&MYFILE.Close();
```

Related Topics

ReadRowset, WriteString, WriteLine

ReadRowset

Syntax

ReadRowset ()

Description

The **ReadRowset** method is a file layout method. It instantiates a PeopleCode rowset object based on the file layout definition, then populates the rowset with one transaction from the file. A transaction is considered to be one instance of level zero data contained in a file record, plus all of its subordinate data.



You must specify a file layout using the `SetFileLayout` method before using **ReadRowset**, otherwise it will return NULL.

When **ReadRowset** is executed, it moves the starting point for the next read operation to the beginning of the next rowset, so each transaction in the file can be read in turn by subsequent **ReadRowset** operations. When no more data remains to be read from the file, **ReadRowset** returns NULL.

If you're using the `SetFileId` method with `ReadRowset` to process an input file based on multiple file layouts, `ReadRowset` returns NULL when it reads a FileId file record (line) between the rowsets.

When `ReadRowset` returns NULL, you can use the `IsFileId` property to determine if you've reached the end of the file or a FileId record.



For more information, see [Handling Multiple File Layouts](#).

If the **ReadRowset** encounters a line in the file containing the FileId, and the lines following this are *not* a new rowset, the process considers it to be an invalid FileId. You can specify whether to ignore the invalid record or terminate your PeopleCode with the `IgnoreInvalidId` property.



If you're using the `SetFileId` method with **ReadRowset** to process an input file based on multiple layouts, FileId file records between the rowsets are considered to be valid file records, and won't generate any errors, regardless of the state of the `IgnoreInvalidId` property.



For more information about `ReadRowset` error handling, see [Handling File Layout Errors](#) and `IsEditError` Rowset class property. For more information about `ReadRowset`, see [ReadRowset Example](#).

Considerations for using dates with `ReadRowset`

Single digits in dates in the form MMDDYY or MMDDYYYY must be padded with zeros. That is, if the date in your data is February 3, 2000, the form must be:

02/03/2000

or

02/03/00

The following is *not* valid.

2/3/00

Parameters

None.

Returns

A populated rowset.

Example

The following example reads and processes an entire file. The data in the file is based on a single File layout definition:

```
&MYFILE.GetFile(&SOMENAME, "R");

If &MYFILE.SetFileLayout(FILELAYOUT.SOMELAYOUT) then

    &SOMEROWSET = &MYFILE.ReadRowset();

    While &SOMEROWSET <> NULL

        /* Process the contents of each &SOMEROWSET */

        &SOMEROWSET = &MYFILE.ReadRowset();

    End-While;

End-If;

&MYFILE.Close();
```

Related Topics

CurrentRecord, IgnoreInvalidId, ReadLine, SetFileLayout, WriteRecord, WriteRowset

SetFileId

Syntax

```
SetFileId(fileid, position)
```

Description

The **SetFileId** method is a file layout method. If your input file contains data based on more than one File Layout definition, you must use this method in combination with the CurrentRecord and IsNewFileId properties and the ReadRowset method to process the file correctly.



SetFileId, CurrentRecord and IsNewFileId don't apply to CSV and XML format input files. You can only use FIXED format files to implement multiple file layouts.

At each point in the input file where the structure of the rowset changes, there must be an extra line in the file containing the file record (line), the **FileId** file record (line), that signifies that the change. Each FileId file record must have the following components:

- A value that designates it as a FileId. Only one FileId value is necessary throughout the file, such as "999".
- A name field that identifies the file layout needed for the rowset that follows. This field could contain the name of the file layout as it's defined in Application Designer.

These lines containing the FileId are *not* part of any rowset. They can contain other information, which will be disregarded by the system. The FileId identifier and the file layout names aren't automatically stored anywhere; they only exist in the input file's FileId file records and in your PeopleCode.

To process an input file that requires multiple file layouts:

1. Use SetFileLayout to specify the file layout definition to use.

If you're specifying the initial file layout for this file, it doesn't have to be the correct one. You'll be able to determine the correct file layout to use for the first rowset during a subsequent step.

2. Use **SetFileId** to specify the value of the FileId field, *fileid*, that's used throughout the file to designate FileId file records.



After each execution of SetFileLayout, the **SetFileId** setting is disabled. Be sure to re-specify the FileId value if you expect the file layout to change, so the system continues looking for the next FileId file record.

3. Use ReadRowset to read the next rowset from the file.

The current file record is also stored in the CurrentRecord property as a string. The PeopleCode process determines whether it's the beginning of a new rowset, or a FileId file record according to the *fileid* you specified. If it's a FileId file record, the process sets the IsNewFileId property to True; if not, it sets the IsNewFileId property to False.



If this is the first execution of ReadRowset on the file, it will return NULL upon encountering the initial FileId file record, but will still store that file record in CurrentRecord and set IsNewFileId accordingly.

4. If the rowset returned isn't NULL, process that rowset as necessary.
5. If IsNewFileId is False, go back to step 3 to continue reading rowsets from the file. If IsNewFileId is True, examine CurrentRecord to determine which new file layout to use, and go back to step 1.

You can repeat this procedure one rowset at a time, proceeding through the remainder of the input file. When ReadRowset returns NULL, no more rowset data is available.



For more information about file layout functionality, see Using File Layouts and File Layout.

Parameters

<i>fileid</i>	Specify the value of the FileId field used in the current file.
<i>position</i>	Specify the column in the FileId file record where the FileId field starts. The default is 1.

Returns

None.

Example

```
&rsFile = &MYFILE.ReadRowset();

&CurrentRecord = &MYFILE.CurrentRecord;

&IsNewFileId = &MYFILE.IsNewFileId;

If &MYFILE.IsNewFileId Then

    &FILELAYOUT = FindFileId(&CurrentRecord);

    &MYFILE.SetFileLayout(@"FileLayout." | &FILELAYOUT);

    &MYFILE.SetFileId("999", 1);

End-If;
```

Related Topics

CurrentRecord, IsNewFileId, ReadRowset, SetFileLayout

SetFileLayout

Syntax

```
SetFileLayout(FILELAYOUT.filelayoutname)
```

Description

The **SetFileLayout** method is a file layout method. It associates a specific file layout definition with the file object executing this method, providing easy access to rowset data. The file object must be associated with an open file. With file layout definitions, you can read and write rowsets

as easily as strings. If the file object isn't open or the definition doesn't exist, **SetFileLayout** will fail.

You must execute **SetFileLayout** before you can use any other file layout methods or properties with a file object, otherwise those methods and properties will return NULL or False.



For more information about file layout functionality, see Using File Layouts and File Layout.

Parameters

<i>filelayoutname</i>	Specify as a string the name of an existing file layout definition created in Application Designer.
-----------------------	---

Returns

A boolean value: True if the method succeeds, False otherwise.

Example

The following example opens a data file, associates a file layout definition with it, reads and processes the first rowset from it, and closes the file.

```
&MYFILE.Open(&SOMENAME, "E");

If &MYFILE.SetFileLayout(FILELAYOUT.SOMELAYOUT) then

    &SOMEROWSET = &MYFILE.ReadRowset();

    /* Process the contents of &SOMEROWSET */

Else

    /* Error - SetFileLayout failed */

End-If;

&MYFILE.Close();
```

Related Topics

CurrentRecord, CreateRowset, ReadRowset, IgnoreInvalidId, WriteRowset

SetPosition

Syntax

```
SetPosition(position)
```

Description

The **SetPosition** method sets the current read or write position in the external file associated with the file object executing this method. **SetPosition** will only work with a file, which is opened in Update mode, and is designed to be used in combination with the **GetPosition** method to recover from interruptions during file access.

To start reading or writing from the beginning of a file, you must use **SetPosition** to set a read/write position of **0**, immediately after you open the file in Update mode.



For more information about the appropriate use of **SetPosition**, see [Recovering from File Access Interruptions](#).

Parameters

position

The byte position in the file at which you want to continue reading or writing. This should either be **0** or a value you previously obtained with the **GetPosition** method and saved in a separate file or a database.



In Update mode, any write operation will clear the file of all data that follows the position you set.



Use **SetPosition** only for recovering from file access interruptions. Supplying your own value for **SetPosition** isn't recommended, except for position **0**.

Returns

None.

Example

The following example reopens a file in Update mode, and sets the read position to the last saved checkpoint:

```
&MYFILE = GetFile(&SOMENAME, "U");

If &MYFILE.IsOpen Then

    /* Retrieve the value of the last saved checkpoint, &LASTPOS */

    &MYFILE.SetPosition(&LASTPOS);

    while &MYFILE.ReadLine(&SOMESTRING)

        /* Process the contents of each &SOMESTRING */
```



```
End-While;  
  
&MYFILE.Close();  
  
End-If;
```



The effect of the Update mode is not well defined for Unicode files. Use the Update mode and **SetPosition** only on files stored with the ANSI character set.

Related Topics

GetPosition, Open, and GetFile function

WriteLine

Syntax

```
WriteLine(string)
```

Description

The **WriteLine** method writes one string of text, *string*, to the output file associated with the file object executing this method, followed by a newline character appropriate to the platform where the file is being written. To build a single line using multiple strings, use the WriteString method.

Parameters

<i>string</i>	The string of text to be written.
---------------	-----------------------------------

Returns

None.

Example

The following example adds a line of text to an existing file:

```
&MYFILE.Open("somefile.txt", "A");  
  
&MYFILE.WriteLine("This is the last line in the file.");  
  
&MYFILE.Close();
```

The following example converts a file where the fields are separated with tabs into a file where the fields are separated with commas.

```
Local File &TABFILE, &CSVFILE;  
  
Local string &FILE_NAME, &DATA, &NEWDATA;
```

```

&FILE_NAME = "Test.txt";

&TAB = Char(9);

&TABFILE = GetFile(&FILE_NAME, "r");

&FileName = &TABFILE.Name;

&POS = Find(".", &FileName);

&NEWFILE_NAME = Substring(&FileName, 1, &POS) | "dat";

&CSVFILE = GetFile(&NEWFILE_NAME, "N", %FilePath_Absolute);

If &TABFILE.IsOpen And
    &CSVFILE.IsOpen Then
    While &TABFILE.ReadLine(&DATA);
        &NEWDATA = Substitute(&DATA, &TAB, ",");
        &CSVFILE.WriteLine(&NEWDATA);
    End-While;
    &TABFILE.Close();
    &CSVFILE.Close();
End-If;

```

Related Topics

ReadLine, WriteString, WriteRaw

WriteRaw

Syntax

```
WriteRaw(RawBinary)
```

Description

The **WriteRaw** method writes the contents of *RawBinary* to a file. This can be used for writing images, messages, or other types of raw binary data to a file.

Parameters

<i>RawBinary</i>	Specify the raw binary to be written to the file.
------------------	---

Returns

None.

Example

The following example writes employee photos (GIF files) from a record to a file.

```
Local File &FILE;

Local Record &REC;

Local SQL &SQL;

&REC = CreateRecord(Record.EMPL_PHOTO);

&SQL = CreateSQL("%SelectAll(:1)", Record.EMPL_PHOTO);

&FILE = GetFile("C:\temp\EMPL_PHOTO.GIF", "w", "a", %FilePath_Absolute);

While &SQL1.Fetch(&REC)

    &FILE.WriteRaw(&REC.EMPLOYEE_PHOTO.Value);

End-While;

&FILE.Close();
```

Related Topics

WriteLine, WriteString

WriteRecord

Syntax

WriteRecord (*record*)

Description

The **WriteRecord** method is a file layout method. It writes the contents of the record object *record*, to the output file. (Remember, a record object contains only one row of data from an SQL table.)

You can use this method to build a transaction in the output file, one file record at a time, without having to instantiate and populate a rowset object. You can apply this method to any record whose structure and name matches that of a record defined in the current file layout.



You must execute the SetFileLayout method from the file object before using **WriteRecord**, otherwise it will return False.

Parameters

record Specify the name of an existing record object to be written. You can use Rowset class methods such as GetField and built-in functions such as GetRecord to populate the record with data before writing it to the file.



For more information about records, see the Data Buffer Access and ProcessRequest Class.

Returns

A boolean value: True if the method succeeds, False otherwise.

Example

The following example appends all the data in a record to an existing file:

```
Local File &MYFILE;

&MYFILE = GetFile("record.txt", "A");

If &MYFILE.IsOpen Then
    If &MYFILE.SetFileLayout(FILELAYOUT.VOL_TEST) Then
        &LN = CreateRecord(RECORD.VOLINTER_ORG_TBL);
        &SQL2 = CreateSQL("%Selectall(:1)", &LN);
        While &SQL2.Fetch(&LN)
            &MYFILE.WriteRecord(&LN);
        End-While;
    Else
        /* do error processing - filelayout not correct */
    End-If;
Else
```

```

        /* do error processing - file not open */

End-If;

```

```

&MYFILE.Close();

```

If your file layout is defined as XML format, **WriteRecord** doesn't add the closing tag for the record. You must write it yourself using the WriteLine method. This is because the code has no way of knowing when you want to write children records following the record just written out. The following code shows an example of using WriteLine:

```

Local File &MYFILE;

&MYFILE = GetFile("XMLrecord.txt", "A");

If &MYFILE.IsOpen Then

    If &MYFILE.SetFileLayout(FILELAYOUT.RECORDLAYOUT) Then

        &LN = CreateRecord(RECORD.QA_INVEST_LN);

        &SQL2 = CreateSQL("%Selectall(:1)", &LN);

        While &SQL2.Fetch(&LN)

            WriteRecord(&LN);

            WriteLine("</QA_INVEST_LN>"); /* Add the closing tag */

        End-While;

    Else

        /* do error processing - filelayout not correct */

    End-If;

Else

    /* do error processing - file not open */

End-If;

&MYFILE.Close();

```

Related Topics

CreateRowset, ReadRowset, SetFileLayout, WriteRowset

WriteRowset

Syntax

WriteRowset (*rowset*)

Description

The **WriteRowset** method is a file layout method. It writes the contents of a rowset object, *rowset*, to the output file associated with the file object executing this method. Regardless of whether the rowset contains just one or more than one transaction (level 0 row), executing this method once will write the entire contents of the rowset to the output file.



You must execute the SetFileLayout method from the file object before using **WriteRowset**, otherwise it will return False.

Parameters

rowset

Specify the name of an existing rowset object that was instantiated with the File class CreateRowset or ReadRowset method. You can use Rowset class methods such as Select and built-in functions such as GetLevel0 to populate the rowset with data before writing it to the file.



For more information about rowsets, see Data Buffer Access and Rowset Class.

Returns

A boolean value: True if the method succeeds, False otherwise.

Example

```
Local File &MYFILE;

Local Rowset &FILEROWSET;

&MYFILE = GetFile("c:\temp\EMP_CKLS.txt", "A", %FilePath_Absolute);

If &MYFILE.IsOpen Then

    If &MYFILE.SetFileLayout(FILELAYOUT.EMPL_CHECKLIST) Then

        &FILEROWSET = &MYFILE.CreateRowset();

        &FILEROWSET = GetLevel0();
```

```
        &MYFILE.WriteRowset (&FILEROWSET) ;

    Else

        /* file layout not found, do error processing */

    End-If;

Else

    /* file not opened, do error processing */

End-if;


&MYFILE.Close();
```

Related Topics

CreateRowset, ReadRowset, SetFileLayout, WriteRecord

WriteString

Syntax

```
WriteString(string)
```

Description

The **WriteString** method writes one string of text to the output file associated with the file object executing this method, without any newline character. Each string written will extend the current line in the file.

You can start a new line by using the WriteLine method to write the last part of the current line. WriteLine always adds a newline character appropriate to the platform where the file is being written, whether you supply a character string of any length, or a null string.

Parameters

<i>string</i>	A string variable containing the text to be written.
---------------	--

Returns

None.

Example

The following example opens an empty file, writes two lines of text to it without a final newline, and closes it:

```
&MYFILE.Open("somefile.txt", "W");
```

```
&MYFILE.WriteString("This is the first ");  
  
&MYFILE.WriteString("line in the file.");  
  
&MYFILE.WriteLine("");  
  
&MYFILE.WriteString("This second line is not terminated.");  
  
&MYFILE.Close();
```

Related Topics

ReadLine, WriteLine, WriteRaw

File Class Properties

CurrentRecord

This property is a file layout property. This property returns the current record as a string. **CurrentRecord** is used in combination with the SetFileId and ReadRowset methods and IsNewFileId property when reading files that contain data based on multiple file layouts.

If ReadRowset returns NULL, either the current record is a FileId record or the end of file has been reached. The IsNewFileId allows you to determine which. If it is a FileId, you can parse the string returned by CurrentRecord to determine the file layout definition to be used.



SetFileId, CurrentRecord and IsNewFileId don't apply to CSV and XML format input files. You can only implement multiple file layouts with FIXED format files.



For more information, see ReadRowset, SetFileId, IsNewFileId, and Handling Multiple File Layouts.

This property is read-only.

Example

```
&RECSTRING = &MYFILE.CurrentRecord;
```

IgnoreInvalidId

This property is a file layout property that's used in combination with the ReadRowset method. It returns a boolean value that specifies whether file records with invalid FileIds will be ignored. Each time ReadRowset is executed, it may encounter a file record, that does not qualify as part of the rowset because its File ID isn't part of the current file layout, or because it occurs in an invalid position in the rowset. If **IgnoreInvalidId** is False, the PeopleCode program will

terminate; if **IgnoreInvalidId** is True, ReadRowset will ignore the invalid file record. The default value is True.

This property is read-write.

Example

```
&MYFILE.IgnoreInvalidId = False;
```

IsError

This property is a file layout property. It returns a boolean value indicating whether a field error condition was generated by the last file layout method executed. If an error condition was generated, **IsError** returns True; if not, **IsError** returns False. The default value is False.

This property is read-only.

Example

The following example shows where IsError would be used:

```
&MYFILE.Open(&SOMENAME, "R");

&MYFILE.SetFileLayout(FILELAYOUT.SOMELAYOUT);

&MYROWSET = &MYFILE.ReadRowset();

If &MYFILE.IsError Then

    /* Examine the EditError property of each field in the rowset
       to find the one with the error, and respond accordingly */

End-If;

&MYFILE.Close();
```



For more information, see Handling Multiple File Layouts.

IsNewFileId

This property is a file layout property. It returns a boolean value indicating whether a FileId file record has been encountered. When the ReadRowset method reads a transaction from an input file, it examines the current file record following the transaction. If that file record is a FileId file record, **IsNewFileId** returns True; if not, **IsNewFileId** returns False.

IsNewFileId is used in combination with the SetFileId method and CurrentRecord property when reading files that require multiple file layouts.



SetFileId, CurrentRecord and IsNewFileId don't apply to CSV and XML format input files. You can only use FIXED format files to implement multiple file layouts.



For more information, see ReadRowset, SetFileId and CurrentRecord.

This property is read-only.

Example

```
&MYFILE.SetFileLayout(FILELAYOUT.SOMELAYOUT) then /* Set the first layout */

&MYFILE.SetFileId("999", 1); /* Set the FileId */

&SOMEROWSET = &MYFILE.ReadRowset(); /* Read the first rowset */

While &SOMEROWSET <> NULL

  If &MYFILE.IsNewFileId Then

    /* Examine &MYFILE.CurrentRecord for the next layout */

    /* Set the next layout */

    &MYFILE.SetFileId("999", 1); /* Set the FileId */

  End-If;

  /* Process the current &SOMEROWSET */

  &SOMEROWSET = &MYFILE.ReadRowset(); /* Read the next rowset */

End-While;
```

IsOpen

This property returns a boolean value indicating whether the file is open. If the file object is open, **IsOpen** returns True; if not, **IsOpen** returns False.

This property is read-only.

Example

The following example opens a file, writes a line to it, and closes it:

```
&MYFILE.Open("item.txt", "W");

If &MYFILE.IsOpen Then

  &MYFILE.WriteLine("Some text.");
```

```
&MYFILE.Close();  
  
End-If;
```

Name

This property returns as a string the name of the external file. If no file is currently associated with the file object, **Name** returns a NULL string.

This property is read-only.

Example

```
&TMP = &MYFILE.Name;
```

RecTerminator

This property is a file layout property. It returns the string of characters that will indicate the end of a file record. Read operations use the value of **RecTerminator** to determine where each file record ends and the next one starts, and write operations append the value of **RecTerminator** to each record written.

This property defaults to the newline character appropriate to the platform where the file is being stored: a linefeed on UNIX systems, and a carriage return/linefeed combination on Windows systems. You'll only need to specify a different **RecTerminator** if you anticipate that part of the data in a file field will include the newline character used on the storage platform; you must assign to **RecTerminator** a unique string that you know will *not* appear in a file field.

This property is read-write.

Example

```
&MYFILE.RecTerminator = "##EOR##";
```

Grid Class

A grid looks and behaves like a spreadsheet embedded in a page: it has column headings, row headings, cells, and horizontal and vertical scroll bars. It can be used instead of a single-level scroll. It is analogous to a scroll region on a page. Each row in a grid corresponds to a set of controls in a scroll occurrence. Each cell in a grid corresponds to a field on a page.



For more information, see Grid Control.

Currently, the Grid class only enables you to instantiate grid column objects, which in turn enables you to change grid column attributes without having to write PeopleCode that loops through every row of the grid.



For more information, see GridColumn Class.



PeopleSoft builds a page grid one row at a time. Because the Grid class applies to a complete grid, you can't attach PeopleCode that uses the Grid class to events that occur before the grid is built; the earliest event you can use is the page Activate Event.

An expression of the form

```
&MYGRID.columnname.property
```

is converted to an object expression by using **GetColumn**(*columnname*).

Example

The following examples are equivalent.

Using *columnname*:

```
&MYGRIDCOLUMN = &MYGRID.CHECKLIST_ITEMCODE;
```

Using the GetColumn method:

```
&MYGRIDCOLUMN = &MYGRID.GetColumn("CHECKLIST_ITEMCODE");
```

The following examples are equivalent.

Using *columnname*:

```
&MYGRID.CHECKLIST_ITEMCODE.Visible = False;
```

Using the GetColumn method:

```
&MYGRID.GetColumn("CHECKLIST_ITEMCODE").Visible = False;
```

Using the Grid Class in PeopleCode

The PeopleCode grid object is a reference to a page runtime object for the grid. These particular page runtime objects aren't present until the Component is started.



PeopleSoft builds a page grid one row at a time. Because the Grid class applies to a complete grid, you can't attach PeopleCode that uses the Grid class to events that occur before the grid is built; the earliest event you can use is the page Activate Event.

If you're using the grid within a secondary page, the runtime object for the grid isn't created until the secondary page is run. The grid object can't be obtained until then, which means that the

earliest PeopleCode event you can use to activate a grid that's on a secondary page is the Activate event for the secondary page.

The attributes you set for displaying a page grid only remain in effect while the page is active. When you switch between pages in a component, you'll have to reapply those changes *every time* the page is displayed.

In addition, the Activate event associated with a page fires every time the page is displayed. Any PeopleCode associated with that Activate event will run, which may undo the changes you made when the page was last active. For example, if you hide a grid column in the Activate event, then display it as part of a user action, when the user tabs to another page in the component, then tabs back, the Activate event will run again, hiding the grid column again.

If a user at runtime hides a column of a grid, tabs to another page in the component, then tabs back to the first page, the page is refreshed and the grid column is displayed again.

When you place a grid on a page, the grid is automatically named the same as the name of the primary record of the scroll for the grid. This is the name you use with the GetGrid function. You can change this name on the Record tab of the Grid properties.



There is no visible property for a grid, just a grid column. However, you can still hide an entire grid. Remember, many of the Rowset methods and properties will work on a grid. If you want to hide an entire grid, get the rowset for that grid, use the HideAllRows Rowset class method.

Declaring a Grid or Grid Column Object

Grids are declared using the Grid data type. For example,

```
Local Grid &MYGRID;
```

Grid Columns are declared using the GridColumn data type. For example:

```
Local GridColumn &MYGRIDCOL;
```

Scope of a Grid or Grid Column Object

Both the grid and grid column objects can only be instantiated from PeopleCode.

A grid is a control on a page. You will generally only use these objects in PeopleCode programs that are associated with an online process, not in an Application Engine program, a message subscription, a Component Interface, and so on.

In addition, PeopleSoft builds a page grid one row at a time. Because the Grid class applies to a complete grid, you can't attach PeopleCode that uses the Grid class to events that occur before the grid is built; the earliest event you can use is the page Activate Event.

Grid Class Built-In Function

GetGrid

Grid Class Method

GetColumn

Syntax

```
GetColumn(columnname)
```

Description

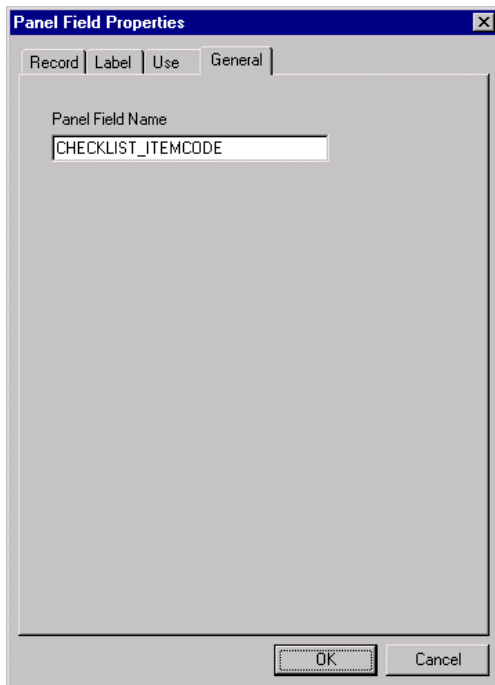
The **GetColumn** method instantiates a grid column object from the GridColumn class, and populates it with a grid column from the grid object executing this method. Specify the grid column name in the page field properties for that field, consisting of any combination of upper case letters, digits and "#", "\$", "@", and "_".

To specify a grid column name:

1. Open the page in Application Designer, select the grid and access the page field properties.
2. Select the Columns tab on the grid properties.
3. Either double-click on the grid column you want to name, or select the column and click on the Properties button.
4. On the General tab, type the grid column name in the Page Field Name field.



Although it's possible to base multiple grid columns on the same record field, the Page Field Name you enter for each grid column must be unique, not just for that grid, but for that page. If you have two grids on a page, every page field name must be unique to the page.



Specifying a Grid Column Name

Parameters

columnname

Specify a string containing the value of the Page Field Name on the General tab of the grid column's properties. You must have previously entered a value for the Page Field Name for the grid column you want to work with.

Returns

A GridColumn object populated with the grid column specified as the parameter to this method.

Example

```
local Grid &MYGRID;

local GridColumn &MYGRIDCOLUMN;

&MYGRID = GetGrid(PAGE.EMPLOYEE_CHECKLIST, "EMPL_GRID");

&MYGRIDCOLUMN = &MYGRID.GetColumn("CHECKLIST_ITEMCODE");
```

The following function loops through rows on a grid. The function will find each row that is selected. It does this through the Selected property of the Row class of PeopleCode. Data is then moved from the selected row to a new row, on a different grid, in the same page. The way in which this function is written, data is moved from &SCROLL_SHELF to &SCROLL_CART. These are two different rowset objects, of two different grids, on the same page. Note that the two grids in this example are on the same occurs level.

```
/* Moving data between grids on the same occurs level of the same page */
```

```

Local Rowset &SCROLL_CART, &SCROLL_SHELF;

Function move_rows(&SCROLL_CART As Rowset, &SCROLL_SHELF As Rowset);

    &I = 1;

/* loop to find whether row is selected */

Repeat
    If &SCROLL_SHELF.GetRow(&I).Selected = True Then
        If All(&SCROLL_CART(1).GetRecord(1).QEPC_ITEM.Value) Then
            &SCROLL_CART.InsertRow(&SCROLL_CART.ActiveRowCount);
        End-If;

/* if it is selected move data to other grid */

&SCROLL_SHELF.GetRow(&I).GetRecord(1).CopyFieldsTo(&SCROLL_CART.GetRow(&SCROLL_C
ART.ActiveRowCount).GetRecord(1));

/* delete row from current grid so data disappears */

    &SCROLL_SHELF.DeleteRow(&I);

    &I = &I - 1;

End-If;

    &I = &I + 1;

Until &I = &SCROLL_SHELF.ActiveRowCount + 1;

/* end of loop
*****/

```



```

End-Function;

/***** end of function *****/

/* Creating the rowset object */

&SCROLL_CART = GetLevel0() (1).GetRowset (SCROLL.QEPC_CART);
&SCROLL_SHELF = GetLevel0() (1).GetRowset (SCROLL.QEPC_SHELF);

/* calling the function */

move_rows (&SCROLL_CART, &SCROLL_SHELF);

```

Related Topics

GetGrid

Grid Class Property

gridcolumn

If a grid column name is used as a property, it accesses the grid column with that name. This means the following code:

```
&Mycolumn = &MyGrid.gridcolumnname;
```

acts the same as

```
&Mycolumn = &MyGrid.GetColumn(columnname);
```

Label

This property returns a string specifying the label that appears as the title of the grid.



You can't use this property to set labels longer than 100 characters. If you try to set a label of more than 100 characters, the label will be truncated to 100 characters.



Always put any changes to labels in the **Activate** event. This way the label is set every time the page is accessed.

This property is read-write.

GridColumn Class

Grid columns are page fields that comprise a grid, which is itself a page field. Similarly, a group of GridColumn objects comprises a Grid object. The GridColumn class enables you to change grid column attributes without having to write PeopleCode that loops through every row of the grid.

This class has no methods or built-in functions, only properties.



In order to use a GridColumn object, you must instantiate it from a grid object. This requires you to first instantiate a Grid object using the GetGrid built-in function.



For more information, see Grid Class.

GridColumn Class Properties

Enabled

This property specifies whether the fields in the column are enabled (that is, can be edited) or if they are "disabled", that is, un-editable. All columns are enabled by default.

This property is read-write.

Example

```
If Not &ValidID Then

    &MYGRID = GetGrid(PAGE.EMPLOYEE_CHECKLIST, "EMPL_GRID");

    &MYGRIDCOLUMN = &MYGRID.GetColumn("CHECKLIST_ITEMCODE");

    &MYGRIDCOLUMN.Enabled = False;

End-If;
```

Label

This property returns a string specifying the display label for the GridColumn object executing the property, as distinct from the grid column's Record Field Name and Page Field Name. This

property overrides the equivalent settings on the Label tab of the grid column's Page Field Properties.



You can't use this property to set labels longer than 100 characters. If you try to set a label of more than 100 characters, the label will be truncated to 100 characters.



If you change the column label, then change the field label, you may overwrite the column label with the field label. The grid object is part of a page, *not* the data buffers, while the field is part of the data buffers. To avoid this problem, always put any changes to column labels in the **Activate** event. This way the label is set every time the page is accessed.

This property is read-write.

Example

```
&MYGRIDCOLUMN.Label = "Checklist Item";
```

Name

This property returns the name of the grid column, as a string. This value comes from the Page Field Name on the General tab in the Page Field Properties of the GridColumn object executing the property. This property was the value used to instantiate the GridColumn object.

This property is read-only.

Example

```
&PF_NAME = &MYGRIDCOLUMN.Name;
```

Visible

This property returns a boolean value which specifies whether a grid column should be hidden from view. Set this property to False in order to hide the grid column, and to True in order to unhide the grid column. This property defaults to True.

The **Visible** property will also hide grid columns that are displayed as tabs in the PeopleSoft Internet Architecture.

If you specify "Show Column on Hide Rows" in Application Designer, the column headers and labels of a grid display at runtime, even when the rest of the column is hidden. You can't override this value using PeopleCode.



For more information about grid properties, see Grid Control.

This property is read-write.



In previous releases of PeopleTools, the methodology for hiding a grid column was to use the Hide function in a loop to hide each cell in the column, one row at a time. This method of hiding grid columns still works. However, your application may experience deteriorated performance if you continue to use this method.

Example

```
&MYGRIDCOLUMN.Visible = False;
```

The following example checks for the value of a field in every row of the grid. If that value is "N" for every row, the column is hidden.

```
&RS = GetRowset (Scroll.EX_SHEET_LINE);

&HIDE = True;

While (&HIDE)

    For &I = 1 To &RS.ActiveRowCount;

        &OUT_OF_POLICY = &RS(&I).EX_SHEET_LINE.OUT_OF_POLICY.Value;

        &NO_RECEIPT_FLG = &RS(&I).EX_SHEET_LINE.NO_RECEIPT_FLG.Value;

        If &OUT_OF_POLICY = "Y" Then

            &OUT_OF_POLICY_HIDE = False;

            &HIDE = False;

        End-If;

        If &NO_RECEIPT_FLG = "Y" Then

            &NO_RECEIPT_HIDE = False;

            &HIDE = False;

        End-If;

    End-For;

    If &HIDE = True Then

        &HIDE = False;

    End-If;

End-While;

If Not (&OUT_OF_POLICY_HIDE) Then
```

```
        GetGrid(Page.EX_SHEET_LINE_APV1, "EX_SHEET_LINE").OUT_OF_POLICY.Visible =  
True;  
  
Else  
  
        GetGrid(Page.EX_SHEET_LINE_APV1, "EX_SHEET_LINE").OUT_OF_POLICY.Visible =  
False;  
  
End-If;  
  
  
If Not (&NO_RECEIPT_HIDE) Then  
  
        GetGrid(Page.EX_SHEET_LINE_APV1, "EX_SHEET_LINE").NO_RECEIPT_FLG.Visible =  
True;  
  
Else  
  
        GetGrid(Page.EX_SHEET_LINE_APV1, "EX_SHEET_LINE").NO_RECEIPT_FLG.Visible =  
False;  
  
End-If;
```

Internet Script Classes

An Internet Script is a specialized PeopleCode function that generates dynamic web content. Internet Scripts interact with web clients (browsers) using a request-response paradigm based on the behavior of the Hypertext Transfer Protocol.

Internet Scripts work with PeopleSoft Internet Architecture. You must have PeopleSoft Internet Architecture set up correctly before you can run an Internet Script.

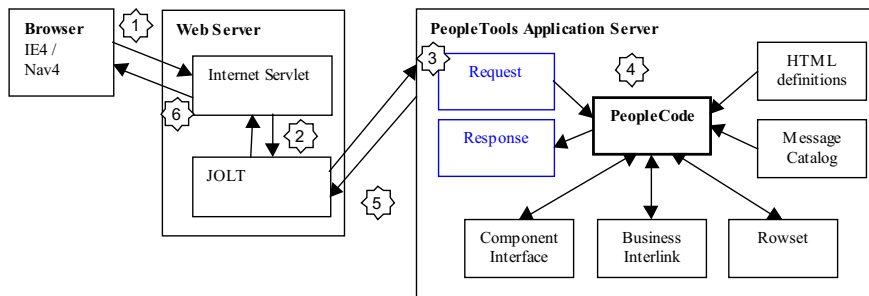


For more information, see PeopleSoft Internet Architecture Administration.

This document describes how an Internet Script works in your application, how to create an Internet Script, and the Internet Script classes that you use in your PeopleCode.

The Big Picture

The following flow chart shows the interaction of an Internet Script (PeopleCode) in a PeopleSoft Internet Architecture application.



Communication Flow Chart for Internet Client Scripts

11. An HTTP Request arrives at the web server specifying in its URL the **Presentation Relay Servlet** (iclientservlet). A query string parameter included in the URL specifies which PeopleCode program to run. No component is specified because none is needed. The PeopleCode program (Internet Script) runs directly. The following is an example of the URL:

`http://serverx/servlets/iclientservlet/peoplesoft8/?ICType=Script&ICScriptProgramName=WEBLIB_BEN_401k.PAGES.FieldFormula.iScript_Home401k`
12. The servlet is invoked by the web server. It serializes all necessary information from the Web Server Request and Response objects, then makes a call through JOLT to the PeopleSoft Application Server service passing the serialized object information.
13. The Application Server unpacks all of the object information and creates PeopleSoft versions of the Request and Response objects. It then calls the function in the PeopleCode program specified in the query string parameter. The PeopleCode Request and Response objects are available to the PeopleCode program through the system variables %Request and %Response.
14. The PeopleCode program is then responsible for generating every aspect of the HTML page that is to be returned to the browser. It has access to the Request object to view items such as form fields, query string parameters, cookies and headers. It also has access to Component Interfaces, Business Interlinks, and Rowset objects to interact with ERP applications. HTML definitions are available to include language sensitive blocks of HTML. They can also contain JavaScript. The Message Catalog is also available for language sensitive messages. The Response object is used to "write" the generated HTML back to the browser.
15. When the PeopleCode program is finished running, the headers, cookies, and HTML in the PeopleSoft Response object are serialized and returned through JOLT to the Presentation Relay Servlet.
16. The Presentation Relay Servlet then unpacks the Response information generated by the PeopleCode, and plays it back to the web server response object, which in turn sends the response to the user's browser.

URL vs. URI

In this document, the term URL refers to the entire URL, including the query string. The following is an example of a URL:

```
http://serverx:port/servlets/iclientservlet/peoplesoft8?ICType=Script&ICScriptProgramName=WEBLIB_BEN_401k.PAGES.FieldFormula.iScript_Home401k
```

A URI does not include the query string (the text following a ? on the URL). You can think of it as a subset of the URL that points to the resource, but does not include any parameters being passed to that resource. From the above example, the URI portion of the URL is as follows:

```
http://serverx:port/servlets/iclientservlet/peoplesoft8
```

Creating Web Libraries

Internet Scripts uses the existing PeopleCode Function Library infrastructure. However, instead of naming your record FUNCLIB_xxx, all Internet Scripts *must* be contained in records named WEBLIB_xxx. All of the existing tools and techniques for working with function libraries (such as Upgrade, Find In, Rename, and so on) also apply to Web Libraries.

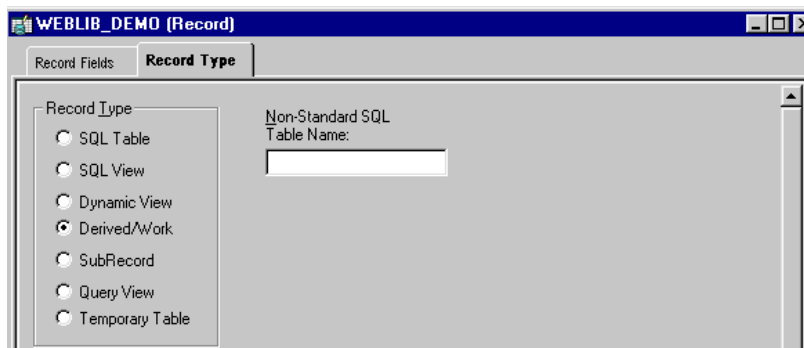


For more information, about FUNCLIBs, see Accessing PeopleCode External Functions.

To create an Internet Script:

1. Create or extend a domain specific WEBLIB (funclib)

WEBLIBs are derived/work record definitions that have their name prefixed with WEBLIB_.



Record Type for Internet Script Records (WEBLIBs)

2. Add a function to a WEBLIB

The name of your function *must* be prefaced with *IScript_*. For example:

```
IScript_HelloWorld
```

```
IScript_FuncHRPage
```



Internet Script functions take no arguments, and do not return a value.

The following Internet Script writes data.

```
Function IScript_HPDefaultCategories()

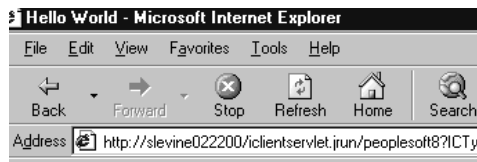
    &ClearDotImage = %Response.GetImageURL(Image.PT_PORTAL_CLEAR_DOT);

    &CatHTML = GetHTMLText(HTML.PORTAL_HP_CATEGORY, &ClearDotImage,
    GetCategories());

    %Response.Write(&CatHTML);

End-Function;
```

The following Internet Script uses both the request and response objects to echo what the user types into an edit control.



Hello World

Say this!

You said: hi!

Example Internet Script

```
Function IScript_HelloWorld()

    %Response.WriteLine("<html><head><title>Hello World</title></head><body>");

    %Response.WriteLine("<h1>Hello World</h1>");

    %Response.WriteLine("<form method=POST action = " | %Request.FullURI | "?" |
    %Request.QueryString | "><input type=submit value='Say this!'">&nbsp;<input
    type=edit name=WhatYouSaid value=" | %Request.GetParameter("WhatYouSaid") |
    "></form>");
```



```
rem echo back what the user typed into the edit box...;

%Response.WriteLine("<p><font face='Arial, Helvetica' size='5' color='blue'
>You said: ");

%Response.WriteLine(%Request.GetParameter("WhatYouSaid"));

%Response.WriteLine("</body></html>");

Return;

End-Function;
```

Internet Script Security

Internet Scripts are secured on your system in a similar fashion to Component Interfaces. After you create a WEBLIB record, and your functions, you must add them just as you would add a page to an application.

To add security to an Internet Script:

1. Navigate to the Maintain Security page and open the Permission List to which you want to give access.



For more information, see Security.

2. Select Web Libraries.

You may need to scroll through the tabs at the top of the page to access the Web Libraries tab.

Select Web Libraries for Permission List

3. Add new WebLibrary for Permission List (optional)

If this is a new Web Library, you need to add it to the permission list. Click the plus button to add a new Web Library, then select the library you want to add from the drop-down.

4. Select **Edit** for the Web Library you want to grant access for.

Selecting Web Library

5. Select the access you wish to give each function in the WEBLIB record.

From the page that displays, you can choose to allow or disallow access on a per function basis. The buttons on the side of the page either grant or disallow access for all functions.

WebLib Permissions

Authorized Web Library
WEBLIB_HPDEMO

Function	Access Permissions
ISCRIP1.FieldFormula.IScript_Dictionary	FULL
ISCRIP1.FieldFormula.IScript_ExciteNews	FULL
ISCRIP1.FieldFormula.IScript_PTOLBalance	FULL
ISCRIP1.FieldFormula.IScript_TechnologyNews	FULL
ISCRIP1.FieldFormula.IScript_YahooStockQuote	FULL

First 1-5 of 5 Last View All

Full Access (All)
No Access (All)

Controlling Permission at Function Level

When to use an Internet Script

Internet Scripts give the developer complete control over the HTML sent to the browser. This gives a developer a great deal of flexibility in creating a user interface. However, there are some responsibilities that the developer inherits when choosing to use Internet Scripts.

- The developer is responsible for creating HTML and JavaScript that is cross-browser compatible.
- The developer is responsible for ensuring that all the text sent to the browser is stored in either the message catalog or in HTML definitions to enable translation of the text.
- Internet Scripts aren't part of the regular Component Processor flow, so the developer is responsible for accessing the database in a multi-language, multi-market and multi-currency sensitive way. (You can do this by using a Component Interface to access the database.)

For these reasons, it's advised that the developer first try to build a page in Application Designer. This approach allows PeopleTools to generate all of the HTML in a cross-browser, multi-language, multi-market and multi-currency sensitive way.

You don't have to use an Internet Script to generate an entire page. The Request and Response objects can be used with an HTML area, so you can choose to develop a portion of your HTML using the Internet Script objects, while allowing the majority of it to be generated by the Component Processor. If you choose to do this, instead of using the Response object's Write and WriteLine methods to output your HTML, you set the value of the HTML area to the HTML that you want to display. Remember though, that just as with an Internet Script, you are responsible for ensuring that the HTML you generate is cross-browser compatible and multi-market sensitive.

So when *should* you use Internet Scripts? Here are two scenarios where Internet Scripts would be an appropriate choice:

- The page being developed can *not* be built using Application Designer. An example of this is a page that requires more than one HTML form. PeopleSoft Internet Architecture places the entire page inside of a single form tag, so no other HTML form tags can be added. In this case the requirements of the page can't be met by pages created in Application Designer, so Internet Scripts should be used.



You should use Component Interfaces for all database access so you have multi-language and multi-currency sensitivity.

- The page being developed doesn't ever access the database. Using a page and Component Processor for this type of page incurs unnecessary processing overhead. An example of this is a page that talks to another website and redisplay HTML from the remote site.

Style Sheets and Styles

PeopleSoft recommends that developers using the Internet Scripts always use styles (also known as classes) defined in the style sheets to specify the attributes (that is, background color, font, size, alignment, borders, and so on) of objects referenced in the Internet Scripts. The Response object provides access to Style Sheets stored in the PeopleSoft database.



For more information about style sheets, see [Creating Style Sheet Definitions](#).

Other Considerations

PeopleSoft does not recommend using any of the following technologies in Internet Scripts:

- ActiveX controls
- Java applets
- Browser plug-ins

Accessing an Internet Script

Viewing an Internet Script requires the assembly of a URL with three basic pieces:

- The URI to a PeopleSoft Internet Architecture web application. All access to the PeopleSoft Internet Architecture is through the Presentation Relay Servlet (iclientservlet). The URI format varies between web servers. For Apache web servers running the servlet on the JServ servlet runner, the URI looks like this:

```
scheme://servername:port/servlets/iclientservlet/installpath/
```

or

```
http://machinename/servlets/iclientservlet/peoplesoft8/
```

- A query string parameter specifying an Internet Type of "Script":

```
?ICType=Script
```

- A second query string parameter specifying the fully qualified name of the Internet Script to execute:

```
&ICScriptProgramName=<recordname>.<fieldname>.<eventname>.<functionname>
```

The following is an example of an Internet Script:

```
http://mlee2038/servlets/iclientservlet/peoplesoft8/?ICType=Script&ICScriptProgramName=WEBLIB_PORTAL.PORTAL_HEADER.FieldFormula.IScript_UniHeader_PIA
```

Error Handling

All errors are handled through the session object. You will need to check the PSMessages collection to see if there are any errors in your code.



For more information, see About Error Handling.

Scope of the Internet Script Classes

The Request, Response, and Cookie classes can only be accessed from PeopleCode. They can only be used as part of a PeopleSoft Internet Architecture application – either as part of a page created in Application Designer or an Internet Script. An Internet Script is always contained in a WEBLIB record. You wouldn't use any of these objects in an Application Engine program, a Component Interface, and so on. However, an Internet Script could call and start a Component Interface, a Business Interlink, and so on.

To access the Request or Response object, use the %Request and %Response system variable. These variables are objects, so you use them with dot notation.

```
%Response.SetContentType("text/HTML");

If %Request.GetParameter("encode") <> "" Then
```

Cookie objects are instantiated from Response object. You can declare the cookie object as data type cookie before any functions. You can also declare variables of type Response and Request, if you wish to assign the %Response and %Request system variables to local variables.

```
Local Cookie &Cookie;

Local Response &Resp;

Local Request &Req;

Function IScript_SetCookie()
```

```
rem set a cookie called "MyCookie" to store my user id in it's value.  Make
the cookie expire when the user's browser session expires;
```

```
&Resp = %Response;

&Req = %Request;

&Cookie = &Resp.CreateCookie("MyCookie");

&Cookie.Value = %UserId;

&Cookie.MaxAge = - 1;
```

```
End-Function;
```

Internet Script Reference

The following is a description of the PeopleCode classes, methods and properties that you use to create an Internet Script.

Request Class

The Request object encapsulates all information from the request issued from the browser. This includes the URI, Query String, Form Fields, QueryString parameters, Cookies, and Headers. The properties for the Request object are primarily read-only.

Parameters

Request parameters are name-value pairs sent by the client to the web server as part of an HTTP request. They include the pairs in the query string part of the URL, as well as data posted in a form, if the request was a post request. Multiple parameter values can exist for any given parameter name. The following methods are available to access parameters:

- GetParameter
- GetParameterNames
- GetParameterValues

The GetParameterValues method returns an array of String objects containing all the parameter values associated with a parameter name. The value returned from the GetParameter method will equal the first value in the array of String objects returned by GetParameterValues.

All form data from both the query string and the post body are aggregated into the request parameter set. The order of this aggregation is that query string data appears before post body parameter data.

Response

The response object encapsulates all the information to be sent back to the browser. This includes the body of the response, content type, response headers, and cookies.

The response object also includes helper methods that retrieve HTML, Image, StyleSheet, and other objects from the database and move them to the webserver, as well as returning strings that can be used to reference these objects in the response content.

The body of the response is created using the response object's Write and WriteLine methods. If the content type of the response is not explicitly set in the PeopleCode program, it will default to "text/html".

Cookies

Cookies are data sent from the client to the server on every request that the client makes. Cookies are stored by the browser, either on disk or in memory, and returned to the server that originally set the cookie.

The Internet Script API makes available the name and value of each cookie that's sent with the request, through the Request object's GetCookieNames and GetCookieValue methods. The API also allows you to set new cookies (or alter existing cookies) through the Response object's CreateCookie method and the Cookie object's properties.

Request Class Methods

GetContentBody

Syntax

```
GetContentBody()
```

Description

This method retrieves the text content of an XML request. This is part of the incoming Business Interlink functionality, which allows PeopleCode to receive an XML request and return an XML response.



For more information, see PeopleSoft Business Interlink Application Developer Guide.

Parameters

None.

Returns

A string.

Example

The following example gets the XML text content of a request, then uses that as input to create a BiDoc. Then the BiDoc methods GetDoc and GetValue are used to access the value of the skills tag.

```
Local BIDocs &rootInDoc, &postreqDoc;

Local string &blob;

Local number &ret;

&blob = %Request.GetContentBody();

/* process the incoming xml(request)- Create a BiDoc */

&rootInDoc = GetBiDoc(&blob);

&postreqDoc = &rootInDoc.GetDoc("postreq");

&ret = &postreqDoc.GetValue("skills", &skills);
```

GetCookieNames

Syntax

```
GetCookieNames()
```

Description

This method returns an array containing names of all the cookies present in this request. If there are no cookies in the request, an empty array is returned. When cookies are present, this method returns an array of strings.

GetCookieValue

Syntax

```
GetCookieValue (name)
```

Description

This method returns a string containing value of the cookie identified *name*. The *name* parameter takes a string value. The match between *name* and the request cookie is case-insensitive. If there is no cookie in the request matching the name, an empty string is returned.

GetHeader

Syntax

```
GetHeader(name)
```

Description

This method returns the value of the header requested by the string *name*. The match between *name* and the request header is case-insensitive. If the header requested does not exist, an empty string is returned.

Example

The following example gets the “Referer” header to see where the request came from:

```
&Referer = %Request.GetHeader("Referer");
```

GetHeaderNames

Syntax

```
GetHeaderNames()
```

Description

This method returns an array of Strings representing the header names for this request.

GetHelpURL

Syntax

```
GetHelpURL(HelpContext)
```

Description

This method returns the help context path (as a string) to the help directory. It’s used to construct a URL to a specific help page.

Example

```
&HelpURLString = %Request.GetHelpURL("hc0110");
```

GetParameter

Syntax

```
GetParameter(name)
```

Description

This method returns the value of a specified query string parameter or posted form data parameter. The match between *name* and the request parameter is case-insensitive. In the event that there are multiple parameter values for a single name, the value returned is the first value in the array returned by the `GetParameterValues` method. If the parameter has (or could have) multiple values, you should use the `GetParameterValues` method in your Internet scripts.

GetParameterNames

Syntax

```
GetParameterNames ()
```

Description

This method returns all the parameter names for this request as an array of Strings, or an empty array if there are no input parameters.

GetParameterValues

Syntax

```
GetParameterValues (name)
```

Description

This method returns the values of the specified parameter (*name*) as an array of Strings, or an empty array if the named parameter does not exist. The *name* parameter takes a string value.

Request Class Properties

BrowserPlatform

This property returns a string containing the value the webserver passes as the browser platform.

This property is read-only.

BrowserType

This property returns a string describing the browser that sent the request.

This property is read-only.

BrowserVersion

This property returns a string describing the version of the browser that sent the request.

This property is read-only.

ExpireMeta

This property returns the refresh meta-tag string that contains the **cmd=expire** parameter (the ExpireMeta string.)

The following is an example of the ExpireMeta string:

```
<meta HTTP-EQUIV='Refresh' Target='_top' CONTENT='10;
URL=/servlets/iclientservlet/peoplesoft810/?cmd=expire'>
```

All Internet script pages should use this property to generate the meta tag to cause the page to expire – it is the same expiration tag used by pages originally created using Application Designer, unless the Internet script generates a menu link in a separate frame that is not supposed to expire.

This tag should be included in the <head> section of the html generated by the Internet Script.

This property is read-only.

Example

```
%Response.Write("<html><head>");
%Response.Write(%Request.ExpireMeta);
%Response.Write("</head><body>");
```

FullURI

This property returns the complete URI up to, but not including, the query string. This method returns a string value.

If you only want to return the request URI (that is, without the scheme, server name or port) use the RequestURI method.

This property is read-only.

Example

From the following code

```
&FullURI = &Request.FullURI();
```

&FullURI contains the following:

```
http://localhost:80/servlets/iclientservlet/HR
```

HTTPMethod

This property returns the HTTP method as a string, (for example, GET, POST, PUT) by which this request was made.

This property is read-only.

LogoutURL

This property returns the complete URL (as a string) to logout of the PeopleSoft session. Use this property to generate a link that causes a page to logout. You should normally not need to include this link on your page.

This property is read-only.

PathInfo

This property returns any path information following the servlet path, but prior to the query string. This property returns null if there is no path information following the servlet path.

This property is read-only.

Example

For the URL

```
http://localhost:80/servlets/iclientservlets/HR?ICType=Panel&...
```

The PathInfo property will return

```
/HR
```

Protocol

This property returns the protocol being used for this request as a string of the form *protocol/major_version.minor_version*. An HTTP 1.0 request, as defined by the HTTP 1.0 specification, should return the string HTTP/1.0. Use the Scheme property instead of the Protocol property when generating hrefs (links).

This property is read-only.

QueryString

This property returns the query string present in the request URL, if any. A query string is defined as any information following a ? character in the URL. If there is no query string, this method returns null. Use the request Parameters methods (GetParameterNames, GetParameterValues, GetParameter) in order to get the values of individual parameters on the query string.

This property is read-only.

RequestURI

This property returns the URI, without the protocol.

This property is read-only.

Example

This example uses the following URL:

```
http://servername/servlets/iclientservlet/peoplesoft8/?ICType=Panel...
```

From the following code

```
&MyRequestURI = &Request.RequestURI;
```

&MyRequestURI contains the following:

```
/servlets/iclientservlet/peoplesoft8/
```

Use the following to build an absolute URL:

```
&myURL = &Request.Scheme | "://" | &Request.ServerName | ":" | &Request.Port |  
&Request.RequestURI | "?" | &Request.QueryString;
```

Scheme

This property returns the scheme, also known as the protocol, used by the request. Common schemes include http, https, ftp and telnet. This property returns a string value.

This property is read-only.

Example

For the URL

```
http://servername/servlets/iclientservlet/peoplesoft8/?ICType=Panel...
```

the Scheme property returns:

```
http
```

ServerName

This property returns the host name of the server on which the servlet is running. This property returns a string value.

This property is read-only.

Example

For the URL

```
http://servername/servlets/iclientservlet/peoplesoft8/?ICType=Panel...
```

the ServerName property returns:

servername

ServerPort

This property returns an integer representing the port on which the browser's request was received (that is, the port on which the server is listening.)

This property is read-only.

Example

For the URL

```
http://servername:80/servlets/iclientservlet/peoplesoft8/?ICType=Panel...
```

the ServerPort property returns:

80

Timeout

This property returns an integer representing the timeout value, in seconds, set in the configuration .properties file.

This property is read-only.

Response Class

The response object encapsulates all information to be returned from the Internet Script to the browser.

Response Class Methods

Clear

Syntax

```
clear()
```

Description

This method removes all cookies and headers as well as deletes any HTML that has been written to the Response object. After using this method, the Response object looks like it did when the script was first called.

CreateCookie

Syntax

```
CreateCookie(name)
```

Description

This method adds a cookie to the response with the name specified by the string *name*. It returns a reference to the cookie object that is used to update the values of this cookie. If the cookie by the specified name already exists, a reference to the existing cookie is returned. This method can be called multiple times to set more than one cookie. Cookie names may not contain the "\$" character.

GetCookie

Syntax

```
GetCookie(name)
```

Description

This method returns the cookie object specified by the string *name*. If the cookie is not already present in this response, a null value is returned. GetCookieNames

Syntax

```
GetCookieNames()
```

Description

This method returns an array of strings that contains the names of all the cookies present in this response. If there are no cookies in the response, an empty array is returned. GetHeader

Syntax

```
GetHeader(name)
```

Description

This method returns the value of the response header requested by the string *name*. The match between *name* and the response header is case-insensitive. If the header requested does not exist, an empty string is returned.

GetHeaderNames

Syntax

```
GetHeaderNames()
```

Description

This method returns an array of Strings representing the header names for this response.

GetImageURL

Syntax

```
GetImageURL (IMAGE.ImageName)
```

Description

This method returns a string which represents the URL of the requested image, for images in the database.

Example

The following example gets the URL to use for the image, and the second uses the URL reference.

```
&strImage = %Response.GetImageURL(Image.ARROW);  
  
%Response.WriteLine("");
```

GetJavaScriptURL

Syntax

```
GetJavaScriptURL (HTML.HTMLName)
```

Description

This method returns a string which represents the URL of the requested HTML object, for JavaScript HTML definitions stored in the database



For more information, see [Creating HTML Definitions](#).

Example

This example assumes the existence in the database of a HTML definition called “HelloWorld_JS”, that contains some JavaScript.

```
Function IScript_TestJavaScript()  
  
    %Response.WriteLine("<script src= " |  
    %Response.GetJavaScriptURL(HTML.HelloWorld_JS) | "></script>");
```



```
End-Function;
```

GetStyleSheetURL

Syntax

```
GetStyleSheetURL (STYLESHEET.stylename)
```

Description

This method returns a URL string pointing to a style sheet created in the Application Designer.

Example

In the following example, the first line gets the URL to use for the style sheet, and the second is the standard HTML to include the style sheet in the HTML document. The stylesheet link must be included in the HTML BEFORE any of the styles in the stylesheet are used.. It should be included in the <head> section of the html document.

```
&strStyleSheet = %Response.GetStyleSheetURL(StyleSheet.BENEFITS_STYLE);

%Response.WriteLine("<link rel=stylesheet href=" | &strStyleSheet | "
type="text/css">");
```

Now to use a style, assign the class attribute of your choice to your HTML tags, as follows:

```
%Response.WriteLine("<p class=PSTEXT>I am some classy text!!</p>");
```

RedirectURL

Syntax

```
RedirectURL(location)
```

Description

This method sends a temporary redirect response to the client using the location specified by the string location. The given location must be an absolute URL. Relative URLs are not permitted. No further output should be made by the Internet Script after calling this method.

Example

```
%Response.RedirectURL(URL);
```

where URL equals

```
http://servername/servlets/iclientservlet/peoplesoft8?ICType=Panel&Menu=ROLE_EMPLOYEE&Market=GBL&PanelGroupName=WEB_HEALTH_PLANS&PlanType=10;
```

SetContentType

Syntax

```
SetContentType (Type)
```

Description

This method sets the content type for this response. The parameter *type* takes a string value. This type may later be implicitly modified by the addition of properties such as the MIME charset property if the service finds it necessary and the appropriate property has not been set.

The content type defaults to “text/html” if it is not set.

SetHeader

Syntax

```
SetHeader (name, value)
```

Description

This method sets a response header with the specified name and value. Both parameters take string values. If the field has already been set with a value, this new value overwrites the previous one.

Write

Syntax

```
Write (String)
```

Description

This method prints *String* to the HTTP output stream.

You can use an HTML string from the Application Designer HTML catalog with the **Write** method if you also use the **GetHTMLText** function, as follows:

```
%Response.Write (GetHTMLTEXT (HTML.MY_HTML) ) ;
```

You can also use an XML string. The following example takes a BiDocs structure that contains an XML response and puts that into a text string. Once this is done, the %Response.Write function can send this as an XML response.

```
Local BiDocs &rootDoc;
```

```
Local string &xmlString;
```

```
&xmlString = %Response.GetContentBody();
```

```
&rootDoc = GetBIDoc(&xmlString);

/* do processing */

&xmlString = &rootDoc.GenXMLString();

%Response.Write(&xmlString);
```

WriteLine

Syntax

```
WriteLine(String)
```

Description

This method adds a carriage control and line feed to the end of the string *String*, then prints the string to the HTTP output stream.

You can use an HTML string from the Application Designer HTML catalog with the **WriteLine** method if you also use the **GetHTMLText** function, as follows:

```
%Response.WriteLine(GetHTMLTEXT(HTML.MY_HTML));
```

Cookie Class

The cookie class encapsulates all information to be sent to the browser for a single cookie. The cookie class is used by the response object to set new cookies. The request object uses a simple name-value pair mechanism for retrieving previously stored cookie values sent with the HTTP request.

Cookie Class Properties

Domain

This property returns the domain of this cookie, or null if not defined. This property is a string value.

This property is read-write.

Example

```
&cookie = &Response.AddCookie("My cookie", "My value");

&cookie.Domain = ".MyDomain.com";
```

MaxAge

This property represents the maximum specified age of the cookie, as a signed number in seconds. The default value is -1. Setting the MaxAge property to a negative value ensures the cookie will not persist on the client when the client session ends. If the MaxAge property is set to zero, the cookie will be deleted immediately from the client.

Value	Meaning
Non Negative Integer	Lifetime of the Cookie in seconds
0	Delete the cookie from the client immediately
-1	Default MaxAge property value (remove the cookie after the client exits)
Negative Integer	Remove the cookie after the client exits

This property is read-write.

Name

This property returns the name of the cookie as a string.

This property is read-only.

Path

This property returns the prefix of all the URL paths for which this cookie is valid, or an empty string if not defined.

This property is read-write.

Secure

This property specifies whether the cookie is to be secured. This property takes a boolean value. The default value is False. Secure cookies will only be sent if the client has established a secure link (such as HTTPS) with the server. This property is read-write.

Value

This property returns the value of the cookie (as a string), or an empty string if it isn't defined.

This property is read-write.

Java Class

Using the PeopleCode Java functions, you can access Java classes, or create instances of Java objects. Calling a Java program from PeopleCode can greatly extend your application. In addition, the PeopleCode integration with Java allows you to pass parameters such as record, record fields or rowsets into your Java program.

The following are the PeopleCode Java functions:

- CreateJavaArray
- CreateJavaObject
- GetJavaClass



For more information about Java in general, see <http://java.sun.com/index.html>

Supported Versions of Java

PeopleSoft supports Java version 1.2.2 for all its platforms. The Java Runtime Engine (JRE) comes bundled with PeopleSoft for the following platforms:

- Sun
- AIX
- HP
- NT

On other platforms, you may have to install the JRE yourself.



For more information, see your platform installation documentation.

Naming Classes and Packages

PeopleSoft recommends following Sun standards for naming your classes and packages.



For more information see the Sun web site, at <http://java.sun.com/docs/codeconv/index.html>.

Java classes delivered with PeopleSoft are generally in the following directories:

- <PS_HOME>\appserv\classes
- <PS_HOME>\class

For example, some of the Java programs delivered with PeopleTools are in the following directory:

```
<PS_HOME>\appserv\classes\com\peoplesoft\tools
```

Setting up your System to use your own Classes

If you only access the classes that come defined with PeopleSoft you don't need to do any additional setup.

If you create your own classes that you want to access with PeopleCode, you'll have to create a PS_CLASSPATH environment variable that points to where your classes are stored. The exact path depends on how you package your classes.

- If you package your Java classes in a JAR file, you have to specify the JAR file name (with the JAR extension) in the PS_CLASSPATH environment variable.

For example, if your JAR file exists in the following directory:

```
c:\myjava\com\mycompany\myproduct\
```

You would have to set your PS_CLASSPATH environment variable to the following:

```
c:\myjava\com\mycompany\myproduct\myjar.jar
```

- If you do **not** store the Java classes you create in a JAR file, the PS_CLASSPATH environment variable must point to the top level of your package hierarchy.

For example, suppose your Java classes were in the following package:

```
c:\myjava\com\mycompany\myproduct\
```

You would set the PS_CLASSPATH environment variable to:

```
c:\myjava
```

Like most environment variables, you can specify more than one entry. On Windows, the PS_CLASSPATH entries are separated by semicolons. On Unix, they're separated by colons. The following is for Windows:

```
c:\myjava;d:\myjava\com\mycompany\myproduct\myjar.jar; . . .
```

The following would be for Unix:

```
/etc/myjava:/home/me/myjava/com/mycompany/myproduct/myjar.jar: . . .
```

This PS_CLASSPATH setting must be set for the process that runs the Java. For example, if Java is being called from PeopleCode that is running on the Application Server, it is the Application

Server process that needs this setting. Similarly, if the Java is being run from PeopleCode in the Application Engine, that that run of the Application Engine has to have this setting.

Also when developing your own classes you must be aware that most Java virtual machines will cache the class definitions. This means that even if you change the class files, a running Java virtual machine (inside, say, an Application Server) that has loaded and is running the old version of the class files. It won't pick up the new version of the class files. You'll have to restart the Application Server to make it reload the new classes.



For more information on setting up an environment variable, see your own system documentation.

From PeopleCode to Java

State Management Concerns

The application server is **stateless** in the sense that it doesn't keep any information (state) for its clients between calls to it. For one reason, calls to the application server can use different actual servers for different calls. When you are using Java in the application server, you should be careful to not leave state in the Java virtual machine that would cause your application to fail if a different application server (which would use a different invocation of the Java virtual machine) was used for subsequent calls. One of the ways you can leave state in the virtual machine is to use static (class) variables.

Similar considerations to these apply using Java in Application Engine programs, though here the difficulty arises when you try to checkpoint and then restart the program. The restart starts with a Java virtual machine invocation that doesn't have any of the state you might have stored into the Java virtual machine before the checkpoint.

JavaObject variables cannot have Global or Component scope because of this lack of ability to save the state of these objects.

An example of this is issuing messages. When you're running with PeopleSoft Internet Architecture and issue a message, the message is produced by an end-user action, so the Application Server gathers up its state to return it to the browser. This state saving attempts to save the current PeopleCode execution state, causing it to issue an error because of the JavaObject.

The solution is to not have any non-Null JavaObject objects when the message is issued.

The following is a simple Java program:

```
public class PC_Java_Test{

    public String pcTest(){
```

```

        String message;

        message = "PeopleCode is successfully executing Java.";

        return message;
    }
}

```

Here is the PeopleCode that calls this Java program. Note that the JavaObject is set to NULL before the message is issued.

```

&java_test = CreateJavaObject("PC_Java_Test");

&java_message = &java_test.pcTest();

&java_test = Null;

WinMessage(&java_message);

```

In SavePreChange, Workflow or SavePostChange PeopleCode the situation is more complicated. Usually messages with a zero style parameter (no buttons other than OK and maybe Explain, therefore no result possible except OK) are queued up by the Application Server. They are output by the browser when the service completes, so the serialization won't happen until after the PeopleCode has finished, so you won't have to set your JavaObject to NULL. With other kinds of messages, you will need to do this.

CreateJavaObject Example

The following is an example program creating a Java object from a sample program that generates a random password.

```

/* Example to return Random Passwords from a Java class */

Local JavaObject &oGpw;

/* Create an instance of the object */

&oGpw = CreateJavaObject("com.PeopleSoft.Random.Gpw_Demo");

&Q = "1";

/* Call the method within the class */

```



```

&NEW_VALUE = &oGpw.getNewPassword(&Q, PSRNDMPSWD.LENGTH);

/* This is just returning one value for now */

PSRNDMPSWD.PSWD = &NEW_VALUE;

```

CreateJavaArray Example

Suppose we had a PeopleCode array of strings (&Parms) that we wanted to pass to a Java method xyz of class Abc. This example assumes that you don't know when you write the code just how many parameters you will have.

```

Local JavaObject &Abc, &RefArray;

Local array of String &Parms;

&Parms = CreateArray();

/* Populate array how ever you want to populate it */

&Abc = GetJavaObject("com.peoplesoft.def.Abc");

/* Create the java array object. */

&JavaParms = CreateJavaArray("java.lang.String[]", &Parms.Len);

/* Populate the java array from the PeopleCode array. */

&RefArray = GetJavaClass("java.lang.reflect.Array");

For &I = 1 to &Parms.Len
    &RefArray.set(&JavaParms, &I - 1, &Parms[&I]);
End-For;

/* Call the method. */

```

```
&Abc.xyz(&JavaParms);
```

GetJavaClass Example

The following example gets a system class.

```
&Sys = GetJavaClass("java.lang.System");

&Sys.setProperty("java.security.policy", "C:\java\policy");

WinMessage("The security property is: " |
&Sys.getProperty("java.security.policy"));

&Props = &Sys.getProperties();

&Props.put("java.security.policy", "C:\java\policy");

&Sys.setProperties(&Props);

WinMessage("The security property is: " |
&Sys.getProperty("java.security.policy"));
```

From Java to PeopleCode

The Java classes delivered with PeopleTools allow you to call PeopleCode from your Java program. Calling into PeopleCode only works from Java code that you have initially called from PeopleCode.

You must call PeopleCode facilities only from the same thread that was used for the call into Java. PeopleTools is not multithreaded.

You cannot call any PeopleCode facility that would cause the server to return to the browser for an end-user action, because the state of the Java computation cannot be saved and restored when the action is complete.



For more information, see Considerations when using the PeopleCode Java Functions.

SysVar Java Class

Use the SysVar Java Class to refer to System Variables, such as %Language or %DBType.

For example, %Session, becomes SysVar.Session()



For more information see System Variables.

SysCon Java Class

Use the SysCon Java Class to refer to system constants, such as %SQLStatus_OK or %FilePath_Absolute.

For example, %CharType_Matched becomes SysCon.CharType_Matched.

Func Java Class

Use the Func Java Class to refer to built-in functions, such as CreateRowset or GetFile.

For example, SetLanguage(LANG_CD) becomes Func.SetLanguage(LANG_CD)

Name Java Class

The **Name** Java Class allows you to use the PeopleSoft reserved item references. This allows you to reference pages, components, records, fieldnames, and so on.

For example, in PeopleCode you can refer to a record field using the following:

```
recname.fieldname
```

With the Name class, you can use a similar construct:

```
new PeopleSoft.PeopleCode.Name("RECNAME", "FIELDNAME");
```

Note that these **must** be in the exact case as the item. As all PeopleTools items are named in upper case, that means you must upper case.

As another example, in PeopleCode you can refer to a page using the following:

```
PAGE.pagename
```

In Java, it would be:

```
new PeopleSoft.PeopleCode.Name("PAGE", "PAGENAME");
```

Using PeopleCode Objects

The existing PeopleCode classes (like Array, Rowset, and so on) have properties and methods you can access.

- PeopleCode classes have the same names, so Record becomes Record, SQL becomes SQL, and so on.
- Methods are accessed by the method name.

- The name of a property is pre-pended with either **get** or **set**, depending on whether you're reading or writing to the property.

For example, to get the `IsChanged` property would be `getIsChanged`. To set the value for a field would be `&MyField.setValue`.

Here is an example of a Java program that uses PeopleCode objects to access the database:

```
/*
 * Class Test
 *
 * This code is used to test the Java/PeopleCode interface.
 */

import PeopleSoft.PeopleCode.*;

public class Test {

/*
 * Test
 *
 * Add up and return the length of all the
 * item labels on the UTILITIES menu,
 * found two different ways.
 */

public static int Test() {
/* Get a Rowset to hold all the menu item records. */
Rowset rs = Func.CreateRowset(new Name("RECORD", "PSMENUITEM"), new Object[]{});
String menuName = "UTILITIES";
int nRecs = rs.Fill(new Object[]{"WHERE FILL.MENUNAME = :1", menuName});

int i;
int nFillChars = 0;
for (i = 1; i <= rs.getActiveRowCount(); i++) {
String itemLabel = (String)rs.GetRow(i)
.GetRecord(new Name("RECORD", "PSMENUITEM"))
.GetField(new Name("FIELD", "ITEMLABEL"))
.getValue();
nFillChars += itemLabel.length();
}

/* Do this a different way - use the SQL object to read each menu
item record. */

int nSQLChars = 0;
Record menuRec = Func.CreateRecord(new Name("RECORD", "PSMENUITEM"));
SQL menuSQL = Func.CreateSQL("%SelectAll(:1) WHERE MENUNAME = :2",
new Object[]{menuRec, menuName});

while (menuSQL.Fetch(new Object[]{menuRec})) {
String itemLabel = (String)menuRec
```

```

        .GetField(new Name("FIELD", "ITEMLABEL"))
        .getValue();
    nSQLChars += itemLabel.length();
}

return nFillChars + 100000 * nSQLChars;
}
}

```

This can be run from PeopleCode like this:

```

Local JavaObject &Test;

Local number &chars;

&Test = GetJavaClass("Test");

&chars = &Test.Test();

&Test = Null;

WinMessage("The character counts found are: " | &chars, 0);

```

Mapping PeopleCode and Java Data Types

The following table describes the matching of types for resolution of overloaded Java methods and basic conversions. The first Java Type/Class is the one that is produced in the absence of any other type of information.

<i>PeopleCode Type</i>	<i>Java Type/Class</i>
Float	double, float
Number	double, float, byte, char, short, int, long
Integer	int, byte, char, short, long
Boolean	boolean
String	java.lang.String
Date	java.sql.Date
Time	java.sql.Time
DateTime	java.util.Date
any kind of object	any kind of object

The following table represents the conversions done to produce the Java class java.lang.Object. In addition to these, the conversions (listed in the table above) from String onwards are done to produce a java.lang.Object.

PeopleCode Type	Java Type/Class
Float, Number	java.lang.Double
Integer	java.lang.Integer
Boolean	java.lang.Boolean

The following table represents other conversions that will be done as required by the signature of a Java method or constructor.

PeopleCode Type	Java Class
Integer, Number, Float	java.lang.Integer, java.lang.Byte, java.lang.Character, java.lang.Short, java.lang.Long, java.lang.Float, PeopleSoft.PeopleCode.intHolder, PeopleSoft.PeopleCode.doubleHolder
String	PeopleSoft.PeopleCode.StringHolder
peoplecode builtin class Xxx	PeopleSoft.PeopleCode.Xxx
JavaObject	corresponding java object

Considerations when using the PeopleCode Java Functions

Some PeopleCode built-in functions can't be called from a Java program. Many of these restrictions arise because you can't serialize Java objects. Inside Java, you can't serialize and save the state of the Java Virtual Machine (JVM.)

This means that you cannot call the following built-in functions:

- Think-time functions, such as DoCancel, Prompt, RevalidatePassword, and so on.



For more information, see Think-Time Functions.

- Math functions, such as Abs, Cos and Sin. Use the Java math functions instead.
- Java class functions, such as CreateJavaArray, CreateJavaObject, or GetJavaClass.
- Deprecated functions, such as SetNextPanel, PanelGroupChanged, and so on. Use the new function instead, like SetNextPage, ComponentChanged, and so on.
- PeopleCode language elements, such as Exit, Return, If, and so on.
- Cursor position, such as SetCursorPos.
- Functions that rely on specific character encoding, such as char, code, exact, codeb, and so on.
- Menu appearance functions, such as CheckMenuItem, HideMenuItem, and so on.

- Transfer functions such as Transfer or TransferPage.
- DoSaveNow isn't allowed. However, DoSave is.

When you're creating your Java program, you should keep the following in mind:

- If you're starting from an online application, should avoid calling anything that will wait a long time.
- If you use third-party Java that require third-party platforms, your limiting your program to just those platforms.

Error Handling and the PeopleCode Java Functions

Java functions throw exceptions to indicate something unusual has happened. However, all exceptions from Java called by PeopleCode are turned into fatal errors. The PeopleCode program is terminated, and the user transaction must be canceled.

PeopleSoft recommends that you write a Java wrapper to handle your errors.

Use either the All or None built-in functions to check values that are returned if you think you may call a Java method that is defined to return a string, but returns a Null object reference instead. Java Null object references are automatically converted into PeopleCode Null object references.

Using the Java Debugging Environment

To use Sun's JPDA V1.0 debugging architecture, you must do the following. These instructions are general: how you actually set up the debugger depends on your system.

To use the Java Debugging Environment (JDB):

1. Download and install a copy of JPDA V1.0.



For more information, see <http://java.sun.com/products/jpda>.

2. Set the path for your system.

Suppose you install JPDA V1.0 in C:\jpda. Set your PATH environment variable to include C:\jpda\bin.

3. Set the path for the application server

For the application server, set the Domain Settings/Add to PATH to include C:\jpda\bin.

4. Set the JavaVM Options

Set the JavaVM Options to be something like the following (see the JPDA documentation for a more complete example):

```
-Xdebug -Djava.compiler=NONE -Xnoagent -  
Xrunjdpw:transport=dt_socket,suspend=n,address=8765,server=y
```

5. Run the debugger.

After starting the tools session and causing it to start the JVM, you can use the JDB command line debugger that comes with JPDA, using a command like the following:

```
jdb -connect com.sun.jdi.SocketAttach:port=8765
```

You can also use the (no cost) Forte for Java Community Edition IDE from Sun or any of the Java IDEs noted on the JPDA pages.



For more information, see <http://www.sun.com/forte/ffj/ce/index.html>.

Declaring a Java Object

You should declare a Java object as type `JavaObject`. For example:

```
Local JavaObject &MyJavaClass;
```



Java objects can only be declared as type `Local`.

Scope of a Java Object

A Java object can only be instantiated from PeopleCode. This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Component Interface PeopleCode, record field PeopleCode, and so on.

PeopleCode Java Functions

The following are the PeopleCode Java functions:

- `CreateJavaArray`
- `CreateJavaObject`
- `GetJavaClass`

Message Class

The Message class is used to create and access an application message from PeopleCode. Application messages are self describing messages that contain application data. Messages are the heart of the messaging architecture. They store the data that is passed between systems.

Message definitions are simply multi-level structures, consisting of records. Publishing a message basically stores the message in a set of relational tables, the application messaging queue.

Messages can be published (created) using an Application Engine program or component. Subscription PeopleCode programs subscribe to messages and process the data.



For more information, see PeopleSoft Application Messaging.

The Message class is built on top of the Rowset, Row, Record, and Field classes, so the PeopleCode written to populate or access a Message looks similar to PeopleCode written to populate or access a Component buffer.



For more information, see Data Buffer Access.

Application Messaging PeopleCode Events

OnPublishTransform and **OnSubscribeTransform** are two PeopleCode events. The PeopleCode programs associated with these events are only associated with message definitions. Only one instance of each of these PeopleCode programs can be defined for a single message. These PeopleCode events should include PeopleCode that executes version logic for messaging.

Another event associated with the message definition is **Subscription** PeopleCode. These events fire when the message is subscribed to. Use this PeopleCode to run your subscription processing for validating and loading the message data.

The **OnRoutePublication** and **OnRouteSubscription** are two PeopleCode events that are associated with the message channel. These routing rule events are used to determine where the messages are routed to and received from.



For more information about how to use the Application Messaging PeopleCode events, see PeopleSoft Application Messaging.

Declaring a Message Object

Messages are declared using the Message data type. For example,

```
Local Message &MSG;
```

Scope of a Message Object

A message object can only be instantiated from PeopleCode. This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Component Interface PeopleCode, record field PeopleCode, and so on.



For more information, see PeopleSoft Application Messaging.

Populating a Message Object

After you've declared and instantiated your message object, you want to populate it with data. If your data is coming from Component buffers, populating your message is very easy.

A message definition contains a hierarchy of records. A Component buffer contains a hierarchy of records. If you want to copy data from a Component buffer rowset to a message, the **structure** of the message and the component must be the same. That is, if you have a record at level 2 in your message and you want that data, you must have the same level 0 and level 1 records in your message as in your component.

For example, suppose your component had the following structure (that is, that PO_INFO and PO_LINE are at the same level, and PO_DETAIL is the child of PO_INFO):

```
PO_HEADER
```

```
    PO_LINE
```

```
    PO_INFO
```

```
        PO_DETAIL
```

If you wanted to include the information in the PO_DETAIL record, you would have to have *at least* the following record structure in your message:

```
PO_HEADER
```

```
    PO_INFO
```

```
        PO_DETAIL
```

Any records that are in the page that aren't in the message (and vice-versa) are ignored.

After you get your message object, you can create a rowset from it. This rowset will have the same structure as the message. If the message is empty, the rowset has no data. If the message has data, the rowset is automatically populated.

The following example is the simplest way of populating a message with data. This assumes that the structure of the message is the same as the structure of the page.

```

/* this gets all the data in the Component buffer */

&RS = GetLevel0();

/* this instantiates a message object */

&MSG = CreateMessage(MESSAGE.MY_MESSAGE);

/* creates a rowset with the same structure as the message */

&MSG_RS = &MSG.GetRowset();

/* this copies all the data from the page to the message */

&RS.CopyTo(&MSG_RS);

/* this publishes the message */

&MSG_RS.Publish();

```

A message rowset is the same as a Component buffer rowset, or any other rowset. It is composed of rows, records, and fields. Suppose you didn't want to get all the data from the Component buffer, but instead wanted to populate just a particular record in your message.

To access a record in a message rowset is the same as accessing a record in a Component buffer rowset. You must instantiate the rowset, then specify the **row** before you can access the record.

The following selects values into a record, then uses the record method **CopyFieldTo** to copy from the Component buffer record to the message record.

```

Local SQL &LN_SQL;

Local Message &STOCK_MSG;

Local Rowset &HDR_RS, &LN_RS;

Local Record &LN_REC, &ln_rec_msg;

&STOCK_MSG = CreateMessage(Message.STOCK_REQUEST);

&HDR_RS = &STOCK_MSG.GetRowset();

&LN_REC = CreateRecord(Record.DEMAND_INF_INV);

&LN_SQL = CreateSQL("Select * from PS_DEMAND_INF_INV where BUSINESS_UNIT= :1
and ORDER_NO = :2", &BUSINESS_UNIT, &ORDER_NO);

```

```

&J = 1;

While &LN_SQL.Fetch(&LN_REC)

    /* copy data into the Level 1 of &STOCK_MSG object */

    &LN_RS = &HDR_RS(&I).GetRowset(1);

    If &J > 1 Then

        &LN_RS.InsertRow(&J - 1);

    End-If;

    &ln_rec_msg = &LN_RS.GetRow(&J).GetRecord(Record.DEMAND_INF_INV);

    &LN_REC.CopyFieldsTo(&ln_rec_msg);

    &J = &J + 1;

End-While;

```

Considerations when Populating a Rowset from a Message

Suppose your message had two rowsets, one at level 0, a second at level 1. In the message, only the level 0 rowset contains any data. When you use `GetRowset` to create a rowset for the entire message, the rowset at level 1 will contain an empty row, even if there isn't any data in it. (This is standard behavior for *all* rowsets.) However, you can use the `IsChanged` property on the record object to determine the status of the data.

The following subscription PeopleCode example traverse the Rowset. (Notice the use of **ChildCount**, **ActiveRowCount** and **IsChanged**).

‘...’ is where application specific code would go.

```

&MSG_ROWSET = &MSG.GetRowset();

For &A0 = 1 To &MSG_ROWSET.ActiveRowCount

    /*-----*/

    /* Process Level 1 Records */

    /*-----*/

```

```

If &MSG_ROWSET(&A0).ChildCount > 0 Then

For &B1 = 1 To &MSG_ROWSET(&A0).ChildCount

&LEVEL1_ROWSET = &MSG_ROWSET(&A0).GetRowset(&B1);

    For &A1 = 1 To &LEVEL1_ROWSET.ActiveRowCount

        If &LEVEL1_ROWSET(&A1).GetRecord(1).IsChanged Then

            . . .

/*****/

/* Process Level 2 Records */

/*-----*/

        If &LEVEL1_ROWSET(&A1).ChildCount > 0 Then

            For &B2 = 1 To &LEVEL1_ROWSET(&A1).ChildCount

                &LEVEL2_ROWSET = &LEVEL1_ROWSET(&A1).GetRowset(&B2);

                For &A2 = 1 To &LEVEL2_ROWSET.ActiveRowCount

                    If &LEVEL2_ROWSET(&A2).GetRecord(1).IsChanged Then

                        ...

/*****/

/* Process Level 3 Records */

/*-----*/

                            If &LEVEL2_ROWSET(&A2).ChildCount > 0 Then

                                For &B3 = 1 To &LEVEL1_ROWSET(&A2).ChildCount

                                    &LEVEL3_ROWSET = &LEVEL2_ROWSET(&A2).GetRowset(&B3);

                                    For &A3 = 1 To &LEVEL3_ROWSET.ActiveRowCount

                                        If &LEVEL3_ROWSET(&A3).GetRecord(1).IsChanged Then

                                            ...

                                            End-If; /* A3 - IsChanged */

```

```

                                End-For; /* A3 - Loop */

                                End-For; /* B3 - Loop */

                                End-If; /* A2 - ChildCount > 0 */

/*-----*/

/* End of Process Level 3 Records */

/*****/

                                End-If; /* A2 - IsChanged */

                                End-For; /* A2 - Loop */

                                End-For; /* B2 - Loop */

                                End-If; /* A1 - ChildCount > 0 */

/*-----*/

/* End of Process Level 2 Records */

/*****/

                                End-If; /* A1 - IsChanged */

                                End-For; /* A1 - Loop */

                                End-For; /* B1 - Loop */

                                End-If; /* A0 - ChildCount > 0 */

/*-----*/

/* End of Process Level 1 Records */

/*****/

                                End-For; /* A0 - Loop */

```

Considerations for Publishing and Subscribing to Partial Records

If you've selected to not publish all the fields in the message, you must be careful when inserting that data into a record.

Deselecting the ***Include*** checkbox in the message definition means that the field is **excluded** from the message definition.

- The field won't be included when generating an XML document (when publishing a message.)
- If the field is present in the XML (that is, it wasn't excluded from the published message) it will be ignored by the subscribing system.

When you insert the data from the message into the database, you *must* set the values for the fields that aren't in the message. You can use the SetDefault record object method to do this. You could also use a Component Interface based on the component the message was created from to leverage the component defaults.



For more information about using Component Interfaces in subscription, see Designing Component Interface Subscribe EIPs.

Considerations when Subscribing to Character Fields

If a message definition has character fields that are defined as uppercase, when the message is subscribed to, character data for those fields is automatically converted to uppercase.

Working with XML

When the message data is loaded from the message queue to the message buffer, certain translations may occur. For example, lowercase data can be converted to uppercase for uppercase fields, or non-matching fields would not be copied over, and so on. The GenXMLString and GenFormattedXMLString methods take the potentially translated message buffer data and generates XML from that data.

The GetRawXML and GetFormattedRawXML methods return the raw data from the message queue, which is already in XML format, before any kind of conversion take place.

Error Handling

In your subscription PeopleCode, you may want to validate the information received in the message. If you find that the data isn't valid, use Exit(1) to end your PeopleCode program. This will set the status of the message to ERROR, log any error messages to the Application Message Queues, and automatically roll-back any database changes you may have made.

There are many ways to validate the data and capture the error messages:

- Use ExecuteEdits on the Message object
- Use ExecuteEdits on the Record object
- Use a Component Interface (See Designing Component Interface Subscribe EIPs)

Write your own validation PeopleCode in the Subscription program:

The following example validates the Business Unit of a message against an array of valid BU's:

```
For &I = 1 To &ROWSET.RowCount;

    &POSITION = &BUArray.Find(&ROWSET(&I).GetRecord(1).BUSINESS_UNIT.Value);
```

```
If &POSITION = 0 Then

    &Status = "ERROR: BU not Found or not Active";

    &ROWSET(&I).BCT_ADJS_MSG_VW.BUSINESS_UNIT.IsEditError = True;

    &ROWSET(&I).BCT_ADJS_MSG_VW.BUSINESS_UNIT.MessageSetNumber = 11100;

    &ROWSET(&I).BCT_ADJS_MSG_VW.BUSINESS_UNIT.MessageNumber = 1230;

    /* The error message 11100-1230 reads: This Business Unit is not a valid, open,
    Inventory Business Unit */

    End-If;

End-For;
```

In the calling PeopleCode, the program calls Exit(1) based on the value of &Status.



For more information, see Processing Messaging Errors.



Note. All errors for Subscription PeopleCode get logged to the application message error table, *not* to the PSMessages collection (on the session object.)

Message Class Built-In Functions

AddSystemPauseTimes

CreateMessage

DeleteSystemPauseTimes

GetMessage

GetMessageInstance

GetPubContractInstance

GetSubContractInstance

ReturnToServer

PingNode

SetChannelStatus

Message Class Methods

Cancel

Syntax

Cancel ()

Description

The Cancel method allows you to programmatically cancel a message, much the same as you can do in the message monitor.



For more information, see Application Message Monitor.

The action of this method depends on the type of the invoking message object:

- If invoked with a message instance object, all pending publication and subscription contracts are canceled.
- If invoked with a publication contract object, the corresponding publication contract is canceled.
- If invoked with a subscription contract object, the corresponding subscription contract is canceled.

The method is only available when the message instance has one of the following statuses:

- Error
- New
- Retry
- Timeout
- Edited

Parameters

None.

Returns

None.

Example

```
/* Load Message Instance */
```

```
&Msg = GetMessageInstance(&PubId, &PubNode, &ChnlName);

/* Or load publication contract or subscription contract

&Msg = GetPubContractInstance(&PubId, &PubNode, &ChnlName, &SubNode);

&Msg = GetSubContractInstance(&PubId, &PubNode, &ChnlName, &MsgName, &SubName);

*/

/* user cancels */

&Msg.Cancel();
```

Related Topics

GetMessageInstance, GetPubContractInstance, GetSubContractInstance functions, Application Message Monitor

Clone

Syntax

```
Clone()
```

Description

The **Clone** method creates an identical copy of the message, copying the data tree of the message executing the method. The **Clone** function sets the following properties for the resulting message object, based on the values set for the message definition:

- Name
- ChannelName
- IsActive

Other properties are set when the message is published or subscribed to (PubID, SubName, and so on), or are dynamically generated at other times (Size, IsEditError, and so on.)

Clone creates a unique version of the message. If the original message changes, it will not be reflected in the cloned object.



For more information, see Object Assignment.

Parameters

None.

Returns

A message object.

Example

Clone could be used in a Hub and Spoke messaging environment. The Hub node uses this method during subscription processing to publish a copy of the messages it receives from the Spokes.

```
&Msg = GetMessage();

&Clone = &Msg.Clone();

&Clone.Publish();
```

The hub's publication routing rules are then used to determine which spokes receive the message.

Related Topics

CopyRowset and CreateMessage function

CopyRowset

Syntax

```
CopyRowset(source_rowset [, record_list]);
```

Where *record_list* is a list of record names in the form:

```
RECORD.source_recname1, RECORD.target_recname1

[, RECORD.source_recname2, RECORD.target_recname2] . . .
```

Description

The **CopyRowset** method copies *all* data from the source rowset to the *like-named* fields in the message object executing the method. This is an easy way to populate a message with data from a component.



CopyRowset will copy all the data, including rows that haven't been modified. If you only want to copy data that has changed in some way, see CopyRowsetDelta.

When the record names in the message and component do not match exactly, use the optional *record_list* to specify the records to be copied from the component into the message.

When you use the **CopyRowset** method to copy the contents from the source rowset to the message object, you are creating a *unique* copy of the object. If you change the original rowset, the message object will not be changed.



Note. You can only execute CopyRowset against a message once. After a message is populated, any other CopyRowsets will be ignored. If you have to populate different levels of a message rowset separately, you can use the CreateRowset method to create a rowset that has the same structure as your message, populate the created rowset, then use CopyRowset to populate your message. You can also use the Rowset **CopyTo** method to populate a message rowset, then populate the PSCAMA record by hand.

Parameters

<i>source_rowset</i>	Specifies the name of the rowset to be copied into the message object.
<i>record_list</i>	Specifies specific source and target records to be copied into the message object.

Returns

None.

Example

The following example copies an entire component into a message object.

```
&Msg = GetMessage();

&Component_Rowset = GetLevel0();

&Msg.CopyRowset(&Component_Rowset);
```

The following example copies a header/line page rowset into a header/line message object, using the *record_list* because the record names don't match:

```
&Msg = GetMessage();

&Component_Rowset = GetLevel0();

&Msg.CopyRowset(&Component_Rowset, RECORD.PO_HDR_VW, RECORD.PO_HDR,
RECORD.PO_LINE_VW, RECORD.PO_LINE);
```

Related Topics

CopyRowsetDelta, GetRowset function, Rowset Class, Fill rowset class method

CopyRowsetDelta

Syntax

```
CopyRowsetDelta(source_rowset [, record_list]);
```

Where *record_list* is a list of record names in the form:

```
[RECORD.source_recname1, RECORD.target_recname1  
[, RECORD.source_recname2, RECORD.target_recname2]] . . .
```

Description

The **CopyRowsetDelta** method copies rows of data that have changed from the source rowset to the *like-named* records and *like-named* fields in the message object executing the method.



Note. **CopyRowsetDelta** copies all the like-named fields from the changed **row** into the message. It is *not* copying just the changed like-named fields.

When the record names in the message and component do not match exactly, use the optional *record_list* to specify the records to be copied from the component into the message. The specified target records must be records in your message, while the specified source records must be records in a rowset that exists in the data buffer and is populated.

This is an easy way to populate a message when the records in the message match the records in the component that the message is published from.

In addition, the **CopyRowsetDelta** method sets the AUDIT_ACTN field in the PSCAMA table for every row in the message. The subscription process can then use PSCAMA.AUDIT_ACTN to determine how to process every row that was published.

The set values match those used in audit trail processing, that is:

- A – Row inserted.
- D – Row deleted.
- C - Row changed (updated), but no key fields changed. The system copies the new value to the Message rowset.
- K - Row changed (updated), and at least one key field changed. The system copies the old value to the Message rowset as well as the new value (see "N" below).
- N - Row changed (updated), and at least one key field changed. The system copies the old value to the Message rowset as well as the new value (see "K" above).
- "" – blank value means that nothing on that row has changed. This is the default value, and is the value used to tag the parent rows of children that have changed.



If a child row is inserted (or changed or deleted) CopyRowsetDelta also copies the parent row (and the parent row of that row, and so on, up to the top row) so the subscriber has a full picture of the transaction. A blank value is set in the AUDIT_ACTN field for these rows to let the subscriber know they don't have to take action; the parent rows are just there for reference.

The Audit_Action values "A", "C", "D" are set when a record is added, changed or deleted, respectively. In some cases like, effective dated records, user may change a key field value, such as Effective Date. In response to such an user action, two records are created, one with an Audit_Action value of "N", and the other with Audit_Action value "K". The "N" record has all the new values, while the "K" record retains the old values.



For more information, see Common Application Messaging Attributes (PSCAMA).

When you use the **CopyRowsetDelta** method to copy the contents from the source rowset to the message object, you are creating a *unique* copy of the object. If you change the original rowset, the message object will **not** be changed.



For more information, see Object Assignment.



You can only execute CopyRowsetDelta against a message once. After a message is populated, any other CopyRowsetDeltas will be ignored. If you have to populate different levels of a message rowset separately, you can use the CreateRowset method to create a rowset that has the same structure as your message, populate the created rowset, then use CopyRowsetDelta to populate your message. You can also use the Rowset **CopyTo** method to populate a message rowset, then populate the PSCAMA record by hand.

Parameters

source_rowset

Specifies the name of the rowset to be copied into the message object.

record_list

Specifies source and target records to be copied into the message object. The target records must be records in your message, while the source records must be records in a rowset that exists in the data buffer *and* is populated.

Returns

None.

Example

The following example copies all the changed rows of data from a component into a message object.

```
&Component_Rowset = GetLevel0();

&Msg.CopyRowsetDelta(&Component_Rowset);
```

Related Topics

CopyRowset, GetRowset function, Rowset Class

ExecuteEdits

Syntax

```
ExecuteEdits([editlevels]);
```

where *editlevels* is a list of values in the form:

```
editlevel1 [+ editlevel2] . . .];
```

and where *editleveln* is one of the following system constants:

- %Edit_DateRange
- %Edit_PromptTable
- %Edit_Required
- %Edit_TranslateTable
- %Edit_YesNo

Description

The **ExecuteEdits** method executes the standard system edits on every field for every record in the message executing the method. All edits are based on the record edits defined for the record that the message is based on. The types of edit(s) performed depends on the *editlevel*. If no *editlevel* is specified, all system edits defined for the record definition are executed, that is:

- Reasonable Date Range (Is the date contained within plus or minus 30 days from the current date?)
- Prompt Table (Is field data contained in the specified prompt table?)
- Required Field (Do all required fields contain data? For numeric or signed fields, it checks for NULL or 0 values.)
- Translate Table (Is field data contained in the translate table?)
- Yes/No (Do all yes/no fields only contain only yes or no data?)

If any of the edits fail, the status of the property `IsEditError` is set to `True` for the Message Object executing the method, as well as any Rowset, Row, Record or Field Objects that could be instantiated from the same data as the Message Object. In addition, the Field class properties **MessageNumber** and **MessageSetNumber** are set to the number of the returned message and message set number of the returned message, for the field generating the error.



For more information, see the `EditError` and `MessageNumber` and `MessageSetNumber` field class properties.

If you want to check the field values only for a particular record, you can use the **ExecuteEdits** method with a record object.



For more information, see `ExecuteEdits` Record class method.

In addition, if you want to dynamically set the prompt tables used for a field (with the `%EditTable` function) you must use the **SetEditTable** method.



For more information, see `SetEditTable` Message class method.

Considerations for `ExecuteEdits` and `SetEditTable`

If an effective date is a key on the prompt table, and the record being edited doesn't contain an `EFFDT` field, the current datetime will be used as the key value.

If a `SETID` is a key on the prompt table, and the record being edited doesn't contain a `SETID` field, the system will look for `SETID` on the other records in the current row first, then search parent rows.

Considerations for `ExecuteEdits` and Numeric Fields

A 0 (zero) may or may not be a valid value for a numeric field. `ExecuteEdits` handles numeric fields in different ways, depending on whether the field is required and a prompt table is used:

- If the numeric field is required and there is *no* prompt edit: 0 is considered valid, so numeric fields never fail the required test.
- If the numeric field is required and there is a prompt edit: Prompt edit is run and the value must be valid.
- If the numeric field is not required and there is a prompt edit: 0 is considered valid, so the prompt edit is skipped.

Parameters

editlevel

Specifies the standard system edits to be performed against every field on every record. If *editlevel* isn't specified, all system edits are performed. *editlevel* can be any of the following system constants:

- %Edit_DateRange
- %Edit_PromptTable
- %Edit_Required
- %Edit_TranslateTable
- %Edit_YesNo

Returns

None.

Example

The following is an example of a call to execute Required Field and Prompt Table edits:

```
&MSG.ExecuteEdits(%Edit_Required + %Edit_PromptTable);
```

The following is an example showing how ExecuteEdits() could be used:

```
&MSG.ExecuteEdits();  
  
If &MSG.IsEditError Then  
  
    Exit(1);  
  
End-If;
```

Related Topics

SetEditTable method, EditError, MessageNumber, MessageSetNumber Field class properties, IsEditError record class property, and Processing Messaging Errors.

GenXMLString

Syntax

```
GenXMLString()
```

Description

The **GenXMLString** method creates an XML string based on the message after it's loaded from the message buffer.



For more information, see [Working with XML](#).

Parameters

None.

Returns

An XML string.

Example

In the following example, the first three lines of code creates the message object and copies the data from a Component buffer into the message object. The last line creates an XML string based on that message object.

```
&MSG = CreateMessage (MESSAGE.PO_INSERT) ;  
  
&RS = GetLevel0() ;  
  
&MSG.CopyRowset (&RS) ;  
  
&XML_STRING = &MSG.GenXMLString() ;
```

Related Topics

[LoadXMLString](#), [GenFormattedXMLString](#), [GetRawXML](#), [GetFormattedRawXML](#)

GenFormattedXMLString

Syntax

```
GenFormattedXMLString()
```

Description

The **GenFormattedXMLString** method creates a formatted XML string based on the message, after it's loaded from the message buffers.



For more information, see [Working with XML](#).

Parameters

None.

Returns

An XML string.

Example

In the following example, the first three lines of code creates the message object and copies the data from a Component buffer into the message object. The last line creates an XML string based on that message object.

```
&MSG = CreateMessage(MESSAGE.PO_INSERT);  
  
&RS = GetLevel0();  
  
&MSG.CopyRowset(&RS);  
  
&XML_STRING = &MSG.GenFormattedXMLString();
```

Related Topics

LoadXMLString, GenXMLString, GetRawXML, GetFormattedRawXML

GetFormattedRawXML

Syntax

```
GetFormattedRawXML([version])
```

Description

The **GetFormattedRawXML** method generates a formatted XML string based on the message before it's loaded from the message buffer.



For more information, see Working with XML.

Parameters

version

Specifies which version of the message you want to retrieve. Must be a valid version, expressed as a character string, of an existing message in the application message queue.

Returns

A formatted XML string.

Related Topics

LoadXMLString, GenXMLString, GenFormattedXMLString, GetRawXML

GetMessageVersion

Syntax

```
GetMessageVersion(version)
```

Description

GetMessageVersion loads data for the specified message version from the application message queue. The specified message version must exist. This method could be used in **OnPublishTransform** or **OnSubscribeTransform** PeopleCode to retrieve one version of a message and transform it into another version.

Parameters

<i>version</i>	Specifies which version of the message you want to retrieve. Must be a valid version, expressed as a character string, of an existing message in the application message queue.
----------------	---

Returns

A boolean value: True if successful, False otherwise.

Example

The following example changes the message from the first version to the second version.

```
Local Message &MSG1, &MSG2;

Local Rowset &MSG1_PERS_DATA0, &MSG2_PERS_DATA0, &MSG1_JOB_DATA1,
&MSG2_JOB_DATA1;

If &MSG1.GetMessageVersion("VERSION_1") Then

    &MSG1_PERS_DATA0 = &MSG1.GetRowset("VERSION_1");

    &MSG2_PERS_DATA0 = &MSG2.GetRowset("VERSION_2");

    /* Version 2 had an extra record field (JOB_DATA.WEB) that needs to be populated
    before it can be published */

    For &I = 1 To &MSG1_PERS_DATA0.ActiveRowCount
        &MSG1_PERS_DATA0(&I).CopyTo(&MSG2_PERS_DATA0(&I));
    End For
End If
```

```

&MSG1_JOB_DATA1 = &MSG1_PERS_DATA0 (&I) .GetRowset (RECORD.JOB) ;

&MSG2_JOB_DATA1 = &MSG2_PERS_DATA0 (&I) .GetRowset (RECORD.JOB) ;

For &J = 1 To &MSG1_JOB_DATA1.ActiveRowCount

    &MSG1_JOB_DATA1 (&J) .CopyTo (&MSG2_JOB_DATA1 (&J)) ;

    &MSG2_JOB_DATA1 (&J) .JOB_DATA.WEB.SetDefault () ;

End-For;

End-For;

&MSG1.Update ("VERSION_1", "VERSION_2");

End-If;

```

Related Topics

GetRowset, Update

GetRawXML

Syntax

```
GetRawXML ([version])
```

Description

The **GetRawXML** method generates an XML string based on the message before it's loaded from the message buffer.



For more information, see Working with XML.

Parameters

version

Specifies which version of the message you want to retrieve. Must be a valid version, expressed as a character string, of an existing message in the application message queue.

Returns

An XML string.

Related Topics

LoadXMLString, GenXMLString, GenFormattedXMLString, GetFormattedRawXML

GetRowset

Syntax

```
GetRowset ([version])
```

Description

The **GetRowset** method returns a standard PeopleCode level 0 rowset object for the specified message version. It creates an empty rowset for the specified message version if it doesn't already exist. If no message version is specified, the default message version is used.

When you use the **GetRowset** method, you are *not* creating a unique copy of the object. You are only making a copy of the reference. If you change the rowset, the original message will be changed.



For more information, see Object Assignment.

Parameters

<i>version</i>	An optional parameter, specifying an existing message version. If <i>version</i> isn't specified, the default message version is used.
----------------	--

Returns

A Rowset object if successful.

Example

The following gets all the SET_CNTRL_REC rows related to the row on the page, then updates SETID with the value from the page. **GetRowset** is used to create a rowset based on the message structure, that is, an unpopulated rowset. Because the rowset is unpopulated, you can use the **Fill** method to populate with data from the page.

```
Local Message &MSG;

Local Rowset &MSG_ROWSET;

If FieldChanged(SETID) Then

    &MSG = CreateMessage(MESSAGE.SET_CNTRL_REC);

    &MSG_ROWSET = &MSG.GetRowset();

    &MSG_ROWSET.Fill("where SETCNTRLVALUE =:1 and REC_GROUP_ID =:2",
        SETCNTRLVALUE, REC_GROUP_ID);

    For &I = 1 To &MSG_ROWSET.ActiveRowCount
```

```
&MSG_ROWSET.GetRow(&I).SET_CNTRL_REC.SETID.Value = SETID;

&MSG_ROWSET.GetRow(&I).PSCAMA.AUDIT_ACTN.Value = "C";

End-For;


&MSG.Publish();


End-If;
```

Related Topics

Clone, [CreateMessage](#), GetMessageVersion, Rowset Class

LoadXMLString

Syntax

```
LoadXMLString(XMLstring)
```

Description

The **LoadXMLString** function loads the message object executing the method with the XML string *XMLstring*.

Parameters

<i>XMLString</i>	Specify the XML string to be loaded into the message object.
------------------	--

Returns

None.

Example

```
&MSG = CreateMessage(MESSAGE.PO_HEADER);

&MSG.LoadXMLString(&XML_STRING);
```

Related Topics

GenXMLString, GenFormattedXMLString

Publish

Syntax

`Publish()`

Description

The **Publish** method publishes a message to the application message **queue** for the default message version. This method will not publish the message if the IsActive property for the message definition, set in Application Designer, is False.



For more information, see Designing Application Messages.

The Publish method can be called from any PeopleCode event, but is most commonly called from an Application Engine PeopleCode step or from a component.

When publishing from a component, you should only publish messages from the SavePostChange event, either from record field or component PeopleCode. This way, if there's an error in your publish code, the data isn't committed to the database. Also, SavePostChange is the last Save event fired, so this way you'll avoid locking the database while the other save processing is happening.

If the message is successfully published, the PubID and PubNodeName properties will be set to the publication ID and publishing system node name.



If you're publishing a message from within an Application Engine program, the message won't actually be published until the calling program performs a COMMIT.

Parameters

None.

Returns

None.

Example

The following simply copies all the data from a page into a message and publishes it.

```
Local Message &MSG;  
  
Local Rowset &ROWSET;  
  
&MSG = CreateMessage(MESSAGE.MESSAGE_PO) ;  
  
&ROWSET = GetLevel0();
```



```
&MSG.CopyRowset (&ROWSET) ;

&MSG.Publish;
```

The following is an example of putting the Incremental Publish PeopleCode on a Record at Level 1 (or 2, 3). If you just do the normal publishing PeopleCode, you will get as many messages as there are records at that level (since the code will fire for each record).

To make sure the code only fires once - use the following code.

```
/*NAMES.EMPLID.WORKFLOW */

Local Message &MSG;

Local Rowset &RS;

&LEVEL = CurrentLevelNumber();

&CURRENT_ROW = CurrentRowNumber(&LEVEL);

&TOT_ROW = ActiveRowCount(EMPLID);

/* Only Publish the message once for all records on this level */

If &CURRENT_ROW = &TOT_ROW Then

    &MSG = CreateMessage(MESSAGE.NAMES);

    &RS = GetRowset();

    &MSG.CopyRowsetDelta(&RS);

    &MSG.Publish();

End-If;
```

Related Topics

IsActive property, Designing Application Messages

Resubmit

Syntax

```
Resubmit()
```

Description

The **Resubmit** method allows you to programmatically resubmit a message, much the same as you can do in the message monitor.



For more information, see Application Message Monitor.

You may want to use this method after an end-user has finished fixing any errors in the message data, and you want to resubmit the message, rerunning the subscription PeopleCode.

The action of this method depends on the type of the invoking message object:

- If invoked with a message instance object, all pending publication and subscription contracts are resubmitted.
- If invoked with a publication contract object, the corresponding publication contract is resubmitted.
- If invoked with a subscription contract object, the corresponding subscription contract is resubmitted.

The method is only available when the message instance has one of the following statuses:

- Error
- Timeout
- Edited
- Canceled

Returns

None.

Example

```
/* Load Message Instance */

&Msg = GetMessageInstance(&PubId, &PubNode, &ChnlName);

/* Or load publication contract or subscription contract
&Msg = GetPubContractInstance(&PubId, &PubNode, &ChnlName, &SubNode);
&Msg = GetSubContractInstance(&PubId, &PubNode, &ChnlName, &MsgName, &SubName);
*/

/* user finishes making changes to the data and saves. Then user resubmits. */
```

```
&Msg.Resubmit();
```



For another example of this method, see [Introducing the Error Handling Utility](#).

Related Topics

GetMessageInstance, GetPubContractInstance, GetSubContractInstance built-in functions, Application Message Monitor

SetEditTable

Syntax

```
SetEditTable(%PromptField, RECORD.recordname)
```

Description

The **SetEditTable** method works with the **ExecuteEdits** method. It is used to set the value of a field on a record that has its prompt table defined as *%PromptField* value. *%PromptField* values are used to dynamically change the prompt record for a field.

There are several steps to setting up a field that you can dynamically change its prompt table.

To set up a field with a dynamic prompt table:

1. Define a field in the DERIVED record called *fieldname*.
2. In the record definition for the field you want to have a dynamic prompt table, define the prompt table for the field as *%PromptField*, where *PromptField* is the name of the field you created in the DERIVED record.



For more information, see [Application Designer, Types of Record Definitions](#).

3. Use **SetEditTable** to dynamically set the prompt table.

%PromptField is the name of the field on the DERIVED work record. *RECORD.recordname* is the name of the record to be used as the prompt table.

```
&MSG.SetEditTable("%EDITTABLE1", Record.DEPARTMENT);

&MSG.SetEditTable("%EDITTABLE2", Record.JOB_ID);

&MSG.SetEditTable("%EDITTABLE3", Record.EMPL_DATA);

&MSG.ExecuteEdits();
```

Every field on a record that has the prompt table field %EDITTABLE1 will have the same prompt table, such as, DEPARTMENT.

Parameters

<i>%PromptField</i>	Specifies the name of the field in the DERIVED record that has been defined as the prompt table for a field in a record definition.
<i>RECORD.recordname</i>	Specifies the name of the record definition that contains the correct standard system edits to be performed for that field in the message.

Returns

None.

Related Topics

ExecuteEdits method, EditError, MessageNumber, MessageSetNumber Field class properties, IsEditError record class property, SetEditTable Record Class method

Update

Syntax

```
Update([versionlist])
```

where *versionlist* is an arbitrary list of message versions in the following format:

```
version1 [, version2] . . .
```

Description

The **Update** method updates a message in the application message queue with the specified message versions. If *versionlist* isn't specified, the default message version is used. This method is commonly used in the OnPublishTransform and OnSubscribeTransform PeopleCode events.

Parameters

<i>versionlist</i>	Specify an optional comma-separated list of message versions. If <i>versionlist</i> isn't specified, the default message version is used.
--------------------	---

Returns

None.

Related Topics

GetMessageVersion, GetRowset, GetPubContractInstance, GetSubContractInstance, PeopleSoft Application Messaging

Message Class Properties

ChannelName

This property references the name of the channel associated with the message definition. This property is set in Application Designer, when the message is created. The message instance keys are a subset of the publication and subscription contract keys. This property is one of the message instance keys: the others are PubID and PubNodeName.



For more information, see Designing Application Messages.

This property is read-only.

Example

```
&CHANNEL = &MSG.ChannelName
```

DoNotPubToNodeName

Use this property to set the node name of a node that you do not want to publish this message to. For example, a single node may publish and subscribe to the same message. You can use this property to prevent the system from subscribing to the same message it publishes. This can help prevent circular processing where the original publisher eventually receives the same message.

Example

```
&MSG.DoNotPubToNodeName = &MSG.PubNodeName;  
  
&MSG.Publish();
```

IsActive

Indicates whether the message definition for the message object executing the property has been set to inactive in Application Designer. This property is True if the message definition is active, False if it's been inactivated from Application Designer. If you have a lot of PeopleCode associated with publishing a message, you might use this property to check if the message is active before you publish it.



For more information, see Designing Application Messages.

This property is read-only.

Example

```
&MSG = CreateMessage(MESSAGE.MY_MESSAGE)

If &MSG.IsActive Then

    /* do PeopleCode processing */

    &MSG.Publish();

Else

    /* do other processing */

End-if;
```

IsDelta

This property indicates if *any* fields in a message populated from page data have been changed since they were first loaded into the component buffer.

This is generally used in the following scenario:

You only want to publish a message if the end-user has changed a value on a page. The problem is that if the end-user makes a change to a field in the level 3 rowset, the `IsChanged` property for the top level row remains *False*. If the rowset is large, and you don't want to iterate through it in order to find a single `IsChanged`, you can use this property to quickly determine if something has changed at a lower level and the message should be published.



This property should only be used when the message and the component do *not* have the same structure (that is, the same records at the same levels). If the message and the component have the same structure use the `CopyRowsetDelta` method instead.

This property takes a boolean value: True if there are changed fields in the message, False otherwise.

This property is read-only.

Example

```
&Msg = CreateMessage(Message.MY_MESSAGE);

&Msg_Rowset = &Msg.GetRowset(); /* get first level in Msg RS */

&Msg_Detail = &Msg_Rowset(1).GetRowset(); /* get detail in Msg */

&Page_Rowset = GetRowset(); /* get page */
```

```

For &I = &Page_Rowset.Rowcount

    &Page_Detail = &Page_Rowset.GetRow(&I).GetRowset();

    &Msg_Detail.CopyRowset(&Page_Detail);

End-For;

/* Find if any data changed and should publish message */

If &Msg.IsDelta Then

    &Msg.Publish();

Else

    /* don't publish message and do other processing */

End-If;

```

IsEditError

This property is True if an error has been found on any field in any record of the current message after executing the ExecuteEdits() method on either a message object or a record object. This property can be used with the Field Class properties MessageSetNumber and MessageNumber to find the error message set number and error message number.



For more information, see ExecuteEdits method and MessageNumber, MessageSetNumber Field class properties

You can use this property in Subscription PeopleCode with **Exit(1)**. Exit(1) will automatically roll back any changes that were made to the database, save the message errors to the message queue, and mark the message status as ERROR.



For more information, see Exit, Processing Messaging Errors.

This property is read-only.

Example

```

&MSG = GetMessage();

&MSG.ExecuteEdits();

If &MSG.IsEditError Then

    Exit(1);

End-If;

```

IsLocal

This property is True if the message object that was just instantiated was published locally. This property takes a boolean value.

This property is read-only.



For more information and an example using this property, see [Introducing the Effective Dating Utility](#).

Example

```
&MSG = GetMessage();  
  
If &MSG.IsLocal Then /* LOCAL NODE */  
  
    /*do processing specific to local node */  
  
else  
  
    /*do processing specific to remote nodes */  
  
End-If;
```

Name

This property returns the name of the message.

This property is read-only.

PubID

This property refers to the publication identifier which is a number. It's generated by the system when the message is published, and is sequential, based on the number of messages published for that channel. The message instance keys are a subset of the publication and subscription contract keys. This property is one of the message instance keys: the others are ChannelName and PubNodeName.

This property is read-only.

Example

```
&MSG.Publish();  
  
&PUBID = &MSG.PubID;
```


PubNodeName

This property refers to a string that contains the name of the publishing node. It is generated by the system when the message is published. The message instance keys are a subset of the publication and subscription contract keys. This property is one of the message instance keys: the others are PubID and ChannelName.

This property is read-only.

Example

```
&MSG = GetMessage();

&PUBNODE = &MSG.PubNodeName;
```

Size

This is the approximate size of the uncompressed XML data generated when the message is published. The size is approximate for the following reasons:

- The maximum size of fields is used except for the case of long text and image fields.
- The active row count is used, without regard to whether the rows have been changed. In a CopyRowsetDelta, rows that are not changed, are not published.
- It doesn't include the size of the XML tags.



For more information about configuring the message size, see Publishing Messages.

This property can be used with the system property %MaxMessageSize in a batch Application Engine program to prevent the application from publishing a message that is too large.

This property is read-only.

Example

```
If &MSG.Size > %MaxMessageSize Then

    &MSG.Publish();

Else

    /*Move more data into the message object */

End-If;
```

SubName

This property refers to a string that contains the name of the subscription process currently processing the message. It is only available when GetMessage() is used in a subscription PeopleCode program.

This property is read-only.

Example

```
&MSG = GetMessage () ;

&SUBNAME = &MSG.SubName
```

SubscriptionProcessId

This property returns a string containing the subscription process identifier. This is a unique sequential number.

This property is only populated if the “Generate Subscription Process Instance” option is turned on in the Subscription Process definition.



For more information, see [Subscribing to Messages](#).

The subscription process identifier corresponds to the subscription process instance, *not* the message instance. If there are multiple subscription processes for the same message each subscription process will have a different, unique ID.

If the subscription process terminated abnormally, its process instance is lost, and a new one is generated on the next retry (that is, there will be a gap in the sequence.)

This property is read-only.

Considerations for using SubscriptionProcessId

- Using the "Run PeopleCode" option from Message Subscription will *not* generate the number. It is only generated if the Application Server is running the PeopleCode.
- If for some reason you have to rerun the subscription, a new number will be assigned.
- Since generating the number is optional, your program *must* contain code to support the option being turned off. Here is a code example from ITEM_SYNC:

```
If All(&MSG.SubscriptionProcessId) Then

    EIP_CTL_ID = Generate_EIP_CTL_ID(4, &MSG.SubscriptionProcessId);

Else

    &EIP_CTL_ID = Generate_EIP_CTL_ID(1, "Random");
```

End-If;

- If the flag is off, this property will be blank. In this case, you may want to adopt a standard for generating a random number, as shown in the above example.

CHAPTER 6

Page Class

The page class is used to manipulate a page in a component. The most common use of this class is to hide or display a page in a component, either based on field values or the level of security of the user.

Generally, the PeopleCode used to manipulate a page object would be associated with PeopleCode in the Activate event.



The page object shouldn't be used until after the page processor has loaded the page: that is, don't instantiate this object in RowInit PeopleCode, use it in PostBuild or Activate instead. For more information about events and the Component Processor, see Component Build Processing in Update Modes.

Note:

An expression of the form

PAGE.pagename.property

is equivalent to **GetPage**(name).property.

Declaring a Page Object

Page objects are declared using the Page data type. For example,

```
Local Page &MYPAGE;
```

Scope of a Page Object

A page object can only be instantiated from PeopleCode.

You will only use this object in PeopleCode programs that are associated with an online process, not in an Application Engine program, a message subscription, a Component Interface, and so on.

In addition, the page object shouldn't be used until after the page processor has loaded the page: that is, don't instantiate this object in RowInit PeopleCode. Use it in PostBuild or Activate instead.

Page Class Built-in Function

GetPage

Page Class Properties

DisplayOnly

Use this property to make a page display-only, or to enable a page, so the fields can be edited. This property takes a Boolean value: True if the page is display-only, False otherwise.



You can't enable or disable the current page.

The value of DisplayOnly for a page is restored to the value set in Application Designer when the user goes to another component. This means you must set DisplayOnly for the page each time a component is accessed.

This property is read-write.

Example

```
If &MYPAGE. DisplayOnly Then  
  
    &MYPAGE.Visible = True;  
  
Else  
  
    &MYPAGE.Visible = False;  
  
End-If;
```

Name

This property returns a reference to the page name definition (a string value.)

This property is read-only.

Example

```
&MYPAGENAME = &MYPAGE.Name;
```

Visible

If this property is True, the page executing this property is displayed. If you set this property to False, the page is hidden. When the visibility is changed, the component tabs and menus are also automatically changed to reflect the new settings. Menus return to their original values when the user goes to another component.



You can't hide the current page.

If you set a page to be hidden in the component definition, you can change the value of the page to be visible at runtime.



For more information, see [Creating Component Definitions](#).

Example

In the following example, a page is hidden based on the value of the current field.

```
If PAYROLE_TYPE = "Global" Then  
  
    PAGE.JOB_EARNINGS.Visible = False;  
  
End-If;
```

PortalRegistry Classes

The portal registry is a tree structure in which content for a portal is organized, classified, and registered.

The PortalRegistry classes (API) are used to update the portal registry. The Portal Administration pages provide a GUI access to the PortalRegistry API. You can also access them using a PeopleCode program you write yourself. How you access the portal registry depends on the type of updates required. Your organization will likely find uses for both methods of updating the portal registry.

This section focuses on access the PortalRegistry classes using PeopleCode.



For more information, see [Introducing the PeopleSoft Portal and Using Portal Administration Features](#).

Portal Registry Overview

A portal is a web site that helps you navigate to other web-based applications and content. The PeopleSoft Portal is an *enterprise portal*. It is much like general purpose portals, except that its main purpose is to help end-users be more effective in accessing information related to performing their jobs.

Each PeopleSoft Portal is defined by one PeopleSoft portal registry. The PeopleSoft portal registry consists of a number of system tables and associated data in a PeopleSoft database. The

portal registry must reside within one PeopleSoft database. There can be more than one portal registry in a PeopleSoft database, but only one portal registry is associated with a PeopleSoft Portal.

The portal registry consists of the following primary parts:

- Folders
- Content references
- Content providers

Folders and content references make up the majority of the portal registry, and provide a hierarchical tree structure to describe various content that is *registered* as part of a PeopleSoft portal.

Content providers are a way to give a logical name to a specific webserver, so content can be registered independent of specific webserver.

The primary function of the portal is to take a target URL (generally a URL for a PeopleSoft component) that comes in from a user's browser, and assemble a page with that content and any other content. The layout and what content to assemble on the page is defined by a *portal template*, which is composed of HTML. The portal attempts to find the content reference associated (that is, registered) with the target URL to get the template, or uses a series of default templates if it cannot get the template from a content reference.

Folders

Folders are the way a hierarchical structure is created within a portal registry. Each folder has a parent, except the root folder which is the top level folder in a portal registry. Each folder can also contain child folders and content references. Folders are roughly analogous to directories within a file system. A folder can be further defined by a number of attributes (description, security, when it expires, and so on) that are useful within the portal environment.



For an example of adding a folder, see Adding a Folder.

Content References

A content reference is simply a reference to a URL. Once you've created an entry for a content reference in the portal registry, it's considered registered. A content reference can be further defined by a number of attributes (description, security, when it expires, and so on) that are useful within the portal environment.

There are a number of distinct types of content references that can be broken down into the following broad categories:

- Target
- Template

- Component

A *target* type of content reference describes a registered URL that a user might reference. Typically, these would be a PeopleSoft Internet Architecture component, such as a page in a transaction. When a user references a URL, the URL is looked up in the registry to find the target content reference.

A *template* type of content reference describes what, if any, other content to put on the page, and where to place that content. The portal attempts to find a template for every URL that is requested.

A *component* type of content reference describes a component that is typically placed on a target page or home page. Examples of components are breadcrumbs, favorites, and so on.

Content Providers

When a content reference is created to register some content (that is, a URL) the specific URI (that is, the scheme, webserver, port, and so on) can be specified in the content reference. However, this has the drawback that every time the URI for a content reference changes (the webserver name changes, the port number changes, and so on), the content reference must be changed. The content provider is a way to create a logical name for a specific webserver, port, and so on. When the content reference is created a content provider name can be specified.

For example, suppose the name of a webserver changed. If you don't use a content provider, you must check all your content references and change them accordingly. If you use a content provider, you only have to change the webserver name in one place, and all the content references that use that content provider now automatically reference the correct webserver.

Security

The same security mechanism is used for both folders and content references. Folders and content references can be marked as public, owner accessible, or can have explicit PeopleSoft permission values set, including cascading the permissions to its child objects (that is, the child objects have the same permissions as the parent objects.)

When marked as public, the folder or content reference is accessible by any user. Public access cannot be cascaded, that is, passed down, to child objects.

When marked as owner accessible, a folder or content reference is only accessible by the same user that created the folder or content reference. Owner accessible cannot be cascaded, that is, passed down, to child objects.

Folders and content references can be marked as accessible by PeopleSoft permissions. This means that if a user is a member of a role, and the role contains the permission list that the folder or content reference is also associated with, the user has access to the folder or content reference. Additionally, when applied to folder permissions the permission can be cascaded. This means that any child of that folder, including a content reference, automatically has that permission added to its permission list.

Attributes

All folders and content references can have any number of attributes added to them. Attributes are simply name/value pairs. These name/value pairs are defined and used by many portal-aware applications, such as the search engine, navigational display, and so on.



For an example of using attributes, see [Using Attributes](#).

Using the PortalRegistry API

The PortalRegistry API provides the entry point to a specific portal registry. Common administrative tasks include adding, deleting, and renaming the registry objects (that is, folders, content references, attributes, and so on.) Additionally, there are many properties associated with every registry object, and all of these properties can be accessed and modified.

You can use the PortalRegistry API in batch mode to make many changes to a portal at once. For example, suppose something changed in your security system. You could write your own PeopleCode program, using the PortalRegistry classes, in an Application Engine program and change the access for all your users at once.

You can also use the PortalRegistry API to exchange data with an external system. For example, suppose an external merchant had an area on your company's portal, and the information there needed to be updated. One way to do this is for the merchant to use an Application Message to send the data to your system, then a subscription program would update the portal using the PortalRegistry API.

Each PortalRegistry object represents one specific PortalRegistry in a system. An empty PortalRegistry object is initially gotten from the PeopleSoft Session object. You can then open an existing PortalRegistry to view or change existing content, create a new PortalRegistry, or delete an existing PortalRegistry.

Security Considerations

The PermissionValue object associated with a folder or content reference, as well as specific properties on the object, work together to form the security for an object.

Using Object Properties

The properties that set permissions for a folder or content reference are the following:

- AuthorAccess
- PublicAccess

The AuthorAccess property determines whether the author of the object has access to the object.

The `PublicAccess` property determines whether the object is generally accessible or not. If this property is set to `True`, all users have access to the object.

These properties only apply to an object. They can **not** be cascaded, that is, passed down to child objects.

The other object property used with security, the `Authorized` property, indicates whether a user is authorized to access an object or not. The value of this property depends on whether the user has access or not.

Accessing Folders and Content References

When you get a folder or content reference collection, only the folders or content references that the end-user is authorized to access are in the collection.

An object is contained in a collection is based on the following algorithm.

- If the object is marked as `PublicAccess` it is automatically accessible.
- If the object is marked as `AuthorAccess` and the current `UserId` is the `Author` it is accessible.
- If the current user is in a permission list (class) that is in the `Permissions` collection for that object.
- If the current user is in a permission list (class) that is in the `CascadedPermissions` collection for that object.

When you access a content reference or folder using a valid name and one of the `Find` methods (`FindCRefByURL`, `FindCRefByName` or `FindFolderByName`) a content reference or folder is returned whether the user is authorized to it or not. When you use these methods, you should always check the `Authorized` property. This is the only property you can view from an object that an end-user isn't authorized to.

Using `PermissionValue` and Cascading Permissions

To set permissions for a folder or content reference, you must access the `PermissionValue` collection for that object using the `Permissions` property. The `PermissionValue` objects refer to permission list values that already exist on your system, such as `ALLPANLS`, `CUSTOMER`, `EMPLOYEE`, and so on.



For more information about Permission Lists, see [Working with Permission Lists](#).

For every `PermissionValue` object, you can chose whether all the child folders and content references of this folder have the same permissions. This is called *cascading*. You cascade permissions by setting the `Cascade` property to `True`.

There are two types of `PermissionList` collections you can access. One is returned by the `Permissions` property, the other by the `CascadedPermissions` property.

Permissions Property

Returns the `PermissionValue` collection containing the permissions set at the current level, that is, set with that folder or content reference. You can add `PermissionValues` to this list. Use the `Cascade` property to cascade the permission you add to child folders and content references.



Folders can contain other folders, but content references can't contain other content references. Therefore, the `Cascade` property is only applicable to folders, not to content references.

CascadedPermissions Property

Returns the `PermissionValue` collection for *all* the permissions for a folder or content reference. It contains any permission list values set in any parent folder and cascaded.



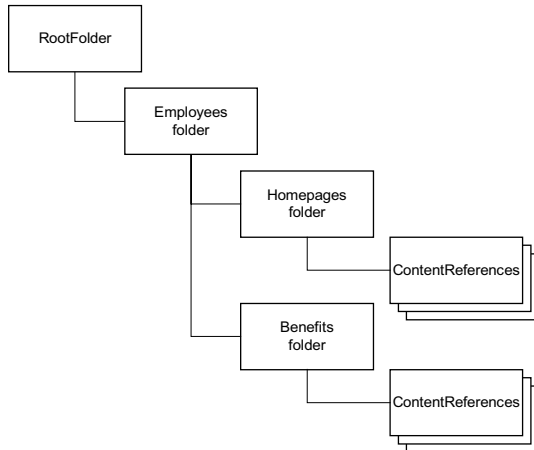
You can *not* add any `PermissionValues` to a collection returned by the `CascadedPermissions` property. You can only add values to the collection returned by the `Permissions` property.

You can add `PermissionValues` for a child folder, but you can *not* delete any of the existing ones that are cascaded. Permissions are augmented, not overwritten, that is, the permissions are the sum of both the parent permissions and whatever is set at the current level.

Therefore, keep the following in mind when working with `PermissionValues`:

- Set permissions only at the level where they're needed.
- Only cascade permissions when necessary.
- Generally, don't set cascaded `PermissionValues` for the root folder.
- Only the values in a `PermissionValue` object cascade. Properties on the folder itself (such as `PublicAccess` or `AuthorAccess`) do **not** cascade.

For example, suppose your `PortalRegistry` had the following hierarchy:



PortalRegistry hierarchy example

The following permission list values are assigned to the PermissionValue objects for the Employees folder, and both of them are cascaded:

- EMPLOYEE
- MANAGER

The Homepages and Benefits folders have the exact same permissions as the Employees folder, that is, all users associated with these two permission lists have access to these folders.

Suppose you decide that you don't want the permission list EMPLOYEE to access the Benefits folder. If you delete the EMPLOYEE PermissionValue from the Benefits PermissionValue collection, you have **not** altered the permissions. The permissions set at the parent folder, that is, at the Employees folder, can't be deleted, they can only be added to.

To make this change, you must:

- Change the Cascaded property for the EMPLOYEEES PermissionValue in the Employees folder to False
- Add EMPLOYEEES as a PermissionValue object to the Homepages folder.



For an example of using security, see Setting Permissions using the PermissionValue Object.

Working with ValidFrom and ValidTo

Folders and content references have both ValidFrom and ValidTo dates. What's the difference and how are they used in the PortalRegistry API?

- A ValidFrom date is when something begins.

- A ValidTo date is when something ends.

For example, suppose your HR department has a page describing the benefits for your employees, and that page changed every calendar year. This means the page for the year 2000 has a **ValidFrom** date of 01/01/2000 and a **ValidTo** date of 12/31/2000. The benefits page for the year 2001 has a ValidFrom date of 01/01/2001 and a ValidTo date of 12/31/2001.

All folders and content references are returned in their collections regardless of the ValidTo and ValidFrom dates. It is up to you to take these dates into account and only display to the end-user those values that should be seen.

In addition, a ValidFrom should always come **before** a ValidTo date. However, there is nothing inherent in these properties to enforce this. You must account for this restriction in your PeopleCode program.

For all newly created folders and content references, the default value for both these properties is null, that is, they begin immediately and do not expire.

Error Handling

All errors for the PortalRegistry classes, like the other APIs, are logged in the PSMessages collection, instantiated from a session object.



For more information see About Error Handling.

The PortalRegistry classes log errors that occur with methods immediately, and errors that occur with properties only after a method is executed.

For example, suppose you specified an invalid name when you were trying to delete a folder. The method (DeleteItem) would return False, and the error would be logged in the PSMessages collection immediately.

Now suppose you created a new folder, and specified an invalid ValidTo date. The error wouldn't be logged in the PSMessages collection until you tried to save your changes.

It depends on your application when you want to check for errors. When users are entering data dynamically, and your program is registering their data in the portal, you may want to check for errors often. If you're using a batch program, you may only want to check for errors after the Open, Save, Insert and Delete methods.

Most methods return a Boolean value indicating success or failure. After the failure of a method, you may want to check the PSMessages collection to determine the exact error.

```
Local ApiObject &MySession;

Local ApiObject &ErrorCol;

Local ApiObject &FolderCol, &Folder, &Registry;

Local Boolean &Open;
```

```
/* Access the current session */

&MySession = %Session;

If &MySession Then

    /* connection is good */

    &Registry = &MySession.GetRegistry();

    &Open = &Registry.Open("Customer");

    If &Open Then

        /* Registry opened successfully */

        /* do processing */

    Else

        /* Do error checking */

        &ErrorCol = &MySession.PSMessages;

        For &I = 1 to &ErrorCol.Count

            /* do processing */

        End-For;

    End-If;

Else

    /* do processing for no connection */

End-If;
```

Life-Cycle of a PortalRegistry Object

At runtime, there are certain things you want to do with a PortalRegistry object, like getting an instance of it, adding content providers, editing content references, and so on. The following is an overview of this process. These steps are expanded in other sections.

- 1a. Execute the GetPortalRegistry method on the PeopleSoft session object to get a PortalRegistry object, OR
- 1b. Use the FindPortalRegistries method on a session object to get a collection of all PortalRegistries. Select a PortalRegistry from the collection.
2. Open the PortalRegistry, using the portal name.
3. Once you have an open PortalRegistry object, you can do many things with it, such as:
 - Add content references to the existing folders
 - Add folders and the content references for them
 - Add or change Content Providers
 - Make changes to the PortalRegistry properties
 - Make changes to the existing content references, folders, content providers or attributes
4. After you make your changes, you must save your work. The Save method must be executed at the appropriate level, such as at the folder level for changes to a folder, at the PortalRegistry level for changes to a ContentProvider, and so on.



For more information see Considerations when Saving Content.

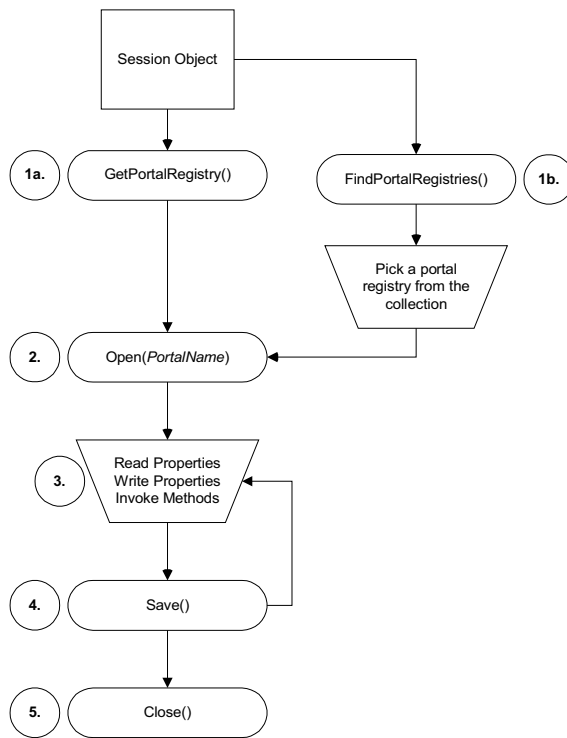
A property value change isn't committed to the database until the parent object is saved.

Some methods commit data to the database only when the parent object is saved. However, other methods cause data to be committed to the database as soon as they are executed, like InsertItem on a folder. Methods that automatically make database changes are noted as such in their description.

5. When you finish all operations for a PortalRegistry, you should close the object.



PeopleSoft recommends that you open and close every PortalRegistry in a single PeopleCode event. You shouldn't open a PortalRegistry object and keep it open across multiple events.



Life-Cycle of a PortalRegistry Object



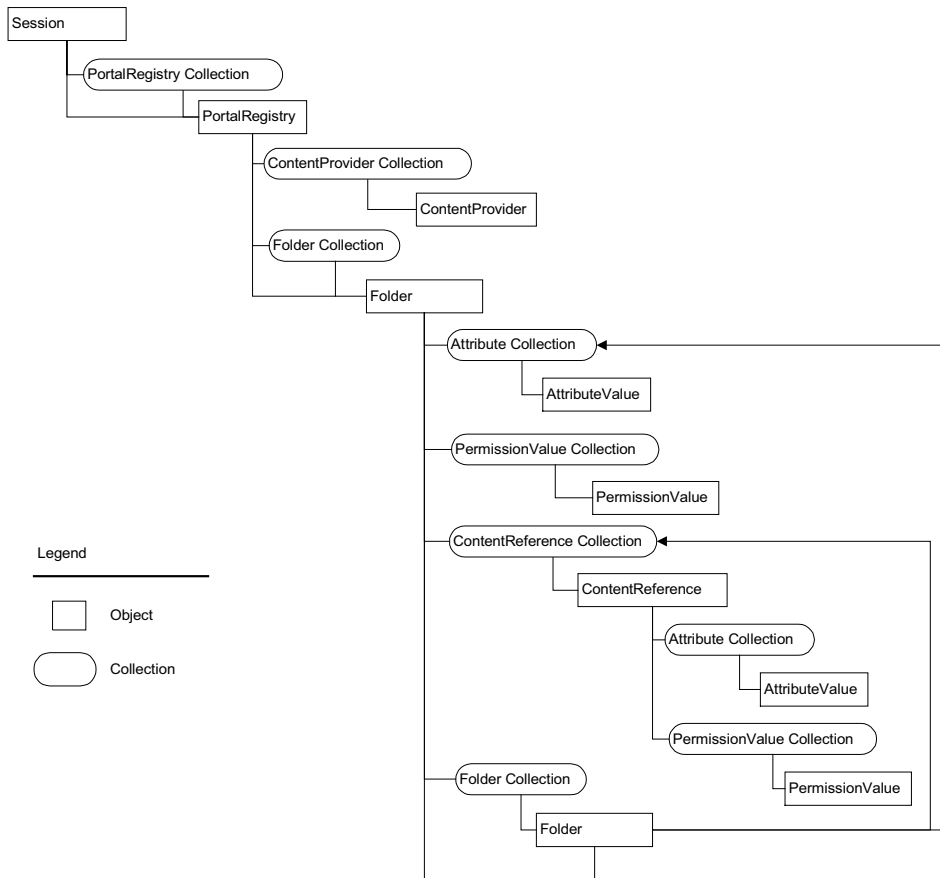
For examples of using the PortalRegistry objects, see PortalRegistry Examples.

PortalRegistry Class Hierarchy

There are many different classes used with the PortalRegistry API. The following flowchart shows the different classes, and how they are interrelated.

Hierarchy Considerations

- The flowchart only shows two levels of folders. In reality, you can embed as many levels of folders as you need.
- From a PortalRegistry object, you must access the root folder before you can access the sub-folders for that PortalRegistry.



PortalRegistry Class Hierarchy

Using Content References

Content references have a number of properties, but several properties work together to define the type of a ContentReference. The values of these properties are interdependent, that is, the value of one indicates the values of others. The properties are:

- UsageType
- TemplateType
- StorageType
- URLType
- ContentProvider and URL

UsageType

This is used as the primary specifier for the type of content reference. There are a number of different types of content references, but all content references can be categorized into the following major types:

- Target
- Template
- Portal Component

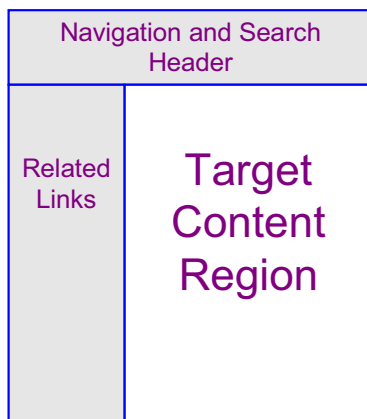
Target

The *target* is the page referenced by a URL in the user's browser. This is the main transaction or page that the user wants, and the portal may place other content around this target page based on the template. The template describes any other content, and where to place it, on the page. The template is either gotten from the content reference or a set of default templates.

Template

A portal *template* defines how the portal creates a user's page. It's an HTML document that describes the content and where the content is placed. The template specifies one target and zero or more portal components.

The portal template in the example below is comprised of four separate template components: one for the navigation and search header, one for related links, one for the target content region, and one for the overall template which specifies where the other components should be placed. At runtime, the target content region would be filled by the HTML returned by the target page, as would the other template component regions.



A Portal Template



For more information, see [Designing Portal Templates](#).

Portal Component

A *portal component* is an HTML document, or something that produces an HTML document. The portal component must be defined within a template.

UsageType Values

The following table matches the general types of target, template and portal component to the actual values of the UsageType property.

General Type	UsageType Value
Target	Target (TARG)
Template	Frame template (FRMT), HTML template (HTMT), or Homepage template (HPGT)
Portal component	Template component (TMPC) or Pagelet (HPGC)

More specifically:

- A UsageType value of TARG specifies a content reference that is a target.
- A UsageType value of FRMT specifies a content reference that is an HTML frame template.
- A UsageType value of HTMT specifies a content reference that is an HTML template.
- A UsageType value of HPGT specifies a content reference that is a PeopleSoft home page.
- A UsageType value of HPGC specifies a content reference that is a PeopleSoft home page component.
- A UsageType value of TMPC specifies a content reference that is a PeopleSoft template component.

TemplateType

This property is valid only when the UsageType property is a target. For target type content references (TARG) this controls whether the portal looks for and uses a template to wrap the target.

TemplateType	Meaning
NONE	There is no template for this target
HTML	There is some kind of HTML template for this target

StorageType

In general, content references contain information on where to get the content, and do not store the content. However, content references that are template or portal component types can have their content accessible directly from the content reference. In these cases, the Data property is valid and can be read or written, and the data is stored locally in the portal database.

StorageType Value	Meaning
LOCL	Local: Data property on content reference is

StorageType Value**Meaning**

valid.

RMTE

Remote: Data property is not valid.

When UsageType is a target this property must be set to RMTE and the URLType property should specify what format the Content Provider and URL are in.

When UsageType is either a template or a portal component this property can be set to either LOCL or RMTE.

Note that when StorageType is LOCL, it specifies a static template or portal component. But, when StorageType is RMTE it can specify either:

- a dynamically generated template or portal component
- a static template or portal component

In both cases, the ContentProvider and URL properties specifies how to get the template or portal component.

URLType

This property is only valid when StorageType is remote (RMTE). This property gives information about what format the URL is in.



For more information about different types of URLs, see Types of Portals Available.

URLType Value	Meaning
UPGE	Frame template (FRMT), HTML template (HTMT), or Homepage template (HPGT)
USCR	Target (TARG)
UEXT	Template component (TMPC) or Pagelet (HPGC)

The URL property is always required (it's one of the parameters for the InsertItem method.) The format of this parameter/property depends on the other properties.

ContentProvider and URL

The content provider and the URL property work together and are interrelated. The content provider is used to 'register' a logical name for a web server (the web server name, the port, and so on) not the actual details. This way, content references don't change when a web server changes.

For example, suppose you had content that you referenced on the HRMS web server. However, the machine name for that server changed. You can change the URI of the ContentProvider, instead of changing every content reference that referred to that content.

Considerations when Using ContentProvider and the URL property

When a content reference is created it is ‘registered’ with its URL (the portal, and others, typically find a content reference by its URL). The content provider is used to create a logical name for the web server, port, and so on, so these details are not included in a content reference’s URL. When details of a web server change, such as at installation time, only the URI for the content provider must change. You don’t have to change the URL for any content references.

When the content provider is specified, the content provider’s URI is concatenated with the URL property. The portal API automatically adds a question mark (?) between the URI and the URL property *only* if the URL property contains a value. This implies that the URL property is really just the “query string” part of the URL in this case.

When the content provider is not specified, the URL property needs to be the absolute URL necessary to get to the content.

The format of the URL property depends on other properties.



For an example, see PortalRegistry Examples.

Format when StorageType is Local (LOCL) and UsageType is Homepage Component (HPGT)

When the UsageType property is Homepage template (HPGT), and the StorageType is Local (LOCL), the URL property must indicate this is a template, and have the following format:

```
<TEMPLATE>templatename
```

The following are examples:

```
<TEMPLATE>PORTAL_ADMIN
```

```
<TEMPLATE>PORTAL_DEFAULT
```

Format when UsageType is Target (TARG) and URLType is PeopleSoft Component (UPGE)

When the URLType property is PeopleSoft Component (UPGE) the URL property must indicate the menu name, market and component name, in the following format. The parts in bold are required. You must specify the parts in *italics*:

```
ICType=Panel&Menu=ComponentName&Market=Market&PanelGroupName=ComponentName
```

The following are examples:

```
ICType=Panel&Menu=PORTAL_PERS_HOMEPAGE&Market=GBL&PanelGroupName=PORTAL_HOMEPAGE
```

```
ICType=Panel&Menu=PORTAL_ADMIN&Market=GBL&PanelGroupName=PORTAL_MENU_EXPORT
```

Format when URLType is PeopleSoft Script (USCR)

When the URLType property is PeopleSoft Script (USCR), the URL property must indicate the Internet Script record, field, event, and Internet Script name, in the following format. The parts in bold are required. You must specify the parts in *italic*:

```
ICType=Script&ICScriptProgramName=WEBLIB_RECORDNAME.EventName.IScript_IScriptName
```

The following are examples:

```
ICType=Script&ICScriptProgramName=WEBLIB_PORTAL.PORTAL_HEADER.FieldFormula.IScript_UniHeader_PIA
```

```
ICType=Script&ICScriptProgramName=WEBLIB_PORTAL.PORTAL_NAV.FieldFormula.IScript_Portal_Trans_Dyn
```

Format when URLType is External (UEXT)

If the URLType is Non-PeopleSoft URL, or External, (UEXT), and if there is *no* ContentProvider for the ContentReference, the URL property must be a qualified URL. If there is a ContentProvider, the URL property can specify the end part of the qualified URL.

For example, suppose the URI for your ContentProvider is the following:

```
http://pt03.peoplesoft.com/servlets/iclientservlet/peoplesoft8/
```

The URL for a ContentReference could be the following:

```
HR/main.html
```

If you had no ContentProvider, the URL property for the ContentReference would contain the actual URL, that is:

```
http://pt03.peoplesoft.com/servlets/iclientservlet/peoplesoft8/ HR/main.html
```

Summary

The following table summarizes the interrelations between the different content reference properties.



For an example of adding a content reference, see [Adding a Content Reference](#). □

UsageType	TemplateType	StorageType	URLType	URL Format
Target (TARG)	HTML	Remote (RMTE)	Component (UPGE)	Specifies a PeopleSoft component by menu, market, component.
Target (TARG)	HTML	Remote (RMTE)	Internet Script (USCR)	Specifies a PeopleSoft Internet Script record, field, event and function name.
Target (TARG)	HTML	Remote (RMTE)	External (UEXT)	Specifies an external page.
Target (TARG)	NONE	Remote (RMTE)	Component (UPGE)	Specifies a PeopleSoft component by menu, market, component.
Target (TARG)	NONE	Remote (RMTE)	Internet Script (USCR)	Specifies a PeopleSoft Internet Script record, field, event and function name.
Target (TARG)	NONE	Remote (RMTE)	External (UEXT)	Specifies an external page.
Frame template (FRMT), HTMP template	NONE	Remote (RMTE)	Internet Script (USCR)	Specifies a PeopleSoft Internet Script record, field, event and function name.
Frame template (FRMT), HTMP template	NONE	Local (LOCL)	N/A	N/A – HTML is available through the Data property.
Homepage (HPGT)	NONE	Local (LOCL)	N/A	<USER>userid
Template component (TMPC), Homepage component	N/A	Remote (RMTE)	Component (UPGE)	Specifies a PeopleSoft component by menu, market, component

(HPGC)				
Template component (TMPC), Homepage component (HPGC)	N/A	Remote (RMTE)	Internet Script (USCR)	Specifies a PeopleSoft Internet Script record, field, event and function name.
Template component (TMPC), Homepage component (HPGC)	N/A	Remote (RMTE)	External (UEXT)	Specifies an external page.
Template component (TMPC), Homepage component (HPGC)	N/A	Local (LOCL)	N/A	N/A – HTML is available through the Data property.

Naming Conventions

If you create two content references with the exact same URL, the second one will fail.

For example, suppose you create an external content reference with a URL of www.peoplesoft.com. Then you create a second content reference that has a content provider of **PeopleSoft**, whose URI is www.peoplesoft.com. The creation of the second content reference fails because the URL already exists.

If you have multiple content providers with the same URI, FindCRefByURL looks for the specified content reference with all those content providers.

If you have multiple content providers with the same URI, but no registered content references, the system uses the template from the alphabetically first content provider it finds.

Considerations when Deleting Content

You must be extremely careful when you delete any content. There may be more than one object relying on the content you delete.

- If you delete a template for a ContentReference, the default template specified with the ContentProvider is used. If there's no template for the ContentProvider or no content provider, the template for the PortalRegistry is used. However, if you delete the template for the ContentReference, the ContentProvider and the PortalRegistry, you'll receive a runtime error.
- If you delete a homepage template for a user, the system tries to use the default user's template first, before resorting to the portal default template.



If you delete a content reference that's a template, you must make sure no other content references are using it before you delete it. Otherwise, you may get unpredictable results.



A content reference doesn't have to have a ContentProvider. However, if it does, and you delete the ContentProvider, you will "lose" the content reference. Without the ContentProvider, you've lost part of the name required to find the content reference.



If you delete a folder, you delete **all** content in the folder. If you delete a folder that contains other folders, that is, a parent folder, all the child folders, as well as all the content references are deleted as well.



If you delete a PortalRegistry, you delete **everything**. Your entire PortalRegistry is gone, all the folders, content references, templates, and so on. Do not delete a PortalRegistry object unless you are absolutely certain you want to.

Considerations when Saving Content

The following PortalRegistry classes have Save methods:

- PortalRegistry
- Folder
- content reference

This means if you make changes to a folder, you must save the folder. If you make changes to a folder and only save at the PortalRegistry level, your changes are **not** saved.

Some classes do **not** have a Save method. For these classes, you must save the parent object.

- If you change a ContentProvider, you must use the PortalRegistry Save method.
- If you change an AttributeValue you must use the Save method with the content reference or folder that contains the AttributeValue.
- If you change a PermissionValue, you must use the Save method with the content reference or folder that contains the PermissionValue.

Declaring a PortalRegistry Object

PortalRegistry objects, and all objects instantiated from a PortalRegistry object, are declared as data type ApiObject. For example,

```
Local ApiObject &MyRegistry;  
  
Local ApiObject &MyFolder, &MyAttributeValue;
```



PortalRegistry objects can only be declared as Local.

Considerations using Local Variables

When a local variable has a reference to an object and the end-user click the Back button on a browser, the local variable is set to NULL. This is always logged in the trace file.

Considerations using Global Variables

Global variables are not available to a portal or applications on separate databases. They are only available on applications and Portals in the **same** database.

Scope of a PortalRegistry Object

A PortalRegistry object can be instantiated from the following language environments:

- PeopleCode
- OLE/COM
- C/C++
- Java

For example programs of how to access the PortalRegistry objects using language environments other than PeopleCode, see PortalRegistry Examples.

PortalRegistry Reference

The following sections go into more detail of the properties, methods, and other objects that can be used with a PortalRegistry object. If you don't want to programmatically change a portal registry, you can use the Portal Administration Tool pages instead.



For more information, see Introducing the PeopleSoft Portal.

Session Object Methods

PortalRegistry objects don't have any built-in functions. They are instantiated from a session object.



For more information, see Session Class.

FindPortalRegistries

Syntax

```
FindPortalRegistries (Name)
```

Description

The **FindPortalRegistries** method, used with the session object, returns a reference to a PortalRegistry Collection, filled with zero or more PortalRegistry objects matching the *Name* parameter. The *Name* parameter takes a string value.

You can use a partial key to get a smaller subset of the PortalRegistry collection. For example, if you wanted to get a collection of all the PortalRegistry objects whose names started with the letter "B", you could specify just the letter B for *Name*:

```
&MyColl = &MySession.FindPortalRegistries("B");
```

Parameters

<i>Name</i>	Specify the name of the PortalRegistry object you want to find. This parameter takes a string value.
-------------	--

Returns

A PortalRegistry Collection object.

Example

The following example will return a collection with references to all PortalRegistry objects.

```
Local ApiObject &MySession;  
  
Local ApiObject &MyColl;  
  
&MySession = %Session;  
  
&MyColl = &MySession.FindPortalRegistries("");
```

GetPortalRegistry

Syntax

```
GetPortalRegistry()
```

Description

The GetPortalRegistry method returns an empty PortalRegistry object. You can then open or delete an existing PortalRegistry, or create a new one.

Parameters

None.

Returns

An empty PortalRegistry object.

Example

```
Local ApiObject &MyPortal;  
  
&MyPortal = %Session.GetPortalRegistry();  
  
PORTAL_NAME = %Request.GetParameter("PORTAL_NAME");  
  
&Portal.Open(PORTAL_NAME);
```

PortalRegistry Class

A PortalRegistry object is returned from the following:

- The GetPortalRegistry session method
- The PortalRegistry Collection Methods First, ItemByName or Next



For an example, see Changing PortalRegistry Properties.



In addition to the methods listed below, the PortalRegistry class has methods used with the Search API. For more information, see Search Classes.

PortalRegistry Class Methods

Close

Syntax

```
Close()
```

Description

The Close method closes the PortalRegistry object, that is, this method sets the object to the state it was in immediately after the GetPortalRegistry was done on the PeopleSoft Session object. Any unsaved changes will be discarded. The **Close** method can only be used on an open PortalRegistry, not a closed one. This means you must have opened the PortalRegistry with the Open method before you can close it.

Parameters

None.

Returns

A Boolean value: True if the PortalRegistry object is successfully closed, False otherwise.

Example

```
&Rslt = &MyRegistry.Close();

If Not &Rslt Then

    /* registry not closed - do error processing */

End-if;
```

Related Topics

Open, Save

Create

Syntax

```
Create(RegistryName)
```

Description

The **Create** method creates a new PortalRegistry in the PortalRegistry object called *RegistryName*. The specified registry must be a **new** registry. The Create method will return False if the registry already exists.

The new PortalRegistry is immediately committed to the database. If you change any property of a PortalRegistry after you create the object, use the Save method to commit your changes to the database.

When a registry is created, a folder called **Root** is automatically created. This is the root folder for the registry.



If you're using Visual Basic, you must check that the PortalRegistry is actually created. If you use a duplicate name, a zero is returned, but no error is raised.

Parameters

RegistryName

The name of the registry you want to create. This parameter takes a string value. If you specify a registry that already exists, this method will return a False value.

Returns

A Boolean value: True if the PortalRegistry object is successfully created, False otherwise.

Example

```
&PORTAL_NAME = MY_PORTAL_RECORD.PORTALNAME;

&MyPortal = %Session.GetPortalRegistry();

If NOT &MyPortal.Create(&PORTAL_NAME) Then;

    /* portal not created - do error processing */

End-If;
```

Related Topics

Open, Save, Close, Delete

Delete

Syntax

```
Delete(RegistryName)
```

Description

The **Delete** method deletes the PortalRegistry **from the database**, including any data and tables.



If you delete a PortalRegistry, you delete **everything**. Your entire PortalRegistry is gone, all the folders, content references, templates, and so on. Do not delete a PortalRegistry object unless you are absolutely certain you want to.



Note. The PortalRegistry classes execute all methods "interactively", that is, as they happen. The item won't be marked for deletion, then actually deleted later. The item will be deleted from the database **as soon as** the method is executed.

The **Delete** method can only be used with a **closed** registry, it cannot be used on an open registry. Before you use the **Delete** method, you must explicitly close the PortalRegistry object (with the Close method.) The Delete method will return False if you try to delete an open PortalRegistry object.

Parameters

RegistryName

The name of the registry you want to delete. This parameter takes a string value. If you specify a registry that doesn't exist, this method returns a False value.

Returns

A Boolean value: True if the PortalRegistry object is successfully deleted, False otherwise.

Example

The following example deletes all PortalRegistry objects that start with Customer.

```
Local ApiObject &MySession;  
  
Local ApiObject &MyPortal;  
  
    &MySession = GetSession();  
  
    &MySession.Connect(1, "EXISTING", "", "", 0);  
  
    &MyPortal = &MySession.GetPortalRegistry();  
  
    &MyPortal.Delete("HRMS_99");
```

Related Topics

Close, Open

FindCRefByName

Syntax

FindCRefByName (*Name*)

Description

The **FindCRefByName** method returns the content reference object corresponding to *Name*. The name is a unique identifier for each content reference.

Considerations on Returned Content References

This method returns content reference objects that aren't yet valid as well as ones that are no longer valid. When you create your program, you must check for these properties (ValidTo and ValidFrom) if you don't want to use them.

This method returns content reference objects that you aren't authorized to. When you use create your program, you should always check the Authorized property. This is the only property you can view from an object that you're not authorized to view.

Parameters

<i>Name</i>	A unique name within the registry that identifies the content reference. This parameter takes a string value. This parameter uses the name, not the label, of a content reference.
-------------	--

Returns

What this method returns depends on the condition of the content reference:

- If the content reference name is valid and the end-user has access to the content reference, a content reference object is returned.
- If the content reference is valid but the end-user doesn't have access to it a content reference object is returned, however, the only property you can access is the Authorized property.
- If you specified an invalid name this method returns NULL.

Example

The following example returns a content reference object:

```
&CRef = &Portal.FindCRefByName("GLOBAL_PAYROLL");
```

Related Topics

Content Reference Class

FindCRefByURL

Syntax

FindCRefByURL (*URL*)

Description

The **FindCRefByURL** method returns the content reference object corresponding to *URL*. The URL specified by *URL* must be an absolute URL.

If you have multiple content providers with the same URI, FindCRefByURL looks for the specified content reference with all those content providers.



For more information on the different types of URLs, see Types of Portals Available.

Considerations on Returned Content References

This method returns content reference objects that aren't yet valid as well as ones that are no longer valid. When you create your program, you must check for these properties (ValidTo and ValidFrom) if you don't want to use them.

This method returns content reference objects that the end-user isn't authorized to. When you create your program, you should always check the Authorized property. This is the only property you can view from an object that the end-user isn't authorized to view.

Parameters

<i>URL</i>	A URL that represents the content. This parameter takes a string value. This URL must be an absolute URL. This parameter is case insensitive.
------------	---

Returns

What this method returns depends on the condition of the content reference:

- If the end-user has access to the URL, a content reference object is returned.
- If the content reference is registered, but the end-user doesn't have access to it, a content reference is returned, but the only property you can access is the Authorized property.
- If a URL isn't registered or is invalid, this method returns NULL.

Example

The following example finds a content reference from a URL:

```
&UserCRef = &Portal.FindCRefByURL("http://www.PeopleSoft.Com");
```

Related Topics

Content Reference Class, GetQualifiedURL PortalRegistry method, QualifiedURL content reference property

FindFolderByName

Syntax

```
FindFolderByName (Name)
```

Description

The **FindFolderByName** method returns the Folder object corresponding to *Name*. The name is a unique identifier for each folder.

Considerations on Returned Folders

This method returns Folder objects that aren't yet valid as well as ones that are no longer valid. When you create your program, you must check for these properties (ValidTo and ValidFrom) if you don't want to use them.

This method returns folder objects that you aren't authorized to. When you use create your program, you should always check the Authorized property. This is the only property you can view from an object that you're not authorized to view.

Parameters

<i>Name</i>	A unique name within the registry that identifies the folder. This parameter takes a string value. This parameter takes the name of a folder, not the label.
-------------	--

Returns

What this method returns depends on the condition of the folder:

- If the folder name is valid and the end-user has access to the folder, a folder object is returned.
- If the folder is valid but the end-user doesn't have access to it a folder object is returned, however, the only property you can access is the Authorized property.
- If you specified an invalid name this method returns NULL.

Example

The following example returns a folder named ROOT:

```
&MyFolder = &MyPortal.FindFolderByName("ROOT");
```

The following example returns the folder object for an already instantiated content reference:

```
&Folder = &Portal.FindFolderByName(&CRef.ParentName);
```

Related Topics

Folder Class

GetQualifiedURL

Syntax

```
GetQualifiedURL(ContentProvider, RelativeURL)
```

Description

The **GetQualifiedURL** method returns an absolute URL as it should appear in a link. It concatenates the URI from *ContentProvider* with the value from *RelativeURL*. Once an absolute URL is retrieve, you can pass it to the FindCRefByURL method to return the content reference.



For more information about the different types of URLs, see Types of Portals Available.

This method is only used with more advanced portal applications.

Parameters

<i>ContentProvider</i>	Specify the ContentProvider for the URL you need.
<i>RelativeURL</i>	Specify the relative URL for the content reference.

Returns

An absolute URL as a string.

Example

The following creates a hyperlink for either the same or for a different database.

```
REM String together the query string to pass for the URL;

&MENU = "PORTAL_COMPONENTS";

&PNLGRPNAME = PANELGROUP.EO_PE_TASKLIST_DTL;

&MKT = "GBL";

&MYURL = "ICType=Panel" | "&Menu=" | &MENU | "&Market=" | &MKT |
"&PanelGroupName=" | &PNLGRPNAME;

REM -- The content Provider Name and URL querystring is required.--;
```

```
&CP_Name = "HRMS";  
  
LINKFIELD = &Portal.GetQualifiedURL(&CP_Name, &MYURL);
```

Related Topics

QualifiedURL content reference property, FindCRefByURL

Open

Syntax

```
Open(RegistryName)
```

Description

The **Open** method opens the PortalRegistry specified by the parameters. The registry must already exist. The **Open** method can only be used with a **closed** PortalRegistry, it cannot be used on an open registry.

Parameters

<i>RegistryName</i>	The name of the registry you want to open. This parameter takes a string value. If you specify a registry that doesn't exist, this method will return a False value.
---------------------	--

Returns

A Boolean value: True if the PortalRegistry object is successfully opened, False otherwise.

Example

In the following example, the name of the portal is stored as the value of a field in a record.

```
&PORTAL_NAME = EO_PE_REG_AET.PORTAL_NAME;  
  
&Portal = %Session.GetPortalRegistry();  
  
&Portal.Open(&PORTAL_NAME);
```

Related Topics

Open, Save, Close, Delete

Save

Syntax

```
Save()
```

Description

The **Save** method saves changes you've made to the PortalRegistry or to a ContentProvider. It does **not** save any changes you've made to a folder, content reference, and so on.



Note. To save changes for a folder or content reference use the save method associated with that object.

Parameters

None.

Returns

A Boolean value: True if the PortalRegistry object is successfully saved, False otherwise.

Example

```
If Not (&MyPortal.Save ()) Then  
  
    /* do error checking */  
  
End-if:
```

Related Topics

Save folder method, Save content reference method

PortalRegistry Class Properties

ContentProviders

This property returns a reference to a ContentProvider Collection.

This property is read-only.

DefaultTemplate

This property returns or sets the name of the default template for this PortalRegistry object as a string.

If you delete a template for a content reference, and none of the other content references on a page have a template, the default template specified with the ContentProvider is used. If there's no template for the ContentProvider, the template specified with this property is used.

If you want to return a reference to the content reference that contains the template specified by this property, use the TemplateObject property.

This property only takes the first 30 characters of a value. If you specify a value longer than 30 characters, the remaining characters are ignored.

This property is read-write.

Description

This property returns or sets the description of this PortalRegistry object as a string

The length of this property is 256 characters.

This property is read-write.

Name

This property returns the name of the PortalRegistry object as a string.

The length of this property is 30 characters.

This property is read-only.

RootFolder

This property returns the root folder object for this PortalRegistry object.

This property is read-only.

TemplateObject

This property returns a reference to a content reference object that contains the template specified by the DefaultTemplate property. If no template is specified with DefaultTemplate, this property returns NULL.

This property is read-only.

PortalRegistry Collection

A PortalRegistry Collection is returned by the FindPortalRegistries session class method.

PortalRegistry Collection Methods

First

Syntax

First()

Description

The **First** method returns the first PortalRegistry object in the PortalRegistry collection.

Example

```
&MyRegistry = &MyCollection.First();
```

Item

Syntax

```
Item(number)
```

Description

The **Item** method returns the PortalRegistry object with the position in the PortalRegistry collection specified by *number*.

Parameters

<i>number</i>	Specify the position number in the collection of the PortalRegistry object that you want returned.
---------------	--

Returns

A PortalRegistry object if successful, NULL otherwise.

Example

```
For &I = 1 to &MyCollection.Count  
  
    &MyRegistry = &MyCollection.Item(&I);  
  
    /* Do processing */  
  
End-For;
```

Next

Syntax

```
Next()
```

Description

The **Next** method returns the next PortalRegistry object in the PortalRegistry collection. You can only use this method after you have used the **First** method: otherwise the system doesn't know where to start.

Example

```
&MyRegistry = &MyCollection.Next();
```

PortalRegistry Collection Property

Count

This property returns the number of PortalRegistry objects in the PortalRegistry collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

ContentProvider Class

A content reference object specifies a web site that supplies content to the portal.



For examples, see Adding a ContentProvider and Changing ContentProvider Properties.

ContentProvider Class Properties

DefaultTemplate

This property returns or sets the name of the default template to be used with this ContentProvider as a string. You must specify the name of an existing template.

If you delete a template for a content reference, and none of the other content references on a page have a template, the default template specified with this property is used.

If you want to return a reference to the content reference that contains the template specified by this property, use the TemplateObject property.



If you have multiple content providers with the same URI, but no registered content references, the system uses the template from the alphabetically first content provider it finds.

This property is read-write.

Description

This property returns or sets a description of the ContentProvider as a string.

The length of this property is 256 characters.

This property is read-write.

Name

This property returns the name of the ContentProvider as a string.

The length of this property is 30 characters.

This property is read-only.

TemplateObject

This property returns a reference to a content reference object that contains the template specified by the DefaultTemplate property. If no template is specified with DefaultTemplate, this property returns NULL.



If you have multiple content providers with the same URI, but no registered content references, the system uses the template from the alphabetically first content provider it finds.

This property is read-only.

URI

This property sets or returns the URI of the ContentProvider as a string.

When you want to specify a secure site, you can use the https scheme.

The URI must be of the following form:

```
protocol://host:port/path/file/
```



Note. You must add the trailing backslash to the ContentProvider URI. If you don't, relative links in template components in those ContentProviders aren't constructed correctly.

The length of this property depends on your system database limit for LONG fields.

This property is read-write.

Example

Accessing the Content Provider or URI from the Registry

```
REM - Get the Content Provider Object and then the Content Provider's URI.--;
```

```
&CP = &Portal.ContentProviders.ItemByName("HRMS");
```

```
&URI = &CP.URI;
```

ContentProvider Collection

A ContentProvider collection is returned by the ContentProviders PortalRegistry property.

ContentProvider Collection Methods

DeleteItem

Syntax

```
DeleteItem(ContentProviderName)
```

Description

The **DeleteItem** method deletes the ContentProvider object identified by *ContentProviderName* from the ContentProvider Collection.

This method is not executed automatically. It is only executed when the PortalRegistry is saved.



A content reference doesn't have to have a ContentProvider. However, if it does, and you delete the ContentProvider, you will "lose" the content reference. Without the ContentProvider, you've lost part of the name required to find the content reference.

Parameters

<i>ContentProviderName</i>	Specify the name of a ContentProvider existing in the ContentProvider collection.
----------------------------	---

Returns

A Boolean value: True if the ContentProvider was deleted, False otherwise.

Example

```
If Not &MyCP.DeleteItem("HRMS_1999") Then
```

```
        /* Do error processing */  
    End-if;
```

First

Syntax

```
First()
```

Description

The **First** method returns the first ContentProvider object in the ContentProvider collection.

Parameters

None.

Returns

A ContentProvider object.

Example

```
&MyContentProvider = &MyCollection.First();
```

InsertItem

Syntax

```
InsertItem(ContentProviderName)
```

Description

The **InsertItem** method inserts the ContentProvider object identified by *ContentProviderName* from the ContentProvider Collection.

This method is not executed automatically. It is only executed when the PortalRegistry is saved.

Parameters

<i>ContentProviderName</i>	Specify the name of a ContentProvider existing in the ContentProvider collection.
----------------------------	---

Returns

A reference to the new ContentProvider object if the method executed successfully, NULL otherwise.

Example

```
&CProvider = &MyPortal.ContentProviders.InsertItem(&Dept_Name);
```

ItemByName

Syntax

```
ItemByName (Name)
```

Description

The **ItemByName** method returns the ContentProvider object with the name *Name*.

Parameters

<i>Name</i>	Specify the name of an existing ContentProvider within the ContentProvider collection. If you specify an invalid name, the object will be NULL.
-------------	---

Returns

A ContentProvider object if successful, NULL otherwise.

Example

```
&CProvider = &MyPortal.ContentProviders.ItemByName("HRMS");
```

ItemByURI

Syntax

```
ItemByURI (URI)
```

Description

The **ItemByURI** method returns the ContentProvider object specified by *URI*.

Parameters

<i>URI</i>	Specify the URI of an existing ContentProvider within the ContentProvider collection. If you specify an invalid URI, the object will be NULL.
------------	---

Returns

A ContentProvider object if successful, NULL otherwise.

Example

The following example accesses the URI and Content Provider of the target content:

```
&URI = %Request.RequestURI;  
  
&CP = &Portal.ContentProviders.ItemByURI (&URI);
```

Next

Syntax

```
Next ()
```

Description

The **Next** method returns the next ContentProvider object in the ContentProvider collection. You can only use this method after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

ContentProvider object.

Example

```
&MyContentProvider = &MyCollection.Next();
```

ContentProvider Collection Properties

Count

This property returns the number of ContentProvider objects in the ContentProvider Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

Folder Class

Folder objects are instantiated from other classes as follows:

- From a PortalRegistry object with the RootFolder property or the FindFolderByName method.
- From a Folder Collection with the First, ItemByName or Next methods



For an example of adding a folder, see Adding a Folder.

Folder Class Method

Save

Syntax

`Save()`

Description

The **Save** method saves any changes you've made to the folder, for example, a changed description or ValidFrom date.

Using this method also saves any changes you've made to PermissionValue or AttributeValue objects associated with this folder.

Parameters

None.

Returns

A Boolean value: True if the Folder object is successfully saved, False otherwise.

Example

```
If Not (&MyFolder.Save()) Then  
    /* do error checking */  
End-If;
```

Related Topics

Save PortalRegistry method, Save content reference method

Folder Class Properties

Attributes

This property returns an Attribute Collection containing the AttributeValue objects for this folder.

This property is read-only.

Visual Basic Considerations

If you're using Visual Basic, you must first instantiate the folder before you can access the Attributes.

The following code is **invalid**:

```
Set oAttrColl = oPortal.RootFolder.Attributes
```

The following code is valid:

```
Set oFolder = oPortal.RootFolder  
  
Set oAttrColl = oFolder.Attributes
```

Author

This property returns the author (PeopleSoft user ID) for this folder as a string.

This property is read-only.

AuthorAccess

This property specifies whether the author of the folder has access to the folder. This property takes a Boolean value. The default value for this property for a newly created object is True. This property is not cascaded.

This property is read-write.

Authorized

This property specifies whether the user is authorized to view this Folder.

This property is used when you access a particular Folder using FindFolderByName. If you've specified a valid Folder with this method, a Folder is always returned, whether you have authorized to view it or not. This is the only property you can view from an object that you are not authorized to.

The initial value of this property depends on the other permission properties (PublicAccess and AuthorAccess) as well as the permission list values in the PermissionValue object associated with this folder.

This property is read-only.

CascadedPermissions

This property returns a PermissionValue Collection. This collection contains the value of the permissions for all the child and parent objects (up to the root folder). If you want to determine only the permissions of the object use the Permissions property instead.



You can *not* add any PermissionValue objects to a collection returned by the CascadedPermissions property. You can only add values to the collection returned by the Permissions property.

This property is read-only.

ContentRefs

This property returns the Content Reference Collection for this folder.

This property is read-only.

CreationDate

This property returns the creation date for this folder as a string.

This property is read-only.

Description

This property returns or sets the description for this folder as a string.

The length of this property is 256 characters.

This property is read-write.

Folders

This property returns a reference to the Folder Collection for this folder.

This property is read-only.

Label

This property returns or sets the label for this folder as a string.

The length of this property is 30 characters.

This property is translatable.

This property is read-write.

Name

This property returns the name for this folder as a sting. The name is a unique identifier for each folder.

Every folder name must be unique in across the portal, not just in the parent folder.

This property is *not* translatable. However, the value for the Label property is translatable.

This property is read-only.

ParentName

This property returns the parent folder name for this folder as a string. This property is only valid if the folder is contained within another folder. If the folder using this property is the root folder, this property returns -1.

This property is read-only.

Path

The **Path** property returns a path to this folder, with each element of the path separated by a period. The path has the following syntax:

```
FolderLabel{Name}.[ChildFolderLabel{Name}]. . .
```

This property is read-only.

Example

```
Departments{Root}.HR{EastCoast}.AdministerWorkforce{Global}
```

Permissions

This property returns a PermissionValue Collection. This collection contains the value of the permissions for this folder. If you want to determine the permissions for all the parent objects (up to the root folder) use the CascadedPermissions property.

This property is read-only.

Product

This property returns or sets the PeopleSoft product for this folder as a string.

The length of this property is four characters.

This property is read-write.

SequenceNumber

The sequence number is used when returning a collection. The default order of the returned folders is based on the sequence number. Use this property to reorder folders.

If there are duplicates in the sequence number, the folders are returned in alphabetically.

The length of this property is four characters.

This property is read-write.

PublicAccess

This property indicates whether a folder is generally accessible or not, that is, if this property is set to True, any user can access the folder. This property is not cascaded.

This property takes a Boolean value.

The default value for this property for a newly created object is False.

This property is read-write.

ValidFrom

This property returns or sets the date this folder is valid from as a string.

This property is read-write.

ValidTo

This property returns or sets the date this folder is valid until as a string.

This property is read-write.



The PortalRegistry API never uses the ValidTo and ValidFrom fields to determine what to return in a collection. It is up to you to check for these values in your application.

Folder Collection

The Folder Collection object provides access to a collection of folders in a Folder object.

The Folder Collection is instantiated from the Folders Folder object property.

Folder Collection Methods

DeleteItem

Syntax

```
DeleteItem(FolderName)
```

Description

The **DeleteItem** method deletes the folder object identified by *FolderName* **from the database**. If the folder contains other folders, all child folders and their contents will also be deleted.



If you delete a folder, you delete **all** content in the folder. If you delete a folder that contains other folders, that is, a parent folder, all the child folders, as well as all the content references are deleted as well.



Note. The PortalRegistry classes execute all methods "interactively", that is, as they happen. The item won't be marked for deletion, then actually deleted later. The item will be deleted from the database **as soon as** the method is executed.

Parameters

<i>FolderName</i>	Specify the name of a folder existing in the folder collection.
-------------------	---

Returns

A Boolean value: True if the folder was deleted, False otherwise.

Example

```
If Not &MyFolderColl.DeleteItem("MYFOLDER") Then  
    /* Folder not deleted. Do error checking */  
End-If;
```

First

Syntax

```
First()
```

Description

The **First** method returns the first Folder object in the Folder collection.

Parameters

None.

Returns

Folder object.

Example

```
&MyFolder = &MyCollection.First();
```

InsertItem

Syntax

```
InsertItem(FolderName)
```

Description

The **InsertItem** method inserts the folder object identified by *FolderName* from the Folder Collection. It returns a reference to the new folder object. You must specify a unique *FolderName*, or you'll receive an error.



Note. The PortalRegistry classes execute all methods "interactively", that is, as they happen. The item won't be marked for insertion, then actually inserted later. The item will be inserted into the database **as soon as** the method is executed.

Parameters

FolderName

Specify the name of a folder existing in the folder collection. This parameter takes a string value. You must specify a name, not a label, for a folder.

Returns

A reference to the new Folder object if the method executed successfully, NULL otherwise.

Example

```
&DepthHPFldr = &MyPortal.Folders.InsertItem("PORT0145");
```

ItemByName

Syntax

```
ItemByName(Name)
```

Description

The **ItemByName** method returns the Folder object with the name *Name*.

Parameters

Name Specify the name of an existing folder within the folder collection. If you specify an invalid name, the object will be NULL. You must specify a name, not a label.

Returns

A folder object if successful, NULL otherwise.

Example

```
&DeptFldr = &MyPortal.RootFolder.Folders.ItemByName(&Dept_Name);
```

Next

Syntax

```
Next ()
```

Description

The **Next** method returns the next Folder object in the Folder collection. You can only use this method after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A folder object.

Example

```
Local ApiObject &MySession, &Root, &Folders, &MyFolder;
```

```
&MySession = %Session;
```

```
&MyPortal = &MySession.GetPortalRegistry("ADMIN");
```

```
&Root = &MyPortal.GetRoot();
```

```
&Folders = &Root.Folders;
```

```
&MyFolder = &Folders.First();
```

```
For &I = 1 to &Folders.Count  
  
    /* Do processing on folders */  
  
    If &I <> &Folders.Count  
        &MyFolder = &Folders.Next();  
    End-If;  
  
End-For;
```

Folder Collection Property

Count

This property returns the number of Folder objects in the Folder Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

Content Reference Class

The content reference object provides access to some kind of content. The type of content depends on the content reference.

content reference objects are instantiated from other classes as follows:

- From a PortalRegistry object with the FindCRefByURL or FindCRefByName properties.
- From a Content Reference Collection (instantiated from a folder) with the First, ItemByName or Next methods



For an example, see Adding a Content Reference.

Content Reference Class Methods

Save

Syntax

```
Save ()
```

Description

The **Save** method saves any changes you've made to the content reference, for example, a changed description. It also performs some validation.

Using this method also saves any changes you've made to `PermissionValue` or `AttributeValue` objects associated with this content reference.

Parameters

None.

Returns

A Boolean value: True if the content reference and its associated object saved successfully, False otherwise.

Example

```
If NOT (&MyCRef.Save ()) Then  
  
    /* save failed, do error processing */  
  
End-If;
```

Related Topics

Save PortalRegistry method, Save Folder method

Content Reference Class Properties

Attributes

This property returns an Attribute Collection containing the AttributeValues for this content reference object.

This property is read-only.

Author

This property returns the author (PeopleSoft user ID) for this content reference object as a string.

This property is read-only.

AuthorAccess

This property specifies whether the author of the content reference has access to the content reference. This property takes a Boolean value. The default value for this property is True.

This property is read-write.

Authorized

This property specifies whether the user is authorized to view this content reference.

This property is used when you access a particular content reference using FindCRefByURL or FindCRefByName. If you've specified a valid content reference with either of these methods, a content reference is always returned, whether you have authorized to view it or not. This is the only property you can view from an object that you are not authorized to.

The initial value of this property depends on the other permission properties (PublicAccess and AuthorAccess) as well as the permission list values in the PermissionValue object associated with this content reference.

This property is read-only.

CascadedPermissions

This property returns a PermissionValue Collection. This collection contains the value of the permissions for all the parent objects (up to the root folder). If you want to determine only the permissions of the object use the Permissions property instead.



You can *not* add any PermissionValue objects to a collection returned by the CascadedPermissions property. You can only add values to the collection returned by the Permissions property.

This property is read-only.

ContentProvider

This property returns or set the name of the ContentProvider for the content reference as a string.

If no ContentProvider was specified when the content reference was created, this property returns NULL.

This property is read-write.

CreationDate

This property returns the creation date for this content reference object as a string.

This property is read-only.

Data

This property returns the data for this content reference. This property is only valid when the StorageType property is LOCL.

The length of this property depends on your system database limit for LONG fields.

This property is read-write.

Example

```
&MyData = &MyCRef.Data;
```

Description

This property returns or sets the description for this content reference object as a string.

The length of this property is 256 characters.

This property is read-write.

Label

This property returns or sets the label for this content reference object as a string.

The length of this property is 30 characters.

This property is translatable.

This property is read-write.

Name

This property returns the name for this content reference object as a string. The name is a unique identifier for each content reference object.

Every content reference name must be unique in the portal, not just in the parent folder.

This property is read-only.

ParentName

This property returns the parent folder name for this content reference object as a string.

This property is read-only.

Example

The following example uses the ParentName to return a folder object for the content reference.

```
&Folder = &Portal.FindFolderByName(&CRef.ParentName);
```

Path

The **Path** property returns a path to this content reference, with each element of the path separated by a period. The path has the following syntax:

```
ContentReferenceLabel{Name}. [ChildContentReferenceLabel{Name}]. . .
```

This property is read-only.

Permissions

This property returns a PermissionValue Collection. This collection contains the value of the permissions for this content reference. If you want to determine the permissions for all the parent objects (up to the root folder) use the CascadedPermissions property.

This property is read-only.

Product

This property returns or sets the PeopleSoft product for this content reference object as a string.

The length of this property is four characters.

This property is read-write.

PublicAccess

This property indicates whether a folder is generally accessible or not, that is, if it will always be included in the general content reference collection or not. This property takes a Boolean value.

The default value for this property is False.

This property is read-write.

QualifiedURL

This property returns an absolute URL. If the content reference has a ContentProvider associated with it, the URI from the ContentProvider is concatenated with the URL from the content reference. If there is no ContentProvider, this property returns the text in the URL property.



For more information about the different types of URLs, see [Types of Portals Available](#).

This property is read-only.

SequenceNumber

The sequence number is used when returning a collection. The default order of the returned content references is based on the sequence number. Use this property to reorder content references. This property takes a number value.

If there are duplicates in the sequence number, the content references are returned alphabetically.

The length of this property is four characters.

This property is read-write.

StorageType

In general, content references contain information on where to get the content, and do not store the content. However, content references that are template or portal component types can have their content accessible directly from the content reference. In these cases, the Data property is valid and can be read or written, and the data is stored locally in the portal database.

StorageType	Meaning
LOCL	Local: Data property on content reference is valid.
RMTE	Remote: Data property is not valid.

When UsageType is a target this property must be set to RMTE and the URLType property should specify what format the Content Provider and URL are in.

When UsageType is either a template or a portal component this property can be set to either LOCL or RMTE.

Note that when StorageType is LOCL, it specifies a static template or portal component. But, when StorageType is RMTE it can specify either:

- a dynamically generated template or portal component
- a static template or portal component

In both cases, the ContentProvider and URL properties specifies how to get the template or portal component.

RMTE is the default value for a new content reference.

The length of this property is four characters.

This property is read-write.

Template

This property returns or sets the name of the template used with this content reference as a string. You must specify the name of an existing template.

This property uses the name *not* the label of a content reference.

This property is only used when UsageType is specified as Target (TARG) and TemplateType is specified as HTML.

If you want to return a reference to the content reference that contains the template specified by this property, use the TemplateObject property.

The length of this property is 30 characters.

This property is read-write.

TemplateObject

This property returns a reference to a content reference object that contains the template specified by the Template property. If no template is specified with Template, this property returns NULL.

This property is read-only.

TemplateType

This property indicates whether or not a template should be used to wrap the content. This property takes a string value.

Valid values are:

Value	Description
HTML	An HTML template should wrap the data (URL) in the content reference.
NONE	No template should be used. The text should not be wrapped.

HTML is the default value on a new content reference.

Use the NONE value if the URL should not appear in the portal. An example is when the content reference is a template itself.

If the homepage template for the user is un-retrievable, the system tries to use the default user's template first, before resorting to the portal default template.

When the content reference describes content, this property should be set to HTML.



For more information see Using Content References.

This property is read-write.

URL

This property returns or sets the URL for this content reference object as a string. The URL returns **exactly** as it appears in the database.

If you're setting a URL, you must use a unique URL.

The absolute URL, that is, the URL from the ContentProvider concatenated with this URL must be unique.



For more information about qualified URLs, see Types of Portals Available.

You receive an error if the URL you use is already registered.

If you want to retrieve a qualified URL, that is, one that contains the ContentProvider URI, use the QualifiedURL property.

The length of this property depends on your system database limit for LONG fields.

This property is read-write.

The format of the value for this property depends on the setting of other properties.



For more information see Using Content References.

URLType

This property indicates what kind of PeopleCode definition is used to build the page for the content reference. This property is only used for content that has the StorageType property set as **Remote**. If the type is External, a PeopleCode definition is **not** used. This property takes a string value.

Valid values are:

<i>Value</i>	<i>Description</i>
UPGE	PeopleSoft Component
USCR	PeopleSoft Script (iScript)
UEXT	Non-PeopleSoft URL (External)

UPGE is the default value on a new content reference.

This property is read-write.

UsageType

This property indicates what the content reference is used for. Several other properties depend on this property.



For more information see Using Content References.

This property takes a string value.

The length of this property is four characters.

Value	Description
FRMT	Frame template: the content reference is a frame-based template
HTMT	HTML template: the content reference is an HTML template
HPGT	Homepage template: the content reference is a homepage template
HPGC	Pagelet: the content reference is a pagelet used in the homepage.
TARG	Target: the content reference is the target content its template determines what else needs to be loaded and how it will look
TMPC	Template component: the content reference is part of the template, like a Related Links component or a Favorites component.

TARG is the default value on a new content reference.

This property is read-write.

ValidFrom

This property returns or sets the date this content reference is valid from as a string.

This property is read-write.

ValidTo

This property returns or sets the date this content reference is valid until as a string.

This property is read-write.



The PortalRegistry API never uses the ValidTo and ValidFrom fields to determine what to return in a collection. It is up to you to check for these values in your application.

Content Reference Collection

The content reference Collection provides access to the collection of content references in a Folder object.

The content reference Collection is instantiated from the ContentRefs Folder property.

Content Reference Collection Methods

DeleteItem

Syntax

```
DeleteItem(ContentReferenceName)
```

Description

The **DeleteItem** method deletes the content reference object identified by *ContentReferenceName* from the content reference Collection.

If you delete a template for a content reference, and none of the other content references on a page have a template, the default template specified with the ContentProvider is used. If there's no template for the ContentProvider, the template for the PortalRegistry is used. However, if you delete the template for the content reference, the ContentProvider and the PortalRegistry, you'll receive a runtime error.



Note. The PortalRegistry classes execute all methods "interactively", that is, as they happen. The item won't be marked for deletion, then actually deleted later. The item will be deleted from the database **as soon as** the method is executed.

Parameters

<i>ContentReferenceName</i>	Specify the name of a content reference existing in the content reference collection.
-----------------------------	---

Returns

A Boolean value: True if the content reference was deleted, False otherwise.

Example

```
If Not &MyCRef.DeleteItem("Test_CRef") Then  
    /* can't delete test data. Do error processing */  
End-if;
```


First

Syntax

```
First()
```

Description

The **First** method returns the first content reference object in the content reference collection.

Parameters

None.

Returns

A content reference object.

Example

```
&MyCRef = &MyCollection.First();
```

InsertItem

Syntax

```
InsertItem(ContentReferenceName, ContentProvider, URL)
```

Description

The **InsertItem** method inserts the content reference object identified by *ContentReferenceName* into the content reference Collection.



Note. The PortalRegistry classes execute all methods "interactively", that is, as they happen. The item won't be marked for insertion, then actually inserted later. The item will be inserted into the database **as soon as** the method is executed.

Parameters

<i>ContentReferenceName</i>	Specify the name of a new content reference. This parameter takes a string value. If you specify a name that already exists in the collection, you'll get an error.
<i>ContentProvider</i>	Specify a ContentProvider. This parameter takes a string value. If you specify a fully qualified URL, you can specify a NULL (that is, two quotation marks with no space between them ("")).

URL

Specify a URL that contains the content. The format of this parameter depends on other properties, such as the type of content reference, where the data is stored, and so on.



For more information, see Using Content References.

Returns

A reference to the new content reference object if the method executed successfully, NULL otherwise.

Example

The following example inserts an external content reference that is a template:

```
&URL = "<TEMPLATE>" | &ITEMNAME;

&MyCRef = &CRefColl.InsertItem(&ITEMNAME, "", &URL);
```

The following example inserts a content reference where URLType is iScript (USCR):

```
&URL =
ICType=Script&ICScriptProgramName=WEBLIB_PORTAL.PORTAL_NAV.FieldFormula.IScript_
Portal_Trans_Dyn

&MyCRef = &CRefColl.InsertItem(&ITEMNAME, "HRMS", &URL);
```

ItemByName

Syntax

```
ItemByName (Name)
```

Description

The **ItemByName** method returns the content reference object with the name *Name*.

Parameters

Name	Specify the name of an existing content reference within the content reference collection. If you specify an invalid name, the object will be NULL.
------	---

Returns

A content reference object if successful, NULL otherwise.

Example

```
&MyCRef = &CRefColl.ItemByName("PORTAL_ADMIN);
```

Next

Syntax

```
Next ()
```

Description

The **Next** method returns the next content reference object in the content reference collection. You can only use this method after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

content reference object.

Example

```
&MyCRef = &MyCollection.Next();
```

Content Reference Collection Property

Count

This property returns the number of content reference objects in the content reference Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

AttributeValue Class

The AttributeValue object provides access to attributes associated with either folders or content references.

AttributeValue objects are instantiated from an Attribute Collection Methods with the First, InsertItem, ItemByName or Next methods.



For an example of using attributes, see [Using Attributes](#).

AttributeValue Class Properties

Label

This property returns the label of the AttributeValue as a string. This property works with the Translatable property. If Translatable is set to True, the value of Label can be translated.

The length of this property is 30 characters.

This property is read-write.

Name

This property returns the name of the AttributeValue as a string.

The length of this property is 30 characters.

This property is read-only.

Example

```
&AttrColl = &Folder.Attributes;

&Attr = &AttrColl.First();

&Scroll = GetLevel0().GetRow(1).GetRowset(Scroll.PORTAL_FLDR_ATR);

&I = 1;

While All(&Attr)

    &Record = &Scroll.GetRow(&I).GetRecord(Record.PORTAL_FLDR_ATR);

    &Record.PORTAL_ATTR_NAM.Value = &Attr.Name;

    &Record.PORTAL_ATTR_VAL.Value = &Attr.Value;

    &Attr = &AttrColl.Next();

/* need this check so we don't insert extra blank row */

If All(&Attr) Then

    &Scroll.InsertRow(&I);
```

```
&I = &I + 1;  
  
End-If;  
  
End-While;
```

Translatable

This property specifies if the AttributeValue is translatable. This property takes a Boolean value: True if the AttributeValue can be translated, False otherwise.

If this property is set to True, the value of the Label can be translated.

This property is read-write.

Value

This property returns the value of the AttributeValue as a string.

The length of this property depends on your system database limit for LONG fields.

This property is read-write.

Example

If you want to specify more than a single value for an AttributeValue, you can specify several values separated by a semicolon. For example:

```
&MyAtt.value = "401k;benefits;dependants;HR";
```

Attribute Collection

The Attribute Collection object provides access to the collection of Attribute in a Folder or a content reference object.

An Attribute Collection is instantiated from other classes as follows:

- From a content reference object with the Attributes property
- From a Folder object with the Attributes property.

Attribute Collection Methods

DeleteItem

Syntax

```
DeleteItem(AttributeValueName)
```

Description

The **DeleteItem** method deletes the AttributeValue object identified by *AttributeValueName* from the Attribute Collection.

This method is not executed automatically. It is only executed when the parent object is saved.

Parameters

<i>AttributeValueName</i>	Specify the name of a AttributeValue existing in the Attribute Collection.
---------------------------	--

Returns

A Boolean value: True if the AttributeValue was deleted, False otherwise.

First

Syntax

```
First()
```

Description

The **First** method returns the first AttributeValue object in the Attribute Collection.

Parameters

None.

Returns

An AttributeValue object.

Example

```
&MyAttributeValue = &MyCollection.First();
```

InsertItem

Syntax

```
InsertItem(AttributeValueName)
```

Description

The **InsertItem** method inserts the AttributeValue object identified by *AttributeValueName* from the Attribute Collection.

This method is not executed automatically. It is only executed when the parent object is saved.

Parameters

AttributeValueName Specify the name of a AttributeValue existing in the Attribute Collection.

Returns

A reference to the new AttributeValue object if the method executed successfully, NULL otherwise.

Example

```

For &I = 1 To &Rowset.ActiveRowCount

    If &CompName = "PORTAL_CREF_ADM" Then

        &Record = &Rowset.GetRow(&I).GetRecord(Record.PORTAL_CATR_DV);

    Else

        &Record = &Rowset.GetRow(&I).GetRecord(Record.PORTAL_FATR_DV);

    End-If;

    If &Record.PORTAL_ATTR_NAM.Value <> "" Then

        &Attr = &AttrColl.InsertItem(&Record.PORTAL_ATTR_NAM.Value);

        &Attr.Value = &Record.PORTAL_ATTR_VAL.Value;

    End-If;

End-For;

```

ItemByName

Syntax

ItemByName (*Name*)

Description

The **ItemByName** method returns the AttributeValue object with the name *Name*.

Parameters

Name Specify the name of an existing AttributeValue within the Attribute Collection. If you specify an invalid name, the object will be NULL.

Returns

An AttributeValue object if successful, NULL otherwise.

Example

```
&Attr = &AttrColl.ItembyName("RELLINK");
```

Next**Syntax**

```
Next ()
```

Description

The **Next** method returns the next AttributeValue object in the Attribute Collection. You can only use this method after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

AttributeValue object.

Example

```
&MyAttributeValue = &MyCollection.Next();
```

Attribute Collection Property**Count**

This property returns the number of AttributeValue objects in the Attribute Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

PermissionValue Class

The PermissionValue object provides access to permission lists associated with either folders or content references.



You can *not* add any PermissionValue objects to a collection returned by the CascadedPermissions property. You can only add values to the collection returned by the Permissions property.

PermissionValue objects are instantiated from a PermissionValue Collection with the First, InsertItem, ItemByName or Next methods.



For more information about PermissionValue objects, see Security Considerations.



For an example of using security, see Setting Permissions using the PermissionValue Object.

PermissionValue Class Properties

Cascade

This property indicates whether the current permission should be granted to all child folders.

This property is only valid with folders, not with content references.

This property takes a Boolean value. The default value for a new PermissionValue object is False.

This property is read-write.

Name

Specify the name of a permission list as the name of this object. You must specify a permission list that has already been created. This property takes a string value.

The length of this property is 30 characters.

This property is read-write.

PermissionValue Collection

A PermissionValue collection is returned by the following:

- CascadedPermissions folder property
- Permissions folder property
- CascadedPermissions content reference property
- Permissions content reference property

What is contained in the PermissionValue collection depends on the property that created it.



For more information, see Security Considerations.

PermissionValue Collection Methods

DeleteItem

Syntax

```
DeleteItem (PermissionValueName)
```

Description

The **DeleteItem** method deletes the PermissionValue object identified by *PermissionValueName* from the PermissionValue Collection.

This method is not executed automatically. It is only executed when the parent object is saved.

Parameters

<i>PermissionValueName</i>	Specify the name of a PermissionValue existing in the PermissionValue collection.
----------------------------	---

Returns

A Boolean value: True if the PermissionValue was deleted, False otherwise.

Example

```
If Not &MyPValColl.DeleteItem("ALLPNLS") Then  
    /* do error processing */  
End-If;
```

First

Syntax

```
First()
```

Description

The **First** method returns the first `PermissionValue` object in the `PermissionValue` collection.

Parameters

None.

Returns

A `PermissionValue` object.

Example

```
&MyPermissionValue = &MyCollection.First();
```

InsertItem

Syntax

```
InsertItem(PermissionValueName)
```

Description

The **InsertItem** method inserts the `PermissionValue` object identified by *PermissionValueName* into the `PermissionValue` Collection.

This method is not executed automatically. It is only executed when the parent object is saved.



You can *not* add any `PermissionValue` objects to a collection returned by the `CascadedPermissions` property. You can only add values to the collection returned by the `Permissions` property.

Parameters

PermissionValueName Specify the name of an existing permission list.

Returns

A reference to the new `PermissionValue` object if the method executed successfully, `NULL` otherwise.

Example

```
&MyPermV = &MyPermVColl.InsertItem("ALLPNLS");

If Not &MyPermV Then

    /* do error processing */

End-If;
```

ItemByName

Syntax

```
ItemByName (Name)
```

Description

The **ItemByName** method returns the PermissionValue object with the name *Name*.

Parameters

<i>Name</i>	Specify the name of an existing PermissionValue within the PermissionValue collection. If you specify an invalid name, the object will be NULL.
-------------	---

Returns

A PermissionValue object if successful, NULL otherwise.

Example

```
&MyPVal = &MyPValColl.ItemByName("CUSTOMER");
```

Next

Syntax

```
Next ()
```

Description

The **Next** method returns the next PermissionValue object in the PermissionValue collection. You can only use this method after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

PermissionValue object.

Example

```
&MyPermissionValue = &MyCollection.Next();
```

PermissionValue Collection Property
Count

This property returns the number of PermissionValue objects in the PermissionValue Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

PortalRegistry Examples

There are several actions you want to perform using a PortalRegistry, such as adding a ContentProvider, adding a folder, changing permissions, and so on. The following examples covers the most usual cases. In addition, there are example programs of accessing the PortalRegistry classes using language environments other than PeopleCode.

Changing PortalRegistry Properties

In the following example, you're changing the default template used by a PortalRegistry. The following is the complete code sample: the steps explain each line.

```
Local ApiObject &MySession;

Local ApiObject &MyPortal, &MyPortalColl;

/* Access the current session */

&MySession = %Session;

If NOT &MySession Then

    /* do error processing */
```

```

End-if;

&MyPortalColl = &MySession.FindPortalRegistries();

For &I = 1 to &MyPortalColl.Count

    &MyPortal = &MyPortalColl.Item(&I);

    If &MyPortal.DefaultTemplate = "HR99" Then
        &MyPortal.DefaultTemplate = "HR00"
        &MyPortal.Save();
    End-If;

End-For;

```

To change PortalRegistry properties:

1. Get a Session object.

Before you can access a PortalRegistry object, you must get a session object. The session controls access to the registry, provides error tracing, allows you to set the runtime environment, and so on. Because you want to use the existing session, use the %Session system variable (instead of the GetSession function.)

```
&MySession = %Session;
```

2. Get a PortalRegistry collection.

Use the FindPortalRegistries method with no parameters in order to return a collection of all the PortalRegistries because you want to check all the registries for the invalid template.

```
&MyPortalColl = &MySession.FindPortalRegistries();
```

Use the Count property on the collection to loop through all the registries.

```
For &I = 1 to &MyPortalColl.Count
```

3. Get a PortalRegistry object.

You can access a PortalRegistry object from the PortalRegistry collection using the Item method.

```
&MyPortal = &MyPortalColl.Item(&I);
```

4. Check for the invalid template and correct if necessary.

Use the `DefaultTemplate` property both to check for the invalid template name, and to set the correct template. Save the `PortalRegistry` when you've made a correction.

```
If &MyPortal.DefaultTemplate = "HR99" Then

    &MyPortal.DefaultTemplate = "HR00"

    &MyPortal.Save();

End-If;
```

You may want to do error checking after you save the `PortalRegistry`.

Adding a ContentProvider

The following example adds a `ContentProvider` to an existing `PortalRegistry`. The following is the complete code sample: the steps explain each line.

```
Local ApiObject &MyPortal;

Local ApiObject &MyCPColl, &MyCProvider;

&MyPortal = %Session.GetPortalRegistry();

If NOT &MyPortal.Open(MYRECORD.PORTAL_NAME) Then

    /* Do error handling */

End-if;

/* Add a ContentProvider */

&MyCPColl = &MyPortal.ContentProviders;

&MyCProvider = &MyCPColl.InsertItem("HRMS_00");

&MyCProvider.URI = "http://MYMACHINE103100/servlets/iclientservlet/HRMS/";

&MyCProvider.DefaultTemplate = "MYPORTAL_HRMS";
```

```

&MyCProvider.Description = "Updated Content Provider for HRMS";

If NOT(&MyPortal.Save()) Then

    &ErrorCol = &MySession.PSMessages;

    For &I = 1 to &ErrorCol.Count

        &Error = &ErrorCol.Item(&I);

        /* do error processing */

    End-For;

&ErrorCol.DeleteAll();

End-If;

```

To add a ContentProvider:

1. Get a Session object. and a PortalRegistry.

Before you can access a PortalRegistry object, you must get a session object. The session controls access to the registry, provides error tracing, allows you to set the runtime environment, and so on. Because you want to use the existing session, use the %Session system variable (instead of the GetSession function.) In addition, you want to get a PortalRegistry. Using the GetPortalRegistry method returns a reference to an unpopulated PortalRegistry object.

```

&MyPortal = %Session.GetPortalRegistry();

```

2. Open the PortalRegistry

After you get a PortalRegistry object, you want to open it, that is, populate it with data. Use the **Open** method to do this. The Open method returns a Boolean value, and this example uses that value to do error checking to make sure that the PortalRegistry is actually opened. In addition, the name of the PortalRegistry is kept in the record MYRECORD, in the field PORTAL_NAME.

```

If NOT &MyPortal.Open(MYRECORD.PORTAL_NAME) Then

    /* Do error handling */

End-if;

```

3. Access the ContentProvider collection.

You can only add a ContentProvider to a ContentProvider collection. So you must access a ContentProvider collection first, using the ContentProviders property on a PortalRegistry.

```

&MyCPColl = &MyPortal.ContentProviders;

```


4. Add the ContentProvider.

You must use the name of the ContentProvider with the InsertItem method. This example uses HRMS_00. This method does **not** execute automatically, that is, the ContentProvider isn't actually inserted into the database until the PortalRegistry is saved.

```
&MyCProvider = &MyCPColl.InsertItem("HRMS_00");
```

5. Further define the ContentProvider.

The URI of a ContentProvider is used in conjunction with content references to define the location of content. So you should define the URI. If a template isn't found for any of the content references on a page at runtime, the system next tries to use the DefaultTemplate defined for a ContentProvider, so you should define a default template.

```
&MyCProvider.URI = "http://MYMACHINE103100/servlets/iclientservlet/HRMS/";
```

```
&MyCProvider.DefaultTemplate = "MYPORTAL_HRMS";
```

```
&MyCProvider.Description = "Updated Content Provider for HRMS";
```

6. Save the PortalRegistry and check for errors.

Because there is no Save method with a ContentProvider, you must save the PortalRegistry in order to complete your changes.

You can check if there were any errors using the PSMessages property on the session object.

```
If NOT(&MyPortal.Save()) Then

    /* save didn't complete */

    &ErrorCol = &MySession.PSMessages;

    For &I = 1 to &ErrorCol.Count

        &Error = &ErrorCol.Item(&I);

        /* do error processing */

    End-For;

    &ErrorCol.DeleteAll();

End-if;
```

If there are multiple errors, all errors are logged to the PSMessages collection, not just the first occurrence of an error. As you correct each error, you want to delete it from the PSMessages collection.



For more information see About Error Handling.



If you've called the PortalRegistry API in an Application Engine program, all errors are also logged in the Application Engine error log tables.

Changing ContentProvider Properties

In the following example, you'll be changing the name of a DefaultTemplate for all ContentProviders that have a specific DefaultTemplate. The following is the complete code sample: the steps explain each line.

```

Local ApiObject &MyPortal;

Local ApiObject &MyCPColl, &MyCProvider;

&MyPortal = %Session.GetPortalRegistry();

If NOT &MyPortal.Open(MYRECORD.PORTAL_NAME) Then

    /* Do error handling */

End-if;

/* Add a ContentProvider */

&MyCPColl = &MyPortal.ContentProviders;

&MyCProvider = &MyCPColl.First();

&MyCount = 1;

For &I = 1 to &MyCPColl.Count

    /* Check template name and change if necessary */

    If &MyCProvider.DefaultTemplate = "INV_DEPT_SALES" Then

        &MyCProvider.DefaultTemplate = "UK_INV_DEPT_SALES";

    End-If;

```

```
/* If not the end of the collection, get next ContentProvider */
```

```

If &MyCount <> &MyCPColl.Count
    &MyCount = &MyCount + 1;
    &MyCProvider = &MyCPColl.Next();
End-If;

```

```
End-For;
```

```
If NOT(&MyPortal.Save()) Then
```

```
&ErrorCol = &MySession.PSMessages;
```

```
For &I = 1 to &ErrorCol.Count
```

```
    &Error = &ErrorCol.Item(&I);
```

```
/* do error processing */
```

```
End-For;
```

```
&ErrorCol.DeleteAll();
```

```
End-If;
```

To change a ContentProvider:

1. Get a Session object. and a PortalRegistry.

Before you can access a PortalRegistry object, you must get a session object. The session controls access to the registry, provides error tracing, allows you to set the runtime environment, and so on. Because you want to use the existing session, use the %Session system variable (instead of the GetSession function.) In addition, you want to get a PortalRegistry. Using the GetPortalRegistry method returns a reference to an unpopulated PortalRegistry object.

```
&MyPortal = %Session.GetPortalRegistry();
```

2. Open the PortalRegistry

After you get a PortalRegistry object, you want to open it, that is, populate it with data. Use the **Open** method to do this. The Open method returns a Boolean value, and this example uses that value to do error checking to make sure that the PortalRegistry is actually opened. In

addition, the name of the PortalRegistry is kept in the record MYRECORD, in the field PORTAL_NAME.

```
If NOT &MyPortal.Open(MYRECORD.PORTAL_NAME) Then

    /* Do error handling */

End-if;
```

3. Access the ContentProvider collection.

You can only add a ContentProvider to a ContentProvider collection. So you must access a ContentProvider collection first, using the ContentProviders property on a PortalRegistry.

```
&MyCPColl = &MyPortal.ContentProviders;
```

4. Get the first ContentProvider in the collection.

This example loops through all the ContentProviders in the collection, starting with the first one, then using the **Next** method to access each one, to the end of the list. This example uses a second counter to determine the end of the list.

```
&MyCProvider = &MyCPColl.First();

&MyCount = 1;
```

5. Check the name of the DefaultTemplate and change it if necessary.

This example uses the Count property with the ContentProvider collection to determine how many times to loop through. Then it checks the existing name of the DefaultTemplate. If it's the incorrect name, the name is changed to the correct name.

```
For &I = 1 to &MyCPColl.Count

    /* Check template name and change if necessary */

    If &MyCProvider.DefaultTemplate = "INV_DEPT_SALES" Then

        &MyCProvider.DefaultTemplate = "UK_INV_DEPT_SALES";

    End-If;
```

6. Check the second counter to determine if at the end of the collection.

If the program hasn't reached the end of the ContentProvider collection, access the next ContentProvider.

```
If &MyCount <> &MyCPColl.Count

    &MyCount = &MyCount + 1;

    &MyCProvider = &MyCPColl.Next();
```

```
End-If;
```

7. Save the PortalRegistry and check for errors.

Because there is no Save method with a ContentProvider, you must save the PortalRegistry in order to complete your changes.

You can check if there were any errors using the PSMessages property on the session object.

```
If NOT(&MyPortal.Save()) Then

    /* save didn't complete */

    &ErrorCol = &MySession.PSMessages;

    For &I = 1 to &ErrorCol.Count

        &Error = &ErrorCol.Item(&I);

        /* do error processing */

    End-For;

    &ErrorCol.DeleteAll();

End-if;
```

If there are multiple errors, all errors are logged to the PSMessages collection, not just the first occurrence of an error. As you correct each error, you want to delete it from the PSMessages collection.



For more information see About Error Handling.



If you've called the PortalRegistry API an Application Engine program, all errors are also logged in the Application Engine error log tables.

Adding a Folder

The following example checks to see if a folder for a user exists. If it doesn't, it creates one.

The folder hierarchy for this example is as follows. The Users folder is where the portal stores things such as user homepages:

```
Root folder

    Users folder

        UserId1 folder
```

```

        UserId2 folder

```

```

        UserId3 folder

```

```

        . . .

```

The following is the complete code sample: the steps explain each line.

```

Local ApiObject &MyPortal;

Local ApiObject &UserFldrColl, &UserFldr;

&MyPortal = %Session.GetPortalRegistry();

If NOT &MyPortal.Open(MYRECORD.PORTAL_NAME) Then

    /* Do error handling */

End-if;

&UserFldrColl = &MyPortal.RootFolder.Folders.ItemByName("Users").Folders;

&UserFldr = &UserFldrColl.ItemByName(%UserId);

If &UserFldr = Null Then

    /* add Folder */

    &UserFldr = &UserFldrColl.InsertItem(%UserId);

    &UserFldr.Description = %UserId;

    /* Set dates */

    &UserFldr.ValidFrom = %Date;

    &ToDate = AddToDate(%Date, 1, 1, 0);

    &UserFldr.ValidTo = &ToDate;

```

```

/* Set properties */

&UserFldr.PublicAccess = True;

/* save the folder */

&UserFldr.Save();

End-If;

```

To add a folder:

1. Get a Session object. and a PortalRegistry.

Before you can access a PortalRegistry object, you must get a session object. The session controls access to the registry, provides error tracing, allows you to set the runtime environment, and so on. Because you want to use the existing session, use the %Session system variable (instead of the GetSession function.) In addition, you want to get a PortalRegistry. Using the GetPortalRegistry method returns a reference to an unpopulated PortalRegistry object.

```
&MyPortal = %Session.GetPortalRegistry();
```

2. Open the PortalRegistry

After you get a PortalRegistry object, you want to open it, that is, populate it with data. Use the **Open** method to do this. The Open method returns a Boolean value, and this example uses that value to do error checking to make sure that the PortalRegistry is actually opened. In addition, the name of the PortalRegistry is kept in the record MYRECORD, in the field PORTAL_NAME.

```

If NOT &MyPortal.Open(MYRECORD.PORTAL_NAME) Then

    /* Do error handling */

End-if;

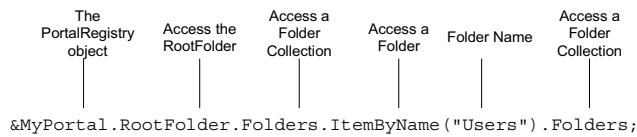
```

3. Access the User folder collection.

One of the strengths of dot notation is being able to string together many methods and properties into a single line of code. The result of this single line of code is to return a reference to the folder collection within the Users folder.

```
&UserFldrColl = &MyPortal.RootFolder.Folders.ItemByName("Users").Folders;
```

To following breaks the above code down:



PortalRegistry Syntax

The second `Folders` property returns a reference to a folder collection.

Note that in this kind of code, you must access the RootFolder for the PortalRegistry before you can access the folders collection.

If there were additional folders in your hierarchy, you could continue in the same way, using `ItemByName` and `Folders`.

4. Check if the user already has a folder.

The system variable %UserId returns the user ID of the current user. Then this example uses the ItemByName method on the folder collection to see if the user already has a folder. This assumes that the user folders are names according to the UserId. The ItemByName method returns a reference to the folder if it exists. If the folder does not exist, ItemByName returns NULL.

```
&UserFldr = &UserFldrColl.ItemByName(%UserId);
```

```
If &UserFldr = Null Then
```

5. Add a folder if one doesn't exist.

Use the UserId as the name of the folder, as well as for the description.

```
&UserFldr = &UserFldrColl.InsertItem(%UserId);
```

```
&UserFldr.Description = %UserId;
```

- 6.** Set the ValidFrom and ValidTo dates.

For this example, the folder should be accessible from the date it's created, so the example sets the `ValidFrom` property to be today's date.

When you first create a folder, the ValidTo date, that is, the date that this folder "expires", is set null, that is an empty string. This means it never expires. In this example, we set the ValidTo date to one year and one day after the current date.

```
&UserFldr.ValidFrom = %Date;  
&ToDate = AddToDate(%Date, 1, 1, 0);  
&UserFldr.ValidTo = &ToDate;
```


7. Set permissions for the folder.

There are several properties that work together to determine the access of a folder or content reference. This example sets the `PublicAccess` property to `True`, which means that everyone has access to it, and it's included in all collections of folders.

```
&UserFldr.PublicAccess = True;
```

This example only sets the permission properties on the folder. It doesn't set all the possible permissions, such as the `PermissionValue` objects for the folder.



For more information, see [Security Considerations](#).

8. Save the folder.

After you have finished your changes to the folder, you should save it.

```
&UserFldr.Save();
```

You may want to check for errors after you save each folder.

Adding a Content Reference

The following code example adds a content reference that's a target component to the Approve Expenses folder.

The folder hierarchy for this example is as follows:

```
Root folder
```

```
    Expenses folder
```

```
        Create Expenses folder
```

```
        Review Expenses folder
```

```
        Approve Expenses folder
```

The following is the complete code sample: the steps explain each line.

```
Local ApiObject &MyPortal;

Local ApiObject &CRef, &CRefColl;

Local ApiObject &Fldr;

&MyPortal = %Session.GetPortalRegistry();
```

```
If NOT &MyPortal.Open(MYRECORD.PORTAL_NAME) Then

    /* Do error handling */

End-if;

&Fldr =
&MyPortal.RootFolder.Folders.ItemByName("Expenses").Folders.ItemByName("Approve
Expenses");

/* add content reference */

&CRef = &Fldr.ContentRefs.InsertItem(&CRefname, MYCP, MYRECORD.MYURL);

&CRef.Description = &CRefname;

/* Set dates */

&CRef.ValidFrom = %Date;

&ToDate = AddToDate(%Date, 1, 1, 0);

&CRef.ValidTo = &ToDate;

/* Set content reference types */

&CRef.UsageType = "TARG"; /* Target content reference */

&CRef.URLType = "UPGE"; /* Component type of content reference */

&CRef.StorageType = "RMTE"; /* Template is remote. */

&CRef.Template = "EXPENSES_TEMPLATE"; /* name of template */

&CRef.TemplateType = "HTML"; /* type of Template */

/* Set URL */

&CRef.URL =
"ICType=Panel&Menu=MANAGE_EXPENSES&Market=GBL&PanelGroupName=APPROVE_FINAL"

/* Save the content reference */

&CRef.Save();
```

To add a content reference:

1. Get a Session object. and a PortalRegistry.

Before you can access a PortalRegistry object, you must get a session object. The session controls access to the registry, provides error tracing, allows you to set the runtime environment, and so on. Because you want to use the existing session, use the %Session system variable (instead of the GetSession function.) In addition, you want to get a PortalRegistry. Using the GetPortalRegistry method returns a reference to an unpopulated PortalRegistry object.

```
&MyPortal = %Session.GetPortalRegistry();
```

2. Open the PortalRegistry

After you get a PortalRegistry object, you want to open it, that is, populate it with data. Use the **Open** method to do this. The Open method returns a Boolean value, and this example uses that value to do error checking to make sure that the PortalRegistry is actually opened. In addition, the name of the PortalRegistry is kept in the record MYRECORD, in the field PORTAL_NAME.

```
If NOT &MyPortal.Open(MYRECORD.PORTAL_NAME) Then
```

```
    /* Do error handling */
```

```
End-if;
```

3. Access the folder collection.

One of the strengths of dot notation is being able to string together many methods and properties into a single line of code. The result of this single line of code is to return a reference to the folder collection within the s folder.

```
&Fldr =  
&MyPortal.RootFolder.Folders.ItemByName("Expenses").Folders.ItemByName("Approve  
Expenses");
```

See the Adding a Folder for explanation on the break down of this line of code.

4. Add the content reference.

You can only add content references from the collection of content references for the folder. After you have the content reference collection, use the InsertItem method to add the content reference. The ContentProvider for this content reference is named MYCP. The URL is stored in the record MYRECORD in the field MYURL.

```
&CRef = &Fldr.ContentRefs.InsertItem(&CRefname, MYCP, MYRECORD.MYURL);
```

```
&CRef.Description = &CRefname;
```

Note that the InsertItem method adds the content reference to the database as soon as it's executed. You could check for errors at this point to see if the content reference was added correctly.

5. Set the ValidFrom and ValidTo dates.

For this example, the content reference should be accessible from the date it's created, so the example sets the ValidFrom property to be today's date.

When you first create a content reference, the ValidTo date, that is, the date that this content reference "expires", is set to null, that is an empty string. This means it never expires. In this example, we set the ValidTo date to one year and one day after the current date.

```
&CRef.ValidFrom = %Date;

&ToDate = AddToDate(%Date, 1, 1, 0);

&CRef.ValidTo = &ToDate;
```

6. Set the type parameters.

Many content reference properties work together to set the type of content reference.



For more information see Using Content References.

In this example, the content reference is a target. It's a component type of content reference, which means it's a PeopleSoft definition. The template to be used for displaying this content reference is stored remotely, its name is EXPENSES_TEMPLATE, and it's an HTML template.

```
&CRef.UsageType = "TARG"; /* Target content reference */

&CRef.URLType = "UPGE"; /* Component type of content reference */

&CRef.StorageType = "RMTE"; /* Template is remote. */

&CRef.Template = "EXPENSES_TEMPLATE"; /* name of template */

&CRef.TemplateType = "HTML"; /* type of Template */
```

7. Set the URL.

The URL property of the content reference indicates where the content actually is. As this content reference is referencing a page, the URL has a specific format to indicate which page.

```
&CRef.URL =
"ICType=Panel&Menu=MANAGE_EXPENSES&Market=GBL&PanelGroupName=APPROVE_FINAL"
```

8. Save the content reference.

After you have finished adding the content reference, you should save it.

```
&CRef.Save();
```

You may want to check for errors after you save each content reference.

Setting Permissions using the PermissionValue Object

The following code example adds permissions for a folder through the PermissionValue object.

The following is the complete code sample: the steps explain each line.

```

Local ApiObject &MyPortal;

Local ApiObject &UserFldr;

Local ApiObject &PermV, &PermVColl;


&MyPortal = %Session.GetPortalRegistry();


If NOT &MyPortal.Open(MYRECORD.PORTAL_NAME) Then

    /* Do error handling */

End-if;


&UserFldr = &MyPortal.RootFolder.Folders.ItemByName("Users");


&CascadedColl = &UserFldr.CascadedPermissions;


&PermV = &CascadedColl.ItemByName("VENDOR1");


If NOT &PermV Then

    &PermVColl = &UserFldr.Permissions;

    &PermVColl.InsertItem("VENDOR1");

    &UserFldr.Save();

End-If;

```

To set permissions for a folder, using the PermissionValue object:

1. Get a Session object. and a PortalRegistry.

Before you can access a PortalRegistry object, you must get a session object. The session controls access to the registry, provides error tracing, allows you to set the runtime environment, and so on. Because you want to use the existing session, use the %Session system variable (instead of the GetSession function.) In addition, you want to get a

PortalRegistry. Using the GetPortalRegistry method returns a reference to an unpopulated PortalRegistry object.

```
&MyPortal = %Session.GetPortalRegistry();
```

2. Open the PortalRegistry

After you get a PortalRegistry object, you want to open it, that is, populate it with data. Use the **Open** method to do this. The Open method returns a Boolean value, and this example uses that value to do error checking to make sure that the PortalRegistry is actually opened. In addition, the name of the PortalRegistry is kept in the record MYRECORD, in the field PORTAL_NAME.

```
If NOT &MyPortal.Open(MYRECORD.PORTAL_NAME) Then

    /* Do error handling */

End-if;
```

3. Access the User folder.

One of the strengths of dot notation is being able to string together many methods and properties into a single line of code. The result of this single line of code is to return a reference to the folder named Users.

```
&UserFldr = &MyPortal.RootFolder.Folders.ItemByName("Users");
```

4. Access the entire, cascaded PermissionValue collection.

There are two types of PermissionValue collections you can access for a folder.

- One contains only the permission list values that are set at that folder. This collection is returned with the Permissions property.
- The other contains all the permission list values for folder, that is, it contains any permission list values set in any parent folder and cascaded. This collection is returned with the CascadedPermissions property.



You can *not* add any PermissionValue objects to a collection returned by the CascadedPermissions property. You can only add values to the collection returned by the Permissions property.

This example uses the a CascadedPermissions collection to check if the permission exists, because we don't want to add a permission if it already exists. Then it uses the Permissions collection to add the value.

```
&CascadedColl = &UserFldr.CascadedPermissions;
```



For more information, see Using PermissionValue and Cascading Permissions.

5. Check to see if the permission list value exists.

Before adding the new PermissionValue object, you want to make sure one by that name doesn't already exist.

```
&PermV = &CascadedColl.ItemByName("VENDOR1");
```

6. If the PermissionValue doesn't exist, add it.

The ItemByName returns a reference to a PermissionValue object if it exists, or NULL if it doesn't. So this example checks for NULL, and adds the PermissionValue if it doesn't exist.

Notice that this example is **not** changing the **Cascaded** property with the PermissionValue object. The default value for a new PermissionValue object is False. As this example does not want to cascade the permissions for this permission list, it retains the default value.

In addition, if the PermissionValue object is added, the folder is saved. The InsertItem method only executes after the parent object, the folder, is saved.

```
If NOT &PermV Then
    &PermVColl = &UserFldr.Permissions;
    &PermVColl.InsertItem("VENDOR1");
    &UserFldr.Save();
End-If;
```

You may want to check for errors after you save the folder.

Using Attributes

The following example uses attributes for a folder to determine what is displayed to an end-user. The end-user still has access to the content, however, it isn't displayed. This program doesn't cover how to hide or display content: it just shows the function used to determine if a folder should be displayed or not.

The following is the complete code sample: the steps explain each line.

```
Function checkVisible(&FolderId As ApiObject) Returns Boolean
    &colAttrib = &FolderId.Attributes;

    If &colAttrib.ItemByName("PORTAL_HIDE_FROM_NAV") <> Null Then
        If &colAttrib.ItemByName("PORTAL_HIDE_FROM_NAV").value = "TRUE" Then
            &display = False;
        End-If;
    End-If;
```

```
Return &display;
```

```
End-Function;
```

To use attributes for a folder:

1. Access the folder attribute collection.

A reference to a folder is passed into the function. Then the example uses the `Attributes` property on the folder to access the attribute collection for the folder.

```
&colAttrib = &FolderId.Attributes;
```

2. Check to see if the folder should be hidden.

This is actually two checks. One is checking to see if there is an attribute with the folder called `PORTAL_HIDE_FROM_NAV`. If the folder has such an attribute, the second check determines the value of this attribute. If both are true, the folder should be hidden, so the variable `&display` is set to `True`, and returned.

```
If &colAttrib.ItemByName("PORTAL_HIDE_FROM_NAV") <> Null Then
    If &colAttrib.ItemByName("PORTAL_HIDE_FROM_NAV").value = "TRUE" Then
        &display = False;
    End-If;
End-If;

Return &display;
```

Visual Basic Example

```
Dim oCref As ContentRef
Dim oCrefColl As ContentRefCollection

Set oSession = CreateObject("PeopleSoft.Session")
iResult = oSession.Connect(1, AppServStr, OperID, OperPasswd, 0)

Set oPortal = oSession.GetPortalRegistry
```



```

iResult = oPortal.Open("PORTAL")

Set oCrefColl = oPortal.RootFolder.ContentRefs
Set oCref = oCrefColl.InsertItem("papi012")
iResult = oPortal.Close

iResult = oPortal.Open("PORTAL")

Set oCref = oPortal.FindCRefByName("papi012")

oPortal.Close
oSession.Disconnect
Exit Sub

```

C/C++ Example

```

/*****
*
* psportal_test.cpp
*
*****
*
* Confidentiality Information:
*
* This module is the confidential and proprietary information of
PeopleSoft, Inc.; it is not to be copied, reproduced, or
* transmitted in any form, by any means, in whole or in part,
* nor is it to be used for any purpose other than that for which it
* is expressly provided without the written
* permission of PeopleSoft.
*
*

```

```

* Copyright (c) 1988-1999 PeopleSoft, Inc. All Rights Reserved.      *
*                                                                      *
*****
*                                                                      *
* SourceSafe Information:                                             *
*                                                                      *
* $Logfile::                                                         $*
* $Revision::                                                         $*
* $Date::                                                             $*
*                                                                      *
*****/
/*****
*      includes                                                         *
*****/

#ifdef PS_WIN32
#include "stdafx.h"
#endif

#include "bcdef.h"
#include "apiadapterdef.h"
#include "peoplesoft_peoplesoft_i.h"

#include <stdio.h>
#include <iostream.h>
#include <wchar.h>

/*****
*      general defines and macros                                     *

```

```

*****/

/*****

*      globals      *

*****/

HPSAPI_SESSION      hSession;

/*****

*      function prototypes      *

*****/

void DisconnectSession();

void GetFolder(HPSAPI_SESSION hSession, LPTSTR pszPortalName);

void OutError(HPSAPI_SESSION hSession);

/*****

* Function:      main      *

*      *      *

* Description:      *

*      *      *

* Returns:      *

*****/

void main(int argc, char* argv[])

{

    // Declare variables

    PSAPIVARBLOB      blobExtra;

    TCHAR szServer[80] = _T("//LCHE0110:900");

    TCHAR szUserid[30] = _T("PTMO");

    TCHAR szUserPswd[80] = _T("PT");

```

```

TCHAR szPortalName[80] = _T("PORTAL");

memset(&blobExtra, 0, sizeof(PSAPIVARBLOB));

// Establish a PeopleSoft Session.

HPSAPI_SESSION hSession = PeopleSoft_Session_Create();

if (PeopleSoft_Session_Connect(hSession, 1, szServer, szUserid, szUserPswd,
blobExtra))
{
    GetFolder(hSession, szPortalName);
}
else
{
    // Connect to AppServer Failed. Error out.

    if (PeopleSoft_Session_GetErrorPending(hSession))
    {
        wprintf(L"\nConnect to AppServer Failed.\n");

        OutError(hSession);
    }
    else
        wprintf(L"No error pending from session.\n");
}

DisconnectSession();
}

```

```

/*****

*      function implementations      *

*****/

```

```

/*****

* Function:      DisconnectSession                      *
*
*
* Description:   Disconnects the Session object.        *
*
*
* Returns:       None                                  *
*****/

void DisconnectSession()
{
    // Disconnect current session to free memory and session variables.
    if (hSession)
    {
        if (PeopleSoft_Session_Disconnect(hSession))
        {
            PeopleSoft_Session_Release(hSession);
        }
        else
        {
            wprintf(L"Disconnect to AppServer Failed.\n");
        }
    }
}

```

```

/*****

* Function:      GetFolder                              *
*
*
* Description:   Get root folder and display folder name *
*
*
*****/

```

```

* Returns:      None
*
*****/

void GetFolder(HPSAPI_SESSION hSession, LPTSTR pszPortalName)
{
    LPTSTR          szStr;

    HPSAPI_PORTALREGISTRY hPortal;

    HPSAPI_FOLDER      hRootFolder;

    hPortal=(HPSAPI_PORTALREGISTRY)
    PeopleSoft_Session_GetPortalRegistry(hSession);

    PortalRegistry_PortalRegistry_Open(hPortal, pszPortalName);

    if (PortalRegistry_PortalRegistry_Open(hPortal, pszPortalName) == false)
        _tprintf(_T("Open portal failed\n"));
    else
    {
        hRootFolder = PortalRegistry_PortalRegistry_GetRootFolder(hPortal);

        szStr = PortalRegistry_Folder_GetName(hRootFolder);

        _tprintf(_T("root folder name = %s\n"), szStr);
    }
}

/*****

* Function:      OutError
*
*
* Description:   Output error message from session psMessage object
*
*
* Returns:      None
*
*****/

void OutError(HPSAPI_SESSION hSession)

```

```

{
    LPTSTR                szStr;

    HPSAPI_PSMESSAGECOLLECTION hMsgColl;

    HPSAPI_PSMESSAGE hMsg;


    hMsgColl=(HPSAPI_PSMESSAGECOLLECTION)
    PeopleSoft_Session_GetPSMessages(hSession);


    wprintf(L"Get psMessage collection ok.\n");


    int i;

    PSI32 count;

    count=(PSI32) PeopleSoft_PSMessagesCollection_GetCount;

    wprintf(L"Count= %d\n", count);


    for (i=0; i<count; i++)
    {
        hMsg=(HPSAPI_PSMESSAGE) PeopleSoft_PSMessagesCollection_Item(hMsgColl,
i);

        szStr=PeopleSoft_PSMessages_GetText(hMsg);

        wprintf(L"\nMessage from session: %s\n",szStr);

    }

}

```

ProcessRequest Class

The ProcessRequest class provides properties and a method for scheduling a pre-defined process or job. You must define the process or the job (using Process Scheduler Manager) *before* you can schedule it using the ProcessRequest class.

The properties of this class contain the same values as those required by Process Scheduler Manager for scheduling a process or job. Values you provide for these properties may override the equivalent values set in Process Scheduler Manager, depending on the override settings you make in Process Scheduler pages.



For more information, see Process Scheduler.

Starting with PeopleSoft 8, PeopleSoft recommends using the ProcessRequest class instead of the ScheduleProcess PeopleCode function.

Restrictions on Use in PeopleSoft Internet Architecture, Three-Tier Mode and Windows Client

The **ProcessRequest** RunLocation property specifies whether the scheduled process is to be run on the client or on the server.

If the ProcessRequest object is called in a program running on the application server, the process cannot run on the client; that is, you can't set the RunLocation property to "CLIENT". This will cause a runtime error and the transaction will be canceled.

If the PeopleCode program where the ProcessRequest is called runs on the application server, then COBOL and SQR processes must be set to run on the server. If the PeopleCode program runs on the client (which could happen in either 2-tier or 3-tier mode), then COBOL or SQR processes can run on either the client or the server.

Example

The following is an example of using the ProcessRequest class. It runs on the server.

```
Local ProcessRequest &RQST;

/*****
* Construct a ProcessRequest Object.
*****/

&RQST = CreateProcessRequest();

/*****
* RunControlID and ProcessType, ProcessName or JobName
* are required properties.
*****/
```



```

*****/

&RQST.ProcessType = "SQR Report";

&RQST.ProcessName = "XRFWIN";

&RQST.RunControlID = RUN_CNTL_ID;

/*****
* RunLocation can be Client, Server or a Process Scheduler Server      *
* Name. If the Run Location is Server, the Process Request would be    *
* scheduled on Process Scheduler server for the default Operating      *
* System.                                                                *
* A default Operating System is defined on the System Settings page    *
* of Process Scheduler Manager.                                         *
*****/

&RQST.RunLocation = "PSNT";

/*****
* OutDestTypes are None, File, Printer, Window, Email and Web. Window is
Supported for Run Location Client only. In this example, letting Process
Scheduler resolve the output directory at runtime.                      *
*****/

&RQST.OutDestType = "%OutputDirectory%";

&RQST.OutDestFormat = "SPF";

&RQST.OutDest = "C:\TEMP\";

/*****
* DateTime when the Process Request will be scheduled to Run
Also TimeZone
*
*****/

&RQST.RunDateTime = %Datetime;

```

```

&RQST.TimeZone = %ServerTimeZone;

/*****

* Schedules the process based on the object properties, setting the *
* ProcessInstance and Status properties. *
* Status of zero means Success. *
*****/

&RQST.Schedule();

&PRCSSTATUS = &RQST.Status;

If &RQST.Status = 0 then

    /* Schedule succeeded. */

    &PRCSINSTANCE = &RQST.ProcessInstance;

Else

    /* Process (job) not scheduled, do error processing */

End-If;

```

Declaring a ProcessRequest Object

ProcessRequest objects are declared using the ProcessRequest data type. For example,

```
Local ProcessRequest &RQST;
```

Scope of a ProcessRequest Object

A ProcessRequest object can only be instantiated from PeopleCode. However, if you need to access the Process Schedule Manager from outside of PeopleCode, like in a Visual Basic program, you can use the Process Schedule Manager API.



For more information, see Process Scheduler.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Component Interface PeopleCode, record field PeopleCode, and so on.

ProcessRequest Class built-in Function

CreateProcessRequest

ProcessRequest Class Method

Schedule

Syntax

```
Schedule()
```

Description

The **Schedule** method inserts a request in the Process Request table which will run according to the values in the properties of the ProcessRequest object. In order to successfully schedule a process of job, certain properties are required.

- If you're scheduling a job, you must assign values to the following properties:
 - RunControlID
 - JobName
- If you're scheduling a process, you must assign values to the following properties:
 - RunControlID
 - ProcessName
 - ProcessType

Parameters

None.

Returns

None. If you want to verify that the method executed successfully, check the value of the Status property.

Example

```
&MYRQST.Schedule();  
  
If &MYRQST.Status = 0 then  
  
    /* Schedule succeeded. */  
  
Else
```

```
/* Process (job) not scheduled, do error processing */  
End-If;
```

Related Topics

RunControlID, JobName, ProcessName, ProcessType properties and CreateProcessRequest function

ProcessRequest Class Properties

EmailAttachLog

This property specifies whether or not a log file is attached to the email sent at completion of this job (or process). This property takes a Boolean value.

This property is read-write.

Example

```
&RQST.EmailAttachLog = False; /* Do not attach Log File */
```

EmailSubject

This property specifies the text used in the subject of the email sent at completion of this job (or process). This property takes a string value.

This property is read-write.

Example

```
&RQST.EmailSubject = "SQR Report: Cross Reference Listing";
```

EmailText

This property specifies the text of the email sent at the completion of this job (or process). This property takes a string value.

This property is read-write.

Example

```
&RQST.EmailText = "This text will be displayed as the text of this email ";
```

You can also use the text from a message in the message catalog for this property.

```
&RQST.EmailText = MsgGetText(65, 117, "Sample text for email with two  
parameters", &MessageParm1, &MessageParm2);
```

JobName

This property contains the name of a job you've defined in Process Scheduler.

If you want to schedule a job, you must assign valid values to **JobName** and RunControlID in order for the Schedule method to succeed.

If you're scheduling a job, you don't need to set the ProcessType property.

This property is read-write.

Example

```
&MYRQST.JobName = "3SQR";
```

LanguageCd

This property allows you to specify a language code for the process or job. If you don't specify a language code, first Process Scheduler will look in the process run control table to retrieve a language code. If there isn't a value there, Process Scheduler will use the user's language code specified in the User Definition table.

This property takes a string value.

This property is read-write.



For more information about PeopleSoft language codes, see International Language Architecture.

Example

```
&MYRQST.LanguageCd = "ESP" /* Spanish */
```

OutDest

This property specifies the output destination for the process or job to be scheduled as a string. Valid values depend on your setting for the OutDestType property:

- If OutDestType is FILE, **OutDest** must be the name of a file or a directory. If the OutDest property isn't set, it defaults to the setting in the Process profile.
- If OutDestType is PRINTER, **OutDest** must be the name of a printer. If the OutDest property isn't set, it defaults to the setting in the Process profile.
- If OutDestType is WEB or EMAIL, OutDest should contain the list of User IDs, Role IDs, or email addresses (for email only). If you don't set this property, it's automatically assigned to the person who submitted the request.
- If OutDestType is WEB, Process Scheduler uses the **OutDest** property to determine who has access to the output.

- If OutDestType is EMAIL, Process Scheduler uses the **OutDest** property to determine who to send the report output to.

In both these cases, in order to identify whether it's a User ID or a Role ID, the value has to be preceded by one of the following:

- **U:***Username* or **User:***Username* for a UserID
- **R:***Rolename* or **Role:***Rolename* for a Role ID

If the ID is **not** preceded by either of these identifiers, Process Scheduler assumes it's an email address.



Each ID **must** be separated by a semi-colon (;).

The following are two examples:

```
&RQST.OutDest = "U:PTDMO;R:Employee;robert_smith@peoplesoft.com"
```

```
&RQST.OutDest =  
"User:PS;Role:Employee;sue_line@XYZ.com;simon_gree@peoplesoft.com"
```

- For any other value of OutDestType, this property has no effect.

You can specify directory and printer names using the UNC (Uniform Naming Convention) protocol.

This property is read-write.

Example

```
&MYRQST.OutDest = "C:\TEMP";
```

OutDestFormat

This property specifies the output format for the process or job to be scheduled as a string. Valid values depend on your settings for the ProcessType and OutDestType properties:

ProcessType	OutDestType	Valid OutDestFormats	Default
APPENGINE	NONE	NONE	NONE
COBOL	NONE	NONE	NONE
CRYSTAL	EMAIL	DOC, HTM, RPT, RTF, TXT, WKS, XLS	RPT
CRYSTAL	FILE	DOC, HTM, RPT, RTF, TXT, WKS,	RPT

		XLS	
CRYSTAL	PRINTER	RPT	RPT
CRYSTAL	WEB	DOC, HTM, RPT, RTF, TXT, WKS, XLS	RPT
CRYSTAL	WINDOW	RPT	RPT
CUBEBUILDER	NONE	NONE	NONE
nVision-Report	EMAIL	HTM, XLS	XLS
nVision-Report	FILE	HTM, XLS	XLS
nVision-Report	PRINTER	HTM, XLS	XLS
nVision-Report	WEB	HTM, XLS	XLS
nVision-Report	DEFAULT	DEFAULT	DEFAULT
nVision-ReportBook	EMAIL	HTM, XLS	XLS
nVision-ReportBook	FILE	HTM, XLS	XLS
nVision-ReportBook	PRINTER	HTM, XLS	XLS
nVision-ReportBook	WEB	HTM, XLS	XLS
nVision-ReportBook	DEFAULT	DEFAULT	DEFAULT
SQR	EMAIL	CSV, HP, HTM, LP, PDF, PS, SPF,OTHER	SPF
SQR	FILE	CSV, HP, HTM, LP, PDF, PS, SPF,OTHER	SPF
SQR	PRINTER	HP, LP, PS, WP	HP
SQR	WEB	CSV, HP, HTM, LP, PDF, PS, SPF,OTHER	SPF
SQR	WINDOW	SPF	SPF
WinWord	NONE	NONE	NONE
OTHER	NONE	NONE	NONE

This property is read-write.

Example

```
&MYRQST.OutDestFormat = "RTF";
```

OutDestType

This property specifies the type of output for the process or job to be scheduled as a string.

The value specified for this property will be used *only* if the Output Destination Type for the pre-defined process or job is specified as **Any**. Otherwise any value specified for this property will be ignored.

Valid (and default) values depend on your settings for the ProcessType and RunControlID properties:

ProcessType	RunLocation	Valid OutDestTypes	Defaults
APPENGINE	CLIENT or SERVER	NONE	NONE
COBOL	CLIENT or SERVER	NONE	NONE
CRYSTAL	CLIENT	NONE, FILE, PRINTER, WINDOW	FILE
CRYSTAL	SERVER	NONE, EMAIL, FILE, WEB, PRINTER	FILE
CUBEBuilder	CLIENT or SERVER	NONE	NONE
nVision	CLIENT	NONE, FILE, PRINTER, DEFAULT	FILE
nVision	SERVER	NONE, EMAIL, FILE, WEB, PRINTER, DEFAULT	FILE
SQR	CLIENT	NONE, FILE, PRINTER, WINDOW	FILE
SQR	SERVER	NONE, EMAIL, FILE, WEB, PRINTER	FILE
WinWord	CLIENT or SERVER	NONE	NONE
OTHER	CLIENT or	NONE	NONE

	SERVER		
--	--------	--	--

This property is read-write.

Example

```
&MYRQST.OutDestType = "FILE";
```

ProcessInstance

This property is a system-generated identification number. Process Scheduler assigns a **ProcessInstance** at runtime to each process or job it successfully schedules. You can also use this property in conjunction with the SubmitJobItem property to submit job items yourself with different output parameters.

This property is read-write.

Example

```
&MYRQST.Schedule();

If &MYRQST.Status = 0 then

    /* process successfully scheduled */

    &ProcInst = &MYRQST.ProcessInstance;

Else

    /* do error processing */

End-If;
```

ProcessName

This property specifies the name of a predefined process as a string.

In order to successfully schedule a process, you must assign valid values to ProcessName, ProcessType and RunControlID (that is, for the Schedule method to succeed.)

This property is read-write.

Example

```
&MYRQST.ProcessName = "XRFWIN";
```

ProcessSeq

This property specifies the sequence in which jobs are processed. This property is only used when you want to specify jobs yourself.

This property is read-write.

ProcessType

This property specifies the name of a predefined process type as a string.

In order to successfully schedule a process, you must assign valid values to ProcessName, ProcessType and RunControlID (that is, for the Schedule method to succeed.)

The valid values for ProcessType depend on the types of processes you have defined in your system. There are generic process types that are delivered with your installation of PeopleSoft. These process types may include the following:

- Application Engine
- COBOL SQL
- Crw Online
- Crystal
- Cube Builder
- Database Agent
- Message Agent API
- PSJob
- SQR Process
- SQR Report
- SQR Report For WF Delivery
- WinWord
- nVision-Report
- nVision-ReportBook

If you define your own processes, you can use the name of that process with the ProcessType property. For example, suppose you create a custom process named "Custom CBL Programs." You could use this as follows:

```
&MyRqst.ProcessType = "Custom CBL Programs";
```

Note that spaces are included in the string for ProcessType.

This property is read-write.

Example

Note that spaces are included in the string for ProcessType.

```
&MYRQST.ProcessType = "Application Engine";
```

RunControlID

This property returns a string that serves, along with the user's ID, as a key that identifies a predefined group of parameters to be used by a process or a job at runtime.

In order to successfully schedule a process, you must provide valid values for **RunControlID**, **ProcessName** and **ProcessType**.

In order to successfully schedule a job, you must provide valid values for **RunControlID** and **JobName**.

This property is read-write.

Example

```
&MYRQST.RunControlID = "MYRUNCONTROLID";  
  
or  
  
&MYRQST.RunControlID = PRCSAMPLEREC.RUN_CNTL_ID;
```

RunDateTime

This property contains a **DateTime** value that specifies when the scheduled process or job will run.

If you don't specify a value for this property, and there is no date time set for the pre-defined process or job, the process or job will run as soon as the **Schedule** method is executed.

This property is read-write.

Example

The following example schedules the process or job to run immediately when the **Schedule** method is executed:

```
&MYRQST.RunDateTime = %Datetime;
```

RunLocation

This property specifies the host computer on which the scheduled process or job should run. This property takes a string value.

This property should only be used when you're scheduling processes. Jobs always run on the server.

The value specified for this property will be used *only* if the run location for the pre-defined process is specified as **Both**. Otherwise any value specified for this property will be ignored.

Valid values for **RunLocation** are:

- CLIENT
- SERVER
- A specific server name, such as PSNT

This property is read-write.

Example

```
&MYRQST.RunLocation = "CLIENT";
```

or

```
&MYRQST.RunLocation = "PSNT";
```

RunRecurrence

This property specifies the frequency with which a process or job is to be run as a string. The **RunRecurrence** value you use must be the name of a **Recurrence Definition** defined in Process Scheduler Manager in order to successfully schedule a job or process (that is, for the Schedule method to succeed.)

This property is read-write.

Example

```
&MYRQST.RunRecurrence = "M-F at 5pm";
```

SubmitJobItem

Use this property to submit jobs yourself with different output parameters for each process in the job. This property takes a Boolean value. The default value of this property is True.

If you want to submit job items yourself with different output parameters for each process in the job, do the following:

1. Set the properties for the job header.
2. Set SubmitJobItem as False.
3. Execute the Schedule method.
4. Get the ProcessInstance.
5. Loop through all the processes (job items) within the job and set the ProcessInstance for each process equal to the ProcessInstance of the job header.
6. Set the ProcessName and ProcessSeq for each process.
7. Set any other properties you wanted, such as the OutDest, the OutDestFormat, and so on.

8. Execute the Schedule method for each process.

If you want to submit the job header only and have the system execute Schedule() for each process within the job, do the following:

9. Set the properties for the job header.
10. Set SubmitJobItem to True (this is the default value for this property).
11. Execute the Schedule method for the header.

There is no need to execute the Schedule method for each process. The system will do that for you.

Example

The following code example submits a job and job items with the same output type, format and definition:

```
Local ProcessRequest &RQST;

&RQST = CreateProcessRequest();

&RQST.JobName = "3CRYSTAL";

&RQST.RunControlID = RUN_CNTL_ID;

&RQST.RunLocation = "Server";

&RQST.OutDestType = "File";

&RQST.OutDestFormat = "RPT";

&RQST.OutDest = "C:\TEMP\";

&RQST.RunDateTime = %Datetime;

/* Set SubmitJobItem = TRUE if you want Process Scheduler to submit all the job
items for you. In this case, all the job items are submitted with the same
output type, output format and output destination. This is the default value
for this property. */

&RQST.SubmitJobItem = TRUE;

&RQST.Schedule();
```

```

&PRCSSTATUS = &RQST.Status;

If &RQST.Status = 0 then

    /* Schedule succeeded. */

    &PRCSINSTANCE = &RQST.ProcessInstance;

Else

    /* Process (job) not scheduled, do error processing */

End-If;

```

The following code example submits a job and job items with the different output type, format and definitions:



If you choose to set different output type and format for each job items, you must schedule all the job items. Process Scheduler will not schedule any job items for you.

```

Local ProcessRequest &RQST;

&RQST = CreateProcessRequest();

&RQST.JobName = "3CRYSTAL";

&RQST.RunControlID = RUN_CNTL_ID;

&RQST.RunLocation = "Server";

&RQST.OutDestType = "File";

&RQST.OutDestFormat = "RPT";

&RQST.OutDest = "C:\TEMP\";

&RQST.RunDateTime = %Datetime;

&RQST.TimeZone = "EST";

/* Set SubmitJobItem = False, Process Scheduler will not submit the job items
for you. You are responsible for submitting all the job items.
*/

&RQST.SubmitJobItem = FALSE;

/* save the job name for job items. */

```

```

&wsJobName = &RQST.JobName;

&RQST.Schedule();

&PRCSSTATUS = &RQST.Status;

If &RQST.Status = 0 then

    /* Schedule succeeded. */

    &wsPRCSINSTANCE = &RQST.ProcessInstance;

    /* Schedule all the job items for the job */

    /* If you don't submit job items yourself, no job items will be scheduled
    from Process Scheduler. */

    ScheduleJobItems();

Else

    /* Process (job) not scheduled, do error processing */

End-If;

```

The following is the function called from the previous code example, used to submit job items. It assumes looping through a rowset which contains the job header and a list of job items for the job 3CRYSTAL.

```

Function ScheduleJobItems();

/* get all job items for the job */

&selectjobitems = "SELECT DISTINCT F.DESCR, I.PRCNAME, I.PRCSTYPE,
T.GENPRCSTYPE, I.PRCJOBSEQ, I.PRCJOBNAME, P.PNLGRPNAME, F.OUTDESTSRC,
F.OUTDESTTYPE FROM PS_PRCJOBDEFN D, PS_PRCJOBPNL P, PS_PRCJOBITEM I,
PS_PRCJOBGRP G, PS_PRCDEFN F, PS_PRCSTYPEDEFN T WHERE P.PNLGRPNAME = :1 AND
D.PRCJOBNAME = P.PRCJOBNAME AND D.PRCJOBNAME = I.PRCJOBNAME AND

```

```
D.PRCJOBNAME = G.PRCJOBNAME AND I.PRCSTYPE = F.PRCSTYPE AND I.PRCNAME =
F.PRCNAME AND I.PRCSTYPE = T.PRCSTYPE AND D.PRCJOBNAME = :2";
```

```
&RSJobItemList.Select(Record.PRCRQSTDLGLIST, &selectjobitems, %Component,
&wsJobName);
```

```
For &I = 1 To &RSJobItemList.ActiveRowCount
```

```
    &RQST = CreateProcessRequest();
```

```
    &RQST.JobName = "3CRYSTAL";
```

```
    &RQST.RunControlID = RUN_CNTL_ID;
```

```
    &RQST.RunLocation = "Server";
```

```
    &RQST.RunDateTime = %Datetime;
```

```
    &RQST.TimeZone = "EST";
```

```
    &RQST.ProcessType =
    &RSJobItemList.GetRow(&I).PRCRQSTDLGLIST.PRCSTYPE.Value;
```

```
    &RQST.ProcessName =
    &RSJobItemList.GetRow(&I).PRCRQSTDLGLIST.PRCNAME.Value;
```

```
/* Use the process instance returned from the job header request */
```

```
&RQST.JobInstance = &wsPRCSINSTANCE;
```

```
/* loop through job item. Job Sequence is used to track the sequence of
execution of the job items if the run mode is "Serial". In this case, you need
to make sure the first job item runs before the second job item, and so on. */
```

```
    &RQST.JobSeq = &I;
```

```
    &RQST.JobName = &wsJobName;
```

```
/* for each job item, you can assign different outdesttype, outdestformat,
outdest */
```



```

    &RQST.OutDestType =
&RSJobItemList.GetRow(&I).PRCSRQSTDLGLIST.OUTDESTTYPE.Value;

    &RQST.OutDestFormat =
&RSJobItemList.GetRow(&I).PRCSRQSTDLGLIST.OUTDESTFORMAT.Value;

    &RQST.OutDest = &RSJobItemList.GetRow(&I).PRCSRQSTDLGLIST.OUTDEST.Value;

&RQST.Schedule();

If &RQST.Status = 0 then

    /* Schedule succeeded. */

Else

    /* Process (job) not scheduled, do error processing */

End-If;

End-For;

```

The following example submits two jobs, each with different output types and formats:

```

&RQST = CreateProcessRequest();

&RQST.JobName = "MYJOB";

&RQST.RunControlID = RUN_CNTL_ID;

&RQST.RunLocation = "Server";

&RQST.RunDateTime = %Datetime;

    &RQST.TimeZone = %ServerTimeZone;

    &RQST.ProcessType = "SQR Report";

    &RQST.ProcessName = "XRFWIN";

/* Use the process instance returned from the job header request*/

&RQST.JobInstance = &wsPRCSINSTANCE;

/* first job item in the list has job seq = 1 */

&RQST.JobSeq = 1;

```

```
&RQST.JobName = &wsJobName;

&RQST.OutDestType = "PRINTER";

&RQST.OutDestFormat = "SPF";

&RQST.OutDest = "\\PLE_PRINT_01\P12";

&RQST.Schedule();

If &RQST.Status = 0 then
    /* Schedule succeeded. */
Else
    /* Process (job) not scheduled, do error processing */
End-If;

&RQST = CreateProcessRequest();

&RQST.JobName = "MYJOB";

&RQST.RunControlID = RUN_CNTL_ID;

&RQST.RunLocation = "Server";

&RQST.RunDateTime = %Datetime;

&RQST.TimeZone = %ServerTimeZone;

&RQST.ProcessType = "Crystal";

&RQST.ProcessName = "XRFFIELDS";

/* Use the process instance returned from the job header request*/

&RQST.JobInstance = &wsPRCSINSTANCE;

/* Second job item in the list has job seq = 2 */

&RQST.JobSeq = 2;
```

```
&RQST.JobName = &wsJobName;

&RQST.OutDestType = "Email";
&RQST.OutDestFormat = "PDF";
&RQST.OutDest = "name@company.com";
&RQST.Schedule();

If &RQST.Status = 0 then
    /* Schedule succeeded. */
Else
    /* Process (job) not scheduled, do error processing */
End-If;
```

Status

This property returns a number based on the result of the last execution of the Schedule method.

Valid returns are:

- Zero if the method succeeded
- Non-zero otherwise

This property is read-only.

Example

```
&MYRQST.Schedule();

If &MYRQST.Status = 0 then
    /* Schedule succeeded. */
Else
    /* Process (job) not scheduled, do error processing */
End-If;
```

TimeZone

This property contains a timezone value that specifies when the scheduled process or job will run. If no value is used for this property, the server timezone is used. This property takes a string value.

This property is read-write.

Example

```
&MyRqst.TimeZone = "EST";
```

Query Classes

You create queries to extract the data you need from your PeopleSoft database. You can use the Query classes in your PeopleCode to create a new query, or to modify or delete an existing query. You might also use the Query classes to create a SQL statement to be used with the SQL object. Your application should instantiate the appropriate query objects when it needs to work with queries, call the appropriate methods and properties, then close the objects when it is finished.

You can *not* run a query using the Query API. You must access PeopleTools Query in order to run a query.

Creating or deleting a query **object** does not create or delete query information. You must call the method for that query object directly to create or delete database information, that is, the **Create** or **Delete** method.

All of the classes, and most of the properties and methods that make up the Query Classes have a GUI representation in PeopleSoft Query. This document assumes that the reader has working knowledge of PeopleSoft Query.



For more information, see PeopleSoft Query.

With most of the classes of objects in PeopleTools, when you use a **Getxxx** method, you are fully instantiating an object. However, for the Query class, when you use **GetQuery** (from the session object), you get an unpopulated query, just the structure of a generic query, with no data. To open an existing query, you must use the **Open** method.

There aren't any built-in functions for the Query classes: objects are instantiated from other objects or from a session object.



For more information, see Session Class.

Query Classes Overview

The Query API is made up of many classes. The following are the primary parts, generally used when updating a query:

Query	The query definition.
QueryRecords	The records that are part of the existing query definition.
QueryOutputFields	The fields that you've selected to be part of the query definition.
QuerySelectedFields	The fields that make up the QueryRecord. They may or may not be part of the actual query definition.
QueryCriteria	The criteria for the query.
QueryDBRecords and QueryDBRecordFields	All the records available to be used as QueryRecords, as well as all the fields available.

Query

A database query. You must get a query and before you can access any of the other Query classes. You can then create a new query and save it to the database, or access an existing query and modify it.

QueryRecords

These are the records that are part of an existing query definition. In the PeopleSoft Query, these are the records listed on the query tab. The following example shows two QueryRecords, MAKE_ANL_DB_BU and BUS_UNIT_TBL_FS.

Record Query Fields Criteria SQL Properties

Query Name: DW_BUSINESS_UNITS Description: List of Business Units

Chosen Records Find First 1-2 of 2 Last

Record (Table) Name	Description	Any join	Add fields	
MAKE_ANL_DB_BU				
Check Fields to Add				
<input type="checkbox"/> OPRID			Use as criteria	
<input type="checkbox"/> RUN_CNTL_ID			Use as criteria	
<input type="checkbox"/> BUSINESS_UNIT			Use as criteria	
BUS_UNIT_TBL_FS	PS/Financials Business Units			
Check Fields to Add				
<input type="checkbox"/> BUSINESS_UNIT			Use as criteria	
<input type="checkbox"/> DESCR			Use as criteria	
<input type="checkbox"/> DESCRSHORT			Use as criteria	

QueryRecords as shown in PeopleSoft Query

QueryOutputFields

These are the fields that you've selected to be part of the query definition. They're called output fields because when you run the query, these are the fields that make up the output columns.

In PeopleSoft Query, these are the fields listed on the Fields tab.

Record Query Fields Criteria SQL Properties

Select Fields Find First 1-2 of 2 Last

Col.:	Field Name:	Heading:		
1	EMPLID	A.EMPLID	Show properties	Use as criteria
2	ABSENCE_TYPE	A.ABSENCE_TYPE	Show properties	Use as criteria

QueryOutputFields as shown in PeopleSoft Query

QuerySelectedFields

These are the fields that make up the QueryRecord. They may or may not be part of the actual query definition.

In PeopleSoft Query, these are the fields under the Query tab:

Record (Table) Name:	Description:
ABSENCE_HIST	EE Absence History

[Any join](#) [Add fields](#)

Check Fields to Add

<input checked="" type="checkbox"/> EMPLID	Use as criteria
<input checked="" type="checkbox"/> ABSENCE_TYPE	Use as criteria
<input type="checkbox"/> BEGIN_DT	Use as criteria
<input type="checkbox"/> RETURN_DT	Use as criteria
<input type="checkbox"/> DURATION_DAYS	Use as criteria
<input type="checkbox"/> DURATION_HOURS	Use as criteria
<input type="checkbox"/> REASON	Use as criteria
<input type="checkbox"/> PAID_UNPAID	Use as criteria
<input type="checkbox"/> EMPLOYER_APPROVED	Use as criteria
<input type="checkbox"/> COMMENTS	Use as criteria

QuerySelectedFields as shown in PeopleSoft Query

QueryCriteria

The selection criteria for the query. Each QueryCriteria object is made up of the following:

- Logical** Any criteria objects after the first must include have a Logical value, either AND or OR.
- Expression 1** A field or value you want to base the selection criterion on.
- Operator** A mathematical or other operator used to specify the relationship between Expression 1 and Expression 2.
- Expression 2** A field or other value, also called a comparison value, used with Expression 1.

In PeopleSoft Query, a QueryCriteria is under the Criteria tab. The following shows two criteria.

Query Name: DW_BUSINESS_UNITS **Description:** List of Business Units

Criteria Find First 1-2 of 2 Last

Logical:	Expression 1:	Condition Type:	Expression 2:
[]	ARUN CNTL ID	equal to	Not supported for edit.
And	ABUSINESS_UNIT	equal to	BUSINESS_UNIT

QueryCriteria as shown in PeopleSoft Query

Expression can be made up of constant values, fields, subqueries, and so on.



For more information about using expression in the Query API, see *Working with Criteria and Expressions*. For more information about using expressions in a query, see *PeopleSoft Query*.

QueryDBRecords and QueryDBRecordFields

A QueryDBRecord is a record in the database that can be used as a QueryRecord. The list of records is controlled by security: the only records displayed as QueryDBRecords are records accessible by the user.

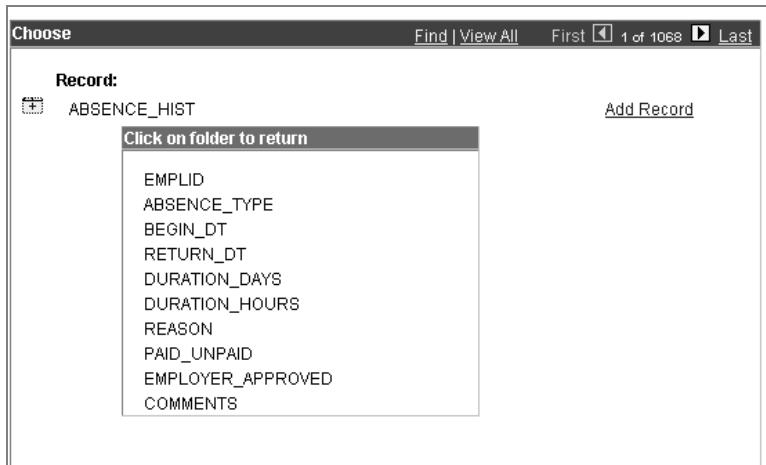
The QueryDBRecordFields are the fields that make up the QueryDBRecords.

In PeopleSoft Query, the QueryDBRecords are under the Record tab.

Record		
Query		
Fields		
Criteria		
SQL		
Properties		
Choose		
Find View All		
First 1-15 of 1088 Last		
Record:	Description:	
ABSENCE_HIST		Add Record
ACCDNT_TYPE_TBL		Add Record
ACCESS_GRP_TBL		Add Record
ACCOMPLISHMENTS		Add Record
ACCOMP_TBL		Add Record
ACTN_REASON_TBL		Add Record
ACTN_RSN_LANG		Add Record
AEINSTANCENBR		Add Record
AELOCKMGR		Add Record
AEONLINEINST		Add Record
AEREQUESTPARM		Add Record
AEREQUESTTBL		Add Record
AERUNCONTROL		Add Record
AERUNCONTROLPC		Add Record

QueryDBRecords as shown in PeopleSoft Query

In PeopleSoft Query, after you click on the plus sign next to a record, the QueryDBRecordFields are displayed:



QueryDBFields as shown in PeopleSoft Query

Understanding QueryOutputFields and QuerySelectedFields

A QuerySelectedField is a field that's part of a QueryRecord. If you select a QuerySelectedField, it becomes a QueryOutputField, that is, it becomes one of the columns in the SQL output.

QuerySelectedFields and QueryOutputFields have all the same methods and properties, so in this documentation, they're described together under the heading, QueryField.

Collections in the Query Classes

A *collection* is a set of similar things, like a group of already existing queries or QueryRecords. Like everything else in the query classes, collections have a GUI representation in PeopleSoft Query.

For example, when you select want to open a query, the search page returns a list of all the available queries. This is equivalent to using the FindQueries session class method to get a Query collection.

Open		Find View All	First 1 1-15 of 111 Last
	Query	Description	
Private	<u>UNTITLED</u>		
Public	<u>ADDRESSLIST</u>	AddressList	
Public	<u>CM_ATTRIBUTES</u>	Attribute mappings	
Public	<u>CM_DIM_CTRL_TBL</u>	Dimension Control Table	
Public	<u>CM_FACT_CTRL_TBL</u>	Fact Control Table	
Public	<u>CM_FACT_MAP_TBL</u>	Fact Map Table	
Public	<u>CM_FIELD_PROPERTIES</u>		
Public	<u>CM_HIER_CTRL_TBL</u>	Hierarchy Control table	
Public	<u>CM_HIER_MAP_TBL</u>	Hierarchy Map Table	
Public	<u>DW_AGGREGATION</u>	Aggregation for the instance	
Public	<u>DW_AGG_LEVEL</u>	Level Mappings	
Public	<u>DW_BUSINESS_UNITS</u>	List of Business Units	
Public	<u>DW_CUBE</u>	Cube element list	
Public	<u>DW_CUBE_DATA</u>	Check of data query existence	
Public	<u>DW_CUBE_DIM</u>	Cube dimension list	

Query collection

The following collections are part of the Query classes:

- QueryDBRecord collection
- QueryDBRecordField collection
- Query collection
- QueryRecord collection
- QueryOutputField collection
- QuerySelectedField collection
- QueryCriteria collection
- QueryExpression collection

Life-Cycle of a Query

At runtime, there are certain things you want to do with a query, like creating one from scratch, updating the criteria for an existing query, running a query, and so on. The following is an overview of this process, and assumes the most usual case for how someone is going to use the Query API. These steps are expanded in other sections.

12. Execute the GetQuery method on the PeopleSoft session object to get a query.
13. Either open the specific query you want using Open, or create a new query using Create.
14. Make changes as appropriate, adding records, field, criteria, and so on.

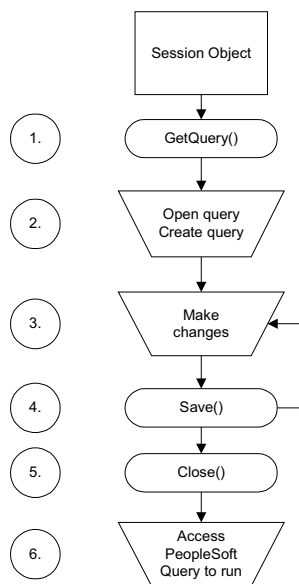
15. Save the changes.

16. Close the query.



PeopleSoft recommends that you open and close every Query in a single PeopleCode event. You shouldn't open a Query object and keep it open across multiple events.

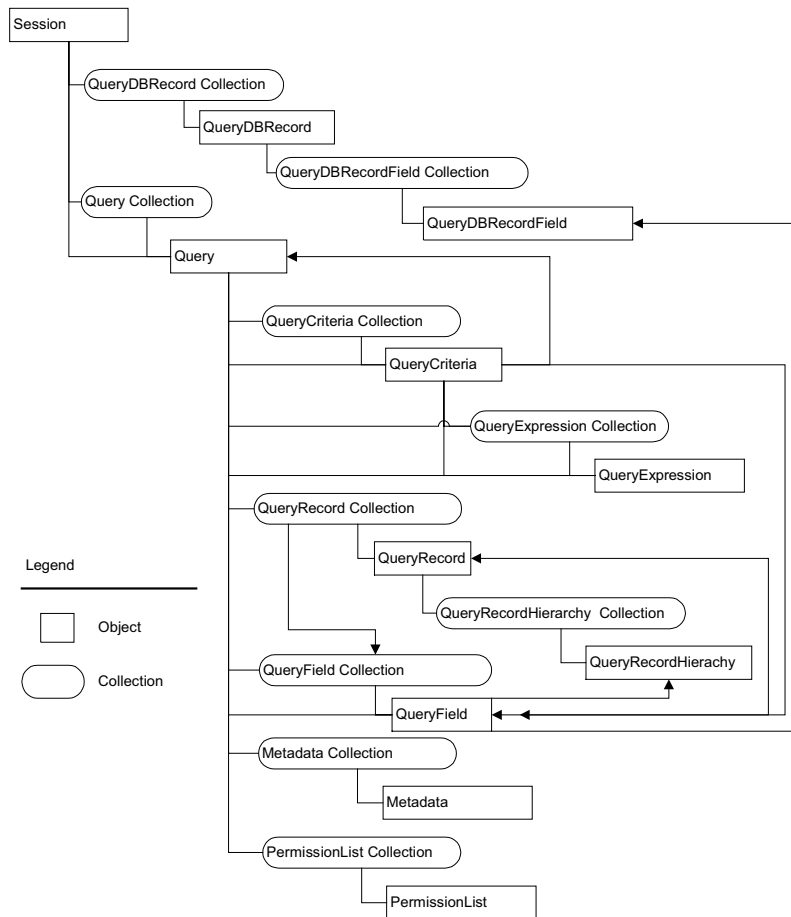
17. Navigate to PeopleSoft Query to run the query.



Life-Cycle of Query API

Query Class Hierarchy

There are many different classes used with the Query API. The following flowchart shows all the different classes and how they're interrelated.



Query API class hierarchy

Working with Criteria and Expressions

If you run a query after selecting the QueryFields, the system retrieves *all* the data in those columns; that is, it retrieves the data from every row in the QueryRecord or records.

You can select which *rows* of data you want by adding selection criteria to the query.

This document assumes that you know how to use selection criteria. This section only discusses working with the QueryCriteria and QueryExpression objects in the Query API.



For more information about using criteria in general, see [Specifying Query Selection Criteria](#).

Setting the Type

Before you can add a new expression (either Expression 1 or Expression 2) to your QueryCriteria, you must set the type for the expression.

The following code example adds a new criteria, sets the type for the first expression, adds the first expression, then does the same thing for the second expression.

```
&MyCriteria = &MyQuery.AddCriteria();

/* make expression 1 a field */

&MyCriteria.Expr1Type = 1;

/* add the ABSENCE_TYPE field */

&MyCriteria.AddExpr1Field(ABSENCE_HIST, 2);

/* make it not equal to */

&MyCriteria.Operator = 3;

/* Make expression 2 a constant */

&MyCriteria.Expr2Type = 1;

&MyCriteria.AddExpr2Constant("VAC");
```

Considerations Adding New Expressions

When you use any of the QueryCriteria methods to add either a new Expression 1 or Expression 2, you destroy the existing value.

In general, you should only use the QueryCriteria methods to add a new Expression 1 or Expression 2 when you're adding a new criteria to a query.

Operator and Expression 2 Dependencies

Which values are valid for the Expression 2 type (Expr2Type property) are dependant on the value of the Operator property.

The following table describes which Expr2Type values are valid with which values of Operator.

<i>Operator</i>	<i>Expression 2</i>
equal to	Constant

not equal to	Field
greater than	Expression
not greater than	Subquery
less than	
not less than	
Exists	Subquery
not exists	
Like	Constant (with wild cards)
not like	
is null	
is not null	
in tree	Tree Option
not in tree	
Eff Date <=	Field
Eff Date >=	Expression
Eff Date <	Constant
EffDate >	Current Date
First Eff Date	
Last Eff Date	

Using Metadata

There are actually two ways of accessing information about a query. You can use the Query classes (QueryOutputField, QueryRecord, and so on). Or you can use the Metadata property. If you use the Metadata property, the information is presented in a different format. It is also read-only. But using this property can be a quick and easy way to access information about a query.

Each metadata object is a name-value pair. *Name* is an indicator of which metadata property you're accessing, and *Value* is the value of that property.

While the value of each metadata property is unique, the name may or may not be. For example, there is only one description (Descr) for each query, so there is only one metadata property with *name* equal to Descr and *value* equal to the description for the query.

However, there may be more than one record for a query. A metadata property will exist for each record, with *name* equal to Record and *value* equal to one of the records in the query. The same goes for Input Param, Expression, Field and Heading.

The following is a simple PeopleCode program to get the metadata for the ADDRESSLIST query:

```

Local ApiObject &MyQuery, &MyMetacol, &MyMetadata;

Local File &MyFile;

&MyQuery = %Session.GetQuery();

&MyFile = GetFile("Metadata.Txt", "A");

&Rlst = &MyQuery.Open("ADDRESSLIST", True, 1);

&MyMetacol = &MyQuery.metadata;

&MyFile.WriteLine("Name      Value");

For &i = 1 To &MyMetacol.count
    &MyMetadata = &MyMetacol.item(&i);

    &Name = &MyMetadata.name;

    &Value = &MyMetadata.Value;

    &MyFile.WriteLine(&Name | "    " | &Value);
End-For;

```

Here is the sample output:

```

Name      Value

Descr    AddressList

LongDescr

Public/Private

LastUpdDttm  1998-11-24-19.43.12.640000

LastUpdOprId  PPLSOFT

Record  PERSONAL_DATA

Field  EMPLID

Field  NAME

Field  ADDRESS1

```

```
Field  CITY

Field  STATE

Field  ZIP

Field  COUNTRY

Heading  ID

Heading  Name

Heading  Address 1

Heading  City

Heading  St

Heading  Zip

Heading  Cntry

Input Param  EMPLID
```

Running a Query

You can *not* run a query using the Query classes. You must access PeopleTools Query to run all queries. You can create a new query, or change or delete an existing query. You must save your changes to the query object before you can run the query.

Query Security

Security is critical for your business data. Typically, you don't want everyone in your company to have access to all your data. All the standard security features used with PeopleSoft applications are integrated in the PeopleSoft Query, as well as the Query classes.



For more information about security, see Setting up Query Security.

Use the PermissionList property to see which permission lists the have access to the query object.



For more information about permission lists, see Security.

Error Handling with Query Classes

All errors for the Query classes, like the other APIs, are logged in the PSMessages collection, instantiated from a session object.



For more information see About Error Handling.

The query classes log errors "interactively", that is, as they happen. For example, suppose you specified an invalid query name. The error would be logged in the PSMessages collection as soon as you executed the GetQuery method.

It depends on your application when you want to check for errors. However, if you check for errors after every assignment, you may see a performance degradation.

The easiest way to check for errors is to check the number of messages in the PSMessages collection, using the **Count** property. If the Count is 0, there are no errors.

```

Local ApiObject &MySession;

Local ApiObject &ERRORCOL;

Local ApiObject &Query, &QueryList;

&MySession = %Session;

If &MySession Then

    /* connection is good */

    &QueryList = &MySession.FindQueries();

    For &I = 1 to &QueryList.Count

        &Query = &QueryList.Item(&I);

        /* Do processing */

        /* Do error checking */

```

```

&ERRORCOL = &MySession.PSMessages;

If (&ERRORCOL.Count <> 0) Then

    /* errors occurred - do processing */

Else

    /* no errors */

End-If;

End-For;

Else

    /* do processing for no connection */

End-If;

```

Declaring Query Objects

All query objects, like a query collection, a QueryDBRecord, a QueryExpression, and so on, are declared as type ApiObject. For example,

```

Local ApiObject &MyQuery;

Local ApiObject &MyExpression;

```



All Query objects can only be declared as Local.

Scope of the Query Objects

All query objects can only be instantiated from PeopleCode.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Application Engine PeopleCode, record field PeopleCode, and so on.

You can only instantiate a query object from a session object. You **must** instantiate the session object before you can instantiate a query object or query collection. You can either use the GetSession and Connect method from that, or use the %Session system variable if you want to connect to the existing session.

```

Local ApiObject &QueryList;

Local ApiObject &MySession;

```

```
Local boolean &RSLT;

&MySession = GetSession();

&RSLT = &MySession.CONNECT(1, "EXISTING", "", "", 0);

/* OR use &MySession = %Session; If &MySession Then */

If &RSLT Then

    /* connection is good */

    &QueryList = &MySession.FindQueries();

Else

    /* do error processing */

End-if;
```

Session Object Methods

FindQueryDBRecords

Syntax

```
FindQueryDBRecords()
```

Description

The **FindQueryDBRecords** method returns a reference to a QueryDBRecord collection, filled with zero or more records.

Security applies to the results of this list, that is, you have access to all the records your UserID (permission list) allows you to access.

Parameters

None.

Returns

A reference to a QueryDBRecord collection containing 0 or more queries.

Related Topics

QueryDBRecord Collection, QueryDBRecord Class

FindQueries

Syntax

```
FindQueries()
```

Description

The **FindQueries** method returns a reference to a Query collection, filled with zero or more queries.

FindQueries Considerations

FindQueries returns both public and private queries. If you have two queries with the same name, and one is public while the other is private, the first use of the **Item** method returns the private query. The second **Item** call returns the public query.

Parameters

None.

Returns

A reference to a Query collection containing 0 or more queries.

Example

In the following example, all available queries are returned:

```
Local ApiObject &MySession;  
  
Local ApiObject &MyList;  
  
    &MySession = GetSession();  
  
    &MySession.Connect(1, "EXISTING", "", "", 0);  
  
    &MyList = &MySession.FindQueries();
```

Related Topics

FindQueriesByDateRange, Query Collection, Open method

FindQueriesByDateRange

Syntax

```
FindQueriesByDateRange(StartDateString, EndDateString)
```

Description

The **FindQueriesByDateRange** method returns a reference to a Query collection, filled with zero or more queries that match the specified date range.

FindQueriesByDateRange Considerations

FindQueriesByDateRange returns both public and private queries. If you have two queries with the same name, and one is public while the other is private, the first use of the **Item** method returns the private query. The second **Item** call returns the public query.

Parameters

<i>StartDateString</i>	Specify the year, month and day of the beginning date you want to look for. This parameter takes a string value. You can specify the date either as YYYY-MM-DD or YYYY/MM/DD.
<i>EndDateString</i>	Specify the year, month and day of the end date. This parameter takes a string value. You can specify the date either as YYYY-MM-DD or YYYY/MM/DD.

Returns

A reference to a Query collection containing 0 or more queries.

Example

```
Local ApiObject &MySession, &QueryList;

&MySession = %Session;

&Start = GetField(VOLUN_ACT_WRK.START_DT_STR).Value;
&End = GetField(VOLUN_ACT_WRK.END_DT_STR).Value;

&QueryList = &MySession.FindQueriesByDateRange(&Start, &End);
```

Related Topics

FindQueries, Query Collection, Open method

GetQuery

Syntax

```
GetQuery()
```

Description

The **GetQuery** method returns an empty query object. Once you have an empty query object, you can use it to open an existing query (using the Open method) or to create a new query definition (using the Create method).

Parameters

None.

Returns

A reference to an empty query object if successful, NULL otherwise.

Example

```
&MyQuery = &MySession.GetQuery();  
  
If &MyQuery.Open("PHONELIST") Then
```

Related Topics

FindQueries, FindQueriesByDateRange, Open, Create

Query Collection

A query collection is returned from the FindQueries and FindQueriesByDateRange session methods.

Query Collection Methods

First

Syntax

```
First()
```

Description

The **First** method returns the first Query object in the Query collection.

Parameters

None.

Returns

A reference to a Query object if successful, NULL otherwise.

Example

```
&MyQuery = &MyCollection.First();
```

Item

Syntax

```
Item(number)
```

Description

The **Item** method returns the Query object that exists at the *number* position in the Query collection.

Item Considerations

FindQueries returns both public and private queries. If you have two queries with the same name, and one is public while the other is private, the first use of the **Item** method returns the private query. The second **Item** call returns the public query.

Parameters

<i>Number</i>	Specify the position number in the collection of the Query object that you want returned.
---------------	---

Returns

A reference to a Query object if successful, NULL otherwise.

Example

```
For &I = 1 to &QueryColl.Count;  
  
    &MyQuery = &QueryColl.Item(&I);  
  
    /* do processing */  
  
End-For;
```

ItemByName

Syntax

```
ItemByName(Name)
```

Description

The **ItemByName** method returns the Query object with the name *Name*.

Parameters

Name Specify the name of an existing Query within the Query collection. If you specify an invalid name, the object will be NULL.

Returns

A reference to a Query object if successful, NULL otherwise.

Example

```
&MyQuery = &MyCollection.ItemByName("PHONELIST");
```

Next

Syntax

```
Next()
```

Description

The **Next** method returns the next Query object in the Query collection. You can only use this method after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A reference to a Query object if successful, NULL otherwise.

Example

```
&MyQuery = &MyCollection.Next();
```

Query Collection Property

Count

This property returns the number of Query objects in the Query Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

Query Class

A query object is returned from the following:

- The GetQuery session method
- The Query Collection methods First, Item, ItemByName or Next.

Query Class Methods

AddAllFields

Syntax

```
AddAllFields (QueryRecord)
```

Description

The **AddAllFields** method add all the fields in the specified query record to the query definition, that is, makes them all QuerySelectedFields. The query record must already be a part of the query definition.

Parameters

<i>QueryRecord</i>	Specify the name of an existing record in the query that you want to add all the fields from to the query definition.
--------------------	---

Returns

An integer: 0 if successfully added.

Related Topics

AddQueryOutputField, AddQueryRecord, AddQuerySelectedField

AddCriteria

Syntax

```
AddCriteria (Name)
```

Description

The **AddCriteria** method adds new criterion to the query definition. This method returns a reference to a new QueryCriteria object that you can then use to specify details about the criteria.

Parameters

Name This parameter is required, but is unused in this release. You must specify a NULL string (that is, two quotation marks with no space between them ("")).

Returns

A reference to a QueryCriteria object.

Related Topics

Criteria property, QueryCriteria Class, DeleteCriteria

AddExpression

Syntax

```
AddExpression (Name)
```

Description

The **AddExpression** method adds an expression to the query definition. It returns a reference to a new QueryExpression object you can use to specify details about the expression.

Parameters

Name This parameter is required, but is unused in this release. You must specify a NULL string (that is, two quotation marks with no space between them ("")).

Returns

A reference to a QueryExpression object.

Related Topics

QueryExpression Class, Expressions property, DeleteExpression

AddQueryOutputField

Syntax

```
AddQueryOutputField(QueryRecord, index)
```

Description

The **AddQueryOutputField** method adds a query output field to the query definition. This method returns a reference to a new QueryOutputField object that you can then use to specify attributes for of the query definition.

Parameters

QueryRecord

Specify the QueryRecord that contains the field you want to include in the query definition.

Index

Specify the numeric position in the QueryRecord of the field you want to add.

Returns

A reference to a QueryOutputField object.

Related Topics

AddAllFields, QueryField Class

AddQueryRecord

Syntax

```
AddQueryRecord (QueryRecordName)
```

Description

The **AddQueryRecord** method add a query record to the query definition. You must specify a valid record name in *QueryRecordName*. This method returns a reference to the record as a QueryRecord.

Parameters

QueryRecordName

Specify the name of a record to be added to the query definition. You must specify a valid record name.

Returns

A reference to the record as a QueryRecord object.

Related Topics

QueryRecord Class

AddQuerySelectedField

Syntax

```
AddQuerySelectedField (QueryRecord, index)
```

Description

The **AddQuerySelectedField** method adds a query field to the query definition, as a QuerySelectedField. This method returns a reference to a new QuerySelectedField object that you can then use to specify attributes for of the query definition.

Parameters

<i>QueryRecord</i>	Specify the QueryRecord that contains the field you want to include in the query definition.
<i>Index</i>	Specify the numeric position in the QueryRecord of the field you want to add.

Returns

A reference to a QuerySelectedField object.

Related Topics

DeleteField, AddAllFields, QueryField Class

Close

Syntax

```
Close()
```

Description

The **Close** method closes the query, freeing the memory associated with that object, and discarding any changes made to the query since the last save. The **Close** method can only be used on an open query, not a closed query. This means you must have opened the query with the Open or Create methods before you can close it. If you want to save any changes, you must use the Save method before using **Close**.

It's very important to close your query when you're finished processing. Canceling out of a page does *not* close a query. You may receive error messages every other time you run your program if you haven't closed your queries.

Parameters

None.

Returns

None.

Related Topics

Open, Save

Create

Syntax

Create(*queryname*, *Public*, *Distinct*, *Type*, *Description*, *LongDescription*)

Description

The **Create** method creates a new query, based on the parameters passed with the method. The specified must be a *new* query.



If you specify the name of a query that already exists, the existing query will be overwritten by the new query.

The **Create** method can only be used with a *closed* query, it cannot be used on an open query. Before you use the **Create** method, you must explicitly close any open query objects (with the Close method.) You will receive an error if there are any open queries.

After you create a new query, you don't have to open it with the Open method. The existing query object points to the new query.

Creating a new query doesn't create the query in the database. You must save the query (with the Save method) to commit it to the database.

Parameters

<i>Queryname</i>	Name of the query to be created. This parameter takes a string value. This parameter takes 30 characters.
<i>Public</i>	Specify if the query is public or private. This parameter takes a Boolean value: True if the query is public, False if it's a private query.
<i>Distinct</i>	Specify if the query is distinct or not. This parameter takes a Boolean value: True if the query is distinct, False if the query is not distinct (Default).
<i>Type</i>	Specify the type of query. This parameter takes a number value. Valid values are:

Value	Description
1	Query
2	Search
3	View
4	Role

Value	Description
5	DBAgent
6	Acitvequery

Description

Specify a short description for the query. This parameter takes a string value. This parameter takes 30 characters.

LongDescription

Specify a long description for the query. This parameter takes a string value. This parameter takes 256 characters.

Returns

An integer value: 0 means the query was created successfully.

Example

```

/* Use the existing session */

&MySession = %Session;

&MyQry = &MySession.GetQuery();

/* create a new query : public, no-distinct, type-User */
&Rslt = &MyQry.Create("MYQUERY", True, False, 1, "My Query", "My first Query");

If &Rslt = 0 Then
    /* Query created successfully */
Else
    /* do error processing */
End-If;

```

Related Topics

GetQuery, Close, Save

Delete

Syntax

```
Delete()
```

Description

The **Delete** method deletes the specified query *from the database*. The **Delete** method can only be used with an *open* query, it cannot be used on a closed query. Before you use the **Delete** method, you must explicitly open the query object to be deleted (with either the Open or Create method.)

Parameters

None.

Returns

An integer value: 0 means the query was deleted successfully.

Related Topics

Save

DeleteCriteria

Syntax

```
DeleteCriteria(index)
```

Description

The **DeleteCriteria** method deletes the criteria specified by *index* from the query definition.

Parameters

<i>Index</i>	Specify the numeric position in the query definition of the criteria you want to delete.
--------------	--

Returns

An integer value: 0 means the query was deleted successfully.

Related Topics

AddCriteria, Criteria property

DeleteExpression

Syntax

```
DeleteExpression(index)
```

Description

The **DeleteExpression** method deletes the specified expression from the query definition.

Parameters

<i>Index</i>	Specify the numeric position of the expression in the query definition that you want to delete.
--------------	---

Returns

An integer: 0 if successfully deleted.

Related Topics

QueryExpression Class, Expressions property, AddExpression

DeleteField

Syntax

```
DeleteField(index)
```

Description

The **DeleteField** method deletes the query selected field from the query definition. This does **not** delete the field from the QueryRecord, just from the query definition, so it's no longer selected.

Parameters

<i>Index</i>	Specify the position number of the field you want deleted from the query definition.
--------------	--

Returns

An integer value: 0 means the query was deleted successfully.

Related Topics

AddAllFields, QueryField Class

Open

Syntax

```
Open (QueryName, Public, Update)
```

Description

The **Open** method opens the query object specified by the parameters. The **Open** method can only be used with a **closed** query, it cannot be used on an open query. You cannot read or set any properties of a query until after you open it.

Parameters

<i>QueryName</i>	Specify the name of the query to be opened. You must specify an existing query. This parameter takes a string value.
<i>Public</i>	Specify if the query is public or private. This parameter takes a Boolean value: True if the query is public, False if it's a private query.
<i>Update</i>	This parameter is required, but is unused in this release. You must specify a either True or False.

Returns

An integer value: 0 means the query was opened successfully.

Related Topics

GetQuery, Close, Save

QueryOutputFields

Syntax

```
QueryOutputFields ()
```

Description

The **QueryOutputFields** method returns a reference to a QueryField Collection made up of QueryOutputFields.

Parameters

None.

Returns

QueryField collection.

Related Topics

QuerySelectedFields

QueryRecords

Syntax

```
QueryRecords()
```

Description

The **QueryRecords** method returns a reference to a QueryRecord Collection.

Parameters

None.

Returns

A QueryRecord collection.

QuerySelectedFields

Syntax

```
QuerySelectedFields()
```

Description

The **QuerySelectedFields** method returns a reference to a QueryField Collection made up of QuerySelectedFields.

Parameters

None.

Returns

QueryField collection.

Related Topics

QueryOutputFields

Rename

Syntax

```
Rename(NewQueryName)
```

Description

The **Rename** method renames the existing query definition with *NewQueryName*. The **Rename** method can only be used with an *open* query, it cannot be used on a closed query. Before you use the **Rename** method, you must explicitly open the query object to be renamed (with either the **Open** or **Create** method.)

The **Rename** won't actually be committed to the database until after you use the **Save** method.

Parameters

NewQueryName Specify the new name for the existing query.

Returns

An integer value: 0 means the query was renamed successfully.

Related Topics

Open, Close, Save

Save

Syntax

```
Save ()
```

Description

The **Save** method writes any changes to the existing query to the database.

The **Save** method can only be used on an open query, not on a closed query. This means you must have opened the query with the **Open** method before you can save it.

The query object remains open after executing **Save**. You must execute the **Close** method on the object before it will be closed and the memory freed.



If you're calling the Query API from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

Parameters

None.

Returns

An integer value: 0 means the query was saved successfully.

Related Topics

Open, Close

Query Class Properties

Criteria

This property returns a reference to a QueryCriteria Collection.

This property is read-only.

Description

This property returns or sets the short description for the query.

The length of this property is 30 characters.

This property is read-write.

Distinct

This property specifies if the query is distinct or not.

This property takes a Boolean value: True if the query is distinct, False if the query isn't distinct.

This property is read-write.

Expressions

This property returns a reference to a QueryExpression Collection.

This property is read-only.

LongDescription

This property returns or sets the long description for the query.

The length of this property is 256 characters.

This property is read-write.

Metadata

This property returns a Metadata Collection.

This property is read-only.

Example

```
&MetadataList = &MyQuery.Metadata;
```

Name

This property returns the name of the query as a string.

The length of this property is 30 characters.

This property is read-only.

PermissionList

This property returns a PermissionList Collection.

This property is read-only.

Example

```
&PermissionList= &MyQuery.PermissionList;
```

Public

This property specifies whether the query is public or private.

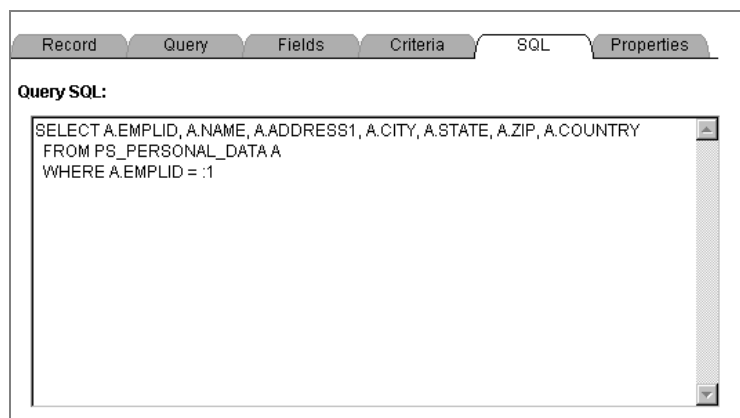
This property takes a Boolean value: True if the query is public, False if the query is private.

This property is read-write.

SQL

This property returns the SQL statement used with the query as a character string.

In the PeopleSoft Query, this is located under the SQL tab.



Query SQL as shown in PeopleSoft Query

This property is read-only.

Type

This property specifies the type of query. This parameter takes a number value. Valid values are:

<i>Value</i>	<i>Description</i>
1	Query
2	Search
3	View
4	Role
5	DBAgent
6	Acitvequery

This property is read-write.

QueryRecord Collection

A QueryRecord collection is returned from the QueryRecords Query class methods.

QueryRecord Collection Methods

First

Syntax

```
First()
```

Description

The **First** method returns the first QueryRecord object in the QueryRecord collection.

Parameters

None.

Returns

A reference to a QueryRecord object if successful, NULL otherwise.

Example

```
&MyQueryRecord = &MyCollection.First();
```

Item

Syntax

Item (*number*)

Description

The **Item** method returns the QueryRecord object that exists at the *number* position in the QueryRecord collection.

Parameters

<i>Number</i>	Specify the position number in the collection of the QueryRecord object that you want returned.
---------------	---

Returns

A reference to a QueryRecord object if successful, NULL otherwise.

Example

```
For &I = 1 to &QueryRecordColl.Count;  
  
    &MyQueryRecord = &QueryRecordColl.Item(&I);  
  
    /* do processing */  
  
End-For;
```

ItemByName

Syntax

ItemByName (*Name*)

Description

The **ItemByName** method returns the QueryRecord object with the name *Name*.

Parameters

<i>Name</i>	Specify the name of an existing QueryRecord within the QueryRecord collection. If you specify an invalid name, the object will be NULL.
-------------	---

Returns

A reference to a QueryRecord object if successful, NULL otherwise.

Example

```
&MyQueryRecord = &MyCollection.ItemByName("PHONELIST");
```

Next

Syntax

```
Next ()
```

Description

The **Next** method returns the next QueryRecord object in the QueryRecord collection. You can only use this method after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A reference to a QueryRecord object if successful, NULL otherwise.

Example

```
&MyQueryRecord = &MyCollection.Next();
```

QueryRecord Collection Property

Count

This property returns the number of QueryRecord objects in the QueryRecord Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

QueryRecord Class

A QueryRecord object is returned from the following:

- the QueryRecord Collection methods First, Item, ItemByName or Next
- the AddQueryRecord Query class method

- the QueryRecord QueryField class method

QueryRecord Class Properties

Alias

This property returns the alias used for the record in the query (that is, A, B, C, and so on.)

This property is read-only.

Name

This property returns the name of the record as a string.

This property is read-only.

QueryFields

This property returns a reference to a QueryField Collection that contains QuerySelectedFields.

This property is read-only.

RecordHierarchy

This property returns a reference to a QueryRecordHierarchy Collection.

The record hierarchy is not related to the query tree hierarchy shown when viewing access groups. Instead, it reflects an actual relationship between the record components, as defined in Application Designer using the Parent Record Name feature.



For more information, see Setting up Query Security.

This property is read-only.

QueryField Collection

A QueryField collection is returned from the following:

- The QueryOutputFields Query class method
- The QuerySelectedFields Query class method
- The QueryFields QueryRecord class method

QueryField Collection Methods

First

Syntax

```
First()
```

Description

The **First** method returns the first QueryField object in the QueryField collection.

Parameters

None.

Returns

A reference to a QueryField object if successful, NULL otherwise.

Example

```
&MyQueryField = &MyCollection.First();
```

Item

Syntax

```
Item(number)
```

Description

The **Item** method returns the QueryField object that exists at the *number* position in the QueryField collection.

Parameters

<i>Number</i>	Specify the position number in the collection of the QueryField object that you want returned.
---------------	--

Returns

A reference to a QueryField object if successful, NULL otherwise.

Example

```
For &I = 1 to &QueryFieldColl.Count;  
  
    &MyQueryField = &QueryFieldColl.Item(&I);  
  
    /* do processing */
```

```
End-For;
```

ItemByName

Syntax

```
ItemByName (Name)
```

Description

The **ItemByName** method returns the QueryField object with the name *Name*.

Parameters

<i>Name</i>	Specify the name of an existing QueryField within the QueryField collection. If you specify an invalid name, the object will be NULL.
-------------	---

Returns

A reference to a QueryField object if successful, NULL otherwise.

Example

```
&MyQueryField = &MyCollection.ItemByName("PHONELIST");
```

Next

Syntax

```
Next ()
```

Description

The **Next** method returns the next QueryField object in the QueryField collection. You can only use this method after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A reference to a QueryField object if successful, NULL otherwise.

Example

```
&MyQueryField = &MyCollection.Next();
```

QueryField Collection Property

Count

This property returns the number of QueryField objects in the QueryField Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

QueryField Class

A QueryField object is returned by the following:

- The AddQuerySelectedField Query class method
- The AddQueryOutputField Query class method
- The QueryField Collection methods First, Item, ItemByName or Next

QueryField Class Properties

Aggregate

This property returns or sets the aggregate for the query field.

This property takes a number value. The valid values are:

<i>Value</i>	<i>Description</i>
1	None
2	Sum
3	Count
4	Minimum
5	Maximum
6	Average

This property is read-write.

ColumnNumber

This property returns or sets the column number for the query field.

This property is read-write.

Format

This property returns the field format for the query field. This property takes a number value. Valid values are:

Value	Description
1	No format
2	Name
3	Phone Number North America
4	Zip/Postal Code North America
5	Social Security Number (SSN)
6	UpperCase
7	Mixed case
8	Raw binary
9	Numbers only
10	Canadian Social Insurance Number (SIN)
11	Phone Number International
12	Zip/Postal Code International
13	Seconds
14	Microseconds
15	Custom

This property is read-only.

HeadingText

This property returns or sets the heading text for the query field.

This property is read-write.

HeadingType

This property returns or sets the heading type for the query field. This property takes a number value. The valid values are:

Value	Description
1	None
2	Text
3	RFT Short
4	RFT Long

This property is read-write.

HeadingUniqueFieldName

This property returns or sets the unique field name for the heading.

This property is read-write.

Name

This property returns the name of the query field as a string.

This property is read-only.

OrderByDirection

This property returns or sets the order by direction. This property takes a number value. Valid values are:

Value	Description
1	Ascending
2	Descending

This property is read-write.

OrderByNumber

This property returns or sets the order by number for the query field.

This property is read-write.

QueryDBRecordField

This property returns a reference to the QueryDBRecordField associated with this QueryField.

Once you have the QueryDBRecordField, you can access the QueryDBRecordField attributes (such as Long name, Type, length, and so on.)

This property is read-only.

QueryRecord

This property returns a reference to the QueryRecord containing this QueryField.

This property is read-only.

QueryRelatedRecords

This property returns the prompt table for the record field as a QueryRecordHierarchy object.

This property is read-only.

TranslateEffDtLogic

This property returns or sets the effective date logic for the QueryField.

The valid values are:

<i>Value</i>	<i>Description</i>
1	Current Date
2	Field
3	Expression

This property is read-write.

TranslateOption

This property returns or sets the translate value for the QueryField. This property takes a number value. The valid values are:

<i>Value</i>	<i>Description</i>
1	None
2	Short
3	Long

This property is read-write.

Type

This property returns the type of the field. Valid values are:

<i>Value</i>	<i>Description</i>
1	Character
2	Long Character

Value	Description
3	Number
4	Signed number
5	Date
6	Time
7	Datetime
9	Image
10	Image Reference

This property is read-only.

QueryCriteria Collection

A QueryCriteria collection is returned from the Criteria Query class property.

QueryCriteria Collection Methods

First

Syntax

```
First()
```

Description

The **First** method returns the first QueryCriteria object in the QueryCriteria collection.

Parameters

None.

Returns

A reference to a QueryCriteria object if successful, NULL otherwise.

Example

```
&MyQueryCriteria = &MyCollection.First();
```

Item

Syntax

```
Item(number)
```


Description

The **Item** method returns the QueryCriteria object that exists at the *number* position in the QueryCriteria collection.

Parameters

<i>Number</i>	Specify the position number in the collection of the QueryCriteria object that you want returned.
---------------	---

Returns

A reference to a QueryCriteria object if successful, NULL otherwise.

Example

```
For &I = 1 to &QueryCriteriaColl.Count;  
  
    &MyQueryCriteria = &QueryCriteriaColl.Item(&I);  
  
    /* do processing */  
  
End-For;
```

Next

Syntax

Next ()

Description

The **Next** method returns the next QueryCriteria object in the QueryCriteria collection. You can only use this method after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A reference to a QueryCriteria object if successful, NULL otherwise.

Example

```
&MyQueryCriteria = &MyCollection.Next();
```

QueryCriteria Collection Property

Count

This property returns the number of QueryCriteria objects in the QueryCriteria Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

QueryCriteria Class

A QueryCriteria object is returned from the following:

- The AddCriteria Query class method
- The First, Item, and Next method of the QueryCriteria Collection.



For more information about QueryCriteria objects in the Query API, see [Working with Criteria and Expressions](#). For more information about query criteria in general, see [Specifying Query Selection Criteria](#).

QueryCriteria Class Methods

AddExpr1Expression

Syntax

```
AddExpr1Expression()
```

Description

The **AddExpr1Expression** method returns a reference to a new a QueryExpression object to be used as an expression for Expression 1. You can then use this object to specify details about the expression using the methods and properties of the QueryExpression Class.



Note. You must set the type for Expression 1 using the Expr1Type property before you can use this method.

Parameters

None.

Returns

A reference to a blank QueryExpression object.

Related Topics

Expr1Expression, AddExpr1Field, Expr1Type

AddExpr1Field

Syntax

```
AddExpr1Field(QueryRecord, index)
```

Description

The **AddExpr1Field** method returns a reference to a new a QueryField object to be used as a field for Expression 1. You can then use this object to specify details about the expression using the methods and properties of the QueryField Class.



Note. You must set the type for Expression 1 using the Expr1Type property before you can use this method.

Parameters

<i>QueryRecord</i>	Specify the QueryRecord that contains the QueryField you want to use.
<i>Index</i>	Specify the numeric location of the QueryField in the QueryRecord that you want to use.

Returns

A reference to a blank QueryField object.

Related Topics

Expr1Field, AddExpr1Expression, Expr1Type

AddExpr2Expression

Syntax

```
AddExpr2Expression()
```

Description

The **AddExpr2Expression** method returns a reference to a new a QueryExpression object to be used as an expression for Expression 2. You can then use this object to specify details about the expression using the methods and properties of the QueryExpression Class.

Parameters

None.

Returns

A reference to a blank QueryExpression object.

AddExpr2Field

Syntax

```
AddExpr2Field(QueryRecord, index)
```

Description

The **AddExpr2Field** method returns a reference to a new a QueryField object to be used as a field for Expression 2. You can then use this object to specify details about the expression using the methods and properties of the QueryField Class.

Parameters

<i>QueryRecord</i>	Specify the QueryRecord that contains the QueryField you want to use.
--------------------	---

<i>Index</i>	Specify the numeric location of the QueryField in the QueryRecord that you want to use.
--------------	---

Returns

A reference to a blank QueryField object.

Related Topics

QueryRecord Class, QueryField Class

AddExpr2Subquery

Syntax

```
AddExpr2Subquery()
```

Description

The **AddExpr2Subquery** method is used to create a subquery for Expression2. This method returns a new Query object that you can use to specify details about the new subquery.



The new subquery created with this method replaces any existing subquery (for this criteria), destroying any existing properties or values.

Parameters

None.

Returns

A reference to a new query object.

Related Topics

Query Class

QueryCriteria Class Properties

Expr1Expression

This property returns a reference to the QueryExpression object that's used as Expression 1.

This property is only valid when Expression 1 exists as an expression. If you want to add an expression for Expression 1, use the AddExpr1Expression method.

This property is read-only.

Expr1Field

This property returns a reference to the QueryField object that's used as Expression 1.

This property is only valid when Expression 1 exists as a field. You can then use the QueryField Class methods and property to manipulate this object.

If you want to add a field for Expression 1, use the AddExpr1Field method.

This property is read-only.

Expr1Type

This property returns or sets the type for Expression 1.



Note. You *must* set the type of expression for every new criteria.

The valid values are:

Value	Description
1	Field
2	Expression

This property is read-write.

Example

The following is used to test the expression in order to determine the property to use to retrieve it:

```
&MyExpr1 = &MyQueryColl.Next();

If &MyExpr1.Expr1Type = 1 Then /* Expression is a Field */

    &OldValue = &MyExpr1.Expr1Field;

    /* do processing */

Else /* Expression 1 is an expression */

    &OldValue = &MyExpr1.Expr1Expression;

    /* Do processing */

End-if;
```

The following is an example showing how to add a field for Expression 1.

```
/* add a new criteria */

&MyCriteria = &MyQuery.AddCriteria();

/* set the type of the first expression to be a field */

&MyCriteria.Expr1Type = 1;
```

```
/* add the first field from the ABSENCE_HIST record */  
  
&MyField = &MyCriteria.AddExpr1Field(ABSENCE_HIST, 1);
```

Expr2Constant

This property returns or sets the constant value for Expression 2. This property takes a string value.

This property is only valid when Expression 2 is defined as a constant.

This property is read-write.

Expr2Expression

This property returns a reference to the QueryExpression object that's used as Expression 2.

This property is only valid when Expression 2 exists as an expression. If you want to add an expression for Expression 2, use the AddExpr2Expression method.

This property is read-only.

Expr2Field

This property returns a reference to the QueryField object that's used as Expression 2.

This property is only valid when Expression 2 exists as a field. You can then use the QueryField Class methods and property to manipulate this object.

If you want to add a field for Expression 2, use the AddExpr2Field method.

This property is read-only.

Expr2Subquery

This property returns a reference to the Query object that's used as a subquery for Expression 2.

This property is only valid when Expression 2 exists as a subquery. If you want to add subquery for Expression 2, use the AddExpr2Subquery method.

This property is read-only.

Expr2Type

This property returns or sets the type for Expression 2. The following is a table with all of the possible valid values for this property. However, the values for this property are dependant upon the Operator property.



For more information, see *Working with Criteria and Expressions*.

All of the valid values are:

Value	Description
1	Constant
2	Field
3	Expression
4	Subquery
6	Current date
8	Bind
9	Const-Const
10	Const-Field
11	Const-Expr
12	Field-Const
13	Field-Field
14	Field-Expr
15	Expr-Const
16	Expr-Field
17	Expr-Expr

The following table describes how to access or change Expression 2 depending on the Expression2 Type.

Expression2 Type	Method or Property for Changing the Expression	Method or Property for Accessing the Expression
Constant	Expr2Constant	Expr2Constant
Field	AddExpr2Field()	Expr2Field
Expression	AddExpr2Expression()	Expr2Expression
Subquery	AddExpr2Subquery()	Expr2Subquery
Const-Const	Expr2Constant, Expr2Constant	Expr2Constant, Expr2Constant
Const-Field	Expr2Constant,	Expr2Constant,

<i>Expression2 Type</i>	<i>Method or Property for Changing the Expression</i>	<i>Method or Property for Accessing the Expression</i>
	AddExpr2Field()	Expr2Field
Const-Expr	Expr2Constant, AddExpr2Expression()	Expr2Constant, Expr2Expression
Field-Const	AddExpr2Field(), Expr2Constant	Expr2Field, Expr2Constant
Field-Field	AddExpr2Field(), AddExpr2Field()	Expr2Field, Expr2Field
Field-Expr	AddExpr2Field(), AddExpr2Expression()	Expr2Field, Expr2Expression
Expr-Const	AddExpr2Expression(), Expr2Constant	Expr2Expression, Expr2Constant
Expr-Field	AddExpr2Expression(), AddExpr2Field()	Expr2Expression, Expr2Field
Expr-Expr	AddExpr2Expression(), AddExpr2Expression()	Expr2Expression, Expr2Expression

This property is read-write.

Example

The following is used to test the expression in order to determine the property to use to retrieve it:

```
&MyExpr2 = &MyQueryColl.Next ();
```

```
If &MyExpr2.Expr2Type = 1 Then /* Expression is a constant */
```

```
    &OldValue = &MyExpr2.Expr2Constant;
```

```
    /* do processing */
```

```
End-if;
```

The following is an example showing how to add a field for Expression 1.

```
/* add a new criteria */
```

```

&MyCriteria = &MyQuery.AddCriteria();

/* set the type of the first expression to be a field */

&MyCriteria.Expr2Type = 4;

/* add the first field from the ABSENCE_HIST record */

&MyField = &MyCriteria.AddExpr2Field(ABSENCE_HIST, 1);

```

Logical

This property returns or sets the logical portion of a criteria.



Note. This property is only valid when there are more than one criteria for a query. Also, this property is required when there is more than one criteria for a query.

The valid values are:

<i>Value</i>	<i>Description</i>
1	And
2	Or

This property is read-write.

Operator

This property returns or sets the operator for the criteria.

The value of this property determines the valid types of the Expression 2, set with the Expr2Type property.



For more information, see Working with Criteria and Expressions.

Valid values are:

<i>Value</i>	<i>Description</i>
1	None (used for initializing a new criteria.)
2	Equal to
3	Not equal to

Value	Description
4	Greater than
5	Not greater than
6	Less than
7	Not less than
12	Exists
13	Does not exist
14	Like
15	Not like
16	Is null
17	Is not null
20	Effective date less than or equal to
21	Effective date greater than or equal to
22	Effective date less than
23	Effective date greater than
24	First effective date
25	Last effective date

This property is read-write.

QueryExpression Collection

A QueryExpression Collection is returned from the Expressions Query class property.

QueryExpression Collection Methods

First

Syntax

```
First()
```

Description

The **First** method returns the first QueryExpression object in the QueryExpression collection.

Parameters

None.

Returns

A reference to a QueryExpression object if successful, NULL otherwise.

Example

```
&MyQueryExpression = &MyCollection.First();
```

Item

Syntax

```
Item (number)
```

Description

The **Item** method returns the QueryExpression object that exists at the *number* position in the QueryExpression collection.

Parameters

<i>Number</i>	Specify the position number in the collection of the QueryExpression object that you want returned.
---------------	---

Returns

A reference to a QueryExpression object if successful, NULL otherwise.

Example

```
For &I = 1 to &QueryExpressionColl.Count;  
  
    &MyQueryExpression = &QueryExpressionColl.Item(&I);  
  
    /* do processing */  
  
End-For;
```

Next

Syntax

```
Next ()
```

Description

The **Next** method returns the next QueryExpression object in the QueryExpression collection. You can only use this method after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A reference to a QueryExpression object if successful, NULL otherwise.

Example

```
&MyQueryExpression = &MyCollection.Next();
```

QueryExpression Collection Property

Count

This property returns the number of QueryExpression objects in the QueryExpression Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

QueryExpression Class

A QueryExpression object is return by the following:

- The AddExpression Query class method
- The First, Item and Next methods of the QueryExpression Collection
- All of the QueryCriteria Class Methods
- The Expr1Expression and Expr2Expression QueryCriteria properties



For more information about QueryExpression objects, see Working with Criteria and Expressions.

QueryExpression Class Properties

Aggregate

This property specifies whether or not the expression is an aggregate function.

This property takes a Boolean value: True if the expression is an aggregate function, False, otherwise.

This property is read-write.

Decimal

This property returns or sets the decimal value of the expression.

This property is only valid with numeric fields.

This property is read-write.

Length

This property returns or sets the length of the expression, as a number.

This property is read-write.

Text

This property returns or sets the text of the expression, as a character string.

This property is read-write.

Type

This property returns or sets the field type of the expression.

Valid values are:

Value	Description
1	Character
2	Long Character
3	Number
4	Signed number
5	Date
6	Time
7	Datetime

Value	Description
9	Image
10	Image Reference

This property is read-write.

QueryRecordHierarchy Collection

A QueryRecordHierarchy collection is returned from the RecordHierarchy QueryRecord property.

The order of each QueryRecordHierarchy object in the collection maps to the order of each tree node as it appears in the tree hierarchy from top to bottom.

QueryRecordHierarchy Collection Methods

First

Syntax

```
First()
```

Description

The **First** method returns the first QueryRecordHierarchy object in the QueryRecordHierarchy collection.

Parameters

None.

Returns

A reference to a QueryRecordHierarchy object if successful, NULL otherwise.

Example

```
&MyQueryRecordHierarchy = &MyCollection.First();
```

Item

Syntax

```
Item(number)
```

Description

The **Item** method returns the QueryRecordHierarchy object that exists at the *number* position in the QueryRecordHierarchy collection.

Parameters

<i>Number</i>	Specify the position number in the collection of the QueryRecordHierarchy object that you want returned.
---------------	--

Returns

A reference to a QueryRecordHierarchy object if successful, NULL otherwise.

Example

```
For &I = 1 to &QueryRecordHierarchyColl.Count;  
  
    &MyQueryRecordHierarchy = &QueryRecordHierarchyColl.Item(&I);  
  
    /* do processing */  
  
End-For;
```

ItemByName

Syntax

ItemByName (*Name*)

Description

The **ItemByName** method returns the QueryRecordHierarchy object with the name *Name*.

Parameters

<i>Name</i>	Specify the name of an existing QueryRecordHierarchy within the QueryRecordHierarchy collection. If you specify an invalid name, the object will be NULL.
-------------	---

Returns

A reference to a QueryRecordHierarchy object if successful, NULL otherwise.

Example

```
&MyQueryRecordHierarchy = &MyCollection.ItemByName("PHONELIST");
```


Next

Syntax

`Next ()`

Description

The **Next** method returns the next QueryRecordHierarchy object in the QueryRecordHierarchy collection. You can only use this method after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A reference to a QueryRecordHierarchy object if successful, NULL otherwise.

Example

```
&MyQueryRecordHierarchy = &MyCollection.Next();
```

QueryRecordHierarchy Collection Property

Count

This property returns the number of QueryRecordHierarchy objects in the QueryRecordHierarchy Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

QueryRecordHierarchy Class

A reference to a QueryRecordHierarchy object is returned by the following:

- The First, Item, ItemByName and Next QueryRecordHierarchy Collection methods.
- The QueryRelatedRecords QueryField property.

Every QueryRecordHierarchy object returned from a collection represents a record node in the Record Hierarchy tree in Query tab of PeopleSoft Query.

The QueryRecordHierarchy object returned from the QueryField represents the prompt table for the record field.

QueryRecordHierarchy Class Properties

Description

This property returns a description of the record node as a string.

This property is read-only.

Level

This property returns the level of the record node in the record hierarchy. 1 is the root node, 2 being a node beneath the root node, 3 being a child of that, and so on.

This property is read-only.

Name

This property returns the name of the record node as a string.

This property is read-only.

ParentFlag

This property returns the parent flag. The valid values are:

<i>Value</i>	<i>Description</i>
0	Record node contains no children nodes.
1	Record node contains children nodes

This property is read-only.

Metadata Collection

A Metadata collection is returned by the Metadata Query class property.

Metadata Collection Methods

First

Syntax

```
First()
```

Description

The **First** method returns the first Metadata object in the Metadata collection.

Parameters

None.

Returns

A reference to a Metadata object if successful, NULL otherwise.

Example

```
&MyMetadata = &MyCollection.First();
```

Item

Syntax

```
Item(number)
```

Description

The **Item** method returns the Metadata object that exists at the *number* position in the Metadata collection.

Parameters

<i>number</i>	Specify the position number in the collection of the Metadata object that you want returned.
---------------	--

Returns

A reference to a Metadata object if successful, NULL otherwise.

Example

```
For &I = 1 to &MetadataColl.Count;  
  
    &MyMetadata = &MetadataColl.Item(&I);  
  
    /* do processing */
```

```
End-For;
```

ItemByName

Syntax

```
ItemByName (Name)
```

Description

The **ItemByName** method returns the Metadata object with the name *Name*.



For more information about valid Metadata object names, see Using Metadata.

Parameters

<i>Name</i>	Specify the name of an existing Metadata within the Metadata collection. If you specify an invalid name, the object will be NULL.
-------------	---

Returns

A reference to a Metadata object if successful, NULL otherwise.

Example

```
&MyMetadata = &MyQuery.Metadata.ItemByName ("Descr");
```

Next

Syntax

```
Next ()
```

Description

The **Next** method returns the next Metadata object in the Metadata collection. You can only use this method after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A reference to a Metadata object if successful, NULL otherwise.

Example

```
&MyMetadata = &MyCollection.Next();
```

Metadata Collection Property

Count

This property returns the number of Metadata objects in the Metadata Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

Metadata Class

A Metadata object is returned from the Metadata Collection methods First, Item, ItemByName or Next.

Metadata Class Properties

Name

This property returns the name of each metadata property as a string.



For more information, see Using Metadata.

The following are the valid values for this property:

Value	Description
Descr	Description
LongDescr	Long description
Public/Private	Specifies if the query is public or private. If the query is private, the name of the owner is listed in Value.
LastUpdDttm	Last updated date and time
LastUpdOprId	The UserId of the user who updated the value last

Record	Record name. May be more than one.
Input Param	Input parameter. May be more than one.
Expression	Expression. May be more than one.
Field	Record field name for the output column. May be more than one.
Heading	Heading name for the output column. May be more than one.

This property is read-only.

Value

This property returns the value for the metadata property as a string.

This property is read-only.

PermissionList Collection

A PermissionList collection is returned by the PermissionList Query class property.

PermissionList Collection Methods

First

Syntax

```
First()
```

Description

The **First** method returns the first PermissionList object in the PermissionList collection.

Parameters

None.

Returns

A reference to a PermissionList object if successful, NULL otherwise.

Example

```
&MyPermissionList = &MyCollection.First();
```

Item

Syntax

Item (*number*)

Description

The **Item** method returns the PermissionList object that exists at the *number* position in the PermissionList collection.

Parameters

<i>number</i>	Specify the position number in the collection of the PermissionList object that you want returned.
---------------	--

Returns

A reference to a PermissionList object if successful, NULL otherwise.

Example

```
For &I = 1 to &PermissionListColl.Count;  
  
    &MyPermissionList = &PermissionListColl.Item(&I);  
  
    /* do processing */  
  
End-For;
```

ItemByName

Syntax

ItemByName (*Name*)

Description

The **ItemByName** method returns the PermissionList object with the name *Name*.

Parameters

<i>Name</i>	Specify the name of an existing PermissionList within the PermissionList collection. If you specify an invalid name, the object will be NULL.
-------------	---

Returns

A reference to a PermissionList object if successful, NULL otherwise.

Next

Syntax

`Next ()`

Description

The **Next** method returns the next `PermissionList` object in the `PermissionList` collection. You can only use this method after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A reference to a `PermissionList` object if successful, `NULL` otherwise.

Example

```
&MyPermissionList = &MyCollection.Next();
```

PermissionList Collection Property

Count

This property returns the number of `PermissionList` objects in the `PermissionList` Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

PermissionList Class

A `PermissionList` object is returned by the `First`, `Item`, `ItemByName` and `Next` `PermissionList` Collection methods.

PermissionList Class Property

Name

This property returns the name of one of the permission lists that have access to the query.



For more information about permission lists, see Security. For more information about query security, see Setting up Query Security.

This property is read-only.

Example

```
If &Perm.Name = "ALLPNLS" Then  
  
    /* do processing */  
  
End-If;
```

QueryDBRecord Collection

A QueryDBRecord collection is returned from the FindQueryDBRecords session method.

QueryDBRecord Collection Methods

First

Syntax

```
First()
```

Description

The **First** method returns the first QueryDBRecord object in the QueryDBRecord collection.

Parameters

None.

Returns

A reference to a QueryDBRecord object if successful, NULL otherwise.

Example

```
&MyQueryDBRecord = &MyCollection.First();
```

Item

Syntax

Item (*number*)

Description

The **Item** method returns the QueryDBRecord object that exists at the *number* position in the QueryDBRecord collection.

Parameters

<i>Number</i>	Specify the position number in the collection of the QueryDBRecord object that you want returned.
---------------	---

Returns

A reference to a QueryDBRecord object if successful, NULL otherwise.

Example

```
For &I = 1 to &QueryDBRecordColl.Count;  
  
    &MyQueryDBRecord = &QueryDBRecordColl.Item(&I);  
  
    /* do processing */  
  
End-For;
```

ItemByName

Syntax

ItemByName (*Name*)

Description

The **ItemByName** method returns the QueryDBRecord object with the name *Name*.

Parameters

<i>Name</i>	Specify the name of an existing QueryDBRecord within the QueryDBRecord collection. If you specify an invalid name, the object will be NULL.
-------------	---

Returns

A reference to a QueryDBRecord object if successful, NULL otherwise.

Example

```
&MyQueryDBRecord = &MyCollection.ItemByName("PHONELIST");
```

Next

Syntax

```
Next ()
```

Description

The **Next** method returns the next QueryDBRecord object in the QueryDBRecord collection. You can only use this method after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A reference to a QueryDBRecord object if successful, NULL otherwise.

Example

```
&MyQueryDBRecord = &MyCollection.Next();
```

QueryDBRecord Collection Property

Count

This property returns the number of QueryDBRecord objects in the QueryDBRecord Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

QueryDBRecord Class

A QueryDBRecord object is returned from the QueryDBRecord Collection methods First, Item, ItemByName or Next.

QueryDBRecord Class Methods

QueryDBRecordFieldByIndex

Syntax

```
QueryDBRecordFieldByIndex(Index)
```

Description

The **QueryDBRecordFieldByIndex** method returns the QueryDBRecordField object that exists at the *index* position in the QueryDBRecordField collection.

Parameters

<i>Index</i>	Specify the position number in the collection of the QueryDBRecordField object you want returned.
--------------	---

Returns

A reference to a QueryDBRecordField object if successful, NULL otherwise.

Related Topics

QueryDBRecordFieldByName, QueryDBRecordField Class

QueryDBRecordFieldByName

Syntax

```
QueryDBRecordFieldByName(Name)
```

Description

The **QueryDBRecordFieldByName** method returns the QueryDBRecordField object with the name *Name*.

Parameters

<i>Name</i>	Specify the name of the QueryDBRecordField object you want returned.
-------------	--

Returns

A reference to a QueryDBRecordField object if successful, NULL otherwise.

Related Topics

QueryDBRecordFieldByIndex, QueryDBRecordField Class

QueryDBRecord Class Properties

Description

This property returns the description of the QueryDBRecord as a string.

This property is read-only.

Name

This property returns the name of the QueryDBRecord as a string.

This property is read-only.

QueryDBRecordFields

This property returns a reference to a QueryDBRecordField Collection.

This property is read-only.

QueryDBRecordField Collection

A QueryDBRecordField collection is returned from the QueryDBRecordFields QueryDBRecord property.

QueryDBRecordField Collection Methods

First

Syntax

```
First()
```

Description

The **First** method returns the first QueryDBRecordField object in the QueryDBRecordField collection.

Parameters

None.

Returns

A reference to a QueryDBRecordField object if successful, NULL otherwise.

Example

```
&MyQueryDBRecordField = &MyCollection.First();
```

Item

Syntax

```
Item(number)
```

Description

The **Item** method returns the QueryDBRecordField object that exists at the *number* position in the QueryDBRecordField collection.

Parameters

<i>Number</i>	Specify the position number in the collection of the QueryDBRecordField object that you want returned.
---------------	--

Returns

A reference to a QueryDBRecordField object if successful, NULL otherwise.

Example

```
For &I = 1 to &Coll.Count;  
  
    &MyQueryDBRecordField = &Coll.Item(&I);  
  
    /* do processing */  
  
End-For;
```

ItemByName

Syntax

```
ItemByName(Name)
```

Description

The **ItemByName** method returns the QueryDBRecordField object with the name *Name*.

Parameters

<i>Name</i>	Specify the name of an existing QueryDBRecordField within the QueryDBRecordField collection. If you specify an invalid name, the object will be NULL.
-------------	---

Returns

A reference to a QueryDBRecordField object if successful, NULL otherwise.

Example

```
&MyQueryDBRecordField = &MyCollection.ItemByName("PHONELIST");
```

Next

Syntax

```
Next ()
```

Description

The **Next** method returns the next QueryDBRecordField object in the QueryDBRecordField collection. You can only use this method after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A reference to a QueryDBRecordField object if successful, NULL otherwise.

Example

```
&MyQueryDBRecordField = &MyCollection.Next();
```

QueryDBRecordField Collection Property

Count

This property returns the number of QueryDBRecordField objects in the QueryDBRecordField Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

QueryDBRecordField Class

A QueryDBRecordField object is returned from the following:

- The QueryDBRecordField QueryField property.
- QueryDBRecord Collection methods First, Item, ItemByName or Next.

QueryDBRecordField Class Properties

Decimal

This property returns the decimal positions for a field. This indicates how many numbers are allowed on the right side of the decimal.

This property is read-only.

Format

This property returns the field format for a field. Valid values are:

<i>Value</i>	<i>Description</i>
1	No format
2	Name
3	Phone Number North America
4	Zip/Postal Code North America
5	Social Security Number (SSN)
6	UpperCase
7	Mixed case
8	Raw binary
9	Numbers only
10	Canadian Social Insurance Number (SIN)
11	Phone Number International
12	Zip/Postal Code International
13	Seconds
14	Microseconds
15	Custom

This property is read-only.

Length

This property returns the length of the field as a number.

This property is read-only.

LongName

This property returns the long name of the field as a string.

This property is read-only.

Name

This property returns the name of the field as a string.

This property is read-only.

Shortname

This property returns the short name of the field as a string.

This property is read-only.

Type

This property returns the type of the field. Valid values are:

Value	Description
1	Character
2	Long Character
3	Number
4	Signed number
5	Date
6	Time
7	Datetime
9	Image
10	Image Reference

This property is read-only.

Record Class

A **record** object, instantiated from the Record class, is a single instance of a data within a row and is based on a record definition. A record object consists of one to n fields.

If a record object is instantiated using **GetRecord** (either the function or the method), the record object that is instantiated references the record from the current row in the data buffers, and its associated data.

If a record object is instantiated using the **CreateRecord** function, the record object that's instantiated is a freestanding record definition with its component set of field objects in the data buffer. The fields created by this function will be initialized to null values, that is, they will *not* contain any data. Default processing will not be performed. You can select into this record object using the **SelectByKey** method. You can also select into this record object using **SQLExec**.

CopyFieldsTo is a commonly used method for the record class. **Name**, **IsChanged**, and **FieldCount** are commonly used properties.

The record class is one of the data buffer access classes.



For more information, see Data Buffer Access.

Note

An expression of the form

```
RECORD.recname.property
```

or

```
RECORD.recname.method(. . .)
```

is converted to an object expression by using **GetRecord**(RECORD.*recname*).

Record Methods used to Create SQL Statements

You can use the following record object methods to build and execute SQL statements:

- Delete
- SelectByKey
- Insert
- Update

The methods that result in a database update (specifically, UPDATE, INSERT, and DELETE) can only be issued in the following events:

- SavePreChange
- WorkFlow
- SavePostChange

Remember that record UPDATES, INSERTs and DELETES go directly to the database server, not to the Component Processor (although you can *look at* data in the buffer using the PeopleCode debugger and other data buffer access classes). If a record method assumes that the database has been updated based on changes made in the component, that record method can be issued only in the SavePostChange event, because before SavePostChange none of the changes made to page data has actually been written back to the database.

If your application is repeating the same instruction many times, such as doing a million INSERTs, you should use the SQL object with the BulkMode property set to True, rather than the record SQL methods.



For more information, see SQL Class.

Declaring a Record Object

Record objects are declared as type Record . For example,

```
Local Record &MYRECORD;
```

Scope of a Record Object

A record can only be instantiated from PeopleCode.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Application Engine PeopleCode, Component Interface PeopleCode, and so on.

Record Class Built-in Functions

CreateRecord

GetRecord

Record Class Methods

CompareFields

Syntax

```
CompareFields (recordobject)
```

Description

The **CompareFields** method compares all like-named fields of the record object executing the method with the specified record object *recordobject*.

Parameters

<i>recordobject</i>	Specify a record object for use in the comparison. The specified record object does not have to refer to the same record as the record object executing the method.
---------------------	---

Returns

A Boolean value; True if all like-named fields have the same value.

Example

```
&REC = GetRecord(RECORD.OP_METH_VW);  
  
&REC2 = GetRecord(RECORD.OPC_METH);  
  
If &REC2.CompareFields(&REC) Then  
    WinMessage("All liked named fields have the same value");  
End-If;
```

Related Topics

CopyChangedFieldsTo, CopyFieldsTo

CopyChangedFieldsTo

Syntax

```
CopyChangedFieldsTo(recordobject)
```

Description

The **CopyChangedFieldsTo** method copies all like-named field values that have changed *from* the record object executing the method *to* the specified record object *recordobject*. This will only copy changed field values. If you want to copy all field values, use the CopyFieldsTo method.

Parameters

<i>recordobject</i>	Specify a record object to be copied to. The specified record object does not have to refer to the same record as the record object executing the method.
---------------------	---

Returns

None.

Example

```
Local Record &REC, &REC2;

&REC = GetRecord(RECORD.OPC_METH);

/* make changes to the values of fields in the record */

&REC2 = CreateRecord(RECORD.OPC_METH_WORK);

&REC.CopyChangedFieldsTo(&REC2);
```

Related Topics

CompareFields, CopyFieldsTo

CopyFieldsTo

Syntax

```
CopyFieldsTo(recordobject)
```

Description

The **CopyFieldsTo** method copies all like-named field values *from* the record object executing the method *to* the specified record object *recordobject*. This will copy all field values. If you only want to copy changed field values, use the CopyChangedFieldsTo method.

Parameters

<i>recordobject</i>	Specify a record object to be copied to. The specified record object does not have to refer to the same record as the record object executing the method.
---------------------	---

Returns

None.

Example

```
Local Record &REC, &REC2;

&REC = GetRecord(RECORD.OPC_METH);

&REC2 = CreateRecord(RECORD.OPC_METH_WORK);

&REC.CopyFieldsTo(&REC2);
```

In the following example, records are copied into a record after being fetched.

```
Component number &displayNum;

Local SQL &sql;

Local Rowset &rs, &rs2;


&level0 = GetLevel0();

&displayNum = WS_NUM_ORDERS;


&rs = GetRowset(Record.WS_ORD_HDR_VW);

&rs.Flush();

WinMessage("1");

&sql = CreateSQL("%selectall(:1) where BUSINESS_UNIT=:2", Record.WS_ORD_HDR_VW,
"M04");

WinMessage("2");

&rec = CreateRecord(Record.WS_ORD_HDR_VW);

For &i = 1 To &displayNum

    If &sql.Fetch(&rec) Then

        &rs.InsertRow(&i);

        &rec.copyfieldsto(&rs.GetRow(&i).WS_ORD_HDR_VW);

        &rs2 = &rs.GetRow(&i).GetRowset(1);

        &rs2.Select(Record.WS_ORD_LINE_VW, "where BUSINESS_UNIT=:1 and
ORDER_NO=:2", &rec.BUSINESS_UNIT.value, &rec.ORDER_NO.value);

        /* Hide rows that do not contain Products */

        If &rs2 = Null Or
None(&rs2.GetRow(1).WS_ORD_LINE_VW.ORDER_INT_LINE_NO.Value) Then

            For &j = 1 To &rs2.ActiveRowCount
```

```
        &rs2.GetRow(&j).Visible = False;

    End-For;

End-If;

End-If;

End-For;


&rs.GetRow(&i).Visible = False;
```

Related Topics

CompareFields, CopyChangedFieldsTo

Delete

Syntax

```
Delete()
```

Description

The **Delete** method uses the key fields of the record and their values to build and execute a DELETE SQL statement which deletes the record (row of data) from the SQL data table.

This method, like the DeleteRow rowset class method, initially marks the record or row as needing to be deleted. At save time the row is actually deleted from the database and cleared from the buffer.

Because this method results in a database change, it can only be issued in the following events:

- SavePreChange
- WorkFlow
- SavePostChange

If your application is repeating the same instruction many times, such as doing a million DELETES, you should use the SQL object with the BulkMode property set to True, rather than the record SQL methods.



For more information, see SQL Class.

For every record deleted by the **Delete** method, if the set language is not the base language and the record has related language records, the **Delete** method tries to do related language processing.



For more information about related language processing, see Understanding Related Language Tables.

Parameters

None.

Returns

The result is True on successful completion, False if the record was not found. Any other conditions cause termination.

Example

Suppose that KEYF1 and KEYF2 are the two key fields of record definition MYRECORD. The following code will delete the database record that has KEYF1 equal to "A" and KEYF2 equal to "X":

```
Local record &REC;  
  
&REC = CreateRecord(RECORD.MYRECORD);  
  
&REC.KEYF1 = "A";  
  
&REC.KEYF2 = "X";  
  
&REC.Delete();
```

Related Topics

SelectByKey, Insert, Update, DeleteRow rowset class method

ExecuteEdits

Syntax

```
ExecuteEdits([editlevel]);
```

where *editlevels* is a list of values in the form:

```
editlevel1 [+ editlevel2] . . . .];
```

and where *editleveln* is one of the following constants:

- %Edit_DateRange
- %Edit_PromptTable
- %Edit_Required
- %Edit_TranslateTable

- %Edit_YesNo

Description

The **ExecuteEdits** method executes the standard system edits on every field in the record. The types of edit(s) performed depends on the *editlevel*. If no *editlevel* is specified, all system edits are executed. All *editlevels* are already defined for the record definition or for the field definition, that is:

- Reasonable Date Range (Is the date contained within the specified reasonable date range?)
- Prompt Table (Is field data contained in the specified prompt table?)
- Required Field (Do all required fields contain data? For numeric or signed fields, checks for NULL or 0 values.)
- Translate Table (Is field data contained in the specified translate table?)
- Yes/No (Do all yes/no fields only contain only yes or no data?)

If any of the edits fail, the status of the property **IsEditError** is set to False for the record. The field property **EditError** is set to True for any fields that are in error. In addition, the Field class properties **MessageNumber** and **MessageSetNumber** are set to the number of the returned message and message set number of the returned message, for each field in error.



For more information, see **EditError**, **MessageNumber** and **MessageSetNumber** Field class properties.

You *must* use the **SetEditTable** method to set the prompt tables for fields that are defined with %EditTable in the record definition.



For more information, see **SetDefault**.

If you're running an Application Engine program, and you want to do set based **ExecuteEdits** (as opposed to row-by-row) consider using the meta-SQL construct %ExecuteEdits.



For more information, see %ExecuteEdits.

Considerations for ExecuteEdits and SetEditTable

If an effective date is a key on the prompt table, and the record being edited doesn't contain an EFFDT field, the current datetime will be used as the key value.

If a SETID is a key on the prompt table, and the record being edited doesn't contain a SETID field, the system will look for SETID on the other records in the current row first, then search parent rows.

For *all other keys*, the value used to "select" the correct row in the prompt table record only comes from the **record** executing the method. No other records (or key field values) are used. You may get unexpected results if not all the keys for the prompt table are filled in (or filled in correctly.)

Considerations for ExecuteEdits and Numeric Fields

A 0 (zero) may or may not be a valid value for a numeric field. ExecuteEdits handles numeric fields in different ways, depending on whether the field is required and a prompt table is used:

- If the numeric field is required and there is *no* prompt edit: 0 is considered valid, so numeric fields never fail the required test.
- If the numeric field is required and there is a prompt edit: Prompt edit is run and the value must be valid.
- If the numeric field is not required and there is a prompt edit: 0 is considered valid, so the prompt edit is skipped.

Parameters

editlevel

Specifies the standard system edits to be performed against every field on every record. If *editlevel* isn't specified, all system edits are performed. *editlevel* can be any of the following system variables.

- %Edit_DateRange
- %Edit_PromptTable
- %Edit_Required
- %Edit_TranslateTable
- %Edit_YesNo

Returns

None.

Example

The following is an example of a call to execute Required Field and Prompt Table edits:

```
&REC.ExecuteEdits(%Edit_Required + %Edit_PromptTable);
```

The following is an example showing how ExecuteEdits() could be used:

```
&REC.ExecuteEdits();
```

```

If &REC.IsEditError Then

    LogError(); /*application specific call */

End-If;

```

Related Topics

SetDefault, IsEditError property, EditError, MessageNumber and MessageSetNumber Field class properties

GetField

Syntax

```
GetField({n | FIELD.fieldname})
```

Description

The **GetField** method instantiates a field object for the specified field associated with the record. This is the *default method* for the record object. This means that any record object, followed by a parameter list, acts as if GetField is specified.



If the field you're accessing has the same name as a record property (that is, Name or FieldCount) you can't use the default method for accessing the field. You must specify **GetField**.

For example, the following is invalid for accessing the value of a field called NAME:

```
&NUMBER = &REC.NAME.Value;
```

You must use the following code to get the value of a field called NAME:

```
&NUMBER= &REC.GetField(FIELD.NAME).Value;
```

Parameters

n | FIELD.*fieldname*

Specify a field to be used for instantiating the field object. You can specify either *n* or FIELD.*fieldname*. Specifying *n* creates a field object for the *nth* field in the record. This might be used if writing code that needs to examine all fields in a record without being aware of the name. Specifying FIELD.*fieldname* creates a field object for the field *fieldname*.



There is no way to predict the order the fields will be accessed. Only use the *n* option if the order in which the fields are processed doesn't matter.

Returns

A field object.

Example

```
&REC.GetField(FIELD.CHARACTER).Value = "Hello";
```

As GetField is the default method for a record object, the following code is identical to the previous code:

```
&REC.CHARACTER.VALUE = "Hello";
```

The following code creates an array containing all the names of all the fields in a related language record.

```
Local array of string &FIELD_LIST_ARRAY;

&FIELD_LIST_ARRAY = CreateArray();

For &I = 1 to &REC_RELATED_LANG.FieldCount

    &FIELD_LIST_ARRAY.Push(&REC_RELATED_LANG.GetField(&I).Name);

End-For;
```

The GetField method requires either FIELD.fieldname or a number. It won't take a string field name. However, you can use the @ operator to convert the string field name into a FIELD.fieldname reference, as follows:

```
&REC = GetRecord();

&REC2 = GetLevel0(1).EMPL_CHECKLIST;

&FIELD2 = &REC.GetField(@ "FIELD." | &REC2.getfield(&I).Name);
```

The following code converts field name strings (using the @ symbol) to component names to get the value of all the fields in a subrecord. Note the code in **bold**.

```
Function get_draft_dst_codes

    /* Load dst id codes into record structures */

    &DSTCODES = CreateRecord(RECORD.DR_DST_CODE_SBR);

    &DRAFTITEM = GetRecord(RECORD.DRAFT_ITEM);

    &DRAFTITEM.CopyFieldsTo(&DSTCODES);

    /* If any missing get from AR dist code, then Draft Type-BU, then BU */

    For &I = 1 To &DSTCODES.FieldCount

        If None(&DSTCODES.GetField(&I).Value) Then
```

```

get_ar_dst_code();

&NAME = &DSTCODES.GetField(&I).Name;

If All(&DST.GetField(@ ("FIELD." | &NAME)).Value) Then

    &DSTCODES.GetField(&I).Value = &DST.GetField(@("FIELD." |
&NAME)).Value;

Else

    If All(&R_DRAFTBU.GetField(@("FIELD." | &NAME)).Value) Then

        &DSTCODES.GetField(&I).Value = &R_DRAFTBU.GetField(@("FIELD." |
&NAME)).Value;

    Else

        &DSTCODES.GetField(&I).Value = &R_ARBU.GetField(@("FIELD." |
&NAME)).Value;

    End-If;

End-If;

End-If;

End-For;

/* Copy the defaulted values back to draft item */

&DSTCODES.CopyFieldsTo(&DRAFTITEM);

End-Function;

```

Related Topics

GetField Function

Insert

Syntax

```
Insert()
```

Description

The **Insert** method uses the field names of the record and their values to build and execute an Insert SQL statement which adds the given record (row of data) to the SQL table.

Because this method results in a database change, it can only be issued in the following events:

- SavePreChange
- WorkFlow

- SavePostChange

If your application is repeating the same instruction many times, such as doing a million INSERTs, you should use the SQL object with the BulkMode property set to True, rather than the record SQL methods.



For more information, see SQL Class.

If you're using a record created using CreateRecord, all fields are initially set to "", 0, or NULL, depending on the type of field. If you don't specify values for these fields, these initial values are written to the database when the insert is executed. If you want the default values for the fields inserted instead, use the SetDefault method prior to using the **Insert** method.

For every record inserted with the **Insert** method, if the set language is not the base language and the record has related language records, the **Insert** method tries to do related language processing.



For more information about related language processing, see Understanding Related Language Tables.

Parameters

None.

Returns

The result is True on successful completion, False if there was a record with those keys already in the database, that is, if a duplicate record was found. Any other conditions cause termination.

Example

Suppose that KEYF1 and KEYF2 are the key fields of record definition MYRECORD, and that this record definition also contains the field definitions MYRF3, MYRF4. The following code will add a MYRECORD record (row of data) to the database, with KEYF1 set to "A", KEYF2 set to "B", MYRF3 set to "X", and MYRF4 set to "Y":

```
Local record &REC;

&REC = CreateRecord(RECORD.MYRECORD);

&REC.KEYF1.Value = "A";

&REC.KEYF2.Value = "B";

&REC.MYRF3.Value = "X";

&REC.MYRF4.Value = "Y";

&REC.Insert();
```

Related Topics

Delete, SelectByKey, Update

SelectByKey

Syntax

```
SelectByKey()
```

Description

The **SelectByKey** method uses the key field names of the record and their values to build and execute a Select SQL statement. The field values are then fetched from the database SQL table into the record object executing the method, and the Select statement is closed.



You can't use this method in dynamic views.

If you don't specify all the key fields for a record, those you exclude are left with a blank value. If you haven't set enough keys to generate a unique record, you won't receive an error and **SelectByKey** won't fetch any data.

SelectByKey returns a single row of data. If you want to fetch more than one row, use the SQL object.



For more information, see SQL Class.

For every record read by the **SelectByKey** method, if the set language is not the base language and the record has related language records, the **SelectByKey** method tries to do related language processing.



For more information about related language processing, see Understanding Related Language Tables.

You can also select into a record using SQLExec and meta-SQL. For example, if you need to use SelectByKey with an effective date, you can use the meta-SQL %SelectByKeyEffDt in a SQLExec statement. In the following example, &RECOBJ is the name of a record created using the **CreateRecord** function.

```
SQLExec("%SelectByKeyEffDt(:1, :2)", &RECOBJ, %Date, &RECOBJ);
```



For more information, see SQLExec.

Parameters

None.

Returns

The result is True on successful completion, False if the record was not found. Any other conditions cause termination.

Example

Suppose that KEYF1 and KEYF2 are the two key fields of record definition MYRECORD. The following code will read the database record that has KEYF1 equal to "A" and KEYF2 equal to "X" into the &REC record object:

```
Local record &REC;  
  
&REC = CreateRecord(RECORD.MYRECORD);  
  
&REC.KEYF1.Value = "A";  
  
&REC.KEYF2.Value = "X";  
  
&REC.SelectByKey();
```

Related Topics

Delete, Insert, Update

SetDefault

Syntax

```
SetDefault()
```

Description

SetDefault sets the value of every field in the record to a null value, or to a default, depending on the type of field.

- If this method is used against data from the Component buffers, the next time default processing occurs, it is set to its default value: either a default specified in its record field definition or one set programmatically by PeopleCode located in a FieldDefault event. If neither of these defaults exist, the Component Processor leaves the field blank.
- If this method is used with a field that isn't part of the data buffer (for example, a field in a record object instantiated with CreateRecord) the field is automatically set to its default value if one is set for the **field**, not for the record field. Any FieldDefault PeopleCode will *not* be run on these types of fields. If you want to set the default values for just one field, use the **SetDefault** field class method.



For more information, see the SetDefault field class method.

Blank numbers correspond to zero on the database. Blank characters correspond to a space on the database. Blank dates and long characters correspond to NULL on the database. **SetDefault** gives each field data type its proper value.

Parameters

None.

Returns

None.

Example

```
&CHARACTER.SetDefault();
```

Related Topics

Default Processing, SetDefault Field class method

SetEditTable

Syntax

```
SetEditTable(%PromptField, RECORD.recordname)
```

Description

The **SetEditTable** method works with the ExecuteEdits method. It is used to set the value of a field on a record that has its prompt table defined as *%PromptField* value. *%PromptField* values are used to dynamically change the prompt record for a field.

There are several steps to setting up a field that you can dynamically change its prompt table.

To set up a field with a dynamic prompt table:

1. Define a field in the DERIVED record called *fieldname*.
2. In the record definition for the field you want to have a dynamic prompt table, define the prompt table for the field as *%PromptField*, where *PromptField* is the name of the field you created in the DERIVED record.



For more information, see Creating Record Definitions.



When you use **SetEditTable**, you don't have to add a hidden field to the page.

3. Use **SetEditTable** to dynamically set the prompt table.

%PromptField is the name of the field on the DERIVED work record. *RECORD.recordname* is the name of the record to be used as the prompt table.

```
&REC.SetEditTable("%EDITTABLE1", Record.DEPARTMENT);

&REC.SetEditTable("%EDITTABLE2", Record.JOB_ID);

&REC.SetEditTable("%EDITTABLE3", Record.EMPL_DATA);

&REC.ExecuteEdits();
```

Every field on a record that has the prompt table field %EDITTABLE1 will have the same prompt table, that is, DEPARTMENT.

Considerations for SetEditTable and ExecuteEdits

The keys used to "select" the correct row in the prompt table record only come from the record object executing the method. All the keys from the component are *not* used. You may get unexpected results if not all the keys for the prompt table are filled in (or filled in correctly.)

Parameters

<i>%PromptField</i>	Specifies the name of the field in the DERIVED record that has been defined as the prompt table for a field in the record definition.
<i>RECORD.recordname</i>	Specifies the name of the record definition that contains the correct standard system edits to be performed for that field in the record.

Returns

None.

Example

```
/* To Set the Prompt Table */

&Der_EditRec = &TSKPRF_Det_Row.GetRecord(Record.DERIVED);

&Prompt_Editname = &Der_EditRec.EDITTABLE.Value;

&Prompt_Edittable = "Record." | &Prompt_Editname;

&TSKPRF_Det_Rec.SetEditTable("%EDITTABLE", @&Prompt_Edittable);
```

```
&Prompt_Editname2 = &Der_EditRec.EDITTABLE2.Value;

&Prompt_Edittable2 = "Record." | &Prompt_Editname2;

&TSKPRF_Det_Rec.SetEditTable("%EDITTABLE2", @&Prompt_Edittable2);


&Prompt_Editname3 = &Der_EditRec.EDITTABLE3.Value;

&Prompt_Edittable3 = "Record." | &Prompt_Editname3;

&TSKPRF_Det_Rec.SetEditTable("%EDITTABLE3", @&Prompt_Edittable3);


&Prompt_Editname4 = &Der_EditRec.EDITTABLE4.Value;

&Prompt_Edittable4 = "Record." | &Prompt_Editname4;

&TSKPRF_Det_Rec.SetEditTable("%EDITTABLE4", @&Prompt_Edittable4);


&Prompt_Editname5 = &Der_EditRec.EDITTABLE5.Value;

&Prompt_Edittable5 = "Record." | &Prompt_Editname5;

&TSKPRF_Det_Rec.SetEditTable("%EDITTABLE5", @&Prompt_Edittable5);


&Prompt_Editname6 = &Der_EditRec.EDITTABLE6.Value;

&Prompt_Edittable6 = "Record." | &Prompt_Editname6;

&TSKPRF_Det_Rec.SetEditTable("%EDITTABLE6", @&Prompt_Edittable6);


&TSKPRF_Det_Rec.ExecuteEdits(%Edit_Required + %Edit_PromptTable);
```

Related Topics

ExecuteEdits Record method, IsEditError Record property, EditError, MessageNumber and MessageSetNumber Field properties

Update

Syntax

```
Update ( [KeyRecord] )
```

Description

The **Update** method uses the changed fields of the record and their values to build and execute an Update SQL statement, updating the SQL table. If you are updating the key fields of the record, you must supply a record object *KeyRecord* that contains the old values of the key record fields.

Because this method results in a database change, it can only be issued in the following events:

- SavePreChange
- WorkFlow
- SavePostChange

If your application is repeating the same instruction many times, such as doing a million UPDATES, you should use the SQL object with the BulkMode property set to True, rather than the record SQL methods.



For more information, see SQL Class.

For every record updated by the **Update** method, if the set language is not the base language and the record has related language records, the **Update** method tries to do related language processing.



For more information about related language processing, see Understanding Related Language Tables.

Considerations Updating Numeric Fields

If you instantiate a record object using CreateRecord, numeric fields are initialized to zero. If you then set a numeric field equal to zero and use the Update method, the system won't recognize the field as being updated. You must set the field equal to some other value, then set it back to 0. The following pseduo-code shows an example:

```
&RECl = CreateRecord("X");

&RECl.key1.value = "A";

/* some math happens here, yielding value B */

If B = 0 Then

    &RECl.field1.value = 1;

End-if;

&RECl.field1.value = B;

&RECl.Update();
```

Parameters

KeyRecord

If you're updating the key fields of the record, you must supply the old key field values in the record object *KeyRecord*.

Returns

The result is True on successful completion, False if the record was not found or (when updating the key fields) a duplicate record was found. Any other conditions cause termination.

Example

Suppose that KEYF1 and KEYF2 are the key fields of record definition MYRECORD, and that this record definition also contains the record field definitions MYRF3, MYRF4. The following code will update the MYRECORD record in the database with KEYF1 set to "A", KEYF2 set to "B", setting MYRF3 to "X" and MYRF4 to "Y":

```
Local record &REC;

&REC = CreateRecord(RECORD.MYRECORD);

&REC.KEYF1.Value = "A";

&REC.KEYF2.Value = "B";

&REC.MYRF3.Value = "X";

&REC.MYRF4.Value = "Y";

&REC.Update();
```

This code will update the MYRECORD record in the database with KEYF1 of "A" and KEYF2 of "B", setting KEYF2 to "C", MYRF3 to "M" and MYRF4 to "N":

```
Local record &REC1, &REC2;

&REC1 = CreateRecord(RECORD.MYRECORD);

&REC2 = CreateRecord(RECORD.MYRECORD);

&REC1.KEYF1.Value = "A";

&REC1.KEYF2.Value = "B";

&REC2.KEYF1.Value = "A";

&REC2.KEYF2.Value = "C";

&REC2.MYRF3.Value = "M";

&REC2.MYRF4.Value = "N";

&REC2.Update(&REC1);
```

Related Topics

Delete, SelectByKey, Insert

Record Class Properties

FieldCount

This property returns the total number of fields contained in the record. This value is a number.

This property is read-only.

Example

```
WinMessage("This record has this many fields : " | &REC.FieldCount);
```

fieldname property

If a field name is used as a property, it accesses the field object with that name. This means the following code:

```
&REC.fieldname
```

acts the same as

```
&REC.GetField(FIELD.fieldname);
```

If you combine the *fieldname* property with the *recname* property, you will obtain the specified field object associated with that record from the row.

```
&ROW.recname.fieldname
```

This property is read-only.

Example

```
&REC.CHARACTER.Enabled = True;
```

IsChanged

This property returns True if any field value on the primary database record of the row has been changed.



This property is only meant to be used with the primary database record. It will not return valid results if used with a work record.

This property is read-only.

Considerations using IsChanged

This property and the IsChanged row property do *not* always return identical values.

If a row in a scroll contains multiple records (such as a primary database record and one or more work records), the IsChanged row property returns True if *any* of the records in the row are changed. The IsChanged record property returns True *only* if a specific record, namely, the primary database record, is changed.

If a row is deleted from a scroll, *only* the primary database record has its IsChanged property marked to False (since the row has been deleted.) Any work records in the row *still* have their IsChanged properties set to True.

Example

```
If &REC.IsChanged Then
    Warning("This Record has been changed");
End-if;
```

IsDeleted

This property is True if the record has been deleted. For a level 0 record, it is possible for a record to be deleted without the whole row being deleted. For other levels, this property is the same as the row property IsDeleted.

This property is read-only.

Example

```
&tmp = &REC.IsDeleted;
```

IsEditError

This property is True if an error has been found for any field associated with the record after executing the **ExecuteEdits** method. This property can be used with the Field class properties **EditError** (to find out which field is in error), and **MessageSetNumber** and **MessageNumber** to find the error message set number and error message number.



For more information, see ExecuteEdits Record method, IsEditError Record property, EditError, MessageNumber and MessageSetNumber Field properties.

This property is read-only.

Example

The following is an example showing how **IsEditError**, along with the **ExecuteEdits** method could be used:

```
&REC.ExecuteEdits();
```

```

If &REC.IsEditError Then

    For &I = 1 to &REC.FieldCount

        If &REC.GetField(&I).EditError Then

            LOG_ERROR(); /* application specific call */

        End-If;

    End-For;

End-If;

```

Name

This property returns the name of the record definition of the record as a string.

If you want to access an SQL table name for a record, use the %Table meta-SQL construct.



For more information, see %Table meta-SQL.

This property is read-only.

Example

```
WinMessage("The name of this record is : " | &REC.Name);
```

ParentRow

This property returns the row object for the row containing the record. If the record object was created with the CreateRecord built-in function, the value for **ParentRow** is null because the record object isn't part of a row.

This property is read-only.

Example

```

&ROWOBJECT = &REC.ParentRow;

&TMP = "The row number of the parent row is " | &ROWOBJECT.RowNumber;

/* note that RowNumber is a property of the row class */

```

RelLangRecName

This property returns the name of the related language record as a string. The value will be a null string ("") if there is no related language record.

Example

```
Local Record &REC;

&REC = GetRecord();

&RECNAME_BASE = &REC.Name;

&RELLANGRECNAME = &REC.RelLangRecName;

SQLExec("select RELLANGRECNAME from PSRECDEFN where RECNAME = :1",
&RECNAME_BASE, &RELLANGRECNAME);
```

Row Class

A **row** object, instantiated from the Row class, is a single row of data that consists of one to *n* records of data. A single row in a component scroll is a row.

A row object is always associated with a rowset object, that is, you cannot instantiate a row object without first either explicitly or implicitly instantiating a rowset object.

A row may have one to *n* child rowsets. For example, a row in a level two scroll may have *n* level three child rowsets.

CopyTo and **GetRecord** are two commonly use methods for this class. **Visible** and **IsChanged** are two commonly used properties for this class.

The row class is one of the data buffer access classes.



For more information, see Data Buffer Access.

If a rowset object is instantiated using the **CreateRowset** function, the rowset object that's instantiated is a standalone rowset. Delete and insert activity on these types of rowsets aren't automatically applied at save time. Use standalone rowsets for work records.



For more information, see Using Standalone Rowsets.

Declaring a Row Object

Row objects are declared as type Row. For example,

```
Local Row &MYROW;
```

Scope of a Row Object

A Row can only be instantiated from PeopleCode.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Application Engine PeopleCode, Component Interface PeopleCode, and so on.

Row Class Built-in Function

GetRow

Row Class Methods

CopyTo

Syntax

```
CopyTo (row)
```

Description

The **CopyTo** method copies *from* the row executing the method *to* the specified target row, copying *like-named* record fields and subscrolls at corresponding levels. The target row object must be from a "related" rowset, that is, built from the same records, but it can be under a different parent at higher levels.

This method copies *all* the records from the row object executing the method to the target row, not just the primary data record associated with the rowset.

Parameters

<i>row</i>	Specify a target row. This must be a row object, not a row number.
------------	--

Returns

None

Example

The following example copies rows from a child rowset (DERIVED_HR) to the current rows.

```
Local Row &ROW;  
  
Local Rowset &RS1;  
  
  
&RS1 = GetRowset();
```

```

For &I = 1 to &RS1.ActiveRowCount

    &ROW = GetRowset (SCROLL.DERIVED_HR) .GetRow (&I) ;

    &ROW.CopyTo (GetRow (&I) ) ;

End-For;

```

Related Topics

ParentRowset, CopyFieldsTo, and CopyChangedFieldsTo Record class methods

GetNextEffRow

Syntax

```
GetNextEffRow()
```

Description

The **GetNextEffRow** method finds the row, in the same rowset as the row executing the method, that has the next effective date (when compared to the row executing the method.)

Parameters

None

Returns

A row object. If there is no row with a later effective date, this method returns a null object.

Example

```

&ROW2 = &ROW.GetNextEff();

If &ROW2 <> NULL Then

    &TMP = &ROW2.RowNumber;

    /* other processing */

Else

    /* no effective dated row - do error processing */

End-if;

```

Related Topics

GetPriorEffRow, GetCurrEffRow Rowset method, DeleteEnabled Rowset property

GetPriorEffRow

Syntax

```
GetPriorEffRow()
```

Description

The **GetPriorEffRow** method finds the row, from the same rowset as the row executing the method, that has the prior effective date to the row executing the method.

Parameters

None

Returns

A row object. If there is no row with a prior effective date, this method returns a null object.

Example

```
&TMP = &ROW.GetPriorEffrow().RowNumber;
```

Related Topics

GetNextEffRow, GetCurrEffRow Rowset method, DeleteEnabled Rowset property

GetRecord

Syntax

```
GetRecord({n | RECORD.recname})
```

Description

The **GetRecord** method creates a record object that references the specified record within the current row object. This is the *default method* for a row object. This means that any row object, followed by a parameter list, acts as if GetRecord is specified.

Parameters

n | RECORD.*recname*

Specify a record to be used for instantiating the record object. You can specify either *n* or RECORD.*recname*. Specifying *n* creates a record object for the *n*th record in the row. This might be used if writing code that needs to examine all records in a row without being aware of the record name. Specifying RECORD.*recname* creates a record object for the record *recname*.



There is no way to predict the order the records will be accessed. Only use the *n* option if the order in which the records are processed doesn't matter.

Returns

A record object.

Example

The following example gets a record, then assigns the name of the record to the variable &TMP:

```
&REC = &ROW.GetRecord(RECORD.QEPC_LEVEL1_REC);

&TMP = &REC.NAME; /* note that NAME is a property of the record class */
```

Because GetRecord is the default method for a row, the following code is identical to the previous:

```
&REC = &ROW.QEPC_LEVEL1_REC;

&TMP = &REC.NAME;
```

The following example loops through all the records for a row:

```
Local rowset &LEVEL1;

Local row &ROW;

Local record &REC;

&LEVEL1 = GetRowset(SCROLL.EMPL_CHECKLIST);

For &I = 1 to &LEVEL1.ActiveRowCount

    &ROW = &LEVEL1.GetRow(&I);

    For &J = 1 to &ROW.RecordCount

        &REC = &ROW.GetRecord(&J);

        /* do processing */

    End-For;

End-For;
```

The following function takes a rowset and a record, passed in from another program. GetRecord won't take a variable for the record, however, using the @ symbol you can convert the string to a component record name.

```
Function Get_My_Row(&PASSED_ROWSET, &PASSED_RECORD)
```

```

For &ROWSET_ROW = 1 To &PASSED_ROWSET.RowCount

    &UNDERLYINGREC = "RECORD." | &PASSED_ROWSET.DBRecordName;

    &ROW_RECORD =
    &PASSED_ROWSET.GetRow(&ROWSET_ROW).GetRecord(@&UNDERLYINGREC);

    /* Do other processing */

End-For;

End-Function;

```

Related Topics

GetRecord function, CreateRecord Function

GetRowset

Syntax

```
GetRowset({n | SCROLL.scrollname})
```

Description

The **GetRowset** method creates a rowset object that references a child rowset of the current row. *scrollname* must specify the primary record for the child rowset.

Parameters

<i>n</i> SCROLL. <i>scrollname</i>	Specify a rowset to be used for instantiating the rowset object. You can specify either <i>n</i> or SCROLL. <i>scrollname</i> . Specifying <i>n</i> creates a rowset object for the <i>n</i> th child rowset in the row. This might be used if writing code that needs to examine all rowsets in a row without being aware of the name. Specifying SCROLL. <i>scrollname</i> creates a rowset object for the record <i>scrollname</i> . <i>scrollname</i> must specify the primary record for the child rowset.
--------------------------------------	---



There is no way to predict the order the rowsets will be accessed. Only use the *n* option if the order in which the rowsets are processed doesn't matter.

Returns

A rowset object.

Example

```
&CHILDROWSET = &ROW.GetRowset (SCROLL.QEPC_LEVEL1_REC) ;
```

Related Topics

scrollname, GetRowset Function

scrollname

Syntax

```
scrollname (n)
```

Description

If *scrollname* is used as a method name for a row, it is used to select a child rowset and a row within that rowset.

```
&ROW.scrollname (n)
```

acts the same as

```
&ROW.GetRowset (SCROLL.scrollname) .GetRow (n)
```

Parameters

<i>n</i>	Must be a valid row number for the selected child rowset.
----------	---

Returns

A row object.

Example

```
&SUBROW1 = &ROW.QEPC_LEVEL1_REC (1) ;
```

Related Topics

GetRowset Row method, GetRowset Function

Row Class Properties

ChildCount

This property returns the number of child rowsets of the row. It is defined by the "structure" of the scroll, so it will be the same for all rows of the rowset.

It might be used, in conjunction with the **GetRowset** method, to write code that examines all child rowsets.

This property is read-only.

Example

```
For &I = 1 to &ROW.ChildCount

    &ROWSET = &ROW.GetRowset (&I) ;

    /* do processing */

End-For;
```

DeleteEnabled

This property determines whether a row can be deleted (the equivalent of the user pressing ALT+8 and ENTER). This property takes a Boolean value.



This property only controls whether an end-user can delete a row. Rows can still be deleted using PeopleCode.

This property is typically used to prevent deletion of an individual row that the other properties controlling deletion would allow to be deleted. The following table goes through the Boolean logic.

<i>Will delete be allowed by user?</i>	<i>Others enable</i>	<i>Others disable</i>
DeleteEnabled = True	Yes	No
DeleteEnabled = False	No	No

The initial value of this property is True.



If **No Row Delete** is selected in Application Designer, setting the DeleteEnabled row property to True will *not* override this value. If you want to control whether a row can be deleted at runtime, you should *not* select **No Row Delete** at design time. Likewise, if the DeleteEnabled **rowset** property is False, setting the DeleteEnabled **row** property to True will *not* override the rowset property. If you want to control whether an individual row can be deleted at runtime, you should not set the DeleteEnabled rowset property to False.



For more information on setting this property, see Scroll Controls. Also, see the DeleteEnabled rowset property.

For consistency, PeopleSoft recommends that either all rows at a level should disable deletions, or they should all allow deletions.

For rows created with non-Component Processor data (such as message rowsets, Application Engine rowsets, and so on) this property has no effect.



Don't use this property with rows from rowsets created using CreateRowset. Rowsets created using the CreateRowset function are standalone rowsets, not tied to the database, so there are no database updates when they are manipulated. Delete and insert activity on these types of rowsets aren't automatically applied at save time.



For more information, see Using Standalone Rowsets.

This property is read-write.

IsChanged

This property returns True if any field value on the primary database record of the row has been changed.



This property is only meant to be used with the primary database record. It doesn't return valid results if used with a work record.



Also note: This property only returns True if the existing row object has a field that has changed. If a field in a lower level rowset has changed, this property returns False.

This property is read-only.

Considerations using IsChanged

This property and the IsChanged record property do *not* always return identical values.

If a row in a scroll contains multiple records (such as a primary database record and one or more work records), the IsChanged row property returns True if *any* of the records in the row are changed. The IsChanged record property returns True *only* if a specific record, namely, the primary database record, is changed.

If a row is deleted from a scroll, *only* the primary database record has its IsChanged property marked to False (since the row has been deleted.) Any work records in the row *still* have their IsChanged properties set to True.

Example

```
&tmp = &ROW.IsChanged;  
  
if &tmp = True then  
  
    Warning("A Field on this row has been changed");  
  
End-If;
```

IsDeleted

This property returns True if the row has been deleted.

This property is read-only.

Example

```
&tmp = &ROW.IsDeleted;
```

IsEditError

This property is True if an error has been found for any field associated with any record in the current row or on any row in any child rowset of the current row after executing the **ExecuteEdits** method. This property can be used with the Field class properties **EditError** (to find out which field is in error), and **MessageSetNumber** and **MessageNumber** to find the error message set number and error message number.



For more information, see ExecuteEdits Record method, IsEditError Record property, EditError, MessageNumber and MessageSetNumber Field properties.

This property is read-only.

Example

The following is an example showing how **IsEditError**, along with the **ExecuteEdits** method could be used:

```

&MSG.ExecuteEdits();

If &MSG.IsEditError Then

&RS = &MSG.GetRowset();

    For &J = 1 to &RS.RowCount

        &ROW = &RS.GetRow(&J);

        If &ROW.IsEditError Then

            &REC = &ROW.GetRecord();

            For &I = 1 to &REC.FieldCount

                If &REC.GetField(&I).EditError Then

                    LOG_ERROR(); /* application specific call */

                End-If;

            End-For;

        End-If;

    End-For;

End-If;

```

IsNew

This property is True if the row is a new (inserted) row.

This property is read-only.

Example

```
&tmp = &ROW.IsNew;
```

ParentRowset

This property returns a rowset object referencing the rowset containing the row.

This property is read-only.

Example

```

&tmp = &ROW.ParentRowset.RowCount;

/* note that rowcount is a property of the Rowset class */

```

***recname* property**

If a record name is used as a property, it accesses the record object with that name. This means the following code:

```
&ROW.somerecname
```

acts the same as

```
&ROW.GetRecord(RECORD.somerecname);
```

This property is read-only.

Example

```
&REC = &ROW.QEPC_LEVEL1_REC;
```

RecordCount

This property returns the number of records in the row. It is defined by the "structure" of the scroll, so will be the same for all rows of the rowset.

This property might be used in conjunction with the **GetRecord** method to write code that examines all records in some way.

This property is read-only.

Example

```
For &I = 1 to &ROW.RecordCount  
  
    &REC = &ROW.GetRecord(&I);  
  
    /* do processing */  
  
End-For;
```

RowNumber

This property returns the row number (starting from 1) of the row within its rowset.

This property is read-only.

Example

The following will return the row number of the current effective date row.

```
&NUMBER = GetRowset(SCROLL.MYRECORD).GetCurrEffRow().RowNumber;
```

Selected

This property returns True if the row is part of a grid and has been selected either by the user or programmatically.

An end-user can select or deselect all rows in a grid by clicking in the upper left-hand corner of a grid (0, 0). The first time an end-user clicks on this location, all rows will be marked as **Selected**. Additional clicks will toggle this selection.

This property is read-write.

Example

The following example determines how many rows in a grid are selected.

```
Function Count_Child_Assets(&GRIDLEVEL1 As Rowset, &NUM_CHILDREN)

    REM *** How many child assets are selected?                                *;

    &GRIDLEVEL1 = GetRowset(SCROLL.PARENT_CHILD_VW);

    For &CHILDROW_COUNT = 1 To &GRIDLEVEL1.ActiveRowCount

        If &GRIDLEVEL1.GetRow(&CHILDROW_COUNT).Selected = True Then

            &NUM_CHILDREN = &NUM_CHILDREN + 1;

        End-If;

    End-For;

End-Function;
```

Style

This property returns the name of the style class if one has been set for the row. You can also use this property to change the style of a row. This property takes a string value.

This property only *overrides* the existing style. It doesn't *change* it. The next time the page is accessed the original style is used.

The PSSTYLE style sheet does *not* contain a default style class for a row. The Style property for all rows is initially NULL (that is, two quotation marks with no space between them ("")). The row assumes the style of the grid alternate row style. Fields in the row assume field styles if set.

Setting the property for a row overrides the grid alternate row style and any page field styles (field styles that were set in Application Designer). It does *not* override any field styles set (using the Field class Style property) for fields in the row.



For more information, see Style field property and Creating Field Definitions.

Setting the Style property back to NULL (that is, two quotation marks with no space between them ("")) turns off the override.

This property is *not* valid for Windows Client applications.

Example

```
&Scroll = GetLevel0().GetRow(1).GetRowset(Scroll.IC_PC_STYLE_2);

&row_class = &Scroll.GetRow(&row);

&row_class.Style = &style;
```

Visible

If this property is True, the row will be visible if displayed on the current page. This property may be set False to hide the row.

When Visible is set to False to hide a row, the row moves to the end of the rowset. This means that the row number of the row being hidden, and any subsequent rows, *changes* as a result of hiding a row. If the row is later made visible again, it is *not* moved back to its original position, but remains at the end of the rowset.

For example, if there are 4 rows in a rowset and row 2 is hidden (that is, Visible = False), that row now becomes row 4. In order to make that row visible again, row 4 must be set to Visible = True. Alternatively, you can use a row object reference: this remains valid even though the row number of the row may have been changed.



If you use the **Sort** method on a rowset with hidden rows, the hidden rows aren't sorted. Only visible rows are sorted.

You cannot hide rows in the current context of the executing program. This means Visible cannot hide the row containing the executing program, or in a child rowset and be executed against its parent rowset. Place your PeopleCode in a parent rowset and execute it against a child rowset.

Example

```
&ROW.Visible = False;
```

The following code uses the visible property to determine if a row is visible or not before updating the value of the row.

```
&ROWSET = GetRowset(SCROLL.LD_SHP_INV_VW)
```

```
For &I = 1 To &ROWSET.ActiveRowCount
```

```
    If &ROWSET(&I).Visible Then
```

```

ITEM_SELECTED.value = "N";

End-If;

End-For;

```

Starts with the last row and works forward to the first. If any changes are made they don't impact the position of rows that still have to be processed.

```

For &I = &ACTIVE_STREAMS to 1 Step -1

    &Row = &STREAM_ROWSET(&I);

    If &Row.QS_STREAM8.STREAM_ROOT_ID.Value <> &STREAM_ROOT_ID Then

        &Row.Visible = False;

    End-if;

End-For;

```

Rowset Class

A **rowset** object, instantiated from a Rowset class, is a collection of rows associated with buffer data. A component scroll is a rowset. You can also have a level 0 rowset.

If a rowset object is instantiated using **GetRowset** (either the function or one of the methods) the rowset object that is instantiated is populated with data according to the context in which it was instantiated.

If a rowset object is instantiated using the **CreateRowset** function, the rowset object that's instantiated is a standalone rowset. Any records and field references created by this function will be initialized to null values, that is, they will *not* contain any data. You can populate this rowset object using the **CopyTo**, **Fill** or **FillAppend** methods.

Default processing isn't performed on a standalone rowset. In addition, a standalone rowset isn't tied to the Component Processor. When you fill it with data, no PeopleCode runs (like RowInsert, FieldDefault, and so on.) Delete and insert activity on these types of rowsets aren't automatically applied at save time. Use standalone rowsets for work records.



For more information, see Using Standalone Rowsets.

You might use a rowset object and the **ActiveRowCount** property to iterate over all the rows of the rowset, or to access a specific row (using the **GetRow** method) or to find the name of the primary record associated with the scroll (**DBName**).

The Rowset class is one of the data buffer access classes.



For more information, see Data Buffer Access.

Declaring a Rowset Object

Rowset objects are declared as type Rowset. For example,

```
Local Rowset &MYROWSET;
```

Scope of a Rowset Object

A Rowset can only be instantiated from PeopleCode.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, record field PeopleCode, and so on.

You can't pass a rowset object in as part of a Component Interface user defined method. (Rowsets aren't common data structures outside of a PeopleSoft system.) However, within a user-defined method for a Component Interface you can use rowset objects.

In an Application Engine program, a rowset object is the equivalent of a record object that contains one row and a single record, that is, the State Record. PeopleSoft suggests using the Record object instead of a rowset object to obtain access to the State Record.

Messages have the same structure as rowsets, that is, hierarchical data structures composed of rows, records, and fields.

File objects can have the same structure as rowsets, that is, hierarchical data structures composed of rows, records, and fields.



For examples of using standalone rowsets and files, see Using Standalone Rowsets.

Rowset Class Built-in Functions

CreateRowset

GetLevel0

GetRowset

Rowset Class Methods

CopyTo

Syntax

```
CopyTo(&DestRowset [, record_list]);
```

Where *record_list* is a list of record names in the form:

```
[RECORD.source_recname1, RECORD.target_recname1  
  
[, RECORD.source_recname2, RECORD.target_recname2]] . . .
```

Description

The **CopyTo** method copies *from* the rowset executing the method *to* the specified destination rowset, copying *like-named* record fields and subscrolls at corresponding levels.

If pairs of source and destination record names are given, these are used to pair up the records and subscrolls *before* checking for like-named record fields.



This method will not work for Application Engine state records.



If you don't specify *record_list*, *both* the record name *and* the field name have to match exactly for data to be copied from one record field to another. If you specify *record_list*, after the records have been paired up, the field names have to match before any data is copied.

If the rowset you are copying *from* has the field level **EditError**, as well as the **MessageNumber** and **MessageSetNumber** properties set, these values will be copied to the rowset you are copying to. For example, suppose you had an application message that had errors. Using the `GetSubContractInstance` function, these errors would be copied into the message object. From there, you could instantiate a message rowset, copy the message rowset into a work rowset and use the work rowset to populate Component buffers. (Once the field properties are set, the `Record`, `Row`, and `Rowset` properties `IsEditError` also get set.)

Parameters

<i>&DestRowset</i>	Specify the rowset to be copied to. This rowset object must have already been instantiated.
<i>SourceRecname</i>	Specify a record to be copied from, in the rowset object being copied from.

DestRename Specify a record to be copied to, in the rowset object to be copied to.

Returns

None.

Example

If you set one rowset equal to another, you haven't made a copy of the rowset. Instead, you have two variables pointing to the *same* data.



For more information, see Object Assignment.

In order to make a clone of an existing rowset, that is, to make two distinct copies, you can do the following:

```
&RS2 = CreateRowset (&RS) ;

&RS.CopyTo (&RS2) ;
```

The following example copies data from one rowset object to another. Because no like-named records exist between the two rowsets, the record names are specified. Only the like-named fields will be copied from one rowset to the other:

```
Local Rowset &RS1, &RS2;

Local String &EMPLID;

&RS1 = CreateRowset (RECORD.PERSONAL_DATA) ;

&RS2 = CreateRowset (RECORD.PER_VENDOR_DATA) ;

&EMPLID = "8001";

&RS1.Fill ("WHERE EMPLID =: 1", &EMPLID);

&RS1.CopyTo (&RS2, RECORD.PERSONAL_DATA, RECORD.PER_VENDOR_DATA) ;
```

The following example copies data from a message into the Component buffers, then calls the page (using TransferPage) to redraw the page. You could do this to fill a page with message data that is in error, so that an end-user can make corrections to the message data.

&WRK_ROWSET0 is the level 0 rowset and &WRK_ROWSET1 is where the data is copied to.

```
/* Get the Message */

&MSG = GetSubContractInstance (&PUBID, &PUBNODE, &CHNLNAME, &MSGNAME, &SUBNAME) ;
```

```

/* Get the Message Rowset */

&MSG_ROWSET = &MSG.GetRowset();

/* Get Level 0 */

&WRK_ROWSET0 = GetLevel0();

/* Create Work rowset */

&WRK_ROWSET1 = GetLevel0()(1).GetRowset(SCROLL.EN_REVISION_TMP);

/* Populate Work Rowset */

&MSG_ROWSET.CopyTo(&WRK_ROWSET1, RECORD.EN_REVISION, RECORD.EN_REVISION_TMP);

SetNextPage("EN_REVISION_MSG");

TransferPage();

```

Related Topics

CopyTo Row class method and CreateRowset Function

DeleteRow

Syntax

```
DeleteRow(n)
```

Description

The **DeleteRow** method deletes the row in the rowset identified by the parameter.

If the program is being run from a component against Component buffer data, a RowDelete PeopleCode event will also fire, followed by the events that normally follow a RowDelete, as if the user had manually pressed ALT+8 and ENTER.

This method initially marks the row as needing to be deleted. At save time the row is actually deleted from the database and cleared from the buffer.

DeleteRow cannot be executed from the same rowset where the deletion will take place, or from a child rowset against a parent. Place your PeopleCode in a parent rowset and execute it against a child rowset.

When **DeleteRow** is used in a loop, you have to process rows from high to low to achieve the correct results, that is, you must delete from the bottom up rather than from the top down. This is necessary because the rows are renumbered after they are deleted (if you delete row one, row two becomes row one).



If you use **DeleteRow** on a rowset created using the **CreateRowset** function, the row isn't automatically be deleted in the database when the page is saved. Rowsets created using the **CreateRowset** function are standalone rowsets, not tied to the database, so there are no database updates when they are manipulated. Delete and insert activity on these types of rowsets aren't automatically applied at save time.



For more information, see [Using Standalone Rowsets](#).

Parameters

<i>n</i>	An integer identifying a row within the rowset object. This must be ≥ 1 and \leq the number of active rows in the rowset (see ActiveRowCount).
----------	---

Returns

An optional Boolean value: True if row is deleted, False otherwise.

Example

In the following example **DeleteRow** is used in a **For** loop. The example checks a value in each row, then conditionally deletes the row. Note the syntax of the **For** loop, including the use of the -1 in the **Step** clause to loop from the highest to lowest values. This ensures that the renumbering of the rows will not affect the loop.

```
For &I = &RS2.ActiveRowCount To 1 Step -1

    If None(&CHECK_SEQ) Then

        &RS2.DeleteRow(&I);

    End-If;

End-For;
```

Related Topics

Flush, FlushRow, InsertRow, and the Insert Record class method

Fill

Syntax

```
Fill([wherestring [, bindvalue] . . . .])
```

Description

The **Fill** method will flush the rowset then read records from the database into successive rows. The records are read from the database tables corresponding to the primary database record of the scroll into that record. The records are selected by the optional *wherestring* SQL clause, in which the optional *bindvalues* are substituted, using the usual bind placeholders (:n).

In general, you will only use this method with rowsets that were created using the `CreateRowset` function.



Because `Flush` always leaves one row in the scroll, there will be one row in the scroll even if you don't read any records.

The actual number of records read into the rowset is an optional return of this method.



This method will not work with Application Engine state records. Also, you can't use this method in dynamic views.

When this method executes, unlike the **Select** method, it will *not* cause any associated PeopleCode to run as part of reading data into the rowset.



Fill only reads the primary database record. It does not read any related records, nor any subordinate rowset records.

For every record read with the **Fill** method, if the set language is not the base language and the record has related language records, the **Fill** method tries to read the related language record and does related language processing.



For more information about related language processing, see [Understanding Related Language Tables](#).

The **Fill** method uses a correlation ID of FILL for the table it reads. You must use the correlation ID if you want to refer to the rowset table name as part of the *wherestring*. You will receive a runtime error if you use the table name as a column prefix instead of the correlation ID.

Parameters

<i>wherestring</i>	Specify a SQL WHERE clause to use for selecting records to fill the rowset. This can be a string or a SQL definition.
<i>bindvalue</i>	Specify optional bind variables to be used with the WHERE clause.

Returns

The number of records read into the rowset.

Example

The following example reads all of the QA_MYRECORD records into a rowset, and returns the number of rows read:

```
&RS = CreateRowset(RECORD.QA_MYRECORD);

&NUM_READ = &RS.Fill();
```

The following example reads all of the QA_MYRECORD records that have a MYRECORD field equal to the value of &UVAL into a rowset, and returns the number of rows read:

```
&NUM_READ = &RS.Fill("where MYRECORD = :1", &UVAL);
```

If you want to re-use a WHERE clause for the *wherestring* you can use the SQL repository, and a SQL object.

```
&NUM_READ = &RS.Fill(SQL.MYWHERE, &UVAL);
```

The following gets all the SET_CNTRL_REC rows related to the row on the page, then updates SETID with the value from the page. **Fill** is used with a rowset that was created from a message that was just created, that is, a rowset that was unpopulated.

```
If FieldChanged(SETID) Then

    &MSG = CreateMessage(MESSAGE.SET_CNTRL_REC);

    &MSG_ROWSET = &MSG.GetRowset();

    &MSG_ROWSET.Fill("where SETCNTRLVALUE =:1 and REC_GROUP_ID =:2",
    SETCNTRLVALUE, REC_GROUP_ID);

    For &I = 1 To &MSG_ROWSET.ActiveRowCount

        &MSG_ROWSET.GetRow(&I).SET_CNTRL_REC.SETID.Value = SETID;

        &MSG_ROWSET.GetRow(&I).PSCAMA.AUDIT_ACTN.Value = "C";

    End-For;
```

```
&MSG.Publish();
```

```
End-If;
```

The following example uses a correlation ID for the table in the Fill SELECT, to enable the use of correlated subqueries in the WHERE clause, such as the usual effective date subquery:

```
&RSOWNER_NAME = CreateRowset (RECORD.PERSONAL_D00);

&RSOWNER_NAME.Fill("where SETID=:1 AND EMPLID=:2 AND
%EffDtCheck (PERSONAL_D00, FILL, :3)", &SETID, &EMPLID, &EFFDT);
```

The **Fill** method implicitly uses Fill as an alias for the Rowset record. This is helpful for complex Fill *where* clauses with subqueries.

```
&oprclasscountries = CreateRowset (Record.SCRTY_TBL_GBL);

&oprclasscountries.Fill("WHERE FILL.OPRCLASS = :1 AND NOT EXISTS (SELECT 'X'
FROM PS_SCRTY_SEC_GBL GBL2 WHERE GBL2.OPRCLASS = FILL.OPRCLASS AND GBL2.COUNTRY
= FILL.COUNTRY AND GBL2.PNLNAME = :2)", &OPRCLASS, %Component);
```

Related Topics

CopyTo, Select, FillAppend, and SQL Class, CreateRowset Function

FillAppend

Syntax

```
FillAppend ([wherestring [, bindvalue] . . . .])
```

Description

The **FillAppend** method reads records from the database into successive rows of the rowset starting *after* the last row that exists in the rowset. Unlike **Fill**, it doesn't first flush the rowset.



FillAppend appends rows after the last *active* row in the rowset. If you have deleted rows in the rowset, they will still be the last rows.

When a rowset is selected into, any autoselected child rowsets will also be read. The child rowsets will be read using a where clause that filters the rows according to the where clause used for the parent rowset, using a subselect.

When a rowset is created using CreateRowset, it contains one empty row. If the rowset hasn't been filled with data, **FillAppend** will fill that row too (so you don't have an empty row at the start of your rowset.)

The records are read from the database tables corresponding to the primary database record of the scroll into that record. The records are selected by the optional *wherestring* SQL clause, in which the optional *bindvalues* are substituted, using the usual bind placeholders (:n).

In general, you will only use this method with rowsets that were created using the `CreateRowset` function.

The actual number of records read into the rowset is an optional return of this method.



This method will not work with Application Engine state records. Also, you can't use this method in dynamic views.

When this method executes, unlike the **Select** method, it will *not* cause any associated PeopleCode to run as part of reading data into the rowset.



FillAppend only reads the primary database record. It does not read any related records, nor any subordinate rowset records.

For every record read with the **FillAppend** method, if the set language is not the base language and the record has related language records, the **FillAppend** method tries to read the related language record and does related language processing.



For more information about related language processing, see [Understanding Related Language Tables](#).

The **FillAppend** method uses a correlation ID of FILLAPPEND for the table it reads. You must use the correlation ID if you want to refer to the rowset table name as part of the *wherestring*. You will receive a runtime error if you use the table name as a column prefix instead of the correlation ID.

Parameters

<i>wherestring</i>	Specify a SQL WHERE clause to use for selecting records to fill the rowset. This can be a string or a SQL definition.
<i>bindvalue</i>	Specify optional bind variables to be used with the WHERE clause.

Returns

The number of records read into the rowset.

Example

The following example reads all of the QA_MYRECORD records that have a MYRECORD field equal to the value of &UVAL into a rowset, and returns the number of rows read:


```
&NUM_READ = &RS.FillAppend("where MYRECORD = :1", &UVAL);
```

If you want to re-use a WHERE clause for the *wherestring* you can use the SQL repository, and a SQL object.

```
&NUM_READ = &RS.FillAppend(SQL.MYWHERE, &UVAL);
```

Related Topics

CopyTo, Select, Fill, and SQL Class, CreateRowset Function

Flush

Syntax

```
Flush()
```

Description

Use the **Flush** method to remove all rows from the rowset and free its associated buffer. Rows that are flushed are not deleted from the database. This function is often used to clear a work scroll before using the Select method.



Flush always leaves one row in the scroll.

You cannot flush the current context of the executing program. This means **Flush** cannot be used for the rowset containing the executing program, or in any child rowset and executed against the parent rowset. Place your PeopleCode in a parent rowset and execute it against a child rowset.



Flush removes all rows and inserts a row. If the rowset is created from component data, the row is initialized (that is, defaults and RowInit PeopleCode are run, if appropriate.) If the rowset is created from message data, an Application Engine program, and so on, the rows are flushed but the row that is inserted is *not* initialized.

Parameters

None.

Returns

None.

Example

The following example checks for the value of the field CHECKLIST_CD. If something exists in this field, the second level rowset is flushed and new values are selected into it.

```

If All (CHECKLIST_CD) Then

    &RS1H.Flush();

    &RS1H.Select(RECORD.CHECKLIST_ITEM, "where Checklist_CD = :1 and EffDt =
    (Select Max(EffDt) from PS_CHECKLIST_ITEM Where CheckList_CD = :2)",
    CHECKLIST_CD, CHECKLIST_CD);

End-If;

```

Related Topics

FlushRow, DeleteRow, InsertRow

FlushRow

Syntax

```
FlushRow(n)
```

Description

Use the **FlushRow** method to remove a specific row from a rowset and from the Component buffer. Rows that are flushed are not deleted from the database.

FlushRow is a specialized and infrequently used method. In most situations, you will want to use **DeleteRow** to remove a row from the Component buffer and remove it from the database as well when the component is saved.

You cannot flush the current context of the executing program. This means **FlushRow** cannot be used for the rowset containing the executing program, or in any child rowset and executed against the parent rowset. Place your PeopleCode in a parent rowset and execute it against a child rowset.

Parameters

<i>n</i>	An integer identifying a row within the rowset object. This must be ≥ 1 and \leq the number of rows in the rowset (see property RowCount).
----------	---

Returns

None.

Example

```
&ROWSET.FlushRow(&ROWNUMBER);
```

Related Topics

DeleteRow, Flush, InsertRow

GetCurrEffRow

Syntax

```
GetCurrEffRow()
```

Description

GetCurrEffRow returns a row object for the row with the current effective date. If the primary record associated with the rowset is not effective-dated this method returns a null object.

Parameters

None

Returns

A row object for the row with the current effective date.

Example

```
&tmp = &ROWSET.GetCurrEffRow().RowNumber;  
  
/* note RowNumber is a property of the row class */
```

Related Topics

GetRow, GetNextEffRow, and GetCurrEffRow Rowset methods, GetRow Function, DeleteEnabled Rowset property

GetRow

Syntax

```
GetRow(n)
```

Description

GetRow returns a row object from a rowset. This is a *default method* for rowsets. This means that any rowset object, followed by a parameter list, acts as if GetRow is specified.

Parameters

<i>n</i>	An integer identifying a row within the rowset object. This must be ≥ 1 and \leq the number of rows in the rowset (see property RowCount).
----------	---

Returns

Returns a row object for the specified row of the rowset.

Example

In the following example, all of the lines of code do the same thing: they return the 5th row from the rowset (&ROWSET is a rowset object.)

```
&ROW = GetRowset().GetRow(5);

&ROW = GetRowset()(5);

&ROW = &ROWSET.GetRow(5);

&ROW = &ROWSET(5);
```

The following example loops through all the rows in a rowset:

```
Local rowset &LEVEL1;

Local row &ROW;

&LEVEL1 = GetRowset(SCROLL.EMPL_CHECKLIST);

For &I = 1 to &LEVEL1.ActiveRowCount

    &ROW = &LEVEL1.GetRow(&I);

    /* do some processing */

End-For;
```

Related Topics

GetRow Function

HideAllRows

Syntax

```
HideAllRows()
```

Description

HideAllRows hides all rows of the rowset. Using this method is equivalent to a loop setting the visible property of each row of the rowset to False.

You cannot hide rows in the current context of the executing program. This means **HideAllRows** cannot hide the rowset containing the executing program, or in any child rowsets and be executed against the parent rowset. Place your PeopleCode in a parent rowset and execute it against a child rowset.

Parameters

None.

Returns

None.

Example

The following example hides the second level scroll if a value exists in the NO_COMMENTS field in the first level of the scroll. The code is running from the first level of the scroll.

```
Local Rowset &RS1, &RS2;

&RS1 = GetRowset();

&RS2 = GetRowset(SCROLL.EMPL_CHKLIST_ITM);

For &I = 1 to &RS1.ActiveRowCount

    If ALL(&RS1.GetRow(&I).EMPLOYEE_CHECKLIST.NO_COMMENTS) Then

        &RS2.HideAllRows();

    End-If;

/*other processing */

End-For;
```

Related Topics

ShowAllRows Rowset method, Visible Row class property

InsertRow

Syntax

```
InsertRow(n)
```

Description

InsertRow programmatically performs the ALT+7 and ENTER (**RowInsert**) function. **InsertRow** inserts a new row in the current rowset *after* the row specified by the parameter if the primary record for the rowset is not effective dated. If the primary record for the rowset is effective dated, the new row is inserted *before* the current row, and all values from the current row are copied into the new row, except for EFFDT, which is set to the current date.

In addition, **InsertRow** propagates the keys from the higher level (if any) into the inserted row.

If the program is being run from a component against Component buffer data, a RowInit PeopleCode event will also fire, followed by the events that normally follow a RowInsert, as if the user had manually pressed ALT+7 and ENTER.

For rowsets corresponding to Component buffer data, the **InsertRow** method only executes in turbo mode: that is, default processing is performed, but only on the row being inserted.

InsertRow cannot be executed from the same rowset where the insertion will take place, or from a child rowset against a parent. Place your PeopleCode in a parent rowset and execute it against a child rowset.



If you use **InsertRow** on a rowset created using the **CreateRowset** function, the row isn't automatically inserted in the database when the page is saved. Rowsets created using the **CreateRowset** function are standalone rowsets, not tied to the database, so there are no database updates when they are manipulated. Delete and insert activity on these types of rowsets aren't automatically applied at save time.



For more information, see [Using Standalone Rowsets](#).

Effective-Dated Row Considerations

When a row is inserted, if that row contains child scrolls, this method also inserts an empty row for any child scrolls. The effective date field for this empty row is also empty. The current date is *not* used.

Parameters

<i>n</i>	An integer identifying a row within the rowset object. This must be ≥ 0 and \leq the number of active rows in the rowset (see property <code>ActiveRowCount</code>). A value of 0 will insert in front of the first row.
----------	--

Returns

An optional Boolean value: True if the row is inserted, False if the row is not inserted.

Example

In the following example, as the primary database record isn't effective-dated, the new row is inserted *after* the second row in the rowset.

```
&ROWSET.InsertRow(2);
```

Related Topics

DeleteRow Rowset class method, Delete record class method

Refresh

Syntax

```
Refresh()
```

Description

Refresh reloads the rowset object from the database using the current page keys. This causes the page to be redrawn. `GetLevel0().Refresh()` refreshes the entire page. If you only want a specific scroll to be redrawn, you can use the refresh method with that rowset. You don't have to use a level 0 rowset with this method. This is particularly useful after using `RemoteCall`.



If a scroll is marked as No Auto Select in Application Designer, **Refresh** will not work on it.



The **Refresh** method clears the existing buffers and does a refresh from the database. You do *not* want to use this method if your rowset is based on message data. Instead, you can use the CopyTo Rowset method combined with `TransferPage`.

Parameters

None.

Returns

None.

Example

The following example refreshes the entire page.

```
&MyLevel0 = GetLevel0().Refresh();
```

The following example refreshes only the second level scroll of the page:

```
&RowSetLevel2.Refresh();
```

Related Topics

[RemoteCall](#)

Select

Syntax

```
Select([parmlist], RECORD.selrecord [, wherestr, bindvars]);
```

Where *parmlist* is a list of child rowsets, given in the following form:

```
SCROLL.scrollname1 [, SCROLL.scrollname2] . . .
```

The first *scrollname* must be a child rowset of the rowset object executing the method, the second *scrollname* must be a child of the first child, and so on.

Description

Select, like the related method **SelectNew**, reads data from the database tables or views into either a row or rowset object.



This method is only valid when used with rowsets that reference data from the Component buffers. You cannot use this method with a message rowset, a rowset from an Application Engine state record, and so on. You can't use this method with rowsets created using the **CreateRowset** function (use the **Fill** method instead.)

Select automatically places child rowsets in the rowset object executing the method under the correct parent row. If it cannot match a child rowset to a parent row, an error will occur.

When a rowset is selected into, any autoselected child rowsets will also be read. The child rowsets will be read using a where clause that filters the rows according to the where clause used for the parent rowset, using a subselect.

If you don't specify any child rowsets in *paramlist*, **Select** selects from a SQL table or view specified by *selrecord* into the rowset object executing the method.

If you specify a child rowset in *paramlist*, **Select** selects from a SQL table or view specified by *selrecord* into the child rowset specified in *paramlist*, under the appropriate row of the rowset executing the method.

If the rowset executing the method is a level 0 rowset, and you specify **Select** without specifying any child rowsets with *paramlist*, the method reads only a single row, because only one row is allowed at level 0.

The rowset executing the method *does not* have to be a level 0 rowset, so the **Select** method can produce the functionality of both the **ScrollSelect** and **RowScrollSelect** functions.



Note for developers familiar with previous releases of PeopleCode. If the rowset executing the method is a level 0 rowset, and you specify a child rowset with *paramlist*, this method functions exactly like **ScrollSelect**. If the rowset executing the method is not a level 0 rowset, and no child rowsets are specified with *paramlist*, the method acts like **RowScrollSelect**.

The record definition of the table or view being selected from is called the *select record*, and identified with **RECORD.selrecord**. The select record can be the same as the primary database record associated with the rowset, or it can be a different record definition that has compatible fields. If you define a select record that differs from the primary database record for the rowset, you can restrict the number of fields that are loaded into the buffers on the client work station by only including the fields that you actually need.

Select accepts a SQL string that can contain a WHERE clause restricting the number of rows selected into the object. The SQL string can also contain an ORDER BY clause to sort the rows.

Select and its related methods generate a SQL SELECT statement at runtime, based on the fields in the select record and the WHERE clause passed to them in the method parameters.

Parameters

<i>paramlist</i>	An optional parameter list, for specifying children rowsets of the rowset executing the method, as the rowset to be selected into. The first <i>scrollname</i> in <i>paramlist</i> must be a child rowset of the rowset executing the method, the second <i>scrollname</i> must be a child of the first child, and so on.
RECORD. <i>selrecord</i>	Specifies the select record. The <i>selrecord</i> record must be defined in Application Designer and be a build SQL table (using Build, Project) as a table or a view, unless <i>selrecord</i> is the same record as the primary database record associated with the rowset. <i>selrecord</i> can contain fewer fields than the primary record associated with the rowset, although it must contain any key fields to maintain dependencies with other records.
<i>wherestr</i>	Contains a WHERE clause to restrict the rows selected from <i>selrecord</i> and/or an ORDER BY clause to sort the rows. The WHERE clause can contain the PeopleSoft meta-SQL functions such as %DateIn() or %CurrentDateIn. It can also contain inline bind variables.
<i>bindvars</i>	A list of bind variables to be substituted in the WHERE clause. The same restrictions that exist for SQLExec exist for these variables. See SQLExec for further information.

Returns

The number of rows read (optional.) This only counts the lines read into the specified rowset. It does not include any additional rows read into any child rowsets of the rowset.

Example

The following example selects into the level 0 rowset, only one row. Depending on the autoselect settings on any child scrolls, they may be reselected too.

```
&LEVEL0 = GetLevel0();

&LEVEL0.Select(RECORD.PERSONAL_DATA, "WHERE. . .");
```

The following example selects into the level 1 scroll specified. Depending on the autoselect settings on any child (level 2) scrolls, they may also be selected.

```
&LEVEL0.Select(SCROLL.EMPL_CHECKLIST, RECORD.EMPL_CHECKLIST, "WHERE. . .");
```

The following example selects into the child scroll of the level 1 rowset. Each row fetched is placed under the appropriate row in &LEVEL1. Note that instead of hard-coding the WHERE clause, the SQL repository is used to access a SQL definition named SELECT_WHERE.

```
&LEVEL1 = &LEVEL0(1).GetRowset(SCROLL.EMPL_CHECKLIST);
```

```
&LEVEL1.Select (SCROLL.EMPL_CHKLIST_ITM, RECORD.EMPL_CHKLIST_ITM,
SQL.SELECT_WHERE) ;
```

The following example first flushes the hidden work scroll, then selects into it based on a field on the page.

```
&RS1H.Flush() ;

&RS1H.Select(RECORD.CHECKLIST_ITEM, "where Checklist_CD = :1 and EffDt = (Select
Max(EffDt) from PS_CHECKLIST_ITEM Where CheckList_CD = :2)", CHECKLIST_CD,
CHECKLIST_CD) ;
```

Related Topics

SelectNew, SelectByKey Record class method

SelectNew

Syntax

```
SelectNew([paramlist], RECORD.selrecord [, wherestr, bindvars]);
```

Where *paramlist* is a list of child rowsets, given in the following form:

```
SCROLL.scrollname1 [, SCROLL.scrollname2] . . .
```

The first *scrollname* must be a child rowset of the rowset executing the method, the second *scrollname* must be a child of the first child, and so on.

Description

SelectNew is similar to **Select**, except that all rows read into the rowset are marked *new* so they are automatically inserted into the database at Save time. This capability can be used, for example, to insert new rows into the database by selecting data using a view of columns from other database tables.



This method is only valid when used with rowsets that reference data from the Component buffers. You cannot use this method with a message rowset, a rowset from an Application Engine state record, and so on. You can't use this method with rowsets created using the CreateRowset function (use the **Fill** method instead.)

Parameters

paramlist

An optional parameter list, for specifying children rowsets of the rowset executing the method, as the rowset to be selected into. The first *scrollname* in *paramlist* must be a child rowset of the rowset executing the method, the second *scrollname* must be a child of the first child, and so on.

<i>RECORD.selrecord</i>	Specifies the select record. The <i>selrecord</i> record must be defined in Application Designer and be a build SQL table (using Build, Project) as a table or a view, unless <i>selrecord</i> is the same record as the primary database record associated with the rowset. <i>selrecord</i> can contain fewer fields than the primary record associated with the rowset, although it must contain any key fields to maintain dependencies with other records.
<i>wherestr</i>	Contains a WHERE clause to restrict the rows selected from <i>selrecord</i> and/or an ORDER BY clause to sort the rows. The WHERE clause can contain the PeopleSoft meta-SQL functions such as %DateIn() or %CurrentDateIn. It can also contain inline bind variables.
<i>bindvars</i>	A list of bind variables to be substituted in the WHERE clause. The same restrictions that exist for SQLExec exist for these variables. See SQLExec for further information.

Returns

The number of rows read (optional.) This only counts the lines read into the specified rowset. It does not include any additional rows read into any child rowsets of the rowset.

Example

The following example selects rows from DATA2 and reads them into a level one scroll. If the user saves the page, these rows will be inserted into DATA1.

```
&LEVEL0.SelectNew(SCROLL.DATA1, RECORD.DATA2, "Where SETID = :1 and CUST_ID
=:2", CUSTOMER.SETID, CUSTOMER.CUST_ID);
```

If you use the same WHERE clause in more than one Select, you may want to use the SQL repository to store the SQL fragment. That way, if you ever want to change it, you will only have to change it in one place.

```
&LEVEL0.SelectNew(SCROLL.DATA1, RECORD.DATA2, SQL.Select_New);
```

Related Topics

Select, SelectByKey Record class method

SetDefault

Syntax

```
SetDefault(recname.fieldname)
```

Description

The **SetDefault** method sets the value of the specified *recname.fieldname* to a null value in every row of the rowset. If this method is being run from a component against Component buffer data,

the next time default processing is performed, this value will be reinitialized to its default value: either a default specified in its record field definition or one set programmatically by PeopleCode located in a FieldDefault event. If neither of these defaults exist, the Component Processor leaves the field blank.

No default processing is performed against data that is not in the Component buffers.

Blank numbers correspond to zero on the database. Blank characters correspond to a space on the database. Blank dates and long characters correspond to NULL on the database. **SetDefault** gives each field data type its proper value.



For more information, about default processing, see Default Processing.

Parameters

<i>recname.fieldname</i>	Specifies a field. The field must be in the specified record, on the rows of the rowset.
--------------------------	--

Returns

None.

Example

This example resets the PROVIDER to its null value. This field will be reset to its default value when default processing is next performed:

```
If COVERAGE_ELECT = "W" Then  
    &ROWSET.SetDefault (INS_NAME_TBL.PROVIDER) ;  
End-if;
```

Related Topics

SetDefault Field class method

ShowAllRows

Syntax

```
ShowAllRows ()
```

Description

ShowAllRows "unhides" all rows of the rowset object executing the method. Using this method is equivalent to a loop setting the visible property of each row of the rowset to True.

ShowAllRows cannot be executed from the same rowset you want to display, or from a child rowset against a parent. Place your PeopleCode in a parent rowset and execute it against a child rowset.

Parameters

None

Returns

None.

Example

```
&ROWSET.ShowAllRows();
```

Related Topics

HideAllRows, Visible Row class property

Sort

Syntax

```
Sort([paramlist], sort_fields);
```

Where *paramlist* is a list of child rowsets, given in the following form:

```
SCROLL.scrollname1 [, SCROLL.scrollname2] . . .
```

The first *scrollname* must be a child rowset of the rowset executing the method, the second *scrollname* must be a child of the first child, etc.

And where *sort_fields* is a list of field specifiers in the form:

```
[recordname.]field_1, order_1 [, [recordname.]field_2, order_2]...
```

Description

Sort programmatically sorts the rows in a rowset. The rows can be sorted on one or more fields.

If you specify a child rowset in *paramlist*, **Sort** selects a set of child rowsets to be sorted. If you don't specify a child rowset in *paramlist*, the rowset executing the method is sorted.



The Sort method only sorts *visible* rows in a rowset. If the rowset you're sorting has hidden rows, the hidden rows will *not* be sorted.

Parameters

<i>paramlist</i>	An optional parameter list, for specifying children rowsets of the rowset executing the method, as the rowset to be sorted. The first <i>scrollname</i> in <i>paramlist</i> must be a child rowset of the rowset executing the method, the second <i>scrollname</i> must be a child of the first child, and so on.
<i>sort_fields</i>	A list of field and order specifiers which act as sort keys. The rows in the rowset will be sorted by the first field in either ascending or descending order, then by the second field in either ascending or descending order, and so on.
<i>[recordname.]field_n</i>	Specifies a sort key field in the rowset. The <i>recordname</i> prefix is required if you've specified a field that is not on the current record.
<i>order_n</i>	A single-character string. "A" specifies ascending order; "D" specifies descending order.

Returns

None.

Example

The first example repopulates a rowset in a page programmatically by first flushing its contents, selecting new contents using **Select**, then sorting the rows in ascending order by EXPORT_OBJECT_NAME:

```
Function populate_rowset;

    &RS1 = GetLevel0() (1).GetRowset(Scroll.EXPORT_OBJECT);

    &RS1.Flush();

    &RS1.Select(RECORD.EXPORT_OBJECT, "where export_type =:
EXPORT_TYPE_VW.EXPORT_TYPE");

    &RS1.Sort(EXPORT_OBJECT_NAME, "A");

End-Function;
```

The second example sorts the rows on child rowset by primary and secondary key fields:

```
&RS1.Sort(Scroll.EN_BOM_COMPS, EN_BOM_COMPS.SETID, "A",
EN_BOM_CMOPS.INV_ITEM_ID, "A");
```

Related Topics

Select Rowset class method, GetRowset Row class method, and GetRowset Function

Rowset Class Properties

ActiveRowCount

This property returns the number of active (non-deleted) rows in the rowset.

This property will return a value of 1 for an empty scroll (**Flush** always leaves an empty row.) You can use the **IsChanged** or **IsNew** properties to check the row.

This property is read-only.

Example

```
&tmp = &ROWSET.ActiveRowCount;
```

DBRecordName

This property returns the name of the primary database record as a string for the rowset.

This property is read-only.

Example

```
&DBNAME = &ROWSET.DBRecordName;
```

DeleteEnabled

This property determines whether a rowset allows rows to be deleted (the equivalent of the user pressing ALT+8 and ENTER). This property takes a Boolean value.



This property only controls whether an end-user can delete a row. Rows can still be deleted using PeopleCode.

The initial value of this property depends on how the scroll was created at design time. If the **No Row Delete** setting is selected on the Use tab of the scroll properties dialog box, DeleteEnabled will be False; otherwise it will be True.



If **No Row Delete** is selected in Application Designer, setting DeleteEnabled to True will *not* override this value. If you want to control whether a rowset allows deletions at runtime, you should *not* select **No Row Delete** at design time.



For more information, on setting this property, see Scroll Controls.

For consistency, PeopleSoft recommends that either all rowsets at a level should disable deletions, or they should all allow deletions.

For rowsets created with non-Component Processor data (such as message rowsets, Application Engine rowsets, and so on) this property has no effect.



Don't use this property with rowsets that are created using CreateRowset. Rowsets created using the CreateRowset function are standalone rowsets, not tied to the database, so there are no database updates when they are manipulated. Delete and insert activity on these types of rowsets aren't automatically applied at save time.



For more information, see Using Standalone Rowsets.

This property is read-write.

Also see the DeleteEnabled Row property, as well as the InsertEnabled rowset property.

EffDt

This property references the effective date of the primary record associated with the rowset. If the primary record associated with the rowset is not effective-dated, this property has a null value. To find the primary record associated with a rowset object, you can use the **DBRecordName** property.



This property isn't valid with rowsets created using the CreateRowset function.

This property is read-only.

Example

```
&tmp = &ROWSET.EffDt;
```

EffSeq

This property references the effective sequence number of the primary record associated with the rowset. If the primary record associated with the rowset does not have an effective sequence number, this property has the value 0. To find the primary record associated with a rowset object, you can use the **DBRecordName** property.



This property isn't valid with rowsets created using the CreateRowset function.

This property is read-only.

Example

```
&tmp = &ROWSET.EffSeq;
```

InsertEnabled

This property determines whether a rowset allows rows to be inserted (the equivalent of the user pressing ALT+7 and ENTER). This property takes a Boolean value.



This property only controls whether an end-user can insert a row. Rows can still be inserted using PeopleCode.

The initial value of this property depend on a value set at design time. If the **No Row Insert** setting is selected on the Use tab of the scroll properties dialog box, InsertEnabled will be False; otherwise it will be True.



If **No Row Insert** is selected in Application Designer, setting InsertEnabled to True will *not* override this value. If you want to control whether a rowset allows inserts at runtime, you should *not* select **No Row Insert** at design time.



For more information on setting this property, see Scroll Controls.

For consistency, PeopleSoft recommends that either all rowsets at a level should disable inserts, or they should all allow inserts.

For rowsets created with non-Component Processor data (such as message rowsets, Application Engine rowsets, and so on) this property has no effect.



Don't use this property with rowsets created using CreateRowset. Rowsets created using the CreateRowset function are standalone rowsets, not tied to the database, so there are no database updates when they are manipulated. Delete and insert activity on these types of rowsets aren't automatically applied at save time.



For more information, see Using Standalone Rowsets.

This property is read-write.

Also see the DeleteEnabled rowset property.

IsEditError

This property is True if an error has been found on any field in any record in any row or child rowset of the current rowset after executing the **ExecuteEdits** method on either a message object or a record object. This property can be used with the Field Class properties **EditError** (to find the field that's in error), **MessageSetNumber** and **MessageNumber** to find the error message set number and error message number.



For more information see ExecuteEdits Record method, IsEditError Record property, EditError, MessageNumber and MessageSetNumber Field properties.

This property is read-only.

Example

The following is an example showing how **IsEditError**, along with **ExecuteEdits** could be used:

```
&REC.ExecuteEdits();

If &ROWSET.IsEditError Then

    For &I = 1 to &ROWSET.ActiveRowCount

        &ROW = &ROWSET(&I);

        For &J to &ROW.RecordCount

            &REC = &ROW.GetRecord(&J);

            For &K = 1 to &REC.FieldCount

                If &REC.GetField(&K).EditError Then

                    LOG_ERROR();

                    /* application specific call */

                End-If;

            End-For;

        End-For;

    End-For;

End-If;
```

Level

This property returns the level, that is, the nesting depth, of the rowset object. The top level rowset has a level number of 0.

This property is read-only.

Example

```
&tmp = &ROWSET.Level;
```

Name

This property refers to the name of the rowset. This property will return different values, based on the type of rowset.

<i>Type of Rowset</i>	<i>Returns</i>
rowset created using GetLevel0	returns an empty string
Component buffer rowset	returns primary record name
Message rowset	returns primary record name
Component Interface rowset	returns primary record name
Application Engine rowset	always returns AESTATE

This property is read-only.

Example

```
&tmp = &ROWSET.Name;
```

ParentRow

This property returns a row object containing a reference to the parent row, that is, the row containing the rowset. If this is a top-level rowset (level 0), the **ParentRow** property has a null value.

This property is read-only.

Example

```
&tmp = &ROWSET.ParentRow.RowNumber;

/* note that RowNumber is a property of the row class */
```

ParentRowset

This property returns a rowset object containing a reference to the parent rowset, that is, the rowset containing the rowset. If this is a top-level rowset (level 0), the **ParentRowset** property has a null value.

This property is read-only.

Example

```
&tmp = &ROWSET.ParentRowset.Level;
```

```
/* note Level is another property of the rowset class */
```

RowCount

This property returns the total number of rows in the rowset. It includes deleted rows. (The **ActiveRowCount** property doesn't include deleted rows.)

This property is read-only.

Example

```
&tmp = &ROWSET.RowCount;
```

TopRowNumber

This property returns the row that is being displayed at the top of the scroll (if any) for the rowset. This property can be used with ActiveX controls contained in a scroll. ActiveX controls aren't bound to page data, so the PSControlInit event runs within the context of the entire page, that is, at level 0.

Generally, this property is used to return the top row number of a scroll. However, sometimes you want to reposition the scroll. For example, if you use SetCursorPos to move the focus to a given field, there is no guarantee that the row containing the field is at the top of the scroll.

This property is read-write.

Example

The following is located in the PSControlInit event, and is used to make sure the ActiveX Chart control displays the correct data.

```
&Scroll = GetRowset (Record.MYSCROLL);

&NewTop = &Scroll.TopRowNumber;

If &NewTop <> &OldTop Then

    /* Update ActiveX Chart with data from row &Scroll.GetRow(&NewTop) */

    &OldTop = &NewTop;

End-If;
```

CHAPTER 7

Search Classes

The Search API provides the programmer the ability to access a search index and query its contents.

Portal users can index and search on content references through the Search API using PeopleCode. The Portal Administration pages provide a GUI access to indexing a portal. This search index is what is used when doing searches on the portal. Users not going through the portal can build their own search index, then use the Search API to run queries against it.

This section focuses on using the Search API to submit and process the results of a search query. It also covers how to use the Search API to build a search index off the Portal Registry.



For more information, see Building and Using Portal Search Indexes.

Search Overview

PeopleSoft has integrated the Verity search engine into PeopleTools.

All searches done using the Search API are executed against a Verity search index. A search index is similar to a record in many ways:

- it has fields that hold information about a document
- it stores one row of information for each document indexed
- it can store information in different languages

You must build a search index before you can use the Search API.

For the portal, you can build a search index either using the Search API (BuildSearchIndex PortalRegistry method) or using the Portal Administration.



For more information, see Building and Using Portal Search Indexes.

If you want to index and search something that isn't registered in the portal, you must build your own search index.



For more information, see Building and Using Portal Search Indexes.

After you've built a search index, you can execute a search against the index, and execute a search query. A search query (SearchQuery object) represents the items in an index that match a query. These matched items are retrieved in the SearchResult collection.

Using the SearchResults Collection

You specify the size of the SearchResult collection when you execute the search. If more SearchResults were returned than can be contained in the single SearchResult collection, you can access the next set of SearchResults by executing the search again, only specifying a different starting point.

For example, suppose your search returns 28 documents that match the query, but the size you specified when you executed the search was 10. You have to execute the search three times to access all the documents that matched the query.

The following pseudo-code could be behind a "Next Matches" button.

- &Start indicates at what document you want to start your search query.
- &Size is the number of search results returned in every search.
- &SearchQuery is the query object you created at the start of your program.

```
&Start = &Start + &Size;  
  
&SearchQuery.Execute(&Start, &Size);
```

Understanding SearchResults

Each SearchResult in the SearchResult collection is composed of the following:

- Key
- Score
- SearchFields

The key uniquely identifies each document in the SearchResult collections. Keys provide a link back to the item that was indexed. The value returned by the Key property depends on the search index. If you're using a search index with the portal, the key contains the content provider and the URL. If you're not using the portal search index, the key is determined by the developer who created the search index.



For more information, see Building and Using Portal Search Indexes.

Each query returns matched documents in relevance-ranked order, with those documents considered most relevant appearing at the top of the list. During search processing, a score is calculated for each retrieved document. This is the value stored in the Score property. A Score is assigned a value from 0.00 to 1.00, where 1.00 represents a perfect match to the search criteria provided.

Each SearchResult contains fields. If this is a non-portal search, these fields were specified by the developer when they created the search index. If this is a portal search, the fields are the following:

Field	Description
VALID_FROM_DATE	Valid from date
VALID_TO_DATE	Valid to date
CREATION_DATE	Creation date
CONTENT_PROVIDER	Content provider name
URL	URL
DESCRIPTION	Description

An external URL has the full URI attached so it might look something like this:

URL: `http://sports.mysports.com/nba/teams/chi/`

Whereas a URL with a content provider specified is comprised of two portions.

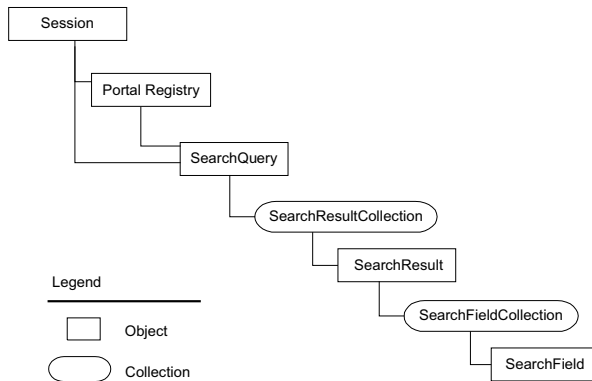
Content Provider: HRMS

URL: `ICType=Panel&Menu=ADMINISTER_WORKFORCE_(U.S.)&Market=GBL&PanelGroupName=ABSE
NCE_HISTORY1`

You can interrogate each field in the SearchField collection to find its name and value.

Search Classes Hierarchy

There are many different classes used with the Search API. The following flowchart shows the different classes, and how they are interrelated.



Search API classes hierarchy

Error Handling

All errors for the Search API, like the other APIs, are logged in the PSMessages collection, instantiated from a session object.



For more information see About Error Handling.

The search classes log errors "interactively", that is, as they happen. For example, suppose you specified an invalid SearchField name. The error would be logged in the PSMessages collection as soon as you executed the ItemByName method.

It depends on your application when you want to check for errors. However, if you check for errors after every assignment, you may see a performance degradation.

The easiest way to check for errors is to check the number of messages in the PSMessages collection, using the **Count** property. If the Count is 0, there are no errors.

```

Local ApiObject &MySession;

Local ApiObject &ERRORCOL;

Component ApiObject &SearchQuery, &SearchResultCol, &SearchResult;

Comoponent Number &Start, &Size;

&MySession = %Session;

If &MySession Then

```



```
/* connection is good */

&SearchQuery = &MySession.GetSearchQuery();

&SearchQuery.IndexName = "PSC";

&SearchQuery.Language = %Language;

&SearchQuery.QueryText = SEARCH_RECORD.USER_QUESTION.Value;

&Start = 1;

&Size = 20;

&SearchResultsCol = &SearchQuery.Execute(&Start, &Size);

If &SearchResult.HitCount > 0 Then

    For &I = 1 to &SearchResultCol.Count

        &SearchResult = &SearchResultCol.Item(&I);

        /* Do processing */

        /* Do error checking */

        &ERRORCOL = &MySession.PSMessages;

        If (&ERRORCOL.Count <> 0) Then

            /* errors occurred - do processing */

        Else

            /* no errors */

        End-If;

    End-For;

Else
```

```
        /* do processing for no returned matches to query */  
  
    End-if;  
  
Else  
  
    /* do processing for no connection */  
  
End-If;
```

Declaring Search Objects

All search objects, like a SearchResult collection, a SearchField, and so on, are declared as type ApiObject. For example,

```
Local ApiObject &SearchResultCol;  
  
Local ApiObject &SearchField;
```



All Search objects can only be declared as Local.

Scope of the Search Objects

A search object can only be instantiated from PeopleCode using a PortalRegistry or Session object.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Application Engine PeopleCode, record field PeopleCode, and so on.



Note. The Search API currently doesn't work in an Application Engine program run on the System 390.

Session Class Method

A SearchQuery object must be instantiated from either a session object or a Portal Registry object. The Search API doesn't have any built-in functions.



For more information, see Session Class and PortalRegistry Classes.

GetSearchQuery

Syntax

```
GetSearchQuery()
```

Description

The **GetSearchQuery** method returns an empty search query object used to start a search.



Note. You must set the search index and the language before you can execute a search based on a search query returned from this method.

Parameters

None.

Returns

An empty search query object.

Example

```
&SearchQuery = %Session.GetSearchQuery();
```

Related Topics

GetSearchQuery PortalRegistry method

PortalRegistry Class Methods

BuildSearchIndex

Syntax

```
BuildSearchIndex(Language)
```

Description

The **BuildSearchIndex** method builds a portal search index for the specified language.

Parameters

Returns

An optional Boolean value: True if the search index is built successfully, False otherwise.

GetSearchQuery

Syntax

```
GetSearchQuery()
```

Description

The **GetSearchQuery** method returns an empty search query object.



When you use this method to get a search query, the index property is already set to the portal registry search index, and the language property is defaulted to the end-user's language.

Parameters

None.

Returns

An empty SearchQuery object with the index and language property already set.

Example

```
&Portal = %Session.GetPortalRegistry();  
  
&Portal.Open("PORTAL");  
  
&SearchQuery = &Portal.GetSearchQuery();
```

Related Topics

GetSearchQuery Session method

SearchQuery Class

The SearchQuery object is used to specify the search index and execute a query against it.

A SearchQuery object is returned by the following:

- The GetSearchQuery method from a PortalRegistry object.
- The GetSearchQuery method from a Session object.

SearchQuery Class Method

Execute

Syntax

```
Execute(Start, Size)
```

Description

The **Execute** method runs a search query beginning at the document number specified by *Start*, and returns the number of results (or less) specified by *Size* in a SearchResult Collection.



Note. You must set the search index and the language before you can execute a search based on a search query returned from the GetSearchQuery session class method. These properties are already set using a search query returned from the GetSearchQuery PortalRegistry class method.

Parameters

<i>Start</i>	Specify the document with which you want to start your query search. This parameter takes an integer, 1 or higher.
<i>Size</i>	Specify the number of results you want returned. This parameter takes an integer value.

Returns

A SearchResult collection.

Example

```
&SearchResultCol = &SearchQuery.Execute(1, 20);
```

Related Topics

SearchResult Collection

SearchQuery Class Properties

HitCount

The **HitCount** property returns the total number of documents that actually match the query.

This property is read-only.

IndexName

The **IndexName** property specifies the SearchIndex this query is to retrieve the result from.



Note. You must set the search index and the language before you can execute a search based on a search query returned from the GetSearchQuery session class method. These properties are already set using a search query returned from the GetSearchQuery PortalRegistry class method.

This property is read-write.

Example

```
&SearchQuery.IndexName = "PSA";
```

Language

The **Language** property specifies the language of the search index.



Note. You must set the search index and the language before you can execute a search based on a search query returned from the GetSearchQuery session class method. These properties are already set using a search query returned from the GetSearchQuery PortalRegistry class method.

This property is read-write.

ProcessedCount

The **ProcessedCount** property returns the total number of documents that were subjected to the search query.

This property is read-only.

QueryText

The **QueryText** property returns the query text to be submitted to the SearchQuery object.

This property is read-write.

SearchResult Collection

A SearchResult collection is returned from the Execute SearchQuery method.

SearchResult Collection Methods

First

Syntax

```
First()
```

Description

The **First** method returns the first SearchResult object in the SearchResult collection. If the SearchResult collection is empty, returns NULL.

Example

```
&MyResult = &MyCollection.First();
```

Item

Syntax

```
Item(number)
```

Description

The **Item** method returns the SearchResult object with the position in the SearchResult collection specified by *number*.

Parameters

<i>Number</i>	Specify the position number in the collection of the SearchResult object that you want returned. Valid values start at 1.
---------------	---

Returns

A SearchResult object if successful, NULL otherwise.

Example

```
For &I = 1 to &MyCollection.Count  
  
    &MyResult = &MyCollection.Item(&I);  
  
    /* Do processing */  
  
End-For;
```

Next

Syntax

Next ()

Description

The **Next** method returns the next SearchResult object in the SearchResult collection. You can only use this method after you have used the **First** method: otherwise the system doesn't know where to start.

Example

```
&MyResult = &MyCollection.Next();
```

SearchResult Collection Property

Count

This property returns the number of SearchResult objects in the SearchResult collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

SearchResult Class

The SearchResult object represents a specific SearchResult in a SearchResult Collection object.

SearchResult objects are instantiated from a SearchResult Collection with the First, ItemByName or Next methods.

SearchResult Class Properties

Key

The **Key** property returns the key for this SearchResult object.

This property is read-only.

Score

The **Score** property returns the score for this SearchResult object.

This property is read-only.

SearchFields

The **SearchFields** property returns a SearchField Collection.

This property is read-only.

SearchField Collection

A SearchField collection is returned by the SearchFields SearchResult property.

SearchField Collection Methods

First

Syntax

```
First()
```

Description

The **First** method returns the first SearchField object in the SearchField collection. If the SearchField collection is empty, returns NULL.

Example

```
&MyField = &MyCollection.First();
```

ItemByName

Syntax

```
ItemByName(Name)
```

Description

The **ItemByName** method returns the SearchField object specified by *Name*.

Parameters

Name Specify the name of the SearchField you want returned.

Returns

A SearchField object if successful, NULL otherwise.

Example

```
&MyField = &MyCollection.ItemByName (VALID_TO_DATE);
```

Next

Syntax

```
Next ()
```

Description

The **Next** method returns the next SearchField object in the SearchField collection. You can only use this method after you have used the **First** method: otherwise the system doesn't know where to start.

Example

```
&MyField = &MyCollection.Next();
```

SearchField Collection Property

Count

This property returns the number of SearchField objects in the SearchField collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

SearchField Class

A SearchField is returned by First, ItemByName and Next SearchField Collection methods.

SearchField Class Properties

Name

The **Name** property returns the name of this SearchField object.

This property is read-only.

Value

The **Value** property returns the value for this SearchField object.

This property is read-only.

Search API Examples

The following PeopleCode programs are examples of how to use the Search API.

General Purpose Search API Example

```

/*-----
   Get a search query object.
   -----*/

&SearchQuery = %Session.GetSearchQuery();

/*-----
   Set the search index to access
   -----*/

&SearchQuery.IndexName = "PSA"

/*-----
   Set the language to access
   -----*/

&SearchQuery.Language = %Language;

/*-----
   Set the text of the query the user entered
   -----*/

&SearchQuery.QueryText = "Java AND C++ AND ('MIT' in EDUCATION)"

```

```

/*-----
Execute the search passing in the starting position and "chunk size" (get me
the

first 20 result or the third 20)
-----*/

&SearchResultsColl = &SearchQuery.Execute(1, 20);

/*-----
Get the first result.
-----*/

&SearchResult = &SearchResultsColl.First();

While (&SearchResult <> NULL)

/*-----
Example of getting the key. In this case assume that the key contains a
file path to a resume.
-----*/

&ResumePath = &SearchResult.Key;

/*-----
Example of getting the value of a Field.
-----*/

&ApplicantIdSearchField =
&SearchResult.SearchFields.ItemByName("APPLICANT_ID")

&ApplicantId = &URLSearchField.Value

/*-----
Another example of a field using dot notation to simplify the code.
-----*/

&Address = &SearchResult.SearchFields.ItemByName("ADDRESS").Value

```

```

/*-----
    Call a function to do something with the search result.
    -----*/

DoSomethingUseful(&ResumePath, &ApplicantId, &Address);

/*-----

    Get the next result of the search.
    -----*/

&SearchResult = &SearchResultsColl.Next();

End-While;

```

Portal Search API Example

```

&SearchKey = %Request.GetParameter("SEARCH_TEXT");

&StartPosition = %Request.GetParameter("START_POSITION")

SearchResultChunkSize = 1000;

/*-----

    Get a portal registry object.
    -----*/

&Portal = %Session.GetPortalRegistry();

/*-----

    Open the desired portal.
    -----*/

&Portal.Open("PORTAL")

/*-----

    Get a search query object.
    -----*/

```

```

&SearchQuery = &PORTAL.GetSearchQuery();

/*-----
   Set the text of the query the user entered
-----*/

&SearchQuery.QueryText = &SearchKey

/*-----

   Execute the search passing in the starting position and "chunk size" (get me
   the first 20 result or the third 20)
-----*/

&SearchResultsColl = &SearchQuery.Execute(&StartPosition,
&SearchResultChunkSize);

/*-----

   Get the first result.
-----*/

&SearchResult = &SearchResultsColl.First();

While (&SearchResult <> NULL)

    /*-----

       Example of getting the key. In the portal case, the key is the URL, but in
       the general case it could be a product id or some kind of other database key.
       -----*/

       &SearchKey = &SearchResult.Key;

    /*-----

       Example of getting the Field. In the portal case, the field is the URL.
       Yes, it is redundant with the key but the key includes some {} around the URL
       and this just makes it easier to get the URL.
       -----*/

       &URLSearchField = &SearchResult.SearchFields.ItemByName("URL")

```

```

&URL = &URLSearchField.Value

/*-----
    Another example of a field using dot notation to simplify the code.
-----*/

&ContentProvider =
&SearchResult.SearchFields.ItemByName("ContentProvider").Value

/*-----
    Call a function to insert the result into the page.
-----*/

AddStuffToPage(&URL, &ContentProvider);

/*-----
    Get the next result of the search.
-----*/

&SearchResult = &SearchResultsColl.Next();

End-While;

&Portal.Close();

```

Session Class

PeopleTools provides a family of APIs for external access into the PeopleSoft system. The Session class is the root of the APIs. It controls access to the PeopleSoft system, controls the environment, and allows you to do error handling for all APIs from a central location.

The PeopleSoft APIs are intended to be used, and are supported, in the following environments.

- PeopleSoft Windows client with either an existing two-tier connection or an existing connection to an application server.
- PeopleSoft Application Engine batch program.
- Non-PeopleSoft program connecting to an application server.

The PeopleSoft APIs are exposed to the following language environments:

- PeopleCode
- OLE/COM
- C/C++
- Java

The APIs that are accessible using a session object are:

- Component Interface Classes
- PortalRegistry Classes
- Tree Classes



Tree Classes are only accessible through PeopleCode.



For more information about each API, see the appropriate documentation.

A session object controls the following:

- Security and access to the PeopleSoft system
- Errors and error handling
- Regional settings (that is, internationalization) for language, dates, times, and numbers
- Tracing

Security and Access to the PeopleSoft System

A session object controls security and access to the PeopleSoft system. You must "sign-on" to the PeopleSoft session with a valid user ID and password when you use the **Connect** method. In addition, there's an external authentication parameter that you can use with the connect.



Additional authentication will be available after this release.

About Error Handling

The session object handles error processing for all the APIs, such as Component Interfaces, the Query API, and so on. From the session object, you can check if there have been any errors.



Note. The exception to this is Subscription PeopleCode. All errors for Subscription PeopleCode get logged to the application message error table.



Warning! For this release, errors for the Tree API that occur in 3-tier mode or in PeopleSoft Internet Architecture do *not* get sent to the PSMessages collection. Instead, they're reported back to the calling application, that is, for 3-tier they get displayed as messages on the application server, and in PIA they come back in the generated HTML.

Error messages include system error messages or messages from the message catalog (a common set of error messages used by all PeopleSoft applications.) For a Component Interface, the session object level errors will also contain any text errors that may have occurred from running ExecuteEdits.

On the session object, you can use the following properties to initially check for errors:

- **ErrorPending** indicates whether there are API errors
- **WarningPending** indicates whether there are API warnings

All errors are contained in the **PSMessages** collection. (The **PSMessages** property on a Session object returns this collection.)

Each item in this collection is a **PSMessage object**. A PSMessage object contains information about the specific error that has occurred, like the explain text for the error, the message set number, and so on. (The type of information depends on the type of error.)

If the error was caused by a Component Interface, a contextual string is attached to the end of the **Text** property of the **PSMessage** object. This contextual string contains the exact location of the error, that is, which field in which data collection, on which row, caused the error. This string is also accessible (by itself) using the **Source** property.

When errors are loaded into the PSMessages collection depends on the type of error. System errors (that is, errors in the connection to the PeopleSoft Session) are logged as they occur. When an API error is logged depends on the API, as some APIs can be run in either interactive mode (meaning errors are logged as they happen) or in non-interactive mode (so errors are only logged when a particular method is run.)

If you are using Visual Basic or another COM environment, the PeopleSoft API system will raise a COM exception the first time **ErrorPending** changes from False to True. It will *not* raise another exception until the PSMessages collection is cleared, which sets **ErrorPending** back to False.



If you've called an API from an Application Engine program, all errors are also logged in the Application Engine error log tables.



For more information about when an error is logged, see the individual APIs.



For more information about session errors, see Accessing a PSMessages Collection.

About Regional Settings

Regional settings allow the user to set the display of numbers, dates, times, and currencies to comply with usage in a specific locale. The session object exposes some of these values to the API through the **RegionalSettings** property. This property returns a regional settings object that has properties you can use to change these settings.



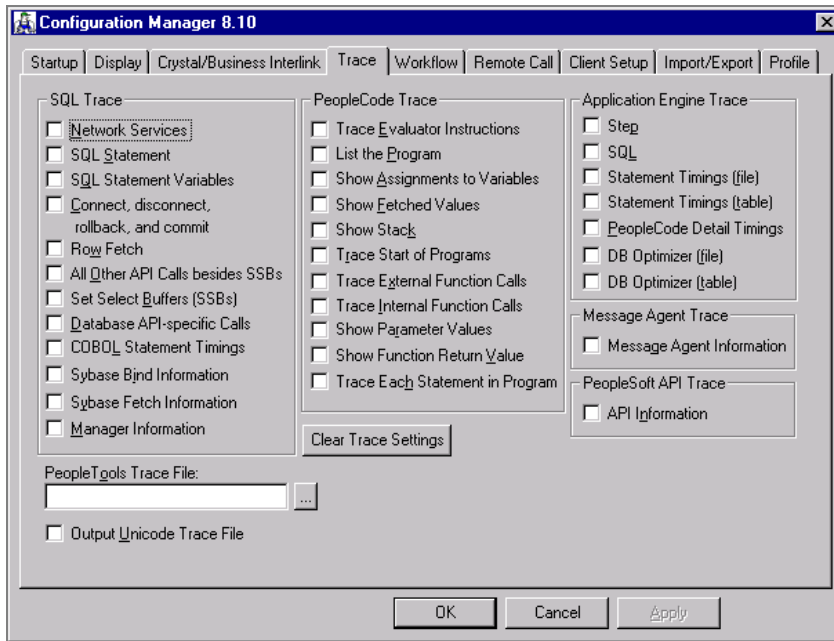
For more information about Regional Settings, see Controlling Windows Regional Settings.



For more information about the Regional Setting object, see Regional Settings Class Properties.

About Trace Settings

The Configuration Manager is a PeopleTool that lets you control the environment in which your PeopleTools session is running. The Trace tab in the Configuration Manager allows you to select the SQL trace options and the PeopleCode trace options that you want. The **TraceSettings** object (assessable from a session object) lets you control many of the options that appear on the Trace tab during your session.



Trace Tab of Configuration Manager

Not every option on the Trace tab is represented with a trace settings property. In particular, the Application Engine trace values and the Message Agent trace values aren't represented.

The initial value of a trace settings property depends on whether the session object used to instantiate the trace settings object was itself instantiated from within a component or from outside a PeopleSoft application.

- If the session object used to instantiate the Trace settings object was itself created within the context of a component (that is, from a running PeopleSoft application) the trace values currently in existence for that component are inherited by the trace settings object. For example, if the SQL trace property `NetworkServices` was set to `True` (the check box was selected) in configuration manager before the session object was instantiated, the value for this property will be initially set to `True` for the trace settings object.
- If the session object used to instantiate the trace settings object was itself *not* instantiated from a component (that is, from outside a PeopleSoft application), the initial value for all Boolean trace setting properties will be `False`. The initial value for the PeopleTools trace file will be the file that is set in the Configuration Manager for the workstation.

For both cases, when you make a change to the trace settings, these changes will continue in the running environment even after the session is disconnected.

Session Class Reference

The following sections go into more detail of the properties and methods that can be used with a session object.

Declaring a Session Object

All session objects are declared as type ApiObject. For example,

```
Local ApiObject &MYSESSION;

Global ApiObject &PSMessage;
```

The following session objects can be declared as Global or Component:

- Session
- PSMessages Collection
- PSMessage

All other session objects *must* be declared as Local.

State Considerations

In the PeopleSoft Internet Architecture, all processing occurs on the server. If you do something in your PeopleCode program that causes a trip back to the user before the end of your PeopleCode program, the PeopleCode program can *not* regain its state, that is, it can't reset local variables, and so on. This applies to all APIs that use the Session object.

If you must go back to the user before the end of your PeopleCode program, you must set all your objects to NULL before you go, or else you'll receive a runtime error.

For example, the following code causes an error:

```
&Session = %Session;

Rem &Session = Null;

WinMessage("Hello");
```

The following example does *not* cause an error:

```
&Session = %Session;

&Session = Null;

WinMessage("Hello");
```

Considerations for Declaring Collections

You *must* declare all collections for C/C++ and COM as objects, or else you'll receive errors at runtime.

For example if you're using RegionalSettings, you should include the following your declarations for a VB program:

```
Dim oRegionalSettings As Object
```

Then, you can use the following to populate this variable:

```
Set oRegionalSettings = oCISession.RegionalSettings  
  
oRegionalSettings.LanguageCd = "GER"
```

Scope of a Session Object

A Session can be instantiated from PeopleCode, from a Visual Basic program, from C++, COM, and so on. For examples, see the example section in Connect.

This object can be used anywhere you have PeopleCode, that is, in Application Engine PeopleCode, record field PeopleCode, and so on.

Session Class Built-in Function

GetSession

Session Class Methods

Connect

Syntax

```
(version, {"EXISTING" | //PSoftApplicationServer:JoltPort}, UserID, Password,  
ExtAuth)
```

The **Connect** method connects a session object to a PeopleSoft application server.

If you already have a PeopleSoft session running (you are already connected to a PeopleSoft application server and are running a component, and so on) you must specify EXISTING, and not the *//PSoftApplicationServer:JoltPort*. Typically, use the parameter value of EXISTING from PeopleCode, not from other language environments.

If you are using an existing connection to the application server, you *cannot* specify a different user ID or password. If you don't specify these values as NULL string, you must specify the exact same user ID (and password) as the one that originally started the session.



If you want to use the existing session, you can use the %Session system variable to return a reference of a session object.

Parameters

<i>version</i>	Specify the API version that the client is expecting. Future releases will use this parameter. For now, you must use a 1.
EXISTING <i>//PSoftApplicationServer::Jolt</i> <i>Port</i>	Specify EXISTING if you're currently connected to an application server. Otherwise, specify the named application server machine and the IP port to connect to on an application server machine.
<i>UserID</i>	Specify the PeopleSoft user ID to use for the connection. This must be a valid user ID. If you are using an existing connection, you can specify a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter. .
<i>Password</i>	Specify the password associated with the user ID to use for the connection. This must match the password assigned for this user ID <i>exactly</i> . If you are using an existing connection, you can specify a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.
<i>ExtAuth</i>	This parameter is required, but it is unused in this release. You must enter a 0 for this parameter.

Returns

A Boolean value: True if the connection completed successfully, False otherwise.

Example

The following is an example of the code you would use if there was already an existing connection to the application server:

```
Local ApiObject &MYSESSION;

&MYSESSION = GetSession();

&MYSESSION.Connect(1, "EXISTING", "", "", 0);
```

The following is an example connecting to a PeopleSoft system using a Visual Basic program. The first thing it does is check for error messages. If there are any errors, the text of the error is displayed to the user.

```
On Error GoTo eMessage

Dim oSession As Object
```

```
Dim oPSMessage As Object

Dim oBC As Object

Set oSession = CreateObject("PeopleSoft.Session")

oSession.Connect 1, "//PSSOFT0061998:7000", "PTDMO", "PTDMO", 0

'Application Specific Processing

eMessage:

If (oSession.PSMessages.Count > 0) Then
    For i = 1 To oSession.PSMessages.Count
        Set oPSMessage = oSession.PSMessages.Item(i)

        MsgBox (oPSMessage.Text)
    Next i
End If
```

Related Topics

[%Session](#)

Disconnect

The **Disconnect** method disconnects the session associated with the session object executing the method from that PeopleSoft session.

This method returns a Boolean value: True if successfully disconnected, False otherwise.

API Instantiation Methods

The following is a list of the other methods used with the session object to instantiate an API object like a Component Interface, a tree structure, and so on.



For more information, see the appropriate documentation.

- FindCompIntfc(*partial_name*): returns a collection of Component Interfaces based on the partial Component Interface name
- FindPortalRegistries(*partial_name*): returns a collection of PortalRegistry objects
- FindQueryDBRecords(): returns a reference to a QueryDBRecord collection, filled with zero or more records
- FindQueries(): returns a reference to a Query collection, filled with zero or more queries
- FindQueriesByDateRange(*StartDateString*, *EndDateString*): returns a reference to a Query collection, filled with zero or more queries that match the specified date range
- FindTree(*SetId*, *UserKeyValue*, *TreeName*, *EffDt*, *BranchName*): returns a collection of tree objects based on the partial values set for the parameters
- FindTreeStructure(*StructureID*): returns a collection of tree structure objects based on the partial tree structure name
- GetCompIntfc(*COMPINTFC.componentname*): returns a Component Interface object
- GetPortalRegistry(): returns a PortalRegistry object
- GetQuery(): returns a query object
- GetSearchQuery(): returns an empty search query object used to start a search
- GetTree(*SetId*, *UserKeyValue*, *TreeName*, *EffDt*, *BranchName*): returns a tree object
- GetTreeStructure(*StructureID*): returns a tree structure object

Session Class Properties

ErrorPending

This property indicates whether there are any error messages pending for the API that is currently running. Once this property has been set, it will return True until the PSMessages collection is cleared (or deleted.)

This property is read-only.

Example

```
If &MYSESSION.ErrorPending Then

    &COUNT = &MYSESSION.PSMessages.Count ;

    For &I = 1 To &COUNT

        &ERROR = &MYSESSION.PSMessages.Item(&I) ;

        &TEXT = &ERROR.Text ;
```



```

        WinMessage("Error text " | &TEXT);

    End-For;

    &MYSESSION.PSMessages.DeleteAll();

```

PSMessages

This property returns a reference to a PSMessages collection object. If there are no messages, this property returns an object reference to PSMessages with a count of zero.



For more information, see PSMessages Collection Methods.

This property is read-only.

Example

```

If &SESSION.PSMessages.Count > NULL Then

    /* there are messages - do processing */

End-if;

```

RegionalSettings

This property returns a reference to a RegionalSettings collection object.



For more information, see Regional Settings Class Properties

This property is read-only.

Repository

This property returns a reference to a Repository object.



For more information, see The PeopleSoft API Repository.

SuspendFormatting

This property turns off the PeopleSoft automatic formatting when coercing a string into a field. Formatting is typically necessary for external Component Interface users (such as for a Visual Basic program,) yet should be suspended for a PeopleCode program. You should not typically

need to set this value yourself, as it's set appropriately based on where the session object is created.

This property takes a Boolean value: True means formatting is suspended, False means it isn't suspended. If you start your session from a PeopleCode program, the default for this property is True. If you don't start the session from a PeopleCode program, the default is False.

This property is read-write.

TraceSettings

This property returns a reference to a TraceSettings collection object.



For more information, see Trace Setting Class Properties

This property is read-only.

WarningPending

This property indicates whether there are any warning messages pending for the API that is currently running.

This property is read-only.

Accessing a PSMessages Collection

Use the PSMessages collection to return all of the messages you've had with the session. The following example finds all the messages listed in the PSMessages collections when the Component Interface Save methods fails. The example assumes that &CI is the Component Interface object reference.

```
If Not &CI.Save() Then

    /* save didn't complete */

    &COLL = &MYSESSION.PSMessages;

    For &I = 1 to &COLL.Count

        &ERROR = &COLL.Item(&I);

        &TEXT = &ERROR.Text;

        /* do message processing */
    End For
End If
```

```
End-For;  
  
&COLL.DeleteAll(); /* Delete all messages in queue */  
  
End-if;
```

As you evaluate each message, you want to delete it from the PSMessages collection. Or, you can delete all the messages in the queue at once using DeleteAll.

If there are multiple errors when saving a Component Interface, all errors are logged to the PSMessages collection, not just the first occurrence of an error.

Considerations for Declaring Collections

You *must* declare all collections for C/C++ and COM as objects, or else you'll receive errors at runtime.

PSMessages Collection Methods

DeleteAll

Syntax

```
DeleteAll()
```

Description

The **DeleteAll** method deletes all the PSMessages objects in the PSMessages collection executing the method. You should use this property after you have processed all the errors in the collection. This method also resets the status of ErrorPending to False.

Returns

This method returns a Boolean value: True if all PSMessages in the collection were successfully deleted, False otherwise.

DeleteItem

Syntax

```
DeleteItem(number)
```

Description

The **DeleteItem** method deletes the PSMessages object that exists at the *number* position in the PSMessages collection executing the method.

Parameters

number Specify the position number in the collection of the PSMMessage object that you want to delete. All other items within the collection will be renumbered.

Returns

This method returns a Boolean value: True if the PSMMessage object was successfully deleted, False otherwise.

Example

```
&ERROR.DeleteItem(&I);
```

First

Syntax

```
First()
```

Description

The **First** method returns the first PSMMessage object in the PSMessages collection object executing the method.

Example

```
&ERROR = &PSMESSAGE.First();
```

Item

Syntax

```
Item(number)
```

Description

The **Item** method returns a PSMMessage object that exists at the *number* position in the PSMessages collection executing the method

Parameters

number Specify the position number in the collection of the PSMMessage object that you want returned.

Returns

A reference to a PSMMessage object, NULL otherwise.Example

```
&PSMSGCOL = &SESSION.PSMessages;
```

```

For &I = 1 to &PSMSGCOL.Count;

    &PSMESSAGE = &PSMSGCOL.Item(&I);

    /* do error processing */

End-For;

```

Next

Syntax

```
Next ()
```

Description

The **Next** method returns the next PSMessages object in the PSMessages collection object executing the method. You can only use this method after you have used either the **First** or **Item** methods: otherwise the system doesn't know where to start.

PSMessages Collection Property

Count

This property returns the number of PSMessages objects in the PSMessages collection object.

This property is read-only.

Example

```
&COUNT = &PSMESSAGE.Count;
```

PSMessage Class Properties

ExplainText

This property returns the explanation text (as a string) for the error message associated with the PSMessages object. You can use this property in conjunction with the **MessageNumber** property (which contains the error message number) and the **MessageSetNumber** property (which contains the error message set number.)



ExplainText should only be accessed when necessary because it requires a second database read or application server roundtrip.

This property is read-only.

MessageNumber

This property returns the error message number (as a number) for the error message associated with the PSMMessage object. You can use this property in conjunction with the **MessageSetNumber** property (which contains the error message set number) and the **ExplainText** property (which contains text explaining the error.)

This property is read-only.

MessageSetNumber

This property returns the error message set number (as a number) for the error message associated with the PSMMessage object. You can use this property in conjunction with the **MessageNumber** property (which contains the error message number) and the **ExplainText** property (which contains text explaining the error.)

This property is read-only.

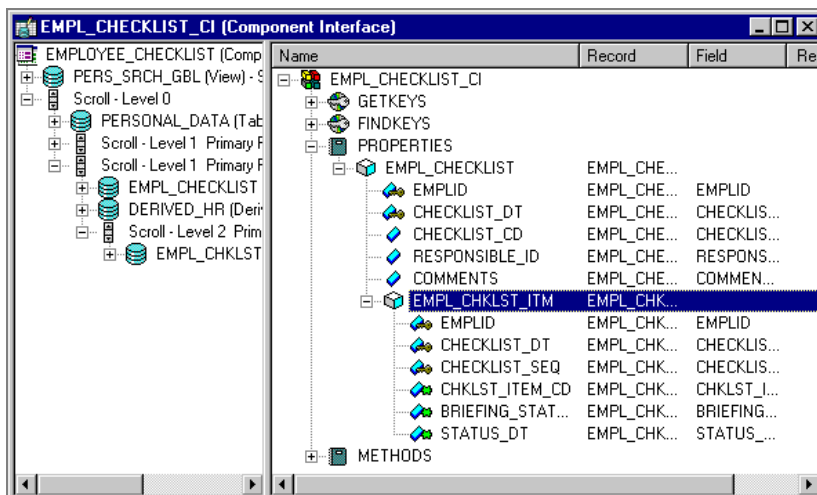
Source

This property returns a string indicating the actual field that's in error. The syntax of the string that's returned is as follows:

```
ComponentInterfaceName.[CollectionName(Row)].[CollectionName(Row)].[CollectionName(Row)]].PropertyName
```

This string is also returned as part of the **Text** property.

For example, suppose you had a Component Interface named EMPL_CHKLIST_CI.



Sample Component Interface

The following indicates that the first level field, EMPLID, has the error:

```
EMPL_CHKLIST_CI.EMPLID
```

The Component Interface EMPL_CHKLIST_BC has a data collection (scroll) called EMPL_CHKLIST_ITM. Suppose that it has 3 rows, and the STATUS_DT field was in error. The **Source** property would return the following string:

```
EMPL_CHKLIST_CI.EMPL_CHKLIST_ITM(3).STATUS_DT
```

You can use the **Source** property in conjunction with the **MessageNumber** property (which contains the error message number), the **MessageSetNumber** property (which contains the error message set number), the **ExplainText** property (which contains text explaining the error) and the **Text** property (which contains the text of the message.)

This property is read-only.

Example

The following example code finds the error messages and displays them to the user. It finds the field that caused the error and displays that as well.

```
Local ApiObject &PSMESSAGE;

Local boolean &FIND;

Local string &SOURCE, &FIELD, &TEXT;

Local number &START, &LEN, &NSTART, &FLEN;

.
.
.

For &I = 1 to &SESSION.PSMessages.Count;

    &PSMESSAGE = &SESSION.PSMessages.Item(&I);

    /* only display errors, not warnings, to user */

    If &PSMESSAGE.Type = 1 Then

        &TEXT = &PSMESSAGE.Text;

        /* find name of field in error */

        &SOURCE = &PSMESSAGE.Source;

        &LEN = Len(&SOURCE);

        /* find last dot before field */

        &FIND = False;

        &START = Find(".", &SOURCE);

        While Not (&FIND)

            &NSTART = Find(".", &SOURCE, &START + 1);
```

```

        If &NSTART = 0 Then

            &FIND = True;

        Else

            &START = &NSTART;

        End-If;

    End-While;

    &FLEN = &LEN - &START;

    &FIELD = Substring(&SOURCE, &START + 1, &FLEN);

    /* display text and field to user */

    Winmessage("You received the following error: " | &TEXT | "For field " |
&FIELD);

End-For;

```

Text

This property returns the text of the message (as a string) for the error message associated with the PSMMessage object. It also includes a contextual string that contains the name of the field that generated the error. The contextual string has the following syntax:

```
{ComponentInterfaceName.[CollectionName(Row).[CollectionName(Row).[CollectionName(Row)].Propertyname}
```

The contextual string (by itself) is available using the **Source** property of the PSMMessage.



For more information, see Source property.

You can use the **Text** property in conjunction with the **MessageNumber** property (which contains the error message number), the **MessageSetNumber** property (which contains the error message set number), and the **ExplainText** property (which contains text explaining the error.)

This property is read-only.

Example

```

Local ApiObject &MYSESSION, &COL, &ERROR;

Local string &TEXT;

.

.

```



```

.

If &MYSESSION.ErrorPending Then

    /* received error */

    &COLL = &MYSESSION.PSMessages;

    For &I = 1 to &COLL.Count

        &ERROR = &COLL.Item(&I);

        &TEXT = &ERROR.Text;

        /* do error processing */

    End-For;

End-if;

```

Regional Settings Class Properties

ClientTimeZone

This property specifies the client time zone. This property takes a string value.

This property is read-write.

CurrencyFormat

This property specifies how currency values are displayed. This property takes a number value. The default value is 1.

In the following valid values, @ represents the character specified with the CurrencySymbol property.

Value	Description
0	@1.1
1	1.1@
2	@.1.1
3	1.1.@

This property is read-write.

CurrencySymbol

This property specifies the currency symbol. This property takes a string value. The default value is "\$".

This property is read-write.

DateFormat

This property specifies the date format defaults. PeopleTools supports the Short Date Format specification (MDY, DMY, or YMD), and the Date separator setting. When you add a Date field in a record definition, you indicate whether you want to display the century. PeopleTools Date and DateTime fields always show leading zeros for month, day, and year.

This property takes a number value.

The valid values for this property are:

Value	Description
0	MDY (default)
1	DMY
2	YMD
3	Default value on system

This property is read-write.

DateSeparator

This property specifies the character used to separate the month, day, year in the date. This property takes a string value. The default value is "/".

This property is read-write.

DecimalSymbol

This property specifies the character used to separate the fractional of the number from the whole. This property takes a string value. The default value for this property is ".".



If you change the value of this property to a comma (,), you should adjust the DigitGroupingSymbol to a different character or some applications may not operate correctly.

This property is read-write.

DigitGroupingSymbol

This property specifies the character used as a thousand or other grouping separator. This property takes a string value. The default value for this property is ",".

This property is read-write.

LanguageCD

This property specifies the language to be used for the transaction. This property takes a string value. The default value for this property is the user's base language.

This property is read-write.



If you change this value dynamically using PeopleCode, this value will ***not*** change back to the original value when you disconnect. You must manually set it back to the original value yourself.

Example

The following example sets the language code, then at the end of the program, sets it back to the original value.

```
Local ApiObject &SESSION;

Local ApiObject &CI;

/** Get Session ApiObject */

&SESSION = GetSession();

&SESSION.Connect(1, "Existing", "", "", 0);

/** Get Message ApiObject */

&MSG = GetMessage();

&RS = &MSG.GetRowset();

/** Get Component Interface */

&CI = &SESSION.GetCompIntfc(compIntfc.NAMES_CI);

&REG_LNG = &SESSION.RegionalSettings.LanguageCd;

For &TRANS_NUM = 1 To &RS.RowCount

/** Store Language Code in Temp Var */
```

```

&LNG = &RS (&TRANS_NUM) .PSCAMA.LANGUAGE_CD.Value;

/** Set the Language for this Transaction */

If &LNG <> &REG_LNG Then

&SESSION.RegionalSettings.LanguageCd = &LNG;

End-If;

/* application specific processing */

&SESSION.RegionalSettings.LanguageCd = &REG_LNG;

&SESSION.Disconnect();

End-For

```

UseLocalTime

This property specifies whether the local time zone of the client should be used to format time fields. This property takes a Boolean value. The default value is True.

This property is read-write.

1159Separator

This property specifies the character used to separate hours/minutes/seconds for time represented on a 12-hour clock (that is, 11:00 PM.) This property takes a string value. The default is "."

This property is read-write.

2359 Separator

This property specifies the character used to separate hours/minutes/seconds for time represented as a 24-hour clock (that is, 23:00.) This property takes a string value. The default is "."

This property is read-write.

Trace Setting Class Properties

Each of the following properties relates to the Trace setting tab on the Configuration Manager.



For more information about the different options, see Configuration Manager.

API

This property is part of the PeopleSoft API trace and sets the **API Information** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

COBOLStmtTimings

This property is part of the SQL trace and sets the **COBOL Statement Timings** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

ConnDisRollbackCommit

This property is part of the SQL trace and sets the **Connect, disconnect, rollback and commit** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

DBSpecificCalls

This property is part of the SQL trace and sets the **Database API-specific Calls** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

ManagerInfo

This property is part of the SQL trace and sets the **Manager Information** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

NetworkServices

This property is part of the SQL trace and sets the **Network Services** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

NonSSBs

This property is part of the SQL trace and sets the **All Other API Calls besides SSBs** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

OutputUNICODE

This property specifies if the trace file is written in ANSI or UNICODE (i.e., **Output Unicode Trace File**).

This property takes a Boolean value: True means the file will be written in UNICODE, False means it will be ANSI.

This property is read-write.

PCExtFcnCalls

This property is part of the PeopleCode trace and sets the **Trace External Function Calls** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

PCFcnReturnValues

This property is part of the PeopleCode trace and sets the **Show Function Return Value** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

PCFetchedValues

This property is part of the PeopleCode trace and sets the **Show Fetched Values** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

PCIntFcnCalls

This property is part of the PeopleCode trace and sets the **Trace Internal Function Calls** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

PCListProgram

This property is part of the PeopleCode trace and sets the **List the Program** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

PCParameterValues

This property is part of the PeopleCode trace and sets the **Show Parameter Values** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

PCProgramStatements

This property is part of the PeopleCode trace and sets the **Trace Each Statement in Program** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

PCStack

This property is part of the PeopleCode trace and sets the **Show Stack** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

PCStartOfPrograms

This property is part of the PeopleCode trace and sets the **Trace Start of Programs** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

PCTraceProgram

This property is part of the PeopleCode trace and sets the **Trace Evaluator Instructions** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

PCVariableAssignments

This property is part of the PeopleCode trace and sets the **Show Assignments to Variables** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

RowFetch

This property is part of the SQL trace and sets the **Row Fetch** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

SQLStatement

This property is part of the SQL trace and sets the **SQL Statement** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

SQLStatementVariables

This property is part of the SQL trace and sets the **SQL Statement Variables** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

SSBs

This property is part of the SQL trace and sets the **Set Select Buffers** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

SybBindInfo

This property is part of the SQL trace and sets the **Sybase Bind Information** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

SybFetchInfo

This property is part of the SQL trace and sets the **Sybase Fetch Information** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

TraceFile

This property specifies the location of the trace file (that is, **PeopleTools Trace File** value.)

This property takes a string value.

This property is read-write.

SQL Class

You can create SQL definitions in Application Designer. These can be entire SQL programs, or just fragments of SQL statements that you want to re-use. PeopleCode provides the SQL class for accessing these SQL definitions in your PeopleCode program at runtime.

The SQL class provides capability beyond that offered by SQLExec. Unlike SQLExec, which fetches just the first SELECTed row, operations for the SQL class allow iteration over all rows fetched. This can dramatically improve performance if you're doing a million operations and you've set the BulkMode property to True.

A list of input (bind) values, and a list of output variables are supported, much as they are in SQLExec. The input and output variables are limited to the same PeopleCode types that can be used with SQLExec, with the addition of a new class called Record.

At runtime, you instantiate a record object from the Record class. A record object is a "one row" instantiation of a record definition.

- When used as an output from an SQL object, each next field in the record is populated with the next column returned.
- When used as an input, several fields in the record are used to replace a single bind marker in the SQL statement. The bind marker for a record describes the kind of substitution to be done.

Both records and other PeopleCode types can be mixed in both the output and input.



For more information, see ProcessRequest Class, Data Buffer Access

At runtime, you instantiate a SQL object from the SQL class. The SQL object is loaded by either a constructor for the object, or an explicit **Open** method call. Optionally, a SQL constructor and the **Open** method support setting the SQL statement through a string parameter. This capability allows ad-hoc SQL statements to be created and executed.

The SQL class has a **Fetch** method for iterating through the rows fetched by a select. A *cursor* is used to control this connection between the runtime SQL object and the database. The cursor is closed automatically when the object goes out of scope. The cursor can be closed before that by using the **Close** method. The status of the connection is available from the Boolean **IsOpen** property.



For more information, see Global SQL Objects and Application Engine Programs

The general status of operations is available from function or method return values or properties. Detailed status is not available, as the operations are designed to terminate on unexpected errors or errors that cannot be reasonably recovered from by application level logic.

Record Class SQL

There are some SQL type operations that you can do with the Record class, such as INSERT, DELETE and UPDATE. The advantage of using the Record class methods is ease of use, re-use of code, and so on. However, if you're doing many iterations of the same operation (like a million UPDATES) you should use the SQL object with the BulkMode property set to True.

The SQL object maintains a state (that is, a cursor). Hence, if your database can take advantage of BulkMode, instead of a million operations, the commands are "bulked up" and committed only once. This can improve performance dramatically.

Creating a SQL Definition

You can create SQL definitions in Application Designer, using the SQL editor. These SQL statements can be entire SQL programs, or just fragments that you want to re-use. Once you have created a SQL definition, you can use it to populate a SQL object (using FetchSQL, Open or GetSQL)



For more information, about using the SQL editor in Application Designer, see Introducing the SQL Editor.

You can also create a SQL statement in PeopleCode (using CreateSQL), save it as a SQL definition (StoreSQL), then access it in Application Designer.

Binding and Executing of SQL Statements

The processing of an SQL statement involves a series of steps.

1. The binding process is the replacement of (variable) input values in the statement, in places indicated by bind placeholders.

The input values are substituted into the SQL statement in place of the bind placeholders. These placeholders have the form `":number"`, or `"%bindop(:number [, parm]...)"` where the *number* indicates which input value is to be substituted, and the *bindop* and *parm* strings indicate what meta-SQL binding function is to be performed.



There must be no spaces between the *bindop* and the left parenthesis.

The following binding meta-SQL functions are used with record objects to substitute various forms of fieldnames and values into the SQL statement. The goal of these binding functions is to enable the writing of SQL and PeopleCode that can manipulate records without dependencies on the exact fields in the records.

- %DTTM
- %InsertValues
- %KeyEqual
- %KeyEqualNoEffDt
- %List
- %OldKeyEqual
- %Table
- %UpdatePairs



For more information, see Meta-SQL.

2. The execution of an SQL statement is the carrying out of the operation of the statement by the database engine.

This view of SQL statement processing is actually simpler than what actually occurs. In actual practice, the binding occurs in two distinct phases. The database engine (or its supporting routines) is aware of only the latter phase. For some operations, some database engines are able to delay the second stage of binding and the execution of an SQL statement, so the statement can be rebound and re-executed (with different input values). The advantage of this is that these bindings and executions can be accumulated and transmitted across the network to the database server, with several database operations being done in one network operation. This is sometimes referred to as *bulk mode*. Since the network time dominates the time taken by most database operations, the performance advantages of bulk mode can be significant.

However, in bulk mode, individual operations might not be executed immediately by the database engine. The result is that the application might not see errors that arise until later operations are performed.

SQL SELECT statements are not bound multiple times, rather the retrieved rows are accumulated and sent across the network many rows at a time, thereby also decreasing the effect of network delays.

Fetching from a Select

Some operations fetch a row of data into a list of output variables. The values of the fields from the select row are assigned in order to the given output variables. If one of these is a reference to a PeopleCode record object, the fields in the record will be assigned the successive values from the row of the select, until all the fields are assigned. Assignment then continues with the next output variable, if any. The number of output fields and variables must equal the number of fields in the row of the select.



For more information, see Data Buffer Access.

Different "Styles" of SELECT

Using the SQL object, you have different methods of using a SELECT statement in your code, depending on your application.

The most direct way is with an Open and Fetch. This will fetch successive rows, and you can process each row after it's fetched.

The pseudo code for this type of select statement is as follows:

```
&SQL.Open("Select . . .", bindpars);

While &SQL.Fetch(resultpars)

    /* process result row */

End-While;
```

Another method is to set up the loop based on something you want to change, and while you still want to change things, continue the loop.

The pseudo code for this type of select statement is as follows:

```
&SQL.Open("Update/Insert/Delete/other. . . :n . . ."); /* Bind references,
but no bind parameters */

While you_want_to_do_changes

    /* set up for changes */

    &SQL.Execute(bindpars);

End-While;
```

Reusing a Cursor

Reusing a cursor means *compiling* a SQL statement just once, but *executing* it more than once. When a SQL statement is compiled, the database checks the syntax and chooses a query path. By only doing this once, but executing this statement several times, you can see an improvement in performance.

In PeopleCode you can reuse a cursor, but there are several conditions:

- You have to use the SQL object. Only SQL objects retain SQL cursor information.
- For INSERT, DELETE and UPDATE statements, you automatically reuse the cursor as long as you don't change the SQL statement as part of the binding process. As long as the SQL statement is textually the same, so only the binds have changed, you get reuse. For example, you won't get reuse if you first bind one kind of record object to %Insert(), then bind another, different kind of record object.
- For SELECTs, you *must* use the meta-SQL shortcuts (%SelectAll, %SelectByKey or %SelectByKeyEffDt), *and* the CreateSQL or GetSQL functions without supplying any bind of buffer parameters. The bind parameters will be supplied in the Execute function. The Fetch parameter must be the fetch buffer, and be the same as the first Execute parameter. You can supply WHERE clauses, ORDER-BY, and so on, on the end of the SQL string containing the meta-SQL.



BulkMode doesn't reuse the cursor in the same way as the SQL object. BulkMode requests that, when the system can reuse the cursor, it also holds all the changes so they can be communicated to the database in batches.

By using one of the forms of the Select meta-SQL, you're guaranteeing the resulting fetched values are all put into one record object (buffer). This means the implementation doesn't have to ask the database for the length and type of each column: the record buffer is already defined. So this sample code uses the SQL object to maintain the state of the connection with the database, and the record object, to maintain a series of fields suitable for database operations.

```
Local SQL &SQL;

Local Record &REC;

&REC = CreateRecord(Record.KP_KPI_DFN);

/* start with select statement, no bind refs, no bind parameters */

&SQL = CreateSQL("%Selectall(:1) Where SETID = :2 and KPI_ID = :3 and EFFDT =
(SELECT MAX(EFFDT) FROM PS_KP_KPI_DFN WHERE SETID = :4 AND KPI_ID = :5 AND EFFDT
<= %DateIn(:6))");
```

Start Loop

```

/* bind and execute the statement */

&SQL.Execute(&REC, &SETID_KPI, &COMPID, &SETID_KPI, &COMPID, &EFFDT);

/* Note record parameter for Fetch statement must be the same as the first
Execute parameter - the results are in this record */

If &SQL.Fetch(&REC) Then

    /*process this record */

End-If;

```

End Loop

```

&SQL.Close(); /* there is no implicit close on a Fetch returning False */

```

The following example comes from an Application Engine program. Because the loop goes in and out of PeopleCode, you need to declare the SQL object as Global. This example is in three parts.

3. Program called by Application Engine before the loop:

```

Global SQL &SQL;

&SQL = CreateSQL("INSERT INTO %Table(MY_WORK) (TRANS , REGISTERWRITE) VALUES
(:1, :2)");

&SQL.BulkMode = True; /* not required for reuse, but will get better performance
on platforms that support bulk insert */

```

4. Program called in the Application Engine loop on SELECT 'X' FROM PSRECDEFN WHERE RECNAME LIKE '%A':

```

Global SQL &SQL;

&var1 = "X";

```

```
&var2 = "Y";
```

```
&SQL.Execute(&var1, &var2);
```

5. Program called by Application Engine after the loop:

```
Global SQL &SQL;
```

```
&SQL.Close();
```

Global SQL Objects and Application Engine Programs

A global variable won't go out of scope until an Application Engine program finishes. However, a global SQL object will be forced closed (i.e., the cursor closed) sooner than that. **Any open global SQL object will be forced closed just before any checkpoint in an Application Engine program.** This is to ensure that the application can be restarted successfully from the checkpoint. After the SQL object is closed, you can reopen the SQL object, or query its properties, (**Status**, **IsOpen**). In the absence of an intervening checkpoint, an open global SQL object will remain open until the Application Engine program finishes.

Declaring a SQL Object

SQL objects are declared as type SQL. For example:

```
Local SQL &MYSQL;
```

Scope of an SQL Object

An SQL object can only be instantiated from PeopleCode.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Application Engine PeopleCode, record field PeopleCode, etc.

Sometimes your SQL statement will change the database. When your SQL changes the database, your code should only be in one of the following events:

- SavePreChange
- WorkFlow
- SavePostChange
- Message Subscription
- FieldChange
- Application Engine PeopleCode action

SQL Class Built-in Functions

CreateSQL

DeleteSQL

FetchSQL

GetSQL

StoreSQL

SQL Class Methods

Close

Syntax

```
Close()
```

Description

The **Close** method closes the SQL object. This terminates any incomplete fetching of select result rows, completes any buffered operations (that is, using **BulkMode**), and disassociates the SQL object from any SQL statement that was open on it.

After **BulkMode** operations, the **RowsAffected** property is not valid.

Parameters

None

Returns

True on successful completion, False if there was a duplicate record error. Any errors associated with buffered operations (that is, using **BulkMode**), other than duplicate record errors, cause termination.

Example

```
&SQL = CreateSQL("%Delete(:1)");

While /* Still something to do */

    /* Set key field values of &ABS_HIST */

    &ABSSQL.Execute(&ABS_HIST);

End-While;

&SQL.Close();
```


Related Topics

Open, CreateSQL function

Execute

Syntax

```
Execute(paramlist)
```

Where *paramlist* is an arbitrary-length list of values in the form:

```
inval1 [, inval2] ...
```

Description

The **Execute** method executes the SQL statement of the SQL object. The SQL object must be open and unbound on a delete, insert or update statement. That is, the CreateSQL, GetSQL, or Open preceding the Execute must have specified a delete, insert or update statement with bind placeholders and must not have supplied any input values.

The values in *paramlist* are used to bind the SQL statement before it gets executed.

When using the optional **BulkMode**, the **Execute** operations may be buffered and are not guaranteed to have been presented to the database until a **Close** is done. Thus, in **BulkMode**, errors that arise may not be reported until later operations are done.



For more information, see BulkMode.

Parameters

<i>paramlist</i>	Specify input values for the SQL string.
------------------	--

Returns

True on successful completion, False for "record not found" and "duplicate record" errors. Any other errors cause termination.

Example

The following example creates a SQL object for inserting. The statement isn't automatically executed when it's created because there aren't any bind variables. The **Execute** occurs after other processing is finished. The name of the record is passed in as the bind variable in the Execute method.

```
&SQL = CreateSQL("%Insert(:1)");

While /* Still something to do */

    /* Set all the field values of &ABS_HIST. */
```

```

        &SQL.Execute(&ABS_HIST);

End-While;

&SQL.Close();

```

The following example creates two SQL objects, one to be used for fetching, the other for updating the record. The first SQL object selects all the records in the &ABS_HIST record that match &EMPLID. The data is actually retrieved using the Fetch method. After values are set on the record, the update is performed by the **Execute**.

```

&SQL1 = CreateSQL("%Select(:1) where EMPLID = :2", &ABS_HIST, &EMPLID);

&SQL_UP = CreateSQL("%Update(:1)");

While &SQL1.Fetch(&ABS_HIST);

    /* Set some field values of &ABS_HIST. */

    &SQL_UP.Execute(&ABS_HIST);

End-While;

&SQL_UP.Close();

```

The following is an example of inserting an array of records:

```

Local SQL &SQL;

Local array of Record &RECS;

/* Set up the array of records. */

. . .

/* Create the SQL object open on an insert statement, and unbound*/

&SQL = CreateSQL("%Insert(:1)");

/* While the array has something in it... */

While &RECS.Len

    /* Insert the first record of the array, and remove it from the array.
    */

    &SQL.Execute(&RECS.Shift());

End-While;

```

Related Topics

Close, CreateSQL function

Fetch

Syntax

```
Fetch(paramlist)
```

Where *paramlist* is an arbitrary-length list of values in the form:

```
outvar1 [, outvar2] ...
```

Description

The **Fetch** method retrieves the next row of data from the SELECT that is open on the SQL object. Any errors result in termination of the PeopleCode program with an error message.

If there are no more rows to fetch, **Fetch** returns as False, the *outvars* are set to their default PeopleCode values, and the SQL object is automatically closed.

Using **Fetch** with a closed SQL object is processed the same as when there are no more rows to fetch.



If you only want to fetch a single row, the SQLExec function can perform better, since it will only fetch a single row from the server. For more information, see SQLExec.

The return of **Fetch** is not optional, that is, you *must* check for the value of the fetch.

Setting Data Fields to Null

This method will *not* set Component Processor data buffer fields to NULL after a row not found fetching error. However, it does set fields that aren't part of the Component Processor data buffers to NULL. It does set work record fields to NULL.

Parameters

paramlist Specify output variables from the SQL Select statement.

Returns

The result of Fetch is True if a row was fetched. If there are no more rows to fetch, the result is False.

Example

In the following example, the **Fetch** method is used first to process a single row, then to process the ABS_HIST record.

```
Local SQL &SQL;

Local Record &ABS_HIST;

&ABS_HIST = CreateRecord(RECORD.ABSENCE_HIST);
```

```

&SQL = GetSQL(SQL.SEL27, 15, "Smith");

While &SQL.Fetch(&NAME1, &BIRTH_DT)

    /* Process NAME1, BIRTHDT from the selected row. */

End-While;

&SQL.Open(SQL.SEL_ABS_HIST, &NAME1, "Smith");

While &SQL.Fetch(&ABS_HIST)

    /* Process ABS_HIST record. */

End-While;

```

The following is an example of reading in an array of record objects:

```

Local SQL &SQL;

Local Record &REC;

Local Array of Record &RECS;

/* Get the SQL object open and ready for fetches. */

&SQL = CreateSQL("%SelectAll(:1) where EMPLID = :2", RECORD.ABSENCE_HIST,
&EMPLID);

/* Create the first record. */

&REC = CreateRecord(RECORD.ABSENCE_HIST);

/* Create an empty array of records. */

&RECS = CreateArrayRept(&REC, 0);

While &SQL.Fetch(&REC)

    /* We got a record, add it to the array and create another.*/

    &RECS.Push(&REC);

    &REC = CreateRecord(RECORD.ABSENCE_HIST);

End-While;

```

Related Topics

Open, GetSQL function

Open

Syntax

```
Open(sql [, paramlist])
```

Where *paramlist* is an arbitrary-length list of values in the form:

```
inval1 [, inval2] ...
```

Description

The **Open** method associates the *sql* statement with the SQL object. The *sql* parameter can be either:

- a string value giving the SQL statement
- a reference to a SQL definition in the form **SQL.sqlname**.

If the SQL object was already open, it is first closed. This terminates any incomplete fetching of select result rows, completes any buffered operations (that is, using **BulkMode**), and disassociates the SQL object from any SQL statement that was open on it.

Opening and Processing sql

If *sql* is a SELECT statement, it is immediately bound with the *inval* input values and executed. The SQL object should subsequently be the subject of a series of **Fetch** method calls to retrieve the selected rows. If you want to only fetch a single row, use the **SQLExec** function instead. If you want to fetch a single row into a PeopleCode record object, use the record **Select** method.

If *sql* is not a SELECT statement, and *either*: there are some *inval* parameters, *or* there are no bind placeholders in the SQL statement, the statement is immediately bound and executed. This means that there is nothing further to be done with the SQL statement and the **IsOpen** property of the returned SQL object will be False. In this case, using the **SQLExec** function would generally be more effective. If you want to delete, insert, or update a record object, use the record **Delete**, **Insert**, or **Update** methods with the record object.

If *sql* is not a SELECT statement, there are no *inval* parameters, *and* there are bind placeholders in the SQL statement, the statement is neither bound nor executed. The resulting SQL object should subsequently be the subject of a series of **Execute** method calls to affect the desired rows.

Setting Data Fields to Null

This method will *not* set Component Processor data buffer fields to NULL after a row not found fetching error. However, it does set fields that aren't part of the Component Processor data buffers to NULL.

Parameters

<i>sql</i>	Specify <i>either</i> a SQL string <i>or</i> a reference to a SQL definition in the form SQL.sqlname .
------------	---

paramlist Specify input values for the SQL string.

Returns

None.

Example

Generally, you'll only use the Open method after you've already gotten a reference to another SQL object. SELECT and SEL_ABS_HIST are the names of the SQL definitions created in Application Designer.

```
Local SQL &SQL;

&SQL = GetSQL(SQL.SELECT);

/* do other processing */

/* get next SQL statement for additional processing */

/* The open automatically closes the previous SQL statement */

&SQL.Open(SQL.SEL_ABS_HIST, &NAME1, "Smith");

While &SQL.Fetch(&ABS_HIST)

    /* Process ABS_HIST record. */

End-While;
```

Related Topics

GetSQL and CreateSQL functions

SQL Class Properties

BulkMode

This property controls the use of bulk mode. Setting this property to True will enable the use of bulk mode, and hence remove any guarantee of the synchronous presentation of error status.



For more information, see Binding and Executing of SQL Statements.

Bulk mode will be used only with those database connections and operations that support it. Bulk mode can be used with any SQL operation, that is, with INSERTs, DELETEs, or UPDATEs.

If you're using an Application Engine program, and have set this property to True, the rows inserted in BulkMode will be committed at the next database commit in your program.

After **BulkMode** operations, the RowsAffected property is not valid.

The default value for **BulkMode** is False.

This property is read-write.

Example

The following code is an example of inserting an array of records using bulk mode:

```
Local SQL &SQL;

Local array of Record &RECS;

/* Set up the array of records. */

. . .

/* Create the SQL object open on an insert statement, and unbound.*/

&SQL = CreateSQL("%Insert(:1)");

/* Try for bulk mode. */

&SQL.BulkMode = True;

/* While the array has something in it... */

While &RECS.Len

    /* Insert the first record of the array, and remove it from the array.
    */

    If not &SQL.Execute(&RECS.Shift) then

        /* A duplicate record found, possibly in bulk mode. There is no way
        to tell which record had the problem. One approach to recovery is to
        fail the transaction and retry it with a process that does only one
        record at a time, that is, doesn't use bulk mode. */

        . . . ;

    End-If;

End-While;
```

IsOpen

This property returns as True if the SQL object is open on some SQL statement.

This property is read-only.

Example

You might use the following in an Application Engine program, after a checkpoint. MYSELECT is the name of a SQL definition created in Application Designer:

```
If Not &MYSQL.IsOpen Then

    &MYSQL.Open (SQL.MYSELECT) ;

End-if;
```

RowsAffected

This property returns the number of rows affected by the last INSERT, UPDATE, or DELETE statement of the SQL object. After BulkMode operations, the **RowsAffected** property is not valid.

This property is read-only.

Example

The following code is an example that determines if a delete statement actually deleted anything:

```
Local SQL &SQL;

/* Create the SQL object and do the deletion. */

&SQL = CreateSQL("Delete from %Table(:1) where EMPLID = :1",
RECORD.ABSENCE_HIST, &EMPLID);

If &SQL.RowsAffected = 0 Then

    /* We didn't delete any rows. */

    . . .

End-If;
```

Status

This property returns the status of the last statement executed. You can use either the constant or the numeric value for this property. The valid values for this property are:

Constant	Value	Description
%SQLStatus_OK	0	No Errors

Constant	Value	Description
%SQLStatus_NotFound	1	Record Not Found
%SQLStatus_Duplicate	2	Duplicate Record Found

This property is read-only.

Example

The following example finds out what went wrong after an update:

```

Local SQL &SQL;

Local Record &NEWREC, &OLDREC;

/* Create and initialize &OLDREC with the keys of the record to be updated.
Create and initialize &NEWREC with the new field values for the record. */
...;

/* Create and execute the update. */

&SQL = CreateSQL("%Update(:1, :2)", &NEWREC, &OLDREC);

Evaluate &SQL.Status

When = %SQLStatus_OK

    /* It worked. */

When = %SQLStatus_NotFound

    /* The OLDREC keys were not found. */

When = %SQLStatus_Duplicate

    /* The NEWREC keys were already there. */

End-Evaluate;

```

TraceName

This property allows you to assign a name to a SQL statement that has been created in PeopleCode using CreateSQL. This name will be used in the Application Engine timings trace. This property takes a string value.



You can *not* associate the `TraceName` property with the execution of a simple `SELECT` statement created with `CreateSQL`. This is because the `SELECT` is executed when the SQL is created, before it has the `TraceName` assigned. If you want to do this, create a SQL object instead.

If this property isn't set, it defaults to a substring of the SQL statement, indicating the operation and table, (for example, `SELECT PS_VOUCHER_LINE`.) It may be useful to set `TraceName` to indicate the origin of the SQL statement.

This property is read-write.

Example

```
&REC = CreateRecord(Record.VOUCHER_LINE);

&SQL = CreateSQL("%selectall (:1) WHERE BUSINESS_UNIT =:2 AND VOUCHER_ID =:3 AND
VOUCHER_LINE_NUM = :4");

&SQL.TraceName = "AEPROG.SECT1.STEP1.SQL2";

&SQL.Execute(&REC, MATCHING_AET.BUSINESS_UNIT, MATCHING_AET.VOUCHER_ID, &count);

If &SQL.Fetch(&REC) Then

    &count2 = &count2 + 1;

End-If;
```

The above example would produce the following in the timings trace.

SQL Statement	Count	Time	Count	Time	Count	Time	Time

PeopleCode							
AEPROG1.SECT1.STEP1.SQL1			169	5.371	169	3.083	4.000
AEPROG1.SECT1.STEP1.SQL2			100	5.371	100	3.083	4.454

							8.454

You can use this parameter with the `Open` statement as well. The following is an example of how this works:

```
&Sql1 = CreateObject("SQL");
```

```

&Sql1.TraceName = "sql1";

&Sql1.Open("Select %FirstRows(1) 'x' FROM psstatus");

&Sql1.Fetch(&Temp);

&Sql2 = CreateObject("SQL");

&Sql2.TraceName = "sql2";

&Sql2.Open("Select 'x' FROM psstatus");

&Sql2.Fetch(&Temp);

```

Value

This property returns the SQL statement associated with the SQL object as a string.

This property is read-only.

Example

If you wanted to report an error in a SQL definition, including the actual SQL executed, you can use the Value property to get the text of the SQL statement:

```

Local SQL &SQL;

/* Execute some SQL. */

&SQL = CreateSQL(SQL.SOMESQL, &EMPLID);

If &SQL.Status = %SQLStatus.NotFound Then

    /* Get the SQL string used. */

    &SQLSTR = &SQL.Value;

    /* Report the error. */

    . . . ;

End-If;

```

Tree Classes

Using the Tree classes in your PeopleCode, you have access to all the functionality of Tree Manager. Your application should instantiate the appropriate tree objects when it needs to work with the tree system database data, call the appropriate methods and properties, then close the objects when it is finished.

Creating or deleting a tree **object** does not create or delete tree system database information. You must call the method for that tree object directly to create or delete database information, that is, the **Create** or **Delete** method.

One instance of a tree or tree structure object can be created to work on multiple database entities. However, only one tree or tree structure can be open at a time. If you open a Tree before closing the one that's currently open, you will receive an error. You must explicitly close a tree or tree structure (using the **Close** method) before you try to open a second one.

All of the classes, and most of the properties and methods that make up the Tree Classes have a GUI representation in Tree Manager. This document assumes that the reader has working knowledge of Tree Manager.



For more information, see Tree Manager.

The following classes make up the Tree classes:

- Branch Collection
- Leaf
- Level
- Level Collection
- Node
- Tree
- Tree Collection
- Tree Structure
- Tree Structure Collection

With most of the classes of objects in PeopleTools, when you use a **Getxxx** method, you are fully instantiating an object. However, for the tree class, when you use **GetTree** (from the session object), you get a closed tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method. Working with closed trees can improve your performance. The same applied to a tree structure: it's closed when you get it from the session object, and you must open it before you can access its properties or change it.

There aren't any built-in functions for the Tree classes: objects are instantiated from identifiers, from other objects, or from a session object.



For more information, see Search Classes.

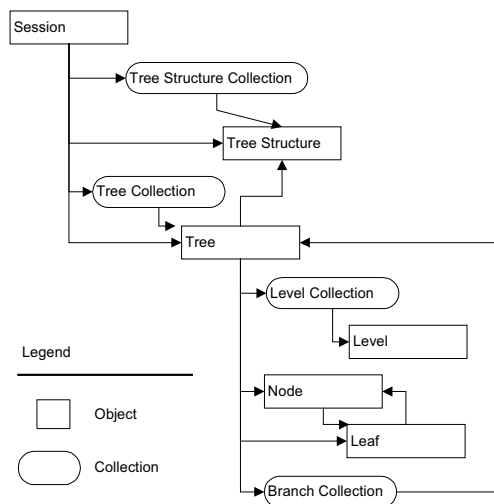
Using the GenerateTree function, you can produce a GUI representation of a tree in the PeopleSoft Internet Architecture.



For more information, see Using the GenerateTree Function.

Relationships between Different Tree Classes

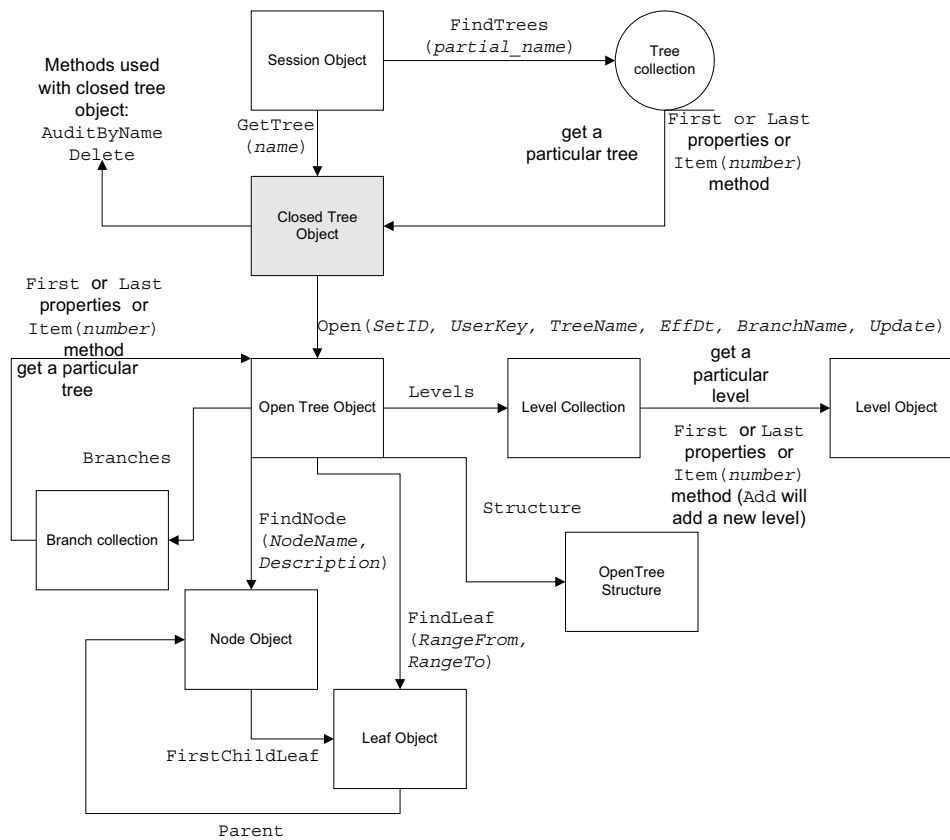
The following diagram shows the Tree classes and their relationships. Objects at the tip of the arrowhead are accessed or created from the object at the left of the arrowhead. Squares indicate objects. Ovals indicate collections.



Tree Classes and their relationships

- From the session object, you can get identifiers for a tree or a tree structure, or you can instantiate a tree collection or a tree structure collection.
- From the tree collection object, you can get an identifier for a tree.
- From the tree structure collection object, you can get an identifier for a tree structure, or you can instantiate a tree structure object.
- From the level collection object, you can instantiate a level object.
- From the branch collection object, you can get an identifier for a tree.
- From the tree object, you can instantiate a tree, a tree structure, a branch collection, a leaf, a node, and a level collection.
- From the node object, you can instantiate a leaf object or a node object.
- From the leaf object, you can instantiate a leaf object or a node object.

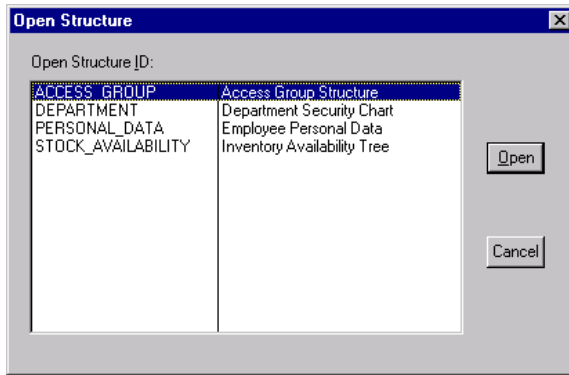
Here's another diagram, listing the different tree classes, with one of the properties or methods used for instantiating one class from another.



Tree Classes flow with methods or properties

Collections in the Tree Classes

A *collection* is a set of similar things, like a group of already existing tree structures or trees. Like everything else in the tree classes, collections have a GUI representation. For example, when you select **Structure**, **Open** from Tree Manager, you get a dialog that lets you select the tree structure you want. This dialog's data is represented in PeopleCode as the *tree structure collection*.



Collection of Tree Structures

The following collections are part of the Tree classes:

- Branch collection
- Level collection
- Tree collection
- Tree structure collection

Error Handling with Trees

The Tree classes log descriptive information regarding errors and warnings to the PSMessages collection, instantiated from a session object.



Warning! For this release, errors that occur in 3-tier mode or in PeopleSoft Internet Architecture do *not* get sent to the PSMessages collection. Instead, they're reported back to the calling application, that is, for 3-tier they get displayed as messages on the application server, and in PIA they come back in the generated HTML.

In your program, use the PSMessages collection to identify and report to the user any errors that are encountered during processing.



For more information, see About Error Handling.

The tree classes log errors "interactively", that is, as they happen. For example, suppose you had created a new tree, and were setting the effective date using the effective date property (**EffDt**). If you used an invalid value for the effective date, the error would be logged in the PSMessages collection as soon as you set the value, not when you saved the tree.

When you want to check for errors depends on your application. However, if you check for errors after every assignment, you may see a performance degradation.

One way to check for errors is to check the number of messages in the PSMessages collection, using the **Count** property. If the Count is 0, no error or warning messages have been logged to the PSMessages collection.

```
Local ApiObject &MYSESSION;

Local ApiObject &ERRORCOL;

Local ApiObject &TREE, TREELIST;

Local boolean &RSLT;

&MYSESSION = GetSession();

&RSLT = &MYSESSION.CONNECT(1, "EXISTING", "", "", 0);

If &RSLT Then

    /* connection is good */

    &TREELIST = &MYSESSION.FindTree("", "", "DEPARTMENT", "", "");

    For &I = 1 to &TREELIST.Count

        &TREE = &TREELIST.Item(&I);

        /* Do processing */

        &TREE.Save();

        /* Do error checking */

        &ERRORCOL = &MYSESSION.PSMessages;

        If (&ERRORCOL.Count <> 0) Then

            /* errors occurred - do processing */

        Else
```



```

                /* no errors */

            End-If;

        End-For;

    Else

        /* do processing for no connection */

    End-If;

```

Verifying Leaves and Nodes being Inserted

InsertChildLeaf, InsertSibNode, InsertChildNode, and so on, can return False or a Null object reference, depending upon the type of error encountered. You may want to declare references to new leaves or nodes as type ANY until after you verify they were actually created. You can do this by using the None or All PeopleCode functions in order to determine whether a valid Leaf or Node Object was created. If a Leaf or Node object was created ALL() returns True and None() returns False.

```

&NewLeaf = &RootNode.InsertChildLeaf("8000", "8999");

If NONE(&NewLeaf) Then

    /* Leaf not inserted, do error processing */

    &Messages = &Session.PSMessages;

    If &Messages.WarningPending Then

        /* do error processing */

    End-if;

End-if;

```

Declaring Tree Objects

All tree objects, like a tree, a tree structure, a node, a level, and so on, are declared as type ApiObject. For example,

```

Local ApiObject &MYTREE;

Global ApiObject &MYNODE;

```

All tree objects can be declared as Local, Component or Global.

Scope of the Tree Objects

All tree objects, that is, trees, tree structures, nodes, leaves, and so on, can *only* be instantiated from PeopleCode.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Application Engine PeopleCode, record field PeopleCode, and so on.

You can only instantiate a tree or tree structure object from a session object. You have to instantiate the session object, and connect to the database, before you can instantiate a tree or tree structure.

```
Local ApiObject &TREELIST;

Local ApiObject &MYSESSION;

Local boolean &RSLT;

&MYSESSION = GetSession();

&RSLT = &MYSESSION.CONNECT(1, "EXISTING", "", "", 0);

If &RSLT Then

    /* connection is good */

Else

    /* do error processing */

End-if;
```



Tree Classes are only accessible through PeopleCode.

Implementing Tree Classes

You will often want to create a new tree. The following procedure covers this action in more detail.

The TreeMover Application Engine program uses the Tree API (and File Layouts) for importing a tree from a flat file or exporting a tree to a flat file.



For more information, see Tree Mover.

To create a new tree:

In this example, you are creating a new tree based on an existing tree structure. The following is the complete code sample: the steps explain each line.

```
Local ApiObject &Session;

Local ApiObject &TreeList, &MyTree;

Local ApiObject &LvlColl;

&Session = GetSession();

&Session.CONNECT(1, "EXISTING", "", "", 0);

&TreeList = &Session.findtree("", "", "", "", "");

If ALL(&TreeList) Then

    &MyTree = &TreeList.first;

    /* create new tree */

    &TreeReturn = &MyTree.Create("", "", "PERSONAL_DATA2", "1999-06-01",
"PERSONAL_DATA");

    &MyTree.description = "test tree";

    /* add level */

    &LvlColl = &MyTree.levels;

    &Level = &LvlColl.add("FIRST LVL");

    &Level.description = "First Level";

    /* add node */

    &MyTree.insertroot("00001");

    /* add new leaf */
```

```

&RootNode = &MyTree.FindRoot();

If ALL(&RootNode) Then
    &NewLeaf = &RootNode.InsertChildLeaf("8000", "8999");

    /* save new tree */
    &RSLT = &MyTree.Save();

    /* Do error checking */
    If Not (&RSLT) Then
        /* errors occurred = do error checking */
        &ERRORCOL = &Session.PSMessages;
        For &I = 1 To &ERRORCOL.count
            /* do error processing */
        End-For;
    Else
        /* no errors - saved correctly - do other processing */
    End-If;
End-if;

End-if;

```

1. Get a session object.

Before you can get a tree, you have to get a session object. The session controls access to the tree, provides error tracing, allows you to set the runtime environment, and so on.

```

&Session = GetSession();

&Session.CONNECT(1, "EXISTING", "", "", 0);

```

2. Get a tree object.

The **FindTree** method specified with no parameters returns a collection of all the trees already created. To ensure you have a valid tree list, use the ALL built-in function. Because we're creating a new tree, it doesn't matter which tree we get originally, so use the **First** method to get the first tree in the collection.

```
&TreeList = &Session.findtree("", "", "", "", "");
```

```
If ALL(&TreeList) Then
```

```
    &MyTree = &TreeList.First;
```

After you execute the **First** method, you only have the **structure** of the tree, that is, a closed tree. You haven't populated the tree with data yet.

3. Create the Tree.

The **Create** method creates a new tree with the name PERSONAL_DATA2. Description is a required property (if you don't specify something for Description you won't be able to save the tree.)

```
&TreeReturn = &MyTree.Create("", "", "PERSONAL_DATA2", "1999-06-01",  
"PERSONAL_DATA");
```

```
&MyTree.description = "test tree";
```

4. Add a level.

In order to add a level, you have to instantiate a level collection. Though there aren't any levels in the tree, you can still access this collection. The **Add** method is used with the level collection to add a new level. Remember, the level name *must be 8 characters or less*. Description is a required property (if you don't specify something for Description you won't be able to save the tree.)

```
&LvlColl = &MyTree.levels;  
  
&Level = &LvlColl.add("FIRST LVL");  
  
&Level.description = "First Level";
```

5. Add the root node.

Because this is a new tree, you must first add the root node.

```
&MyTree.insertroot("00001");
```

6. Add a leaf.

In order to add a new leaf, you have to return a reference to the node object. Using the **All** built-in function ensures that there is a root node before you try insert the leaf with the **InsertChildLeaf** method.

```
&RootNode = &MyTree.FindRoot();
```

```
If ALL (&RootNode) Then

    &NewLeaf = &RootNode.InsertChildLeaf("8000", "8999");
```

7. Save the tree.

When you execute the Save method, the new tree will be saved to the database.

```
&RSLT = &MyTree.Save();
```

The **Save** method returns a Boolean value: True if the save was successful, False otherwise. You can use this value to do error checking.



If you're running the tree API from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

8. Check for errors.

You can check if there were any errors using the PSMessages property on the session object.



Warning! For this release, errors that occur in 3-tier mode or in PeopleSoft Internet Architecture do *not* get sent to the PSMessages collection. Instead, they're reported back to the calling application, that is, for 3-tier they get displayed as messages on the application server, and in PIA they come back in the generated HTML.

```
If Not (&RSLT) Then

    /* errors occurred = do error checking */

    &ERRORCOL = &Session.PSMessages;

    For &I = 1 To &ERRORCOL.count

        /* do error processing */

    End-For;

Else

    /* no errors - saved correctly - do other processing */

End-If;
```

If there are multiple errors, all errors will be logged to the PSMessages collection, not just the first occurrence of an error. As you correct each error, you will want to delete it from the PSMessages collection.



If you've called the Tree API from an Application Engine program, all errors are also logged in the AE error log tables.

Session Object Methods

FindTree

Syntax

```
FindTree(SetId, UserKeyValue, TreeName, EffDt, BranchName)
```

Description

The **FindTree** method returns a tree collection.

You can use wild cards or partial keys with any of the parameters to get a smaller subset of the tree collection. A percent symbol (%) can be used in place of a number of characters. An underscore (_) can be used in place of one character. You can also use a partial value for any key. For example, if you wanted to get a collection of all the trees whose names started with the letter "B", you could specify NULL strings for all the parameters (that is, two quotation marks with no space between them ("")), and specify just the letter B for the *treename*:

```
&MYTREECOL = &MYSESSION.FindTree("", "", "B", "", "");
```

Considerations using Effective Dates

- When you specify a value for *EffDt*, all trees with dates less than or equal to the date specified by *EffDt* will be returned.
- If you use a value for *EffDt* that doesn't match any trees in the database, this method will return a 0.
- If you specify an asterisks ("*") for *EffDt* and no other parameters, the list of returned trees is sorted by effective date, with the max effective date tree listed first. After the tree list is sorted by *EffDt*, it's sorted by *SetID*, user key value, tree name, then branch name.
- If you specify the name of a tree, the list of the returned trees is sorted by effective date, with the max effective date tree listed first.

Parameters

<i>SetID</i>	Specify the table indirection key for the tree. This parameter takes a string value.
--------------	--

If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

UserKeyVal

Specify the User Key Value for the tree. This parameter takes a string value.

If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

TreeName

Specify the name of the tree you want to find. This parameter takes a string value.

EffDt

Specify the effective date for the tree you want to find. This parameter takes a string value.

BranchName

Specify the name of the branch for the tree you want to find. This parameter takes a string value.. If the tree is unbranched, you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

Returns

A tree collection object. If no tree collection object matching the parameters is found, it returns a 0.

Example

The following code populates a tree collection object with a list of all the trees named "DEPARTMENT".

```
Local ApiObject &TREELIST;

Local ApiObject &MYSESSION;

Local boolean &RSLT;

&MYSESSION = GetSession();

&RSLT = &MYSESSION.CONNECT(1, "EXISTING", "", "", 0);

If &RSLT Then

    /* connection is good */

    &TREELIST = &SESSION.FindTree("", "", "DEPARTMENT", "", "");
```



```

Else

    /* do error processing */

End-if;

```

The collection &TREELIST includes all trees that have the name "DEPARTMENT, regardless of SetID, Effective Date, and so on.

FindTreeStructure

Syntax

```
FindTreeStructure (StructId)
```

Description

The **FindTreeStructure** method returns a tree structure collection. *StructId* is a string value.

You can use wild cards or partial keys with *StructId* to get a subset of the list of tree structures. A percent sign (%) can be used in place of a number of characters. An underscore (_) can be used in place of one character. You can also use a partial value for *StructId*. For example, if you wanted to get a collection of all the tree structures whose names started with the letter "B", you could specify just the letter B:

```
&MYTREESCOLL = &MYSESSION.FindTreeStructure("B");
```

Returns

A tree structure collection.

Example

```

Local ApiObject &TLIST;

Local ApiObject &MYSESSION;

Local boolean &RSLT;

&MYSESSION = GetSession();

&RSLT = &MYSESSION.CONNECT(1, "EXISTING", "", "", 0);

If &RSLT Then

    /* connection is good */

    &TLIST = &MYSESSION.FindTreeStructure("DEPT");

Else

```

```

        /* do error processing */

End-if;

```

GetTree

Syntax

```
GetTree (SetID, UserKeyValue, TreeName, EffDt, BranchName)
```

Description

The **GetTree** method returns a closed tree object. Before you can use some of the methods or any of the properties, you must **Open** a tree object.



Note. You can *not* use any of the key properties (KeyName, KeySetId, and so on) with the closed tree object returned by GetTree. You can only use these properties with the trees returned in a collection by one of the Session Find methods.

Parameters

Though all the parameters for GetTree are required, they are unused. You should specify a NULL string (that is, two quotation marks with no blank space between them ("")) for all the parameters.

Returns

A reference to a closed tree object.

Example

To create a new tree, use the following:

```

&MYTREE = %Session.GetTree("", "", "", "", "");

&MYTREE.Create("", "", "PERSONAL_NEW", "05-05-1997", "PERSONAL_DATA");

```

To determine if a tree already exists in the database, use the following code:

```

&MYTREE = %Session.GetTree("", "", "", "", "");

If &MYTREE.Exists("", "", "PERSONAL_OLD", "05-05-1997", "PERSONAL_DATA") = 0 then

    /* Do processing */

End-If;

```

To open an existing tree, use the following code:

```

&MYTREE = %Session.GetTree("", "", "", "", "");

&MYTREE.Open("", "", "PERSONAL_OLD", "05-05-1997", "PERSONAL_DATA", True);

```

GetTreeStructure

Syntax

```
GetTreeStructure (StructId)
```

Description

The **GetTreeStructure** method returns a closed tree structure object. *StructId* is a string value. Before you can use some of the methods or any of the properties, you must **Open** the tree structure object.

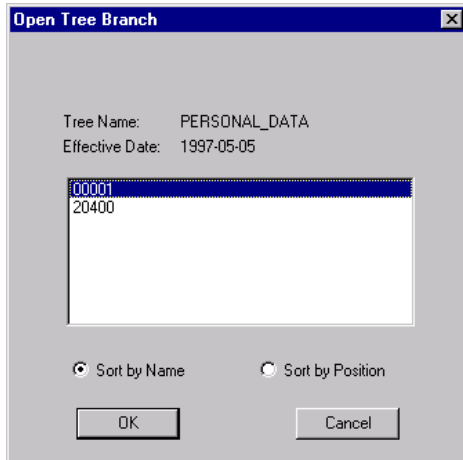
Branch Collections

A branch collection is returned from the Branches property, used with an open tree object.

```
&MYBRANCHCOLL = &MYTREE.Branches;
```

&MYTREE must be a branched tree. If you want to verify whether a tree is branched or not, you can check the value of the IsBranched property of a tree object (Boolean value: True or False.)

The branch collection, like all collections, has a GUI analogy. If you try to open a tree that is branched, a dialog displays to help the user decide which tree branch to open.



Branch Collection GUI analogy

Most of the branch collection methods return a **closed** tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method. Working with closed trees can improve your performance.

Branch Collection Method

Item

Syntax

```
Item(SetID, UserKeyValue, TreeName, EffDt , BranchName)
```

Description

The **Item** method returns the specified branch in the branch collection object executing the method as a **closed** tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method. The branch must be part of the branch collection executing the method.

Parameters

SetID

Specify the table indirection key for the branch (tree). This parameter takes a string value. If the associated tree structure has its **IndirectionMethod** specified as "S", you must specify a SetID.

If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

UserKeyValue

Specify the User Key Value for the branch (tree). This parameter takes a string value. If the associated tree structure has its **IndirectionMethod** specified as "U" or "B", you must specify a User Key Value.

If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

TreeName

Specify the name for the tree the branch is in. This parameter takes a string value.

EffDt

Specify the effective date for this branch. This parameter takes a string value.

BranchName

Specify the name of the branch for the tree. This parameter takes a string value.

Example

```
&MYTREE = &MYBRANCHCOLL.Item("", "", "PERSONAL_DATA", "05-05-1997", "");
```

Branch Collection Properties

Count

Returns the number of branches for the branch collection object.

First

The **First** property returns the first branch in the branch collection object executing the method as a **closed** tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method.

Last

The **Last** property returns the last branch in the branch collection object executing the method as a **closed** tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method.

Next

The **Next** property returns the next branch in the branch collection object executing the method as a **closed** tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method.

You can only use this method after you have used either the **First** or **Item** methods: otherwise the system doesn't know where to start.

Leaf Class

Leaf objects are instantiated from other tree classes as follows:

- From a tree object with the FindLeaf method
- From a node object with the FirstChildLeaf property
- From another leaf object with the NextSib and PrevSib properties

Leaf Class Methods

Cut

Syntax

Cut ()

Description

The **Cut** method cuts the leaf and puts it into a buffer (the clipboard.) You can then use the PasteSib method to add the leaf back as a sibling of a different leaf. You can also use the PasteChild node method to add the leaf back as a child of a different node.

You can only have one object at a time on the clipboard. If you cut another leaf or node, the leaf in the clipboard will be overwritten.

Parameters

None.

Returns

None.

Related Topics

PasteSib, PasteChild, Cut node method

Delete

Syntax

```
Delete()
```

Description

The **Delete** method deletes the leaf object executing the method *from the database*.

Example

```
&MYLEAF = &MYTREE.FindLeaf("8200", "8300");  
  
&MYLEAF.Delete();
```

DeleteByRange

Syntax

```
DeleteByRange(RangeFrom, RangeTo)
```

Description

The **DeleteByRange** method deletes the specified leaf object from the database. The leaf specified by the parameters does *not* have to match the leaf executing the method. That is, if &MYLEAF is associated with a range 8200-8300, you can specify a different range with the **DeleteByRange** method. For example:

```
&MYLEAF = &MYTREE.FindLeaf("8200", "8300");
```

```

        /* some processing */

        &MYLEAF.DeleteByRange("8400", "8500");

```

Parameters

<i>RangeFrom</i>	Specify the starting range of the leaf to be deleted. This parameter takes a string value.
<i>RangeTo</i>	Specify the ending range of the leaf to be deleted. This parameter takes a string value.

Example

```
&MYLEAF.DeleteByRange("8400", "8500");
```

InsertDynSib

Syntax

```
InsertDynSib()
```

Description

The **InsertDynSib** method inserts a dynamic leaf as a sibling leaf to the leaf object executing the method. The leaf will be a *new* node.

A leaf object associated with the new leaf is returned. The new leaf will be inserted as the next sibling leaf. If the new leaf isn't inserted successfully, the method will return NULL.



If you want to insert a sibling leaf with a specific range, use the InsertSib method.

Parameters

None.

Returns

A reference to the new leaf. If this method fails, it returns either False or a Null object reference, depending upon the type of error encountered. The best way to test whether the object was inserted or not is to test the result using either the All or None function.

Example

```
&NEWLEAF = &MYLEAF.InsertDynSib();
```

InsertSib

Syntax

InsertSib(*RangeFrom*, *RangeTo*)

Description

The **InsertSib** method inserts a new leaf as a sibling leaf under the leaf currently executing the method. The leaf specified by the range parameters must be a *new* leaf. You will receive an error if the leaf already exists.

A leaf object associated with the new leaf is returned. The new leaf will be inserted as the next sibling leaf, although there is no explicit ordering of leaves: leaves take the order of the alphanumeric database sort on their range fields.



If you want to insert a dynamic sibling leaf (that is, without specifying a range) use the InsertDynSib method.

Parameters

<i>RangeFrom</i>	Specify the starting range of the new leaf. This parameter takes a string value.
<i>RangeTo</i>	Specify the ending range of the new leaf. This parameter takes a string value.

Returns

A leaf object associated with the new leaf. If this method fails, it returns either False or a Null object reference, depending upon the type of error encountered. The best way to test whether the object was inserted or not is to test the result using either the All or None function.

Example

```
&NEWLEAF = &MYLEAF.InsertSib("10090", "10100");
```

MoveAsChild

Syntax

MoveAsChild(*Node*)

Description

The **MoveAsChild** method moves the leaf executing the method to a different node in the tree. The leaf will become the first child leaf under the node, even though there is no explicit ordering of leaves: leaves take the order of the alphanumeric database sort on their range fields. The specified node will become the new parent to the leaf.

Parameters

Node Specify a node object. This parameter value must be an object, not a string or a name.



If you want to specify a node name, not a node object, use the `MoveAsChildByName` method.

Example

The following example moves leaf 8001 from node 10100 (old parent) to node 00001 (new parent)

```
&MY_LEAF = &MY_TREE.FindLeaf("8001", "8001");

&NEW_PARENT = &MY_TREE.FindNode("00001", "");

If &NEW_PARENT <> Null Then

    &MY_LEAF.MoveAsChild(&NEW_PARENT);

End-If;
```

MoveAsChildByName

Syntax

MoveAsChildByName (*NodeName*)

Description

The **MoveAsChildByName** method moves the leaf executing the method to a different node in the tree. The leaf will become the first child leaf under the node, even though there is no explicit ordering of leaves: leaves take the order of the alphanumeric database sort on their range fields. The specified node will become the new parent to the leaf. The node specified by *NodeName* must be a valid node name.

Parameters

NodeName Specify the name of a node. *NodeName* must be a valid node for the existing tree. This parameter takes a string value.



If you want to specify a node object, not a node name, use the `MoveAsChild` method.

Example

The following example moves leaf 8001 from node 10100 (old parent) to node 00001 (new parent)

```
&MY_LEAF = &MY_TREE.FindLeaf("8001", "8001");

If &MY_LEAF <> Null Then

    &MY_LEAF.MoveAsChildByName("00001");

End-If;
```

MoveAsSib

Syntax

MoveAsSib (*Leaf*)

Description

The **MoveAsSib** method moves the leaf executing the method to a new place in the tree. The current leaf will become the next sibling leaf under the specified leaf object, even though there is no explicit ordering of leaves: leaves take the order of the alphanumeric database sort on their range fields.

Parameters

<i>Leaf</i>	Specify a leaf object. This parameter value must be an object, not a string or a name.
-------------	--



If you want to specify a range, not a leaf object, use the MoveAsSibByRange method.

Example

```
&MYLEAF = &MYTREE.FindLeaf("8000", "8000");

&MYLEAF2 = &MYTREE.FindLeaf("9000", "9000");

&MYLEAF.MoveAsSib(&MYLEAF2);
```

MoveAsSibByRange

Syntax

MoveAsSibByRange (*RangeFrom*, *RangeTo*)

Description

The **MoveAsSibByRange** method moves the leaf executing the method to a new place in the tree. The current leaf will become the next sibling leaf under the specified leaf object.

Parameters

<i>RangeFrom</i>	Specify the starting range of the leaf that you want as the parent of the leaf executing the method. This parameter takes a string value.
<i>RangeTo</i>	Specify the ending range of the leaf that you want as the parent of the leaf executing the method. This parameter takes a string value.



If you want to specify a leaf object, not a range, use the **MoveAsSib** method.

Example

```
&MYLEAF = &MYTREE.FindLeaf("8000", "8000");

&MYLEAF.MoveAsSibByRange("9000", "9000");
```

PasteSib

Syntax

```
PasteSib()
```

Description

The **PasteSib** method makes the leaf from the buffer (clipboard) a sibling to the leaf executing the method. You must use the **Cut** leaf method before you can paste a leaf.

You can only have one object at a time on the clipboard. If you cut another leaf or node, the leaf in the clipboard will be overwritten.

Parameters

None.

Returns

None.

Related Topics

Cut leaf method, PasteChild, Cut node method

Leaf Class Properties

Dynamic

This property specifies whether the leaf has a dynamic range or a specified range. If you set this property to True, the leaf will have a dynamic range. If this is a new leaf, and you do not set this property, the value automatically set is False.



For more information, see [Creating Evenly Gapped Trees](#)

This property is read-write.

HasNextSib

This property returns True if the leaf has a next sibling, that is, it isn't the last leaf listed under the parent node.

This property is read-only.

HasPrevSib

This property returns True if the leaf has a previous sibling, that is, it isn't the first leaf listed under the parent node.

This property is read-only.

IsChanged

This property returns True if the leaf has been edited or changed in some way but the tree containing the leaf hasn't been saved.

This property is read-only.

Example

```
If &MYLEAF.IsChanged Then  
  
    &MYTREE.Save();  
  
End-If;
```

IsDeleted

This property returns True if the leaf has been deleted from the tree but the tree hasn't been saved.

This property is read-only.

IsInserted

This property returns True if the leaf has been inserted as a new leaf in the tree but the tree hasn't been saved.

This property is read-only.

NextSib

This property returns a leaf object associated with the next sibling leaf. The next sibling of a leaf is the leaf appearing under the current leaf. If there is no next sibling and you try to assign it to a variable, you will receive a runtime error.

This property is read-only.

Example

The following code traversed the leaves from top to bottom.

```
While &MYLEAF.HasNextSib  
  
    &MYLEAF = &MYLEAF.NextSib;  
  
    /* do some processing */  
  
End-While;
```

Parent

This property returns a node object associated with the parent node for the leaf.

This property is read-only.

Example

```
&PARENTNODE = &MYLEAF.Parent;  
  
/* do node processing with node object */
```

PrevSib

This property returns a leaf object associated with the previous sibling leaf. The previous sibling of a leaf is the leaf appearing above the current leaf. If there is no previous sibling and you try to assign it to a variable, you will receive a runtime error.

This property is read-only.

Example

The following code traverses the leaves from bottom to top.

```
While &MYLEAF.HasPrevSib  
  
    &MYLEAF = &MYLEAF.PrevSib;  
  
    /* do some processing */  
  
End-While;
```

RangeFrom

This property returns the starting range, as a string, of the leaf.

This property is read-write.

RangeTo

This property returns the ending range, as a string, of the leaf.

This property is read-write.

TreeBranchName

This property returns the branch name of the tree as a string if the tree is branched. If not branched, this property returns a blank string.

This property is read-only.

TreeEffDt

This property returns the effective date of the tree as a string if the tree is effective-dated. If not effective-dated, this property returns a blank string.

This property is read-only.

TreeName

This property returns the name of the tree as a string.

This property is read-only.

TreeSetId

This property returns the SetID of the tree as a string if the tree has a SetID. If the tree doesn't have a SetID, this property returns a blank string.

This property is read-only.

TreeUserKeyValue

This property returns the UserKeyValue of the tree as a string if the tree has a UserKeyValue. If the tree doesn't have a UserKeyValue, this property returns a blank string.

This property is read-only.

Level Collection

Level collection are instantiated from a tree object with the Levels property.

LevelCollection Methods

Add

Syntax

Add (*LevelName*)

Description

The **Add** method adds a *new* level called *LevelName* to the database. The specified level must be a *new* level, that is, it cannot exist in the database. *LevelName* takes a string value.



LevelName must be 8 characters or less.

If no levels currently exist in the tree, the level is added at the top level. If levels already exist in the tree, the new level will be added as the *last* level. You can change the level number of a level using the Number property.

If the new level is the first level, the AllValuesAudit property is automatically set to True.

The new level will *not* be added to the database until the tree is explicitly saved.

This method returns a reference to the new level object.

Item

Syntax

Item (*LevelName*, *LevelNumber*)

Description

The **Item** method returns a reference to the specified level in the level collection executing the method **as an object**. The *LevelName* parameter specifies the name of the level you want to

access. This parameter takes a string value. The *LevelNumber* parameter specifies the number at which the level exists. This parameter takes a number value. For example, suppose your level collection contains the following levels, in this order:

6. CORPORATE
7. COMPANY
8. DIVISION
9. DEPARTMENT
10. BRANCH

You want to access the fourth level, DEPARTMENT. You would use the following code:

```
&MYLEVEL = &LVLCOLLECTION.Item("DEPARTMENT", 4);
```

Remove

Syntax

```
Remove ()
```

Description

The **Remove** method deletes the current level *from the database*. You can only use this method after you have used the **First**, **Next**, or **Item** properties: otherwise the system doesn't know which level to delete from the tree. The rest of the levels in the collection after the deleted level will be moved up in the list and renumbered so that the levels remain consecutively numbered.

Save

Syntax

```
Save ()
```

Description

The **Save** method saves the current level to the database. You can only use this method after you have used the **First**, **Next**, or **Item** methods: otherwise the system doesn't know which level to save.



If you're calling the Tree API from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

Level Collection Properties

Count

Returns the number of levels for the level collection object.

First

The **First** property returns a reference to the first level in the level collection executing the method *as an open object*.

Last

The **Last** property returns a reference to the last level in the level collection executing the method *as an open object*.

Next

The **Next** property returns a reference to the next level in the level collection executing the method *as an open object*. You can only use this method after you have used either the **First** or **Item** properties: otherwise the system doesn't know where to start in the tree.

Level Class

Level objects are instantiated from the level class collection object with the Item method or one of the following properties:

- First
- Last
- Next

Level Class Methods

Create

Syntax

```
Create (LevelName, LevelNumber)
```

Description

The **Create** method adds a *new* level called *LevelName*. The specified level must be a *new* level. You will receive an error if the level already exists.

Parameters

<i>LevelName</i>	Specify the name of the new level. This parameter takes a string value.
<i>LevelNumber</i>	Specify the number of the new level. This parameter takes a number value.

Example

```
&LEVELCOLL = &MYTREE.Levels;  
  
&LEVEL = &LEVELCOLL.First;  
  
&LEVEL.Create("INTERNET", 9);
```

Delete

Syntax

```
Delete()
```

Description

The **Delete** method deletes the level object executing the method from the database.

Example

```
&MYLVLCOLL = &MYTREE.Levels;  
  
&MYLVL = &MYLVLCOLL.Last;  
  
&MYLVL.Delete();
```

Level Class Properties

AllValuesAudit

This property specifies whether Tree Manager permits nodes to skip over this level. To allow nodes to skip this level, specify this parameter as True. If you are creating a new level, and this level is the first level in a tree, this property is automatically set to True. If the level isn't the first level, this property is automatically set to False.



For more information, see [Defining Levels](#).

This property is read-write.

Description

This property returns the description of the level.

This property is read-write.

Name

This property returns the name of the level.

This property is read-write.

Number

This property returns the number of the level.

This property is read-write.

TreeBranchName

This property returns the branch name of the tree as a string if the tree is branched. If not branched, this property returns a blank string.

This property is read-only.

TreeEffDt

This property returns the effective date of the tree as a string if the tree is effective dated. If not effective dated, this property returns a blank string.

This property is read-only.

TreeName

This property returns the name of the tree as a string.

This property is read-only.

TreeSetId

This property returns the SetID of the tree as a string if the tree has a SetID. If the tree doesn't have a SetID, this property returns a blank string.

This property is read-only.

TreeUserKeyValue

This property returns the UserKeyValue of the tree as a string if the tree has a UserKeyValue. If the tree doesn't have a UserKeyValue, this property returns a blank string.

Node Class

Node objects are instantiated from other tree classes, as follows:

- From a tree object with the FindNode method
- From a leaf object with the Parent property
- From another node object with the FirstChildNode, NextSib, PrevSib, and Parent properties

Node Class Methods

Branch

Syntax

```
Branch()
```

Description

The **Branch** method branches the node executing this method, identifying all of its child nodes and leaves as part of that branch. *Branching* means taking a limb of a tree and creating another subtree to hold that limb. (Technically is it not creating a real tree.) The subtree is accessed through the Branches collection.



For more information, see Creating Tree Branches.

After you use the **Branch** method, you can no longer access the child nodes and leaves of the node that executed the method until you close the current tree, open the new branched tree, and find the node again. To unbranch the tree, use the **Unbranch** method.

Example

```
&MYNODE = &MYTREE.FindNode("10900", "");  
  
&MYNODE.Branch();
```

Cut

Syntax

```
Cut ()
```

Description

The **Cut** method cuts the node executing the method and puts it into a buffer (the clipboard.) You can then use the PasteSib method to add the node back as a sibling of a different node. You can also use the PasteChild node method to add the node back as a child of a different node.

You can only have one object at a time on the clipboard. If you cut another leaf or node, the node on the clipboard will be overwritten.

Parameters

None.

Returns

None.

Related Topics

PasteSib, PasteChild, Cut leaf method

Delete

Syntax

```
Delete ()
```

Description

The **Delete** method deletes the node object executing the method *from the database*.

DeleteByName

Syntax

```
DeleteByName (NodeName)
```

Description

The **DeleteByName** method deletes the specified node *from the database*. The *NodeName* parameter takes a string value. The node specified by *NodeName* does *not* have to match the node executing the method. That is, if &MYNODE is associated with node 100200, you can specify a different node with the **DeleteByName** method. For example:

```
&MYNODE = &MYTREE.FindNode("100200", "");
```

```

        /* some processing */

        &MYNODE.DeleteByName("100300");

```

The node name specified by *NodeName* must be a valid node in the existing open tree object. If *NodeName* doesn't exist, you'll receive a runtime error.

Expand

Syntax

```
Expand (ExpandType)
```

Description

The **Expand** method gets the next level of nodes or leaves into memory from the selected node. What leaves or nodes get expanded depends on the *ExpandType*. Valid values are:

<i>Value</i>	<i>Description</i>
0	Expand only nodes
1	Expand nodes and leaves
2	Expand only one level

Example

```

If (&MYNODE.State = 2 AND &MYNODE.HasChildren);

    /* if node is collapsed */

    &MYNODE.Expand(2);

End-if;

```

InsertChildLeaf

Syntax

```
InsertChildLeaf (RangeFrom, RangeTo)
```

Description

The **InsertChildLeaf** method inserts a new leaf (as specified by the range parameters) under the node object executing the method.

A leaf object associated with the new leaf is returned.

The new leaf will be inserted as the child of the current node, although there is no explicit ordering of leaves: leaves take the order of the alphanumeric database sort on their range fields.



You must specify a range with this method. If you want to insert a dynamic range leaf (that is, one with no range) use the InsertDynChildLeaf method.

Parameters

<i>RangeFrom</i>	Specify the starting range of the new leaf. This parameter takes a string value.
<i>RangeTo</i>	Specify the ending range of the new leaf. This parameter takes a string value.

Returns

A leaf object associated with the new leaf. If this method fails, it returns either False or a Null object reference, depending upon the type of error encountered. The best way to test whether the object was inserted or not is to test the result using either the All or None function.

Example

```
&NEWLEAF = &MYLNODE.InsertChildLeaf("10090", "10100");

If None(&NEWLEAF) Then

    /* Error Processing */

End-If;
```

InsertChildNode

Syntax

```
InsertChildNode(NodeName)
```

Description

The **InsertChildNode** method inserts a new node (as specified by *NodeName*) as a child node under the node object executing the method. The node specified by *NodeName* must be a *new* node. You'll receive an error if the node already exists.

A node object associated with the new node is returned. If the new node isn't inserted successfully, the method returns False or NULL.

Parameters

<i>NodeName</i>	Specify a name for the new node. This parameter takes a string value. <i>NodeName</i> must not already exist.
-----------------	---

Returns

A reference to the new node. If this method fails, it returns either False or a Null object reference, depending upon the type of error encountered. The best way to test whether the object was inserted or not is to test the result using either the All or None function.

Example

```
&NEWNODE = &MYNODE.InsertChildNode("100500");

If None(&NEWNODE) Then

    /* Do error processing */

End-If;
```

InsertChildRecord

Syntax

```
InsertChildRecord(NodeName)
```

Description

The **InsertChildRecord** method inserts a new record (as specified by *NodeName*) as a node under the parent node executing this method. This method only works on trees that are Query Access Trees.

A node object associated with the new node is returned, otherwise this method returns False or NULL.

Parameters

<i>NodeName</i>	Specify an existing record name. This parameter takes a string value. This record must not already be a node under the parent node (that is, a parent node can't have the same record as a child node more than once.)
-----------------	--

Returns

A reference to the new node. If this method fails, it returns either False or a Null object reference, depending upon the type of error encountered. The best way to test whether the object was inserted or not is to test the result using either the All or None function.

Example

```
&NEWNODE = &MYNODE.InsertChildRecord("PERSONAL_DATA");

If None(&NEWNODE) Then

    /* Do error processing */

End-if;
```


InsertDynChildLeaf

Syntax

```
InsertDynChildLeaf()
```

Description

The **InsertDynChildLeaf** method inserts a dynamic leaf under the node object executing the method.

A leaf object associated with the new leaf is returned.

The new leaf will be inserted as the child of the current node, although there is no explicit ordering of leaves.



If you want to insert a leaf with a specific range, use the InsertChildLeaf method.

Parameters

None.

Returns

A leaf object associated with the new leaf. If this method fails, it returns either False or a Null object reference, depending upon the type of error encountered. The best way to test whether the object was inserted or not is to test the result using either the All or None function.

Example

```
&NEWLEAF = &MYLNODE.InsertDynChildLeaf();  
  
If None(&NEWLEAF) Then  
  
    /* Do error processing */  
  
End-If;
```

InsertSib

Syntax

```
InsertSib(NodeName)
```

Description

The **InsertSib** method inserts a new node (as specified by *NodeName*) as a sibling node to the node object executing the method. The node specified by *NodeName* must be a *new* node. You'll receive an error if the node already exists.

A node object associated with the new node is returned. If the new node isn't inserted successfully, the method will return NULL.

Parameters

<i>NodeName</i>	Specify a name for the new node. This parameter takes a string value. <i>NodeName</i> must not already exist.
-----------------	---

Returns

A reference to the new node. If this method fails, it returns False.

Example

```
&NEWNODE = &MYNODE.InsertSib("100500");
```

InsertSibRecord

Syntax

```
InsertSibRecord(NodeName)
```

Description

The **InsertSibRecord** method inserts a new record (as specified by *NodeName*) as a sibling node of the parent node executing this method. This method only works on trees that are Query Access Trees.

A node object associated with the new node is returned, otherwise this method returns NULL.

Parameters

<i>NodeName</i>	Specify an existing record name. This parameter takes a string value. This record must not already be a sibling node for the node executing the object (that is, a node can't have the same record as a node more than once.)
-----------------	---

Returns

A reference to the new node. If this method fails, it returns False.

Example

```
&NEWNODE = &MYNODE.InsertSibRecord("PERSONAL_DATA");
```

MoveAsChild

Syntax

```
MoveAsChild(Node)
```

Description

The **MoveAsChild** method moves the node executing the method to a different node in the tree. The node will become the first child node under the named node. The specified node will become the new parent to the child node. The node specified by *Node* must be an existing node in the current tree. All child nodes and details will also be moved with the node.

Parameters

<i>Node</i>	Specify a node object. This parameter value must be an object, not a string or a name.
-------------	--



If you want to specify a node name, not a node object, use the **MoveAsChildByName** method.

Example

The following example moves node 10200 from node 10100 (old parent) to node 00001 (new parent).

```
&MY_NODE = &MY_TREE.FindNode("10200", "");

&NEW_PARENT = &MY_TREE.FindNode("00001", "");

If &NEW_PARENT <> Null Then

    &MY_NODE.MoveAsChild(&NEW_PARENT);

End-If;
```

MoveAsChildByName

Syntax

MoveAsChildByName (*NodeName*)

Description

The **MoveAsChildByName** method moves the node executing the method to a different node in the tree. The node will become the first child node under the parent node. The specified node will become the new parent to the node. The node specified by *NodeName* must be a valid node name. All child nodes and details will also be moved with the node.

Parameters

<i>NodeName</i>	Specify the name of a node. This parameter takes a string value. <i>NodeName</i> must be a valid node for the existing tree.
-----------------	--



If you want to specify a node object, not a node name, use the `MoveAsChild` method.

Example

The following moves node 10200 from node 10100 (old parent) to node 00001 (new parent)

```
&MY_NODE = &MY_TREE.FindNode("10200", "");  
  
If &MY_NODE <> Null Then  
  
    &MY_NODE.MoveAsChildByName("00001");  
  
End-If;
```

MoveAsSib

Syntax

MoveAsSib (*Node*)

Description

The **MoveAsSib** method moves the node executing the method to a new place in the tree. The current node will become the next sibling node under the specified node. All child nodes and details will also be moved with the node.

Parameters

<i>Node</i>	Specify a node object. This parameter value must be an object, not a string or a name.
-------------	--



If you want to specify a node name, not a node object, use the `MoveAsSibByName` method.

Example

```
&MYNODE = &MYTREE.FindNode("20000", "");  
  
&MYNODE2 = &MYTREE.FindNode("20100", "");  
  
&MYNODE.MoveAsSib(&MYNODE2);
```

MoveAsSibByName

Syntax

MoveAsSibByName (*NodeName*)

Description

The **MoveAsSibByName** method moves the node executing the method to a new place in the tree. The current node will become the next sibling node under the specified node. All child nodes and details will also be moved with the node.

Parameters

<i>NodeName</i>	Specify the name of a node. This parameter takes a string value. <i>NodeName</i> must be a valid node for the existing tree.
-----------------	--



If you want to specify a node object, not a node name, use the MoveAsSib method.

Example

```
&MYNODE = &MYTREE.FindNode("20000", "");  
  
&MYNODE.MoveAsSibByName("10100");
```

PasteChild

Syntax

PasteChild ()

Description

The **PasteChild** method makes the node or leaf from the buffer (clipboard) a child to the node executing the method. You must use the **Cut** node or leaf method before you can paste a node or leaf.

You can only have one object at a time on the clipboard. If you cut another node or leaf, the node or leaf on the clipboard will be overwritten.

Parameters

None.

Returns

None.

Related Topics

Cut leaf method, PasteSib, Cut node method

PasteSib

Syntax

```
PasteSib()
```

Description

The **PasteSib** method makes the node from the buffer (clipboard) a sibling to the node executing the method. You must use the **Cut** node method before you can paste a node.

You can only have one object at a time on the clipboard. If you cut another leaf or node, the node on the clipboard will be overwritten.

Parameters

None.

Returns

None.

Related Topics

Cut leaf method, PasteChild, Cut node method

Rename

Syntax

```
Rename (NodeName)
```

Description

The **Rename** method renames the node object executing the method without changing any other properties. The parameter *NodeName* takes a string value. The node specified by *NodeName* must be a **new** node. You'll receive an error if the node already exists. Also, the node specified by *NodeName* can't be branched.

Example

```
&MYNODE = &MYTREE.FindNode("10900", "");  
  
&MYNODE.Rename("10200");
```

SwitchLevel

Syntax

```
SwitchLevel (NewLevelNumber)
```

Description

The **SwitchLevel** methods enables you to move a node down from one level to another.

NewLevelNumber must specify a valid level number. For trees in which the Level Use parameter is set to Strictly Enforce Levels, the new level must be at least one level *lower* than the node's parent level.

If the level specified by *NewLevelNumber* doesn't exist, it's automatically generated and added to the tree.

Parameters

<i>NewLevelNumber</i>	Specify the level number to where you want the node moved.
-----------------------	--

Returns

A zero (0) if node is moved successfully, a different error number otherwise.

Unbranch

Syntax

```
Unbranch()
```

Description

The **Unbranch** method is the opposite of the **Branch** method: that is, it unbranches the node executing the method if it is branched. All of the child node and leaves of the node will become part of the current open tree and accessible in that tree. You can't unbranch the Root Node in a branched tree.



For more information, see [Creating Tree Branches](#)

If the node executing the method isn't branched, you'll receive a runtime error. Use the **IsBranched** property to make sure a node is branched.

Example

```
&MYNODE = &MYTREE.FindNode("10900", "");  
  
&MYNODE.Unbranch();
```

Node Class Properties

AllChildCount

This property returns the number of all the child nodes and leaves of the node, that is, all the nodes and leaves below this node. Valid values are 0 to 2 billion.

This property is read-only.

AllChildNodeCount

This property returns the number of child nodes of the node, that is, all the nodes below this node.

This property is read-only.

Example

To determine all the leaves below a node, use the AllChildNodeCount with the AllChildCount property.

```
&AllCount = (&MyNode.AllChildCount - &MyNode.AllChildNodeCount);
```

ChildLeafCount

This property returns the number of immediate child leaves of the node, that is, all leaves under this node.

This property is read-only.

ChildNodeCount

This property returns the number of immediate child nodes of the node, that is, all nodes under this node.

This property is read-only.

Description

This property returns the description of the node.

This property is read-only.

FirstChildLeaf

This property returns a reference to the first child leaf of the node. This is the leaf that appears highest in the list of children of the node in Tree Manager.

This property is read-only.

Example

```
&NEWLEAF = &MYNODE.FirstChildLeaf;
```

FirstChildNode

This property returns a reference to the first child node of the node. This is the node that appears highest in the list of children of the node in Tree Manager.

This property is read-only.

Example

```
&CHILDNODE = &MYNODE.FirstChildNode;
```

HasChildLeaves

This property returns True if the node has immediate child leaves, False otherwise.

This property is read-only.

HasChildNodes

This property returns True if the node has immediate child nodes, False otherwise.

This property is read-only.

HasChildren

This property returns True if the node has *either* child leaves or child nodes anywhere in the subtree, not just immediately under the node.

This property is read-only.

HasNextSib

This property returns True if the node has a next sibling, that is, if it isn't the last node.

This property is read-only.

Example

```
While &MYNODE.HasNextSib
    &MYNODE = &MYNODE.NextSib;
    /* do some processing */
End-While;
```

HasPrevSib

This property returns True if the node has a previous sibling, that is, if it isn't the first node.

This property is read-only.

IsBranched

This property returns True if the node is branched, False otherwise.

This property is read-only.

IsChanged

This property returns True if the node has been edited or changed in some way but the current tree hasn't been saved.

This property is read-only.

IsDeleted

This property returns True if the node has been deleted but the current tree hasn't been saved.

This property is read-only.

IsInserted

This property returns True if the node was inserted as a new node into the tree but the tree hasn't been saved.

This property is read-only.

Example

```
If &MYNODE.IsInserted Then  
  
    &MYTREE.Save() ;  
  
End-if;
```

IsRoot

This property returns True for both of the following:

- the node is the root node of an unbranched tree
- the node is the top node of an opened tree branch

Otherwise, the property returns False.

This property is read-only.

LastChildLeaf

This property returns a reference to the last child leaf of the node. This is the leaf that appears lowest in the list of children of the node in Tree Manager.

This property is read-only.

Example

```
&NEWLEAF = &MYNODE.LastChildLeaf;
```

LastChildNode

This property returns a reference to the last child node of the node. This is the node that appears lowest in the list of children of the node in Tree Manager.

This property is read-only.

Example

```
&CHILDNODE = &MYNODE.LastChildNode;
```

LevelNumber

This property returns the level number of the node. Valid values are 1-99 for Strict or Loose level trees. This property returns 0 for trees that don't have levels.

This property is read-write.

Name

This property returns the name of the node (as a string.) You must set this property to a valid value if you are creating a *new* node.



Do not use this property to change the name of an existing node. The change will *not* be reflected in the database. You must use the Rename method to change the name of an existing node.

This property is read-write.

NextSib

This property returns a reference to the next sibling node. If there isn't a next sibling node, NULL is returned.

This property is read-only.

Example

```
While &MYNODE.HasNextSib

    &MYNODE = &MYNODE.NextSib;

    /* do some processing */

End-While;
```

Parent

This property returns a reference to the node that is the parent of the node executing the property. If the current node has no parent (is a root node) NULL is returned.

This property is read-only.

Example

```
&PARENT = &MYNODE.Parent;
```

PrevSib

This property returns a reference to the node that is the previous sibling node to the node executing the property. If there isn't a previous sibling, NULL is returned.

This property is read-only.

Example

```
If &MYNODE.PrevSib Then

    /* do processing */

End-if;
```

State

This property returns the *state* of the node, that is, does it have children, and are they expanded or collapsed. (See **Expanded** method.)

The valid values for this property are:

Value	Description
0	Node has no children
1	Children are expanded
2	Children are collapsed

Value	Description
3	Leaves are collapsed

This property is read-only.

Example

```
&VALUE = &MYNODE.State;

If &VALUE = 2 Then

    &MYNODE.Expand(0) ;

End-if;
```

TreeBranchName

This property returns the branch name of the tree as a string if the tree is branched. If not branched, this property returns a blank string.

This property is read-only.

TreeEffDt

This property returns the effective date of the tree as a string if the tree is effective dated. If not effective dated, this property returns a blank string.

This property is read-only.

TreeName

This property returns the name of the tree as a string.

This property is read-only.

TreeSetId

This property returns the SetID of the tree as a string if the tree has a SetID. If the tree doesn't have a SetID, this property returns a blank string.

This property is read-only.

TreeUserKeyValue

This property returns the UserKeyValue of the tree as a string if the tree has a UserKeyValue. If the tree doesn't have a UserKeyValue, this property returns a blank string.

Type

This property returns the type of the node. This property takes a string value. Valid values are:

- "G" – Group: normal unbranched node or record group
- "B" – Branched nodes
- "R" – Query Record: used for Query Access Trees only

This property is read-only.

Tree Class

Tree objects are instantiated from the following:

- From a session object with the GetTree method
- From a branch collection with the First, Last or Next properties or the Item method.
- From a tree collection with the First, Last or Next properties or the Item method.

The following code sample gets a tree, then opens the tree structure associated with that tree:

```
Local string &TREE_NAME;

Local string &TREE_DT;

Local ApiObject &MYSESSION, &VC_TREE, &STRUCT;

/* Get and Open the Tree using the Tree API */

&MYSESSION = GetSession();

&MYSESSION.CONNECT(1, "EXISTING", "", "", 0);

/* Get the Tree Name and Effective Date from the level 0 record */

&TREE_NAME = "CUSTOMER";

&TREE_DT = "1900-01-01";

/* Get and Open the Tree */

&VC_TREE = &MYSESSION.GetTree("BKINV", "", &TREE_NAME, &TREE_DT, "");
```

```
&VC_TREE.OPEN("BKINV", "", &TREE_NAME, &TREE_DT, "", False);

/* Get and Open Tree Structure */

&STRUCT = VC_TREE.GetStructure;
```

Tree Class Methods

Audit

Syntax

```
Audit()
```

Description

The **Audit** method audits the tree object executing the method to determine its validity.



For more information, see Auditing Trees.

The **Audit** method can only be used on an open tree, not on a closed tree. This means you must have opened the tree with the Open method before you can audit it.

This method returns a Boolean value: True if the tree passes all audits, False if one or more errors occurred. If **Audit** returns False, an error is logged.

AuditByName

Syntax

```
AuditByName(SetID, UserKeyValue, TreeName, EffDt, BranchName)
```

Description

The **AuditByName** method audits the tree specified by the parameters passed to it. The **AuditByName** method can only be used with a tree identifier, it cannot be used on a fully instantiated, open tree. Before you use the **AuditByName** method, you must explicitly close any open tree objects (with the Close method.) You will receive an error if there are any open trees.



For more information, see Auditing Trees.

SetID

Specify the table indirection key for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "S", you must specify a SetID.

If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

UserKeyValue

Specify the User Key Value for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "U" or "B", you must specify a User Key Value.

If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

TreeName

Specify the name for this tree. This parameter takes a string value.

EffDt

Specify the effective date for this tree. This parameter takes a string value.

BranchName

This parameter is required, but it is unused in this release. You must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

Returns

This method returns a Boolean value: True if the tree passes all audits, False if one or more errors occurred. If **AuditByName** returns False, a runtime error occurs, and either a message box or a logged error will report the nature of the error found in the tree.

Example

```
&ISVALID = &MYTREE.AuditByName("", "", "PERSONAL_DATA", "05-05-1997", "");
```

Close**Syntax**

```
Close()
```

Description

The **Close** method closes the tree, freeing the memory associated with that object, and discarding any changes made to the tree since the last save. The **Close** method can only be used on an open tree, not a closed tree. This means you must have opened the tree with the **Open** method before

you can close it. If you want to save any changes, you must use the **Save** method before using **Close**.

It's very important to close your tree when you're finished processing. Canceling out of a page does *not* close a tree. You may receive error messages every other time you run your program if you haven't closed your trees.

Copy

Syntax

```
Copy(FromSetId, FromUserKeyValue, FromTreeName, FromEffDt, FromBranchName,  
ToSetId, ToUserKeyValue, ToTreeName, ToEffDt, ToBranchName);
```

Description

The **Copy** method copies from the tree identified by the *From* parameters and copies it to the tree identified with the *To* parameters. The tree specified by the *To* parameters must be a *new* tree. You will receive an error if the tree already exists. The tree specified by the *From* parameters does *not* have to match the tree identifier executing the method.

The **Copy** method can only be used with a **closed** tree, it cannot be used on an open tree. Before you use the **Copy** method, you must explicitly close any open tree objects (with the **Close** method.) You will receive an error if there are any open trees.

To access the new tree, you must use the **Open** method.

Parameters

FromSetId

Specify the table indirection key for the tree to be copied from. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "S", you must specify a SetID.

If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

FromUserKeyValue

Specify the User Key Value for the tree to be copied from. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "U" or "B", you must specify a User Key Value.

If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

FromTreeName

Specify the name for the tree to be copied from. This parameter takes a string value.

<i>FromEffDt</i>	Specify the effective date for the tree to be copied from. This parameter takes a string value.
<i>FromBranchName</i>	This parameter is required, but it is unused in this release. You must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.
<i>ToSetId</i>	Specify the table indirection key for the tree to be copied to. This parameter takes a string value. If the tree structure the tree is based on has its IndirectionMethod specified as "S", you must specify a SetID. If the tree structure doesn't have its IndirectionMethod specified as "S", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.
<i>ToUserKeyValue</i>	Specify the User Key Value for the tree to be copied to. This parameter takes a string value. If the tree structure the tree is based on has its IndirectionMethod specified as "U" or "B", you must specify a User Key Value. If the tree structure doesn't have its IndirectionMethod specified as "U" or "B" you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.
<i>ToTreeName</i>	Specify the name for the tree to be copied to. This parameter takes a string value.
<i>ToEffDt</i>	Specify the effective date for the tree to be copied to. This parameter takes a string value.
<i>ToBranchName</i>	This parameter is required, but it is unused in this release. You must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

Returns

An integer: 0 if copied successfully.

Example

```
&MYTREE.Copy("", "", "PERSONAL_DATA", "05-05-1997", "", "", "", "PERSONAL_EDIT1",
"05-05-1997", "");
```

Create

Syntax

```
Create(SetID, UserKeyValue, TreeName, EffDt, StructureName)
```

Description

The **Create** method creates a new tree, based on the parameters passed with the method. The specified must be a *new* tree. You will receive an error if the tree already exists.

The **Create** method can only be used with a *closed* tree, it cannot be used on an open tree. Before you use the **Create** method, you must explicitly close any open tree objects (with the Close method.) You will receive an error if there are any open trees.

After you create a new tree, you don't have to open it with the Open method. The existing tree object points to the new tree.

The tree structure specified with *StructureName* determines what database fields the new tree is based on, what pages you can use to create tree nodes and detail values, and what record definitions Tree Manager saves tree-related data in. The tree structure must already exist.



For more information, see Creating Trees.

The new tree must be saved (**Save**, **SaveAs**, etc.) for this tree to be saved to the database.

Parameters

SetID

Specify the table indirection key for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "S", you must specify a SetID.

If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

UserKeyValue

Specify the User Key Value for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "U" or "B", you must specify a User Key Value.

If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

TreeName

Specify the name for this tree. This parameter takes a string value.

<i>EffDt</i>	Specify the effective date for this tree. This parameter takes a string value.
<i>StructureID</i>	Specify the name of the tree structure to use for this tree. This parameter takes a string value.

Returns

An integer: 0 if created successfully.

Example

```
&MYTREE.Create("", "", "PERSONAL_NEW", "05-05-1997", "PERSONAL_DATA");
```

Delete

Syntax

```
Delete(SetID, UserKeyValue, TreeName, EffDt, BranchName)
```

Description

The **Delete** method deletes the specified tree *from the database*. The **Delete** method can only be used with a *closed* tree, it cannot be used on an open tree. Before you use the **Delete** method, you must explicitly close any open tree objects (with the Close method.) You will receive an error if there are any open trees.

The specified tree does *not* have to match the tree identifier executing the method.

Parameters

<i>SetID</i>	<p>Specify the table indirection key for the tree. This parameter takes a string value. If the tree structure the tree is based on has its IndirectionMethod specified as "S", you must specify a SetID.</p> <p>If the tree structure doesn't have its IndirectionMethod specified as "S", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.</p>
<i>UserKeyValue</i>	<p>Specify the User Key Value for the tree. This parameter takes a string value. If the tree structure the tree is based on has its IndirectionMethod specified as "U" or "B", you must specify a User Key Value.</p> <p>If the tree structure doesn't have its IndirectionMethod specified as "U" or "B", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.</p>

<i>TreeName</i>	Specify the name for this tree. This parameter takes a string value.
<i>EffDt</i>	Specify the effective date for this tree. This parameter takes a string value.
<i>BranchName</i>	Specify the name of the branch for the tree. This parameter takes a string value. If the tree is unbranched, you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

Returns

An integer: 0 if deleted successfully.

Example

```
&MYTREE.Delete("", "", "PERSONAL_REN2", "05-05-1997", "");
```

Exists

Syntax

```
Exists(SetID, UserKeyValue, TreeName, EffDt, BranchName)
```

Description

The **Exists** method finds an existing tree, as specified by the parameters. The parameters must specify a unique tree. If a tree matching the parameters is found, the method returns 0.

Parameters

<i>SetID</i>	Specify the table indirection key for the tree. This parameter takes a string value. If the tree structure the tree is based on has its IndirectionMethod specified as "S", you must specify a SetID. If the tree structure doesn't have its IndirectionMethod specified as "S", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.
<i>UserKeyValue</i>	Specify the User Key Value for the tree. This parameter takes a string value. If the tree structure the tree is based on has its IndirectionMethod specified as "U" or "B", you must specify a User Key Value.

If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

TreeName

Specify the name for this tree. This parameter takes a string value.

EffDt

Specify the effective date for this tree. This parameter takes a string value.

BranchName

This parameter is required, but it is unused in this release. You must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

Returns

An integer: 0 if tree exists.

Example

```
&RSLT = &MYTREE.Exists("", "", "VENDOR_PER_DATA", "05-03-1998", "");
```

```
If &RSLT = 0 Then
```

```
    /* tree exists, do processing */
```

```
Else
```

```
    /* tree doesn't exist, do error processing */
```

```
End-if;
```

FindLeaf

Syntax

```
FindLeaf(RangeFrom, RangeTo)
```

Description

The **FindLeaf** method finds an existing leaf (specified by the *range* parameters) in the tree executing this method. If the leaf is found, a leaf object is returned. If the leaf isn't found, a NULL is returned.



This method only searches in the expanded portions of a tree. If a node isn't expanded, this method won't search that portion of the tree.

Wild cards can be used with the *range* parameters. An asterisk (*) can be used in place of a number of characters. A question mark (?) can be used in place of one character. You can also provide one of the ranges, and specify a NULL string for the other.

Suppose you wanted to find the leaf with the range 81005 to 82000. The following would be valid:

```
&MyLeaf = &MyTree.FindLeaf("81*", "82000");
```

This would also be valid:

```
&MyLeaf = &MyTree.FindLeaf("8100?", "82000");
```

You could also specify this:

```
&MYLEAF = &MYTREE.FindLeaf("81000", "");
```

If you specify the exact same value for both *RangeFrom* and *RangeTo*, the *RangeTo* parameter is ignored.

Parameters

<i>RangeFrom</i>	Specify the starting range of the leaf to be found. This parameter takes a string value.
<i>RangeTo</i>	Specify the ending range of the leaf to be found. This parameter takes a string value.

Returns

A reference to a leaf object.

Example

```
&MYLEAF = &MYTREE.FindLeaf("81000", "");
```

FindNode

Syntax

```
FindNode(NodeName, Description)
```

Description

The **FindNode** method returns a reference to the node object specified by the parameters passed to the method. If the node is found, a node object is returned. If the node isn't found, a NULL is returned.



If the node you're searching for is in an unexpanded portion of the tree, this method will expand the tree down to that node.

You should specify *either* the node name *or* the description, *not* both. You should specify a NULL string for the other. If both are provided, only the node name will be used and the description will be ignored.

Wild cards can be used with the parameters. An asterisk (*) can be used in place of a number of characters. A question mark (?) can be used in place of one character.

Suppose you wanted to find the node called 10200, with a description of "Human Resources". The following would be valid:

```
&MyNode = &MyTree.FindNode("", "Human*");
```

This would also be valid:

```
&MyNode = &MyTree.FindNode("1020?", "");
```

You could also use the following:

```
&MYNODE = &MYTREE.FindNode("10200", "");
```

Parameters

NodeName

Specify the name of the node you wish to find. This parameter takes a string value. You should only specify either the node name or the description.

Description

Specify the description of the node you wish to find. This parameter takes a string value. You should only specify either the node name or the description.

Returns

A reference to a node object.

Example

```
&MYNODE = &MYTREE.FindNode("10200", "");
```

FindRoot

Syntax

```
FindRoot()
```

Description

The **FindRoot** method returns a reference to the root node object of the tree object executing the method. This method can only be used on an open tree, not on a closed tree. This means you must have opened the tree with the Open method before you can find the root node.

This method will return an error if the tree has no nodes associated with it.

Returns

A reference to a node object.

Example

```
&MYNODE = &MYTREE.FindRoot();
```

InsertRoot

Syntax

```
InsertRoot (NodeName)
```

Description

The **InsertRoot** method inserts a node at the top of the tree object executing the method. *NodeName* is the name of the node you are creating. This parameter takes a string value. The **InsertRoot** method can only be used on an open tree, not on a closed tree. This means you must have opened the tree with the Open method before you can insert any nodes.

You can only insert the root node in a new tree, that is, a tree that was just created with the **Create** method. Also, after you create a new tree, you must insert the root node before you can insert any other nodes.

In a strict level tree, the first level must be defined before you can insert the root node.

This method will return an error if the tree already has nodes in it.

Example

```
&MYTREE.InsertRoot("000001");
```

LeafExists

Syntax

```
LeafExists (RangeFrom, RangeTo)
```

Description

The **LeafExists** method enables you to verify if the leaf specified by the parameters exists in the current tree. Use this method before using FindLeaf to verify a leaf exists before searching for it.



This method only searches in the expanded portions of a tree. If a node isn't expanded, this method won't search that portion of the tree.

Wild cards can be used with the *range* parameters. An asterisk (*) can be used in place of a number of characters. A question mark (?) can be used in place of one character. You can also provide one of the ranges, and specify a NULL string for the other.

Suppose you wanted to check the leaf with the range 81005 to 82000 exists. The following would be valid:

```
&MyResult = &MyTree.LeafExists("81*", "82000");
```

This would also be valid:

```
&MyResult = &MyTree.LeafExists("8100?", "82000");
```

You could also specify this:

```
&MyResult = &MYTREE.LeafExists("81000", "");
```

If you specify the exact same value for both *RangeFrom* and *RangeTo*, the *RangeTo* parameter is ignored.

Parameters

<i>RangeFrom</i>	Specify the starting range of the leaf you're verifying. This parameter takes a string value.
<i>RangeTo</i>	Specify the ending range of the leaf you're verifying. This parameter takes a string value.

Returns

An integer: 0 if the leaf exists.

Related Topics

NodeExists, FindLeaf

Open

Syntax

```
Open(SetID, UserKeyValue, TreeName, EffDt, BranchName, Update)
```

Description

The **Open** method opens the tree object specified by the parameters. The **Open** method can only be used with a **closed** tree, it cannot be used on an open tree. You cannot read or set any properties of a tree until after you open it. When you open a tree, you can specify whether you want to open it for update, or in read-only mode, with the *Update* parameter. If you try updating or writing to a tree opened in read-only mode, you will receive an error.

Parameters

SetID

Specify the table indirection key for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "S", you must specify a SetID.

If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

UserKeyValue

Specify the User Key value for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "U" or "B", you must specify a User Key Value.

If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

TreeName

Specify the name for the tree. This parameter takes a string value.

EffDt

Specify the effective date for the tree. This parameter takes a string value. The format must be in the correct format for the database platform the code is executing on.

BranchName

This parameter is required, but it is unused in this release. You must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

Update

Specify if you want to open the tree for update or in read-only mode. This parameter takes a Boolean value. Valid values for *Update* are:

- True – open tree structure in update mode (read-write)
- False – open tree structure in read-only mode

If you try updating or writing to a tree structure opened in read-only mode, you will receive an error.

Returns

An integer: 0 if opened successfully.

Example

The following example opens a tree object, then tests to see if the tree was actually opened:

```
&RSLT = &MYTREE.Open("", "", "PERSONAL_DATA", "05-05-1997", "", True);
```

```

If &RSLT = 0 Then

    /* normal processing */

Else

    /* error processing tree isn't open */

End-if;

```

NodeExists

Syntax

NodeExists (*NodeName*)

Description

The **NodeExists** method enables you to verify if the node specified by *NodeName* exists in the current tree. Use this method before using FindNode to verify a node exists before searching for it.



If the node you're searching for is in an unexpanded portion of the tree, this method will expand the tree down to that node.

Parameters

<i>NodeName</i>	Specify the name of the node you wish to find. This parameter takes a string value.
-----------------	---

Returns

An integer: 0 if the node exists.

Related Topics

FindLeaf, FindNode, LeafExists

Rename

Syntax

Rename (*FromSetId*, *FromUserKeyValue*, *FromTreeName*, *FromEffDt*, *FromBranchName*, *ToTreeName*);

Description

The **Rename** method renames a tree specified with the *From* parameters to the tree specified with *ToTreeName*. The tree specified by *ToTreeName* must be a *new* tree. You will receive an error if

the tree already exists. The tree specified by the *From* parameters does *not* have to match the tree executing the method.

The **Rename** method can only be used with a *closed* tree, it cannot be used on an open tree. Before you use the **Rename** method, you must explicitly close any open tree objects (with the Close method.) You will receive an error if there are any open trees.

To access the new tree, you must use the Open method.

Parameters

FromSetId

Specify the table indirection key for the tree to be copied from. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "S", you must specify a SetID.

If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

FromUserKeyValue

Specify the User Key Value for the tree to be copied from. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "U" or "B", you must specify a User Key Value.

If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B" you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

FromTreeName

Specify the name for the tree to be copied from. This parameter takes a string value.

FromEffDt

Specify the effective date for the tree to be copied from. This parameter takes a string value.

FromBranchName

This parameter is required, but it is unused in this release. You must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

ToTreeName

Specify the new name of the tree. This parameter takes a string value.

Returns

An integer: 0 if renamed successfully.

Example

```
&MYTREE.Rename ("", "", "PERSONAL_DATA3", "05-05-1997", "", "PERSONAL_REN2");
```

Save

Syntax

`Save()`

Description

The **Save** method writes any changes to the tree executing the method to the database. It also performs audits.



For more information, see Auditing Trees.

The **Save** method can only be used on an open tree, not on a closed tree. This means you must have opened the tree with the **Open** method before you can save it.

The tree object remains open after executing **Save**. You must execute the **Close** method on the object before it will be closed and the memory freed.

This method returns an optional Boolean value: True if the tree was saved with no errors, False if there are one or more errors.



If you're calling the Tree API from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

SaveAs

Syntax

`SaveAs(SetID, UserKeyValue, TreeName, EffDt, BranchName)`

Description

The **SaveAs** method writes any changes to the tree executing the method to the new tree specified with the parameters. It also performs audits.



For more information, see Auditing Trees.

The tree specified with the parameters must be a *new* tree. You will receive an error if the tree already exists. The new tree must have at least one different key field from the original tree.

The **SaveAs** method automatically closes the tree that is associated with the tree object executing the method, *and* associates that tree object with the new tree. Any changes made to the original tree since the last time the tree was saved will be discarded. For example, suppose you opened a

tree named `DEPENDENT_BENEF` and associated it with the variable `&MYTREE`. If you executed the `SaveAs` method with `&MYTREE`, changing the name to `DEPENDENT_BENEF2`, `&MYTREE` would now be associated with `DEPENDENT_BENEF2`, and the original tree, `DEPENDENT_BENEF`, would be closed.

```
&MYTREE.Open("", "", "DEPENDENT_BENEF", "02-02-1999", "", False);

/* do some processing */

&MYTREE.SaveAs("", "", "DEPENDENT_BENEF2", "02-05-1999", "");
```

`&MYTREE` is now associated with `DEPENDENT_BENEF2`, not with the original open tree.



If you're calling the Tree API from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

Parameters

SetID

Specify the table indirection key for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "S", you must specify a SetID.

If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

UserKeyVal

Specify the User Key Value for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "U" or "B", you must specify a User Key Value.

If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

TreeName

Specify the name for this tree. This parameter takes a string value.

EffDt

Specify the effective date for this tree. This parameter takes a string value.

BranchName

This parameter is required, but it is unused in this release. You must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

Returns

This method returns an optional Boolean value: True if the tree was saved with no errors, False if there is one or more errors.

Example

```
&MYTREE.SaveAs("", "", "DEPENDENT_BENEF2", "02-05-1999", "");
```

SaveAsDraft

Syntax

```
SaveAsDraft(SetID, UserKeyValue, TreeName, EffDt, BranchName)
```

Description

The **SaveAsDraft** method writes any changes to the tree executing the method to the new tree specified with the parameters. However, it does *not* perform audits. Trees that are saved in this way are marked as "invalid" and will not be valid (that is, cannot be used with other PeopleTools) until the tree audit is passed during Save or SaveAs.



For more information, see Auditing Trees.

The tree specified with the parameters must be a *new* tree. You will receive an error if the tree already exists. The new tree must have at least one different key field from the original tree.

The **SaveAsDraft** method automatically closes the tree that is associated with the tree object executing the method, *and* associates that tree object with the new tree. Any changes made to the original tree since the last time the tree was saved will be discarded. (See SaveAs)



If you're calling the Tree API from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

Parameters

SetID

Specify the table indirection key for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "S", you must specify a SetID.

If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

UserKeyValue

Specify the User Key Value for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "U" or "B", you must specify a User Key Value.

If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

TreeName

Specify the name for this tree. This parameter takes a string value.

EffDt

Specify the effective date for this tree. This parameter takes a string value.

BranchName

This parameter is required, but it is unused in this release. You must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

Returns

This method returns an optional Boolean value: True if the tree was saved with no errors, False if there is one or more errors.

Example

```
&MYTREE. SaveAsDraft ("", "", "PERSONAL_EDIT2", "05-05-1997", "");
```

SaveDraft**Syntax**

```
SaveDraft()
```

Description

The **SaveDraft** method writes any changes to the tree executing the method to the database. However, it does *not* perform audits. Trees that are saved in this way are marked as "invalid" and will not be valid (that is, cannot be used with other PeopleTools) until the tree audit is passed.



For more information, see Auditing Trees.

The **SaveDraft** method can only be used on an open tree, not on a closed tree. This means you must have opened the tree with the Open method before you can save it.

The tree object remains open after executing **SaveDraft**. You must execute the Close method on the object before it will be closed and the memory freed.

This method returns an optional Boolean value: True if the tree was saved with no errors, False if there is one or more errors.



If you're calling the Tree API from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

Tree Class Properties

AllValues

When the tree is audited, the **AllValues** property specifies whether the audit should verify that the tree includes all user data detail values. This property returns a Boolean value: True, check all values, False, do not check all values.



For more information, see Tree Manager.

If this is a new tree, and you do not set this property, a default value of False is automatically set.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

Example

```
&MyTree.AllValues = False;
```

AuditDetails

Most of the time, when you create a tree structure with Detail records or fields in your tree structure, the terminal nodes of your trees based on this structure have details. However, in some special cases (like in the PeopleSoft Internet Architecture) your trees based on this structure have terminal nodes that won't have details. In these cases, set this property to False.

The default value for this property is True.

Branches

This property returns a reference to a branch collection object. This property is only used with branched trees: it returns NULL for trees that aren't branched. (see Branch Collection Class Methods)

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

BranchLevel

This property returns the level number (as a number) where this branch occurs in the tree. This property is only used with trees with levels: it returns zero for trees that don't have levels.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

Example

```
&LevelNumber = &MyTree.BranchLevel;
```

BranchName

This property returns the name of the specific branch for the tree, as a string. This property is only used with branched trees: it returns NULL for trees that aren't branched.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

Example

```
&BranchName = &MyTree.BranchName;
```

Category

This property returns the category name for the tree, as a string. The category is a high-level grouping under which you can organize your tree structures and tree definitions. If you are creating a new tree, this property is required.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

Example

```
&MyTree.Category = "PurchaseOrders";
```

CheckLeafUserData

In Tree Manager, the details must already exist when there are inserted into the tree. However, sometimes this is impractical with the Tree API. If you're inserting nodes and leaves to create a new tree, and the details don't exist, set this property to False.

The default value for this property is True.

This property is read-write.

Description

This property returns the description of the tree, as a string. If you are creating a new tree, this property is required.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

Example

```
&MyTree.Description = "Departmental Security";
```

DuplicateLeaves

In most trees, you want to make sure that each detail value only appears once in the tree. If you want to permit duplicate values, set this property to True.

If this is a new tree, and you do not set this property, a default value of False is automatically set.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

EffDt

This property returns the effective date of the tree, as a date.

If this is a new tree, and you do not set this property, a default value of the current date is automatically set.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

Example

The following example uses a PeopleCode system variable for setting the effective date.

```
&MYTREE.EffDt = %ClientDate;
```

HasDetailRanges

This property returns True if the tree has any detail ranges.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

IsBranched

This property indicates whether the tree is branched or not. Possible values for this property are True and False: True if the tree is branched, False if it isn't.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

Example

```
If &MyTree.IsBranched Then  
  
    /* Do branched processing */  
  
Else  
  
    /*Do unbranched processing */  
  
End-If;
```

IsChanged

This property returns True if the tree has been changed but not saved, False if the tree is unchanged.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

Example

```
If &MyTree.IsChanged Then  
  
    &MyTree.Save();  
  
End-if;  
  
&MyTree.Close();
```

IsOpen

This property returns True if the tree is open, False if the tree is closed.

This property is read-only.

Example

```
If Not &MyTree.IsOpen Then

    &MyTree.Open("", "", "PERSONAL_DATA", &Date, "", True);

End-If;
```

IsQueryTree

This property returns True if the tree is a Query access tree. It returns False if the tree is not a Query Access Tree or a regular branched or unbranched tree.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

IsValid

This property returns True if the tree is a valid tree that has passed all audits. It returns False otherwise.



For more information, see Auditing Trees.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

KeyBranchName

This property returns the branch name of the tree as a string if the tree is branched. If not branched, this property returns a blank string.

This property can be used with a **closed** tree, that is, before you use the **open** method. You can use this property to open a tree without explicitly knowing its branch

This property is read-only.

KeyEffDt

This property returns the effective date of the tree as a string if the tree is effective dated. If not effective dated, this property returns a blank string.

This property can be used with a **closed** tree, that is, before you use the **open** method. You can use this property to open a tree without explicitly knowing its effective date.

This property is read-only.

KeyName

This property returns the name of the tree as a string.

This property can be used with a **closed** tree, that is, before you use the **open** method. You can use this property to open a tree without explicitly knowing its name.

If the tree has already been opened, you can use the **Name** property to get the tree name, or if you want to create a new tree.

This property is read-only.

KeySetId

This property returns the SetID of the tree as a string if the tree has a SetID. If the tree doesn't have a SetID, this property returns a blank string.

This property can be used with a **closed** tree, that is, before you use the **open** method. You can use this property to open a tree without explicitly knowing its SetID.

This property is read-only.

KeyUserKeyValue

This property returns the UserKeyValue of the tree as a string if the tree has a UserKeyValue. If the tree doesn't have a UserKeyValue, this property returns a blank string.

This property can be used with a **closed** tree, that is, before you use the **open** method. You can use this property to open a tree without explicitly knowing its UserKeyValue.

This property is read-only.

LeafCount

This property returns the number of leaves in the tree. Valid value range is a number between 0 and 2 billion.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

LevelCount

This property returns the number of levels defined for the tree. If the tree doesn't have any levels, this property returns zero (0). See the LevelUse property.

Valid value range is a number between 0 and 99.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

Levels

This property returns a reference to a level collection object. This property is only used with trees that have levels: it returns NULL for trees that don't have levels. (see LevelCollection Methods)

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

LevelUse

This property returns the level use of the tree. The valid values are:

- "S" – Strict levels
- "L" – Loose levels
- "N" – No levels

If this is a new tree, and you do not set this property, the default value is Strict Levels.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

Example

```
If &MyTree.LevelUse = "S" Then  
  
    /* add levels */  
  
End-If;
```

Name

This property returns the name of the tree, as a string. This property is required if you are creating a *new* tree.



Do not use this property to change the name of an existing tree. The change will *not* be reflected in the database. You must use the Rename method to change the name of an existing tree.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

Example

```
&MyNewTree.Name = "DEPT_SECURITY_ADD";
```

NodeCount

This property returns the number of nodes for the tree. A valid tree must have at least one node.

Valid value range is a number between 1 and 2 billion.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

ParentLevel

This property returns the number of the parent branch level in a branched tree. If the tree does *not* have levels or is an unbranched tree, zero (0) is returned. Valid value range is a number between 0 and 100.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

ParentName

This property returns the name of the parent branch in a branched tree, as a string. If the tree does *not* have levels, is an unbranched tree, or if the tree is the root level, NULL is returned.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

PerformanceMethod

This property sets the method to use for select and join statements used to generate SQL with the tree. This property is used with nVision layouts and Query.

The valid values for this property are:

- "L" – Suppress Join: Use Literal Values
- "S" – Sub-SELECT Tree Selector

- "J" – Join to Tree Selector
- "D" – User Application Defaults



For more information, see Tree Manager and PeopleSoft nVision.

If this is a new tree, and you do not set this property, the default value "D" is automatically set.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

PerformanceSelector

This property selects the tree selectors for nVision for the tree. This property is used with nVision layouts and Query.

The valid values for this property are:

- "D" – Dynamic Selectors
- "S" – Static Selectors



For more information, see PeopleSoft nVision.

If this is a new tree, and you do not set this property, the default value "S" is automatically set.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

PerformanceSelectorOption

This property selects the tree selector options for nVision for the tree. This property is used with nVision layouts and Query.

The valid values for this property are:

- "S" – Single values
- "R" – Ranges of values ($\geq \dots \leq$)
- "B" – ranges of values (BETWEEN)



For more information, see PeopleSoft nVision.

If this is a new tree, and you do not set this property, the default value "R" is automatically set.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

SetID

This property returns the SetID for the tree, as a string. This property is only used when the IndirectionMethod property for the tree structure the tree is based on has been set to "S". Otherwise, this property returns NULL.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

SilentMode

This property determines whether error messages are sent as error messages to the display or not. This property takes a Boolean value: True means messages are not sent, False means they are sent. The default value for this property is True.

This property is read-write.

Status

This property returns the effective status of the tree. The valid values are:

- "A" – Active
- "I" – Inactive

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

Structure

This property returns a reference to the tree structure of the tree. Before you can use some of the methods or any of the properties of a tree structure, you must **Open** the tree structure.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

StructureName

This property returns the name (*structure ID*) of the tree structure the tree is based on, as a string. This property can only be set with a new tree. You will receive an error if you try to change the tree structure for an existing tree.

For a new tree, this property must be set before the UserKeyValue or SetID properties can be set.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

TotalNodeCount

This property is set by the user to indicate how many nodes a tree may have. Use this property just after creating a new tree to ensure the proper number and spacing of tree nodes. In most cases, it isn't necessary to set this value. The default Node gap size is automatically calculated by Tree Manager for you.

For example, if you are about to create a large tree through PeopleCode or another programming language, it's a good idea to set this property to the number of nodes you expect the tree will have. It's okay if the tree ends up having fewer nodes. However, you need specify at least the number of nodes you'll be including; otherwise you won't be able to add additional nodes. The system will use this rough estimate to help allocate space for the tree so that it's stored in the database in an efficient manner.



For more information, see [Creating Evenly Gapped Trees](#).

To get the actual number of nodes a tree contains, use the NodeCount property.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

UserKeyValue

Use this property to set the key field value for the Node User Key Field specified from the NodeRecord on the associated tree structure.

This property is only used when the IndirectionMethod property for the tree structure has been set to "U" or "B". Otherwise, this property returns NULL.



For more information, see Tree Manager.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

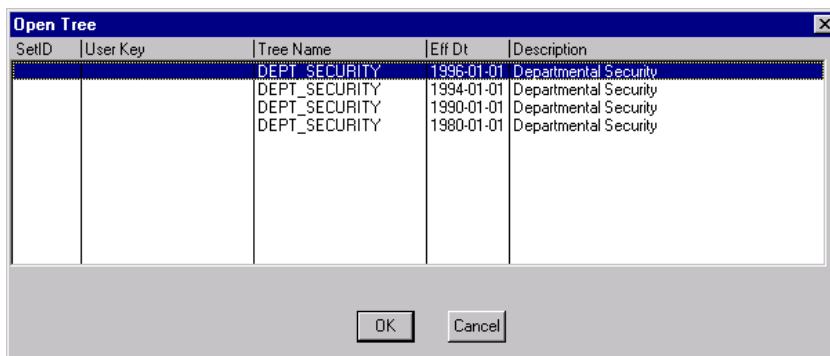
This property is read-write.

Tree Collection

A tree collection is returned from the FindTree method used with a session object:

```
&MYCOLL = &SESSION.FindTree("", "", "DEPT", "", "");
```

All collections have GUI analogies. The following dialog box represents the tree collection.



Tree Collection Class GUI representation

All of the tree collection methods return a **closed** tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method. Working with closed trees can improve your performance.

The following example gets all the trees in a tree collection, opens each, and does some processing:

```
Local ApiObject &SESSION;

Local ApiObject &MYTREE;

Local ApiObject &MYTREECOLL;

Local String &SETID, &USERKEY, &TREENAME, &EFFDT, &BRANCHNAME;

Local Boolean &UPDATE;

&SESSION = GetSession();

&SESSION.Connect(1, "EXISTING", "", "", 0);
```

```

&MYTREECOLL = &SESSION.FindTree("", "", "", "", "");

&MYTREE = &MYTREECOLL.First;

&COUNT = 0;

For &I = 1 to &MYTREECOLL.Count

    &SETID = &MYTREE.KeySetId;

    &USERKEY = &MYTREE.KeyUserKeyValue;

    &TREENAME = &MYTREE.KeyName;

    &EFFDT = &MYTREE.KeyEffDt;

    &BRANCHNAME = &MYTREE.KeyBranchName;

    &UPDATE = True;

    &MYTREE.Open(&SETID, &USERKEYVALUE, &TREENAME, &EFFDT, &BRANCHNAME,
&UPDATE);

        /* do processing */

    &MYTREE.Save();

    &MYTREE.Close();

    &COUNT = &COUNT + 1;

    If &COUNT <> &MYTREECOLL.Count Then;

        &MYTREECOLL.Next;

    End-If;

End-For;

```

Tree Collection Method

Item

Syntax

Item(*SetId, UserKeyValue, TreeName, EffDt, BranchName*)

Description

The **Item** method returns the specified tree in the tree collection object executing the method as a **closed** tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method.

Parameters

SetId

Specify the table indirection key for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "S", you must specify a SetID.

If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

UserKeyVal

Specify the User Key Value for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "U" or "B", you must specify a User Key Value.

If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.

TreeName

Specify the name for this tree. This parameter takes a string value.

EffDt

Specify the effective date for this tree. This parameter takes a string value.

BranchName

Specify the name of the root branch for the tree. This parameter takes a string value. If the tree is unbranched, you must enter a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.



If the tree is branched and you don't specify the correct branch, a tree will not be returned. You can use the KeyBranchName property to return the branch name of an unopened tree.

Example

```
&MYTREE = &MYTREECOLL.Item("", "", "PERSONAL_DATA", "05-05-1997", "");
```

Tree Collection Properties

Count

Returns the number of trees for the tree collection object.

First

The **First** property returns the first tree in the tree collection object executing the method as a **closed** tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method.

Last

The **Last** property returns the last tree in the tree collection object executing the method as a **closed** tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method.

Next

The **Next** property returns the next tree in the tree collection object executing the method as a **closed** tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method. You can only use this method after you have used either the **First** or **Item** properties; otherwise the system doesn't know where to start in the tree.

Tree Structure Class

Tree structure objects are instantiated from the following:

- From a session object with the `GetTreeStructure` method.
- From a tree object with the `Structure` property.
- From a Tree Structure Collection and the `First`, `Last` or `Next` properties or the `Item` method.

Tree Structure Class Methods

Close

Syntax

```
Close()
```

Description

The **Close** method closes the tree structure object executing the method, freeing the memory associated with that object, and discarding any changes made to the tree structure. The **Close** method can only be used on an open tree structure, not on a **closed** tree structure. This means you must have opened the tree structure with the **Open** method before you can save it. If you want to save any changes, you must use the **Save** method before using **Close**.

Copy

Syntax

```
Copy(FromStructId, ToStructId)
```

Description

The **Copy** method copies the tree structure specified with *FromStructId* to the tree structure specified with *ToStructId*. Both parameters take string values. The tree structure specified by *ToStructId* must be a *new* tree structure. You will receive an error if the tree structure already exists. The tree structure specified by *FromStructId* does *not* have to match the closed tree structure executing the method.

For example, if you created a closed tree structure for the ACCESS_GROUP tree structure, that tree structure wouldn't have to be the *FromStructId*.

```
&MyTreeStruct = &MySession.GetTreeStructure(ACCESS_GROUP);  
  
&MyTreeStruct.Copy(PERSONAL_DATA, PERSONAL_DATA2);
```

The **Copy** method can only be used with a **closed** tree structure, it cannot be used on an open tree structure. Before you use the **Copy** method, you must explicitly close any open tree structure objects (with the **Close** method.) You will receive an error if there are any open tree structures.

To access the new tree structure, you must use the **Open** method.

Create

Syntax

```
Create(StructId, Type)
```

Description

The **Create** method creates a tree structure with the name *StructId*, and of the type *Type*. Both parameters take a string value. The tree structure specified by *StructId* must be a *new* tree structure. You will receive an error if the tree structure already exists.

The **Create** method can only be used with a **closed** tree structure, it cannot be used on an open tree structure. Before you use the **Create** method, you must explicitly close any open tree structure objects (with the **Close** method.) You will receive an error if there are any open tree structures.

After you create a new tree structure, you can open it with the **Open** property.

The valid values for *Type* are:

- "D" - create a detail tree structure
- "S" - create a summary tree structure

The new tree structure must be opened, the required properties set (that is, **Description**, and so on) and then saved (**Save** method) for this structure to be saved to the database.

Delete

Syntax

```
Delete (StructId)
```

Description

The **Delete** method deletes the tree structure specified by *StructId* from the database. The **Delete** method can only be used with a tree structure identifier, it cannot be used on a fully instantiated, open tree structure. Before you use the **Delete** method, you must explicitly close any open tree structure objects (with the **Close** method.) You will receive an error if there are any open tree structures.

The tree structure specified by *StructId* does *not* have to match the tree structure identifier executing the method.

Open

Syntax

```
Open (StructId, Update)
```

Description

The **Open** method opens the tree structure specified by *StructId*. *StructId* takes a string value. You cannot read or set any properties of a tree structure until after you open it.

When you open a tree structure, you can specify whether you want to open it for update, or in read-only mode, with the *Update* parameter. *Update* takes a Boolean value. Valid values for *Update* are:

- "True" – open tree structure in update mode (read-write)
- "False" – open tree structure in read-only mode

If you try updating or writing to a tree structure opened in read-only mode, you will receive an error.

Example

```
&MyTreeStruct = &MySession.GetTreeStructure("ACCESS_GROUP");  
  
&MyTreeStruct.Open("ACCESS_GROUP", True);
```

Rename

Syntax

```
Rename (FromStructId, ToStructId)
```

Description

The **Rename** method renames the tree structure specified with *FromStructId* to the new name specified by *ToStructId*. Both parameters take a string value. The tree structure specified by *ToStructId* must be a *new* tree structure. You will receive an error if the tree structure already exists. The change is automatically reflected in the database.

The **Rename** method can only be used with a **closed** tree structure, it cannot be used on an open tree structure. Before you use the **Rename** method, you must explicitly close any open tree structure objects (with the **Close** method.) You will receive an error if there are any open tree structures.

Save

Syntax

```
Save ()
```

Description

The **Save** method writes any changes to the tree structure executing the method to the database.

The **Save** method can only be used on an open tree structure, not on a closed tree structure. This means you must have opened the tree structure with the **Open** method before you can save it.

The tree structure object remains open after executing **Save**. You must execute the **Close** method on the object before it will be closed and the memory freed.



If you're calling the Tree API from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

Tree Structure Class Properties

Description

This property returns the description for the tree structure, as a string. The description is the text that displays in list boxes or reports.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

Example

```
&MYTREESTRUCT.Description = "Department Security Chart";
```

DetailComponent

This property returns the name of the detail component for the tree structure as a string.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

DetailField

This property returns the name of the detail field for the tree structure, as a string. This property is only valid if you're defining a detail tree structure. If you're creating a node-oriented tree structure, this property is not required. This field must exist on the specified DetailRecord.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

Example

The following example sets the **DetailPage**, **DetailRecord** and **DetailField** for a tree structure. Note that the names of the page, record, and field are all capitalized: this is required for all PeopleSoft page, record and field names.

```
&MYTREESTRUCT.DetailPage = "BUS_UNIT_TREE_INV";  
  
&MYTREESTRUCT.DetailRecord = "BUS_UNIT_TBL_IN";  
  
&MYTREESTRUCT.DetailField = "BUSINESS_UNIT";
```

DetailMenu

This property returns the name of the detail menu for the tree structure, as a string. If the page you use for detail values (DetailPage property) is part of a component, and you want to have access to the other pages in that group when you edit a node value, enter the name of the menu and menu bar for the component in the **DetailMenu** and DetailMenuBar properties.

This property is only valid if you're defining a detail tree structure. If you're creating a node-oriented tree structure, this property isn't required.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

DetailMenuBar

This property returns the name of the detail menu bar for the tree structure, as a string. If the page you use for detail values (DetailPage property) is part of a component, and you want to have access to the other pages in that group when you edit a node value, enter the name of the menu and menu bar for the component in the DetailMenu and **DetailMenuBar** properties.

This property is only valid if you're defining a detail tree structure. If you're creating a node-oriented tree structure, this property isn't required.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

DetailMenuItem

This property returns the name of the detail menu item for the tree structure as a string.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

DetailPage

This property returns the name of the detail page for the tree structure as a string. If the page you use for detail values (**DetailPage** property) is part of a component, and you want to have access to the other pages in that group when you edit a node value, enter the name of the menu and menu bar for the component in the DetailMenu and DetailMenuBar properties.

This property is only valid if you're defining a detail tree structure. If you're creating a node-oriented tree structure, this property isn't required.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

Example

The following example sets the **DetailPage**, **DetailRecord** and **DetailField** for a tree structure. Note that the names of the page, record and field are all capitalized: this is required for all PeopleSoft page, record, and field names.

```
&MYTREESTRUCT.DetailPage = "BUS_UNIT_TREE_INV";  
  
&MYTREESTRUCT.DetailRecord = "BUS_UNIT_TBL_IN";  
  
&MYTREESTRUCT.DetailField = "BUSINESS_UNIT";
```

DetailRecord

This property returns the name of the detail record for the tree structure as a string. This property is only valid if you're defining a detail tree structure. If you're creating a node-oriented tree structure, this property isn't required.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

Example

The following example sets the **DetailPage**, **DetailRecord** and **DetailField** for a tree structure. Note that the names of the page, record and field are all capitalized: this is required for all PeopleSoft page, record, and field names.

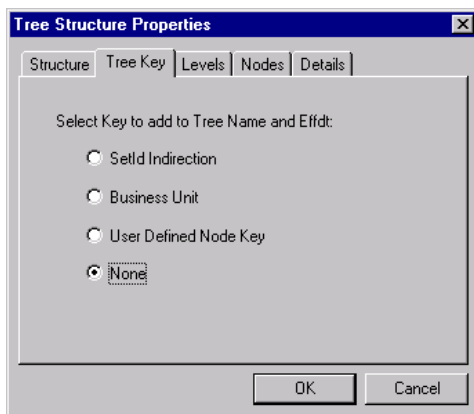
```
&MYTREESTRUCT.DetailPage = "BUS_UNIT_TREE_INV";

&MYTREESTRUCT.DetailRecord = "BUS_UNIT_TBL_IN";

&MYTREESTRUCT.DetailField = "BUSINESS_UNIT";
```

IndirectionMethod

When you look at the properties of a detail tree structure, one of the tabs is Tree Key.



Tree Key tab in tree structure properties

This values for this tab are specified with the IndirectionMethod property. You can set this property to one of the following values:

- "S" – SetID Indirection
- "B" – Business Unit
- "U" – User Defined Node Key

- "N" – None



For more information, see Tree Manager.

Both the *User Defined Node Key* and the *Business Unit* are the key field set with the `NodeRecord` property for this tree structure. If *Business Unit* is set, the key field set with the **NodeRecord** property is a business unit.

If you are creating a new tree structure, and you do not explicitly set this property, **None** ("N") is the default value, and is automatically set.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

KeyName

This property returns the name of the tree structure as a string.

This property can be used with a **closed** tree structure. You can use this property to open a tree structure without explicitly knowing its name.

This property is read-only.

Example

```
Local ApiObject &TLIST;

Local ApiObject &TREESTRUCT;

Local ApiObject &MYSESSION;

Local boolean &RSLT;

Local string &NAME;

&MYSESSION = GetSession();

&RSLT = &MYSESSION.CONNECT(1, "EXISTING", "", "", 0);

If &RSLT Then

    /* connection is good */

    &TLIST = &MYSESSION.FindTreeStructure("");

    For &I = 1 to &TLIST.Count
```

```

        &TREESTRUCT = &TSLIST.Item(&I);

        &NAME = &TREESTRUCT.KeyName;

        &TREESTRUCT.Open(&NAME);

        /* do processing */

    End-For;

Else

    /* do error processing */

End-if;

```

LevelComponent

This property returns the name of the level component for the tree structure as a string.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

LevelMenu

This property returns the name of the level menu for the tree structure as a string. If the page you use for level values (**LevelPage** property) is part of a component, and you want to have access to the other pages in that group when you edit a node value, enter the name of the menu and menu bar for the component in the **LevelMenu** and **LevelMenuBar** properties.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

LevelMenuBar

This property returns the name of the level menu bar for the tree structure as a string. If the page you use for level values (**LevelPage** property) is part of a component, and you want to have access to the other pages in that group when you edit a node value, enter the name of the menu and menu bar for the component in the **LevelMenu** and **LevelMenuBar** properties.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

LevelMenuItem

This property returns the name of the level menu item for the tree structure as a string.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

LevelPage

This property returns the name of the level page for the tree structure object as a string. If you're creating a new tree structure, and you do not explicitly set this property, TREE_LEVEL is the default value, and is automatically set.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

LevelRecord

This property returns the name of the level record for the tree structure object as a string. If you're creating a new tree structure, and you do not explicitly set this property, TREE_LEVEL_TBL is the default value, and is automatically set.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

Name

This property returns the name of the tree structure as a string. This is also called the *structure ID*. You would use this property to set the name (structure ID) of a tree structure. This property is required if you are creating a *new* tree structure.



Do not use this property to change the name of an existing tree structure. The change will *not* be reflected in the database. You must use the Rename method to change the name of an existing tree structure.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

Example

```
&MYTREESTRUCT.Name = "PERSONAL_DATA";
```

NodeComponent

This property returns the name of the node component for the tree structure as a string.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

NodeField

This returns the name of the field used for storing information about nodes for the tree structure as a string. If you're creating a new tree structure, and you do not explicitly set this property, **TREE_NODE** is the default value, and is automatically set. This field must exist on the specified **NodeRecord**.



For more information, see Tree Manager.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

Example

```
&MYTREESTRUCT.NodeField = "DEPT_ID";
```

NodeMenu

This property returns the name of the node menu for the tree structure as a string. If the page you use for node values (**NodePage** property) is part of a component, and you want to have access to the other pages in that group when you edit a node value, enter the name of the menu and menu bar for the component in the **NodeMenu** and **NodeMenuBar** properties.

If you're creating a new tree structure, and you do not explicitly set this property, **TREE_MANAGER** is the default value, and is automatically set.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

NodeMenuBar

This property returns the name of the node menu bar for the tree structure as a string. If the page you use for node values (**NodePage** property) is part of a component, and you want to have access to the other pages in that group when you edit a node value, enter the name of the menu and menu bar for the component in the **NodeMenu** and **NodeMenuBar** properties.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

NodeMenuItem

This property returns the name of the node menu item for the tree structure as a string.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

NodePage

This returns the name of the page used for storing information about nodes for the tree structure as a string. If you're creating a new tree structure, and you do not explicitly set this property, TREE_NODE is the default value, and is automatically set.



For more information, see Tree Manager.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

NodeRecord

This returns the name of the record used for storing information about nodes for the tree structure as a string. If you're creating a new tree structure, and you do not explicitly set this property, TREE_NODE_TBL is the default value, and is automatically set.



For more information, see Tree Manager.

If the IndirectionMethod property of this tree structure has been set to "U" or "B", use this property to specify the record that the **NodeUserKeyField** property will use.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

NodeUserKeyField

Use this property to set the key field from the record specified with **NodeRecord** for use with the **IndirectionMethod**. This property is identical to the **UserKeyValue** property for a tree.

This property is only used when the **IndirectionMethod** property for the tree structure has been set to "U" or "B". Otherwise, this property returns NULL.



For more information, see Tree Manager.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

SummarySetId

This property returns the SetID for the detail tree whose detail values the summary tree structure as a string summarizes. This property is only valid for Summary tree structures. Otherwise, this property returns NULL.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

SummaryLevelNumber

This property returns the level number for the detail tree whose detail values the summary tree structure summarizes. The level number indicates the level in the detail tree whose nodes the summary tree will use as detail values. The top level in the detail tree is 1, the next is level 2, and so on. Valid values are numbers between 1 and 99.

This property is only valid for Summary tree structures. Otherwise, this property returns NULL.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

SummaryTreeName

This property returns the name of the detail tree whose detail values the summary tree structure summarizes, as a string. This property is only valid for Summary tree structures. Otherwise, this property returns NULL.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

SummaryUserKeyValue

This property returns the name of the tree that the node records the tree structure will be summarized to, as a string. This property is only valid for Summary tree structures. This property is only used when the IndirectionMethod property for the tree structure has been set to "U" for *User Defined Keys* or "B" for *Business Unit*. Otherwise, this property returns NULL.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

Type

This property returns the type of the tree structure as a string, whether it is a detail or summary tree structure. The valid values are:

- "D" - detail tree structure
- "S" - summary tree structure

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

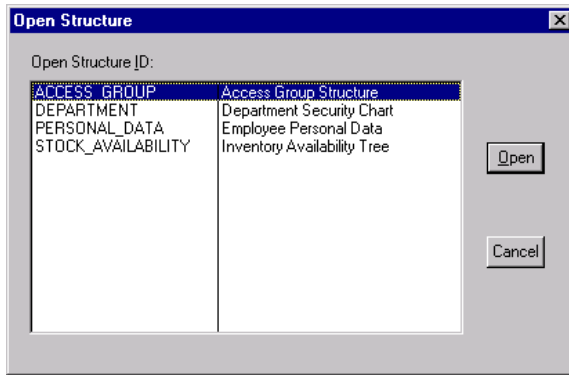
This property is read-write.

Tree Structure Collection

A tree structure collection is returned from the FindTreeStructure method used with a session object:

```
&MYCOLL = &SESSION.FindTreeStructure("");
```

All collections have GUI analogies. The following dialog box represents the tree structure collection:



Tree Structure Collection GUI representation

Tree Structure Collection Method

Item

Syntax

```
Item(StructureID)
```

Description

The **Item** method returns the tree structure specified by *StructureID* in the tree structure collection object executing the method as a **closed** tree structure. A closed tree structure is a tree structure object with just the key fields filled in. The rest of the data is not present. To open the tree structure, you must use the **Open** method. *StructureID* takes a string value.

Tree Structure Collection Properties

Count

Returns the number of tree structures for the tree structure collection object.

First

The **First** property returns the first tree structure in the tree structure collection object executing the method as a **closed** tree structure. A closed tree structure is a tree structure object with just the key fields filled in. The rest of the data is not present. To open the tree structure, you must use the **Open** method.

Last

The **Last** property returns the last tree structure in the tree structure collection object executing the method as a **closed** tree structure. A closed tree structure is a tree structure object with just

the key fields filled in. The rest of the data is not present. To open the tree structure, you must use the **Open** method.

Next

The **Next** property returns the next tree structure in the tree structure collection object executing the method as a **closed** tree structure. A closed tree structure is a tree structure object with just the key fields filled in. The rest of the data is not present. To open the tree structure, you must use the **Open** method. You can only use this method after you have used either the **First** or **Item** properties: otherwise the system doesn't know where to start in the tree.

Summary of Class Methods and Properties

The following is a summary of the methods and properties associated with the Tree classes.

Summary of Methods for Tree Classes

<i>Methods</i>	<i>Branch Coll</i>	<i>Leaf</i>	<i>Level Coll/ Level</i>	<i>Node</i>	<i>Tree/Tree Coll</i>	<i>Tree Structure/ Tree Structure Coll</i>
Add			X			
Audit					X	
AuditByName					X	
Branch				X		
Close					X	X
Copy					X	X
Create			X		X	X
Cut		X		X		
Delete		X	X	X	X	X
DeleteByName				X		
DeleteByRange		X				
Exists					X	
Expand					X	
FindLeaf					X	
FindNode					X	
FindRoot					X	
InsertChildLeaf				X		

Methods	Branch Coll	Leaf	Level Coll/ Level	Node	Tree/Tree Coll	Tree Structure/ Tree Structure Coll
InsertChildNode				X		
InsertChildRecord				X		
InsertDynChildLeaf				X		
InsertDynSib		X				
InsertRoot					X	
InsertSib		X		X		
InsertSibRecord				X		
Item	X		X		X	X
MoveAsChild		X		X		
MoveAsChildByName		X		X		
MoveAsSib		X		X		
MoveAsSibByName				X		
MoveAsSibByRange		X				
Open					X	X
PasteChild				X		
PasteSib		X		X		
Remove			X			
Rename				X	X	X
Save			X		X	X
SaveAs					X	
SaveAsDraft					X	
SaveDraft					X	
SwitchLevel				X		
Unbranch				X		

Summary of Properties for Tree Classes

RO = Read-Only.

Properties	Branch Coll	Leaf	Level	Node	Tree	Tree Structure
AllChildCount				RO		
AllChildNodeCount				RO		
AllValues					X	
AllValuesAudit			X			
AuditDetails					X	
Branches					RO	
BranchLevel					RO	
BranchName					RO	
Category					X	
CheckLeafUserData					X	
ChildLeafCount				RO		
ChildNodeCount				RO		
Count	X		X ²		X ²	X ²
Description			X	RO	X	X
DetailComponent						X
DetailField						X
DetailMenu						X
DetailMenuBar						X
DetailMenuItem						X
DetailPage						X
DetailRecord						X
DuplicateLeaves					X	
Dynamic		X				
EffDt					X	
First	X		X ²		X ²	X ²
FirstChildLeaf				RO		
FirstChildNode				RO		
HasChildLeaves				RO		
HasChildNodes				RO		
HasChildren				RO		
HasDetailRanges					RO	

Properties	Branch Coll	Leaf	Level	Node	Tree	Tree Structure
HasNextSib		RO		RO		
HasPrevSib		RO		RO		
IndirectionMethod						X
IsBranched				RO	RO	
IsChanged		RO		RO	RO	
IsDeleted		RO		RO		
IsInserted		RO		RO		
IsOpen					RO	
IsQueryTree					RO	
IsRoot				RO		
IsValid					RO	
KeyBranchName ¹					RO	
KeyEffDt ¹					RO	
KeyName ¹					RO	RO
KeySetId ¹					RO	
KeyUserKeyValue ¹					RO	
Last	X		X ²		X ²	X ²
LastChildLeaf				RO		
LastChildNode				RO		
LeafCount					RO	
LevelComponent						X
LevelCount					RO	
LevelMenu						X
LevelMenuBar						X
LevelMenuItem						X
LevelNumber				X		
LevelPage						X
LevelRecord						X
Levels					RO	
LevelUse					X	
Name			X	X	X	X

Properties	Branch Coll	Leaf	Level	Node	Tree	Tree Structure
Next	X		X²		X²	X²
NextSib		RO		RO		
NodeComponent						X
NodeCount					RO	
NodeField						X
NodeMenu						X
NodeMenuBar						X
NodeMenuItem						X
NodePage						X
NodeRecord						X
NodeUserKeyField						X
Number			X			
Parent		RO		RO		
ParentLevel					RO	
ParentName					RO	
PerformanceMethod					X	
PerformanceSelector					X	
PerformanceSelectorOption					X	
PrevSib		RO		RO		
RangeFrom		X				
RangeTo		X				
SetId					X	
SilentMode					X	
State				RO		
Status					X	
Structure					RO	
StructureName					X	
SummarySetId						X
SummaryLevelNumber						X
SummaryTreeName						X

Properties	Branch Coll	Leaf	Level	Node	Tree	Tree Structure
SummaryUserKeyVal ue						X
TreeBranchName		RO	RO	RO		
TreeEffDt		RO	RO	RO		
TreeName		RO	RO	RO		
TreeSetId		RO	RO	RO		
TreeUserKeyValue		RO	RO	RO		
TotalNodeCount					X	
Type				RO		X
UserKeyValue					X	

¹ These properties can be used with a closed tree (or tree structure.)

² These properties are used with the tree collection, the tree structure collection, the level collection and the branch collection objects, not the tree, tree structure or level objects.

CHAPTER 8

ActiveX Controls in PeopleTools

This chapter contains documentation relating to the ActiveX controls supported by PeopleSoft: the Microsoft Chart ActiveX control, the Microsoft Tree View ActiveX control, and the Microsoft Image control.

You can add these ActiveX controls to a page in Application Designer.



ActiveX controls are only available in Windows client. They do *not* work in the PeopleSoft Internet Architecture.

Throughout this section, we use typographical conventions to distinguish between different elements of the PeopleCode language, such as bold to indicate function names, italics for arguments, and so on.



For more information, see *Typographical Conventions and Visual Cues*.

Chart Control

The Chart control allows you to plot data in charts according to your specifications. You can create a chart by setting data in the control's properties page, or by retrieving data to be plotted from another source. Possible uses for the chart control include:

- charting dynamic data, such as current prices of selected commodities.
- plotting stored data, such as product prices, for graphic analysis of trends.

The **Chart** control supports the following features:

- True three-dimensional representation.
- Support for all major chart types.
- Data grid population via random data and data arrays.

The **Chart** control is associated with a data grid (DataGrid Object object). This data grid is a table that holds the data being charted. The data grid can also include labels used to identify

series and categories on the chart. The person who designs your chart application fills the data grid with information by inserting data or by importing data from a spreadsheet or array.

Chart Objects

A chart is actually composed of many other objects. Normally you can return a reference to an object by using a property (by that same name) on the parent object. The following flow charts show all the objects that are created from a chart object.

If an object is a "high-level" object, that is, created directly from the chart object, the object description is only listed once. For example, you can get the Backdrop object straight from the Chart control. From the Backdrop object, you can get a Fill, a Frame, and a Shadow object. You can also get the Backdrop object from the Footnote object. All the objects that you can get from a Backdrop object are *not* repeated under the Footnote object.

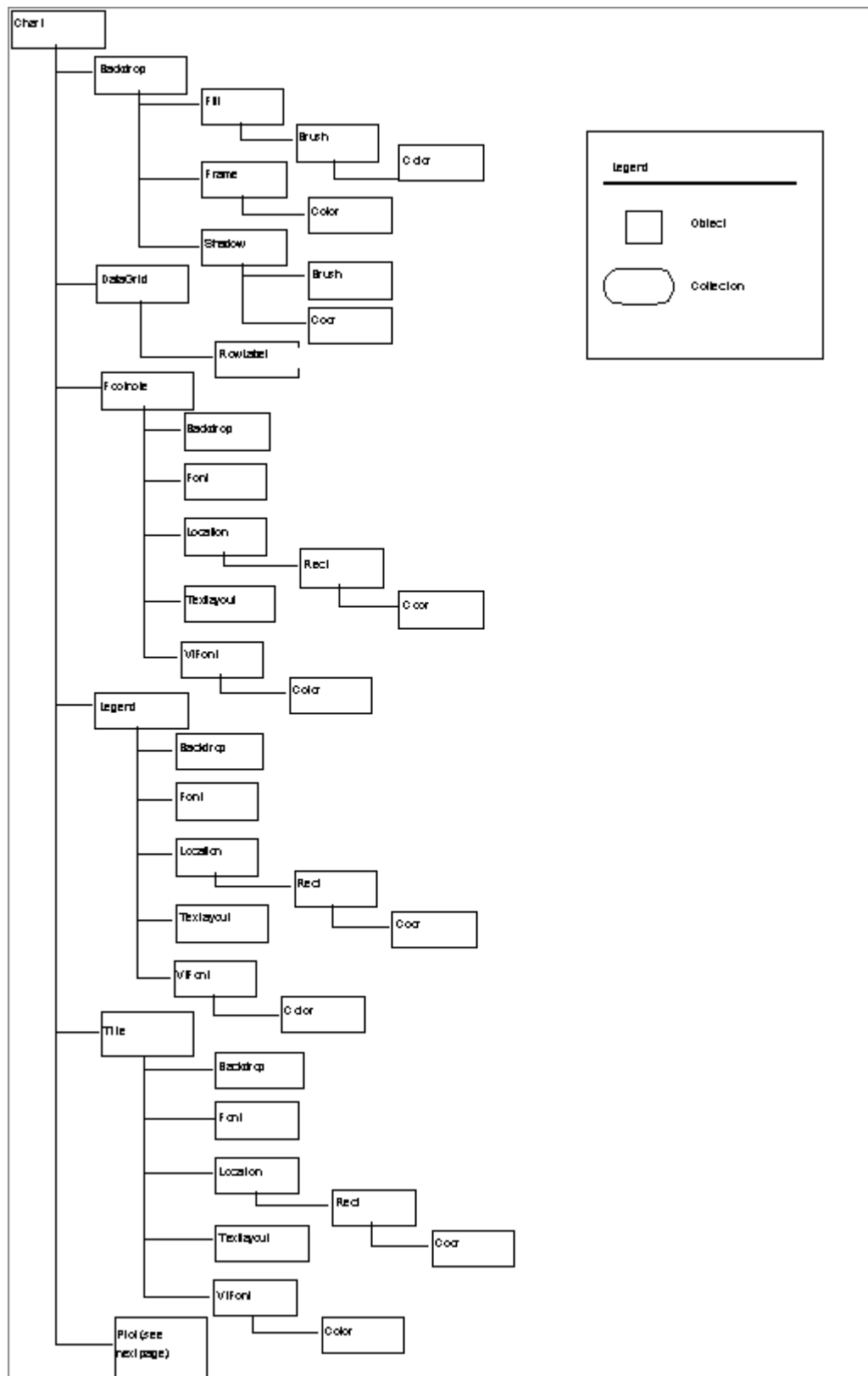


Chart Classes Hierarchy (Part 1)

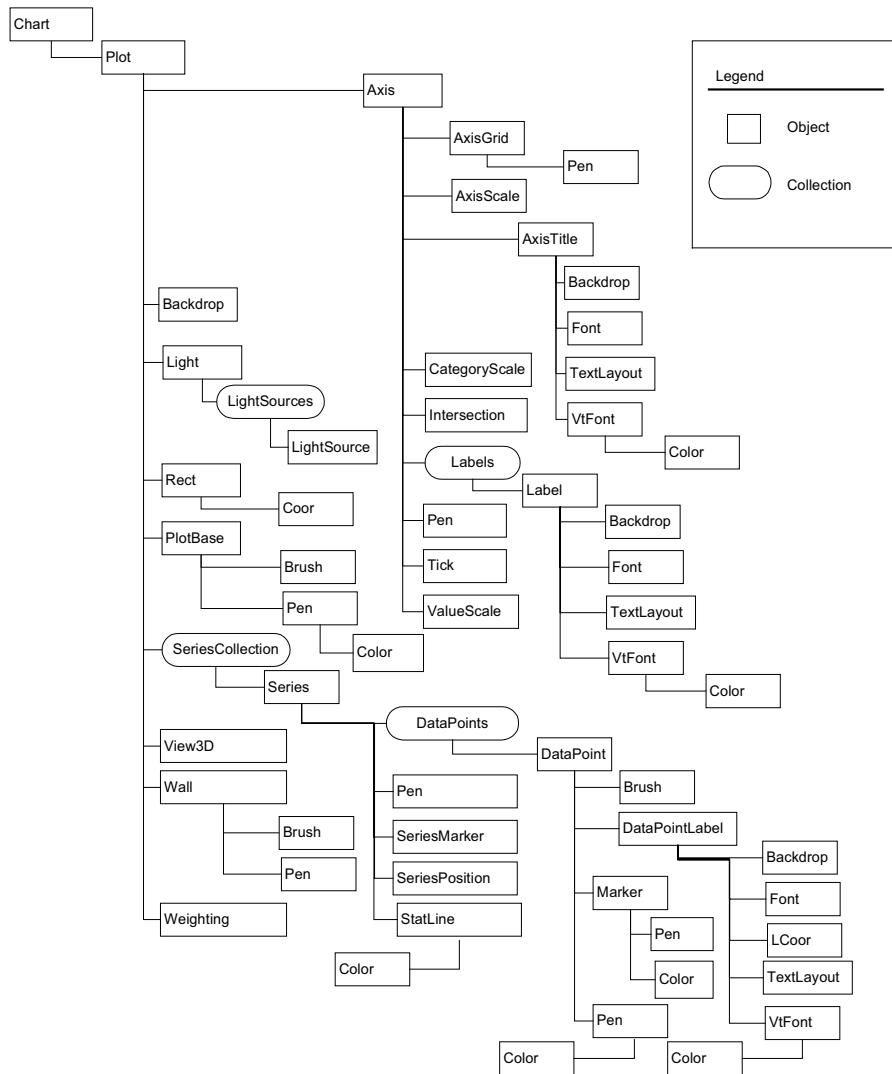


Chart Classes Hierarchy (Part 2)

Setting Indexed Properties

Some of the objects associated with the chart object have special properties (called indexed properties) that can't be set using the native property. These properties can only be set using the PeopleCode ObjectSetProperty built-in function. These exceptions are marked in the documentation.

For example, the ShowGuideLine property is considered a special property. You can use the native property to get a guideline:

```
&MYVALUE = &MYSERIES.ShowGuideLine(AxisID);
```

However, you can't use the ShowGuideLine property to set a value. Instead, use the following code:

```
ObjectSetProperty(&MYCHART, &MYSERIES.ShowGuideLine(AxisID), True);
```

Additional Properties

In addition to the regular properties, there are other properties that are marked as additional in the documentation. In general, the easiest way to tell the difference between a property and a method in PeopleCode is whether it must be followed by parenthesis: methods are, properties aren't. However, because we're implementing controls created in Visual Basic, there will be properties that look like methods, and documented as such.

Setting or Returning a Data Point

Once you have created a chart, you may also want to set or return the value of a particular data point. After you set the column and row, you can either set or get the data point. In the following example, the code will change the third data point for a simple (single-series) chart. This code would be associated with FieldChange PeopleCode.

```
&CHART = GetControl(PAGE.ACTIVEXTEST, "CHART1");

&CHART.column = 1;

&CHART.row = 1;

&CHART.data = RECORD.TEST.CHGFIELD;
```

If the chart has more than one series, you can loop through the series to set values. The following code sample sets the first data point for every series (column) in the chart.

```
&CHART = GetControl();

/* columns equals the number of series */

&VALUE = 0;

For &I = 1 To &CHART.ColumnCount

    &CHART.column = &I;

    &CHART.row = 1;

    &CHART.data = &VALUE;

    &VALUE = &VALUE + 20;

End-For;
```

Returning the data is just as easy. You have to specify the column and row, then you can use the data property. For example, suppose an end-user had adjusted data points on a simple (single-series) chart, and you wanted to store the values. The following code would have to be in a SavePreChange event (due to the **Insert**):

```
&CHART = GetControl(PAGE.ACTIVEXTEST, "chart1");

For &J = 1 To &CHART.rowcount
```

```

&CHART.column = 1;

&CHART.row = &J;

&REC = CreateRecord(RECORD.CHART_DATA);

&REC.getfield(1).value = &CHART.data;

&REC.insert();

End-For;

```

Using the DataGrid

You can either use the individual chart methods and properties for setting or retrieving data, or you can use the **DataGrid**. The **DataGrid** represents a virtual matrix containing labels and data points for the **Chart** control. The **DataGrid** object is configured as rows and columns. You can add and subtract rows, columns, and labels to this matrix to change the appearance of the chart.

A **DataGrid** object is returned by the **DataGrid** property on a **chart** object.

The following example initializes all the values in a DataGrid to null values:

```

&CHART = GetControl();

&ROWS = &CHART.datagrid.RowCount;

&COLUMNS = &CHART.DataGrid.columnCount;

For &I = 1 to &ROWS
    For &J = 1 to &COLUMNS
        &CHART.DataGrid.SetData(&I, &J, 0, 1);
    End-For;
End-For;

```

The following example fills the DataGrid with data passed in from a two dimensional array:

```

Function FillChart(&ARRAY as Array of array of number);

    &CHART = GetControl(PAGE.CHARTTEST, "CHART1");

    &DATAGRID = &CHART.datagrid;

    &ROWS = &DATAGRID.rowcount;

    &COLUMNS = &DATAGRID.columncount;

    For &I = 1 To &ROWS

```

```

For &J = 1 To &COLUMNS

    &CHART.DataGrid.SetData(&I, &J, &ARRAY[&J][&I], 0);

End-For;

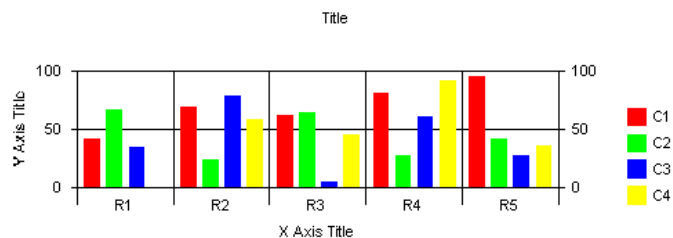
End-For;

End-Function;

```

Manipulating the Chart Appearance

The Chart control has many visible parts, all of which can be programmed. To understand how this is accomplished, it's useful to examine the following figure pointing out the parts of a chart.



Sample chart control

Each of these parts, (the title, the footnote, the X Axis Title, and so on) has a corresponding object in the Chart control which can be used to change the format of practically any element of the chart.

For example, to check, and then change (if necessary) the orientation of your chart legend, you could use the following:

```

&CONTROL = GetControl(PAGE.VOLUNTEER_ORG_TBL, "CHART1");

&ORIENT = &CONTROL.Legend.TextLayout.Orientation;

If &ORIENT <> 1 Then

    /* legend is not Vertical */

    &CONTROL.Legend.TextLayout.Orientation = 1;

End-If;

```



You can't set the **text** of the legend at runtime. You can only set it at design time.

Formatting Fonts

A very basic task with fonts might be to set the text of the chart's title. In order to do this, use the Title object's Text property:

```
&MYCHART = GetControl();

&MYCHART.Title.Text = "Year End Summary";
```

Assigning text to the title will make the title display, even if the chart didn't have a title before you assigned one.

The next question is how to change the font's attributes. In order to format any text attribute on the chart, you must use the VtFont object. For example, to format the title, the following code will work:

```
&MYCHART.Title.Text = "Year End Summary";

&MYFONT = &MYCHART.Title.VtFont;

&MYFONT.Name = "Algerian";

&MYFONT.Style = 2;

/* font style bold */

&MYFONT.Effect = 512;

/* font effect VtFontEffectUnderline */

&MYFONT.Size = 14;

&MYFONT.VtColor.Set(255, 0, 255);
```

You can use the same technique with other parts of chart. The only difference lies in the object model. For example, to format the text attributes of the Legend area use the following code:

```
&MYCHART.Legend.VtFont.Size = 18
```

Change the Scale Using the Type Property

To change the scale of the plot, you must specify that the y axis of the chart is going to be changed (changing the x axis has no visible effect).

In the following example, the AxisScale is set to linear and Tick style to center:

```
&CONTROL = GetControl();

&X_AXIS= &CONTROL.Plot.Axis(0, 1);

&Y_AXIS= &CONTROL.Plot.Axis(1, 1);

&Y_AXIS.AxisScale.Type = 0;

&X_AXIS.Tick.Style = 1;
```

Three-Dimensional Features of the Chart Control

The Chart control comes with many three-dimensional features.

Rotate the Chart

You can manually rotate a 3D chart by using the CTRL key and the mouse. To do this, hold down the CTRL key, click on the chart, then hold down the mouse as you drag across the chart image. The following figures show the same chart, before and after rotation.

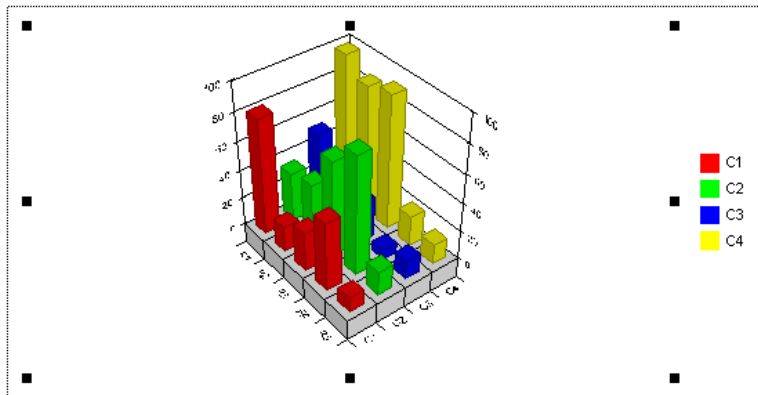


Chart before rotation

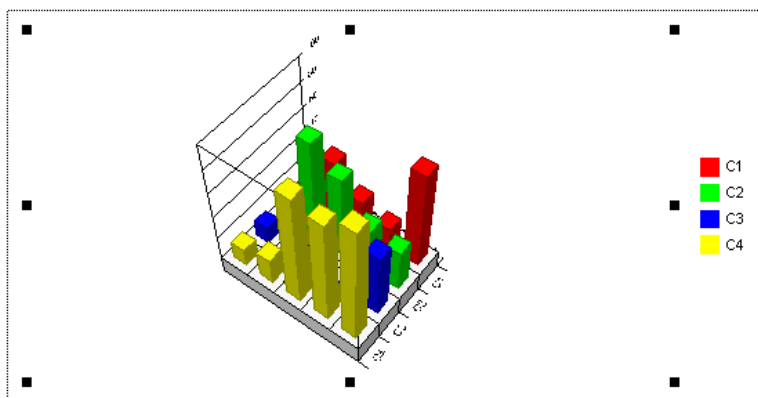


Chart after rotation

You can also rotate a 3D chart by using the Set method of the View3D object. The following will rotate the chart 15 degrees every time the user double-clicks on the chart.

```
Function OnDblClick()

    &MYCHART = GetControl();

    &ROTATE = &MYCHART.plot.view3d.rotation;

    &MYCHART.plot.view3d.rotation = &ROTATE + 15;
```

```
End-Function;
```

Seeing the Light

In addition to rotating a 3D chart, you can also use the Light feature to control how a virtual light shines on the chart. In practice, the appearance of a chart changes as it rotates because its surfaces reflect steady light in continually changing angles.

The following code results in a chart that appears to have a light source shining on it from the lower left of the screen. As the chart is rotated, surfaces which face the light directly get brighter.

```
&MYCHART = GetControl();  
  
&LIGHTCOLL = &MYCHART.plot.light.lightsources;  
  
&LIGHT = &LIGHTCOLL.item(1);  
  
&LIGHT.set(0, 0, 100, 1);
```

By default, a chart comes with one LightSource in the LightSources collection, which is why the **Item** method is passed a 1. The **Set** method takes four parameters: X, Y and Z coordinates, and the intensity of the light.

Light Basics

The LightSource feature is comprised of two parts:

- Ambient light: As its name suggests, this is light that has no specific source and thus no directionality.
- LightSources: a directional light that can be shone on the chart from any angle, with variable intensity. More than one light source can be created.

Ambient Light

Ambient light has a noticeable effect when you are using the LightSources. Ambient light simply bathes the chart evenly in a light that comes from no particular direction. You can set the AmbientIntensity property to any value between 0 and 1, with 0 turning off the light, and 1 increasing it to its maximum. For instance, to turn the ambient on to one quarter of its brightness, set the AmbientIntensity property to .25.



As you experiment with the LightSources, it is useful to set the property to 0. With no ambient light, the effect of the LightSource is accentuated.

The EdgeVisible and EdgeIntensity properties work together to highlight the edges of the chart. When the AmbientIntensity is set to a low value, you can set the EdgeIntensity property to a high value.

The result is that the edges seem to glow, as shown in the following figure:

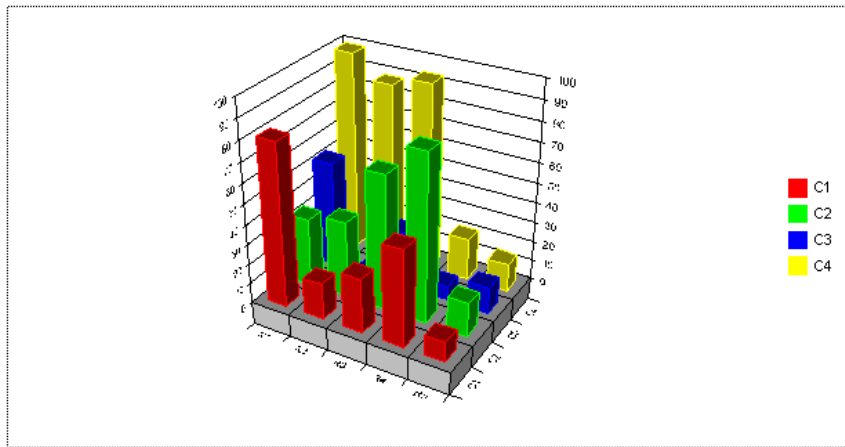


Chart with glowing edges

The code that produced the chart is as follows:

```
&MYCHART = GetControl();

&LIGHT = &MYCHART.plot.light;

&LIGHT.AmbientIntensity = 0;

&LIGHT.EdgeIntensity = 1;
```

Each **LightSource** in the **LightSources** collection has an **Intensity** property which — like the **AmbientIntensity** property — you can set to a value between 0 and 1. This property sets the brightness of the individual **LightSource**.

Each **LightSource** also can be positioned in the virtual space that surrounds the chart by setting the **X**, **Y**, and **Z** properties. By varying these properties, you can specify the direction from which the light will shine on the chart.

Using the Add Method to Add a LightSource

By default, the **Chart** contains one **LightSource** member in the **LightSources** collection. You can, however, add **LightSource** members using the **Add** method. The following sets the default item to shine from "behind", and when the user clicks the chart, the light source will change:

```
Function OnClick()

    &MYCHART = GetControl();

    &LIGHTCOLL = &MYCHART.plot.light.lightsources;

    &LIGHT = &LIGHTCOLL.item(1);

    &LIGHT.set(0, 0, 0, 1);
```

```
&LIGHTCOLL.add(1, 1, 1, 1);  
  
End-Function;
```

The parameters for the Add method include values for X, Y, and Z, which allow you to specify exactly the angle from which the light will shine, and a value for Intensity of the light.

Using the OnPointActivated Event to Change a Data Point

If you've started to explore the Chart control, you will notice that it has a large number of events. These events allow you to program the chart to respond to practically any action of the user. As an example of this programmability, the OnPointActivated event is used in the following example to show how a data point can be changed using the Series and DataPoint parameters. (The OnPointActivated event occurs whenever a data point is double-clicked.) The Series and DataPoint parameters correspond to the Column and Row properties respectively, and can thus be used to set the Data property. In the following example, 10 is added to the selected point.

```
Function OnPointActivated(&SERIES As number, &DATAPOINT As number, &MOUSEFLAGS  
As number, &CANCEL As number)  
  
    &CHART = GetControl();  
  
    &CHART.column = &SERIES;  
  
    &CHART.row = &DATAPOINT;  
  
    &VALUE = &CHART.data;  
  
    &CHART.data = &VALUE + 10;  
  
End-Function;
```

Definitions of Constants

The following constants are used in the Chart controls. The constants are used in the property sheet provided by PeopleTools. You can also use the numeric value in PeopleCode.



You must use the numeric value, not the constant, in your PeopleCode program.

VtBorderStyle

VtBorderStyle constants are defined as follows:

Numeric value	Constant	Description
0	VtBorderStyleNone	No border is placed around the chart control.
1	VtBorderStyleFixedSingle	A single border is placed around the chart control.

VtChAxisId

VtChAxisId constants are defined as follows:

Numeric value	Constant	Description
0	VtChAxisIdX	If the <i>x</i> axis is affected.
1	VtChAxisIdY	If the <i>y</i> axis is affected.
2	VtChAxisIdY2	If the secondary <i>y</i> axis is affected.
3	VtChAxisIdZ	If the <i>z</i> axis is affected.

VtChChartType

VtChChartType constants are defined as follows:

Numeric value	Constant	Description
0	VtChChartType3dBar	3D Bar
1	VtChChartType2dBar	2D Bar
2	VtChChartType3dLine	3D Line
3	VtChChartType2dLine	2D Line
4	VtChChartType3dArea	3D Area
5	VtChChartType2dArea	2D Area
6	VtChChartType3dStep	3D Step
7	VtChChartType2dStep	2D Step
8	VtChChartType3dCombination	3D Combination
9	VtChChartType2dCombination	2D Combination
10	VtChChartType3dHorizontalBar	3D Horizontal Bar
11	VtChChartType2dHorizontalBar	2D Horizontal Bar

	ntalBar	
12	VtChChartType3dClusteredBar	3D Clustered Bar
13	VtChChartType3dPie	3D Pie
14	VtChChartType2dPie	2D Pie
15	VtChChartType3dDoughnut	3D Doughnut
16	VtChChartType2dXY	2D XY
17	VtChChartType2dPolar	2D Polar
18	VtChChartType2dRadar	2D Radar
19	VtChChartType2dBubble	2D Bubble
20	VtChChartType2dHiLo	2D HiLo
21	VtChChartType2dGantt	2D Gantt
22	VtChChartType3dGantt	3D Gantt
23	VtChChartType3dSurface	3D Surface
24	VtChChartType2dContour	2D Contour
25	VtChChartType3dScatter	3D Scatter
26	VtChChartType3dXYZ	3D XYZ

VtChLocationType

VtChLocationType provides location options for chart elements. Valid values are:

Numeric value	Constant	Description
0	VtChLocationTypeTop	Top
1	VtChLocationTypeTopLeft	Top Left
2	VtChLocationTypeTopRight	Top Right
3	VtChLocationTypeLeft	Left
4	VtChLocationTypeRight	Right
5	VtChLocationTypeBottom	Bottom Left

	Left	
6	VtChLocationTypeBottom	Bottom
7	VtChLocationTypeBottom Right	Bottom Right
8	VtChLocationTypeCustom	Custom

VtChMouseFlag

VtChMouseFlag constants are defined as follows:

Numeric value	Constant	Description
4	VtChMouseFlagShiftKeyD own	If the SHIFT key is held down.
8	VtChMouseFlagControlKe yDown	If the CONTROL key is held down.

VtChPartType

VtChPartType constants are defined as follows:

Numeric value	Constant	Description
0	VtChPartTypeChart	Identifies the chart control.
1	VtChPartTypeTitle	Identifies the chart title.
2	VtChPartTypeFootnote	Identifies the chart footnote.
3	VtChPartTypeLegend	Identifies the chart legend.
4	VtChPartTypePlot	Identifies the chart plot.
5	VtChPartTypeSeries	Identifies a chart series.
6	VtChPartTypeSeriesLabel	Identifies a chart series label.
7	VtChPartTypePoint	Identifies an individual data point.
8	VtChPartTypePointLabel	Identifies a data point label.
9	VtChPartTypeAxis	Identifies an axis.
10	VtChPartTypeAxisLabel	Identifies an axis label.
11	VtChPartTypeAxisTitle	Identifies an axis title.
12	VtChPartTypeSeriesName	Identifies a chart series name.
13	VtChPartTypePointName	Identifies an individual data point name.

VtChSeriesType

VtChSeriesType constants are defined as follows:

<i>Numeric value</i>	<i>Constant</i>
-2	VtChSeriesTypeVaries
-1	VtChSeriesTypeDefault
0	VtChSeriesType3dBar
1	VtChSeriesType2dBar
2	VtChSeriesType3dHorizontalBar
3	VtChSeriesType2dHorizontalBar
4	VtChSeriesType3dClusteredBar
5	VtChSeriesType3dLine
6	VtChSeriesType2dLine
7	VtChSeriesType3dArea
8	VtChSeriesType2dArea
9	VtChSeriesType3dStep
10	VtChSeriesType2dStep
11	VtChSeriesType2dXY
12	VtChSeriesType2dPolar
13	VtChSeriesType2dRadarLine
14	VtChSeriesType2dRadarArea
15	VtChSeriesType2dBubble
16	VtChSeriesType2dHiLo
17	VtChSeriesType2dHLC
18	VtChSeriesType2dHLCRight
19	VtChSeriesType2dOHLC
20	VtChSeriesType2dOHLCBar
21	VtChSeriesType2dGantt
22	VtChSeriesType3dGantt
23	VtChSeriesType3dPie
24	VtChSeriesType2dPie
25	VtChSeriesType3dDoughnut

26	VtChSeriesType2dDates
27	VtChSeriesType3dBarHiLo
28	VtChSeriesType2dBarHiLo
29	VtChSeriesType3dHorizontalBarHiLo
30	VtChSeriesType2dHorizontalBarHiLo
31	VtChSeriesType3dClusteredBarHiLo
32	VtChSeriesType3dSurface
33	VtChSeriesType2dContour
34	VtChSeriesType3dXYZ

VtChStats

VtChStats constant used to describe the line type. Valid values are:

Numeric value	Constant	Description
1	VtChStatsMinimum	Shows the minimum value in the series.
2	VtChStatsMaximum	Shows the maximum value in the series.
4	VtChStatsMean	Shows the mathematical mean of the values in the series.
8	VtChStatsStddev	Shows the standard deviation of the values in the series.
16	VtChStatsRegression	Shows a trend line indicated by the values in a series.

VtChUpdateFlags

VtChUpdateFlags constants are defined as follows:

Numeric value	Constant	Description
0	VtChNoDisplay	Absence of update flags; the chart display is not affected. (Defined as 0.)
1	VtChDisplayPlot	Update will cause the plot to repaint.
2	VtChLayoutPlot	Update will cause the plot to lay out.
4	VtChDisplayLegend	Update will cause the legend

		to repaint.
8	VtChLayoutLegend	Update will cause the legend to lay out.
16	VtChLayoutSeries	Update will cause the series to lay out.
32	VtChPositionSection	A chart section has been moved or resized.

VtMousePointer

VtMousePointer constants are defined as follows:

<i>Numeric value</i>	<i>Constant</i>	<i>Description</i>
0	VtMousePointerDefault	Default chart pointer.
1	VtMousePointerArrow	Arrow pointer.
2	VtMousePointerCross	Arrow and hourglass.
3	VtMousePointerIbeam	Ibeam pointer.
4	VtMousePointerIcon	Small square within a square pointer.
5	VtMousePointerSize	Sizing arrows.
6	VtMousePointerSizeNESW	Double arrow pointing northeast and southwest.
7	VtMousePointerSizeNS	Double arrow pointing north and south.
8	VtMousePointerSizeNWSE	Double arrow pointing northwest and southeast.
9	VtMousePointerSizeWE	Double arrow pointing east and west.
10	VtMousePointerUpArrow	Arrow pointing up.
11	VtMousePointerHourGlass	Hourglass pointer.
12	VtMousePointerNoDrop	No drop pointer.
13	VtMousePointerArrowHourGlass	Arrow and hourglass.
14	VtMousePointerArrowQuestion	Arrow and question mark.

15	VtMousePointerSizeAll	Sizing in all directions arrows.
----	------------------------------	----------------------------------

VtPenStyle

The **VtPenStyle** constant that describes the style of pen. The valid values are:

Number value	Constant	Description
0	VtPenStyleNull	No pen is applied
1	VtPenStyleSolid	Solid line pen
2	VtPenStyleDashed	Dashed line pen
3	VtPenStyleDotted	Dotted line pen
4	VtPenStyleDashDot	Dash-dot line pen
5	VtPenStyleDashDotDot	Dash-dot-dot line pen
6	VtPenStyleDitted	Ditted line pen
7	VtPenStyleDashDit	Dash-ditted line pen
8	VtPenStyleDashDitDit	Dash-dit-dit line pen
9	VtPenStyleNative	

VtTextLengthType

VtTextLengthType constants are defined as follows:

Numeric value	Constant	Description
0	VtTextLengthTypeVirtual	Choose this constant to use TrueType virtual font metrics to optimize text layout for printing. TrueType virtual font metrics may not be very accurate for text displayed on the screen. Text displayed on the screen may be a larger or smaller than the virtual metrics requested. Larger text may not fit where it is supposed to and part of a character, a whole character, or even in some cases words may be clipped.
1	VtTextLengthTypeDevice	Choose this constant to optimize text layout for the screen. Text in charts laid out for screen display always

		fits correctly within its chart area. The printed text is generally a bit smaller and so the text may appear in slightly different places.
--	--	--

PeopleCode Events

The following is the list of events for the Chart control.

PeopleCode Event	Argument Description	General Description
OnMouseMove(&BUTTON As number, &SHIFT As number, &X As number, &Y As number)		Occurs when the user moves the mouse over the chart.
OnMouseDown(&BUTTON As number, &SHIFT As number, &X As number, &Y As number)		Occurs when the user presses a mouse button over the chart.
OnMouseUp(&BUTTON As number, &SHIFT As number, &X As number, &Y As number)		Occurs when the user releases a mouse button over the chart.
OnAxisTitleUpdated(&AXI SID As number, &AXISINDEX As number, &UPDATEFLAGS As number)	<i>axisId</i> Integer. An integer that identifies a specific axis. <i>axisIndex</i> Integer. An integer reserved for future use. For this version of Chart control, 1 is the only valid value for this argument. <i>updateFlags</i> Integer. An integer that provides information about the update of the title.	Occurs when an axis title has changed. The event handler determines which axis title is updated and sets <i>axisId</i> to <i>VtChAxisId</i> constant. The <i>updateFlags</i> is set to <i>VtChUpdateFlags</i> constant.
OnAxisLabelUpdated(&AXI SID As number, &AXISINDEX As number, &LABELSETINDEX As number, &LABELINDEX As number, &UPDATEFLAGS As number)	<i>axisId</i> Integer. An integer that identifies a specific axis, as described in Settings. <i>axisIndex</i> Integer. An integer reserved for future use. For this version of Chart control, 1 is the only valid value for this argument. <i>labelSetIndex</i> Integer. An integer that identifies the level of labels you are double clicking on.	Occurs when an axis label has changed. The event handler determines which axis label is updated and sets <i>axisId</i> to <i>VtChAxisId</i> constant. The event handler determines the affect of the update, and sets <i>updateFlags</i> to <i>VtChMouseFlag</i> constant.

	<p>Levels of labels are numbered from the axis out, beginning with 1.</p> <p><i>labelIndex</i> Integer. An integer that is currently unused.</p> <p><i>updateFlags</i> Integer. An integer provides information about the update of the label, as described in Settings.</p>	
OnChartActivated(&MOUSEFLAGS As number, &CANCEL As number)	<p><i>mouseFlags</i> Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings.</p> <p><i>cancel</i> Integer. This argument is not used at this time.</p>	<p>Occurs when the user double clicks the Microsoft Chart control, but not on a specific element in the chart.</p> <p>The event handler determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to <i>VtChMouseFlag</i> constant.</p>
OnAxisActivated(&AXISID As number, &AXISINDEX As number, &MOUSEFLAGS As number, &CANCEL As number)	<p><i>axisId</i> Integer. An integer that identifies a specific axis, as described in Settings.</p> <p><i>axisIndex</i> Integer. An integer reserved for future use. For this version of Chart control, 1 is the only valid value for this argument.</p> <p><i>mouseFlags</i> Integer. An integer that indicates whether a key is held down when the mouse button is clicked, as described in Settings.</p> <p><i>cancel</i> Integer. An integer that is not used at this time.</p>	<p>Occurs when the user double clicks on a chart axis.</p> <p>The event handler determines which axis is activated and sets <i>axisId</i> to <i>VtChAxisId</i> constant.</p> <p>The event handler determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to <i>VtChMouseFlag</i> constant.</p>
OnPlotActivated(&MOUSEFLAGS As number, &CANCEL As number)	<p><i>mouseFlags</i> Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings.</p> <p><i>cancel</i> Integer. This argument is not used at this time.</p>	<p>Occurs when the user double clicks the chart plot.</p> <p>The event handler determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to <i>VtChMouseFlag</i> constant.</p>
OnTitleActivated(&MOUSEFLAGS As number,	<p><i>mouseFlags</i> Integer. Indicates whether a key is</p>	Occurs when the user double clicks the chart title. You

&CANCEL As number)	held down when the mouse button is clicked, as described in Settings. <i>cancel</i> Integer. This argument is not used at this time.	can replace the standard user interface by canceling the event and displaying your own dialog box. The event handler determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to <i>VtChMouseFlag</i> constant.
OnPointSelected(&SERIES As number, &DATAPOINT As number, &MOUSEFLAGS As number, &CANCEL As number)	<i>series</i> Integer. Identifies the series containing the data point. Series are numbered in the order that their columns appear in the data grid, beginning with 1. <i>dataPoint</i> Integer. Identifies the data point's position in the series. Points are numbered in the order that their rows appear in the data grid, beginning with 1. <i>mouseFlags</i> Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings. <i>cancel</i> Integer. This argument is not used at this time.	Occurs when the user clicks a data point. The event handler determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to <i>VtChMouseFlag</i> constant.
OnAxisTitleSelected(&AXISID As number, &AXISINDEX As number, &MOUSEFLAGS As number, &CANCEL As number)	<i>axisId</i> Integer. An integer that identifies a specific axis. <i>AxisIndex</i> Integer. An integer reserved for future use. For this version of Chart control, 1 is the only valid value for this argument. <i>mouseFlags</i> Integer. An integer that indicates whether a key is held down when the mouse button is clicked. <i>cancel</i> Integer. This argument is not used at this time.	Occurs when the user double clicks on an axis title. The event handler determines which axis title is activated and sets <i>axisId</i> to <i>VtChAxisId</i> constant. The event handler determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to <i>VtChMouseFlag</i> constant.
OnKeyDown(&KEYCODE As number, &SHIFT As number)		Occurs when the user presses a key while the chart has the focus.
OnKeyPress(&KEYASCII		Occurs when the user

As number)		presses and releases a key.
OnKeyUp(&KEYCODE As number, &SHIFT As number)		Occurs when the user releases a key while the chart has the focus.
OnClick()		Occurs when the user clicks over the chart.
OnDbClick()		Occurs when the user double clicks over the chart.
OnAxisSelected(&AXISID As number, &AXISINDEX As number, &MOUSEFLAGS As number, &CANCEL As number)	<p><i>axisId</i> Integer. An integer that identifies a specific axis, as described in Settings.</p> <p><i>axisIndex</i> Integer. An integer reserved for future use. For this version of Chart control, 1 is the only valid value for this argument.</p> <p><i>mouseFlags</i> Integer. An integer that indicates whether key is held down when the mouse button is clicked, as described in Settings.</p> <p><i>cancel</i> Integer. An integer that is not used at this time.</p>	<p>Occurs when the user clicks on a chart axis.</p> <p>The event handler determines which axis is selected and sets <i>axisId</i> to <i>VtChAxisId</i> constant.</p> <p>The event handler determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to <i>VtChMouseFlag</i> constant.</p>
OnClick()		
OnPointLabelActivated(&SERIES As number, &DATAPOINT As number, &MOUSEFLAGS As number, &CANCEL As number)	<p><i>series</i> Integer. Identifies the series containing the data point. Series are numbered in the order that their columns appear in the data grid, beginning with 1.</p> <p><i>dataPoint</i> Integer. Identifies the data point's position in the series. Points are numbered in the order that their rows appear in the data grid, beginning with 1.</p> <p><i>mouseFlags</i> Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings.</p> <p><i>cancel</i> Integer. This argument is not used at this time.</p>	<p>Occurs when the user double clicks a data point label.</p> <p>The event handler determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to <i>VtChMouseFlag</i> constant.</p>
OnDataUpdated(&ROW As number, &COLUMN As number)	<p><i>row</i> Integer. Indicates the row in the data grid.</p> <p><i>column</i></p>	Occurs when the chart data grid has changed.

number, &LABELROW As number, &LABELCOLUMN As number, &LABELSETINDEX As number, &UPDATEFLAGS As number)	Integer. Indicates the column in the datagrid. <i>labelRow</i> Integer. Indicates the row label. <i>labelColumn</i> Integer. Indicates the column label. <i>labelSetIndex</i> Integer. Identifies the level of labels. Levels of labels are numbered from the axis out, beginning with 1. <i>updateFlags</i> Integer. Provides information about the update of the data, as described in Settings.	The <i>updateFlags</i> is set to VtChUpdateFlags constant. If row and column are nonzero, the change occurs to the indicated data cell. If <i>labelRow</i> or <i>labelColumn</i> , along with <i>labelSetIndex</i> , are nonzero, the indicated row or column label changes. If none of these are nonzero, no specific information about the change is available.
OnPointUpdated(&SERIES As number, &DATAPOINT As number, &UPDATEFLAGS As number)	<i>series</i> Integer. Identifies the series containing the data point. Series are numbered in the order that their columns appear in the data grid, beginning with 1. <i>dataPoint</i> Integer. Identifies the data point's position in the series. Points are numbered in the order that their rows appear in the data grid, beginning with 1. <i>updateFlags</i> Integer. Provides information about the update of the data point, as described in Settings.	Occurs when a data point has changed. The <i>updateFlags</i> is set to VtChUpdateFlags constant.
OnSeriesSelected(&SERIES As number, &MOUSEFLAGS As number, &CANCEL As number)	<i>series</i> Integer. Identifies the series containing the data point. Series are numbered in the order that their columns appear in the data grid, beginning with 1. <i>mouseFlags</i> Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings. <i>cancel</i> Integer. This argument is not used at this time.	Occurs when the user clicks a chart series. The event handler determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to VtChMouseFlag constant.
OnLegendUpdated(&UPDATEFLAGS As number)	<i>updateFlags</i> Integer. Provides information about the update of the legend, as described in Settings.	Occurs when the chart legend has changed. The <i>updateFlags</i> is set to VtChUpdateFlags constant.

OnLegendActivated(&MOUSEFLAGS As number, &CANCEL As number)	<i>mouseFlags</i> Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings. <i>cancel</i> Integer. This argument is not used at this time.	Occurs when the user double clicks on the chart legend. The event handler determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to <i>VtChMouseFlag</i> constant.
OnPlotUpdated(&UPDATEFLAGS As number)	<i>updateFlags</i> Integer. Provides information about the update of the plot, as described in Settings.	Occurs when the chart plot has changed. <i>The updateFlags is set to VtChUpdateFlags constant.</i>
OnAxisTitleActivated(&AXISID As number, &AXISINDEX As number, &MOUSEFLAGS As number, &CANCEL As number)	<i>axisId</i> Integer. An integer that identifies a specific axis. <i>AxisIndex</i> Integer. An integer reserved for future use. For this version of Chart control, 1 is the only valid value for this argument. <i>mouseFlags</i> Integer. An integer that indicates whether a key is held down when the mouse button is clicked. <i>cancel</i> Integer. This argument is not used at this time.	Occurs when the user double clicks on an axis title. The event handler determines which axis title is activated and sets <i>axisId</i> to <i>VtChAxisId</i> constant. The event handler determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to <i>VtChMouseFlag</i> constant.
OnAxisLabelSelected(&AXISID As number, &AXISINDEX As number, &LABELSETINDEX As number, &LABELINDEX As number, &MOUSEFLAGS As number, &CANCEL As number)	<i>axisId</i> Integer. An integer that identifies a specific axis, as described in Settings. <i>axisIndex</i> Integer. An integer reserved for future use. For this version of Chart control, 1 is the only valid value for this argument. <i>labelSetIndex</i> Integer. An integer that identifies the level of labels you are double clicking on. Levels of labels are numbered from the axis out, beginning with 1. <i>labelIndex</i> Integer. An integer that is currently unused. <i>mouseFlags</i> Integer. An integer that indicates if a key is held	Occurs when the user clicks an axis label. The event handler determines which axis label is selected and sets <i>axisId</i> to <i>VtChAxisId</i> constant. The event handler determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to <i>VtChMouseFlag</i> constant.

	down when the mouse button is clicked, as described in Settings. <i>cancel</i> Integer. An integer that is not used at this time.	
OnSeriesUpdated(&SERIES As number, &UPDATEFLAGS As number)	<i>series</i> Integer. Identifies the series containing the data point. Series are numbered in the order that their columns appear in the data grid, beginning with 1. <i>updateFlags</i> Integer. Provides information about the update of the series, as described in Settings.	Occurs when a chart series has changed. The <i>updateFlags</i> is set to <i>VtChUpdateFlags</i> constant.
OnPointActivated(&SERIES As number, &DATAPOINT As number, &MOUSEFLAGS As number, &CANCEL As number)	<i>series</i> Integer. Identifies the series containing the data point. Series are numbered in the order that their columns appear in the data grid, beginning with 1. <i>dataPoint</i> Integer. Identifies the data point's position in the series. Points are numbered in the order that their rows appear in the data grid, beginning with 1. <i>mouseFlags</i> Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings. <i>cancel</i> Integer. This argument is not used at this time.	Occurs when the user double clicks on a data point. The event handler determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to <i>VtChMouseFlag</i> constant.
OnChartSelected(&MOUSE FLAGS As number, &CANCEL As number)	<i>mouseFlags</i> Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings. <i>cancel</i> Integer. This argument is not used at this time.	Occurs when the user clicks the Microsoft Chart control, but not on a specific element in the chart. The event handler determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to <i>VtChMouseFlag</i> constant.
OnFootnoteUpdated(&UPD ATEFLAGS As number)	<i>updateFlags</i> Integer. Provides information about the update of the footnote, as	Occurs when the chart footnote changes.

	described in Settings.	The <i>updateFlags</i> is set to the <i>VtChUpdateFlags</i> constant.
OnAxisLabelActivated(&AXISID As number, &AXISINDEX As number, &LABELSETINDEX As number, &LABELINDEX As number, &MOUSEFLAGS As number, &CANCEL As number)	<p><i>axisId</i> Integer. An integer that identifies a specific axis, as described in Settings.</p> <p><i>axisIndex</i> Integer. An integer reserved for future use. For this version of Chart control, 1 is the only valid value for this argument.</p> <p><i>labelSetIndex</i> Integer. An integer that identifies the level of labels you are double clicking on. Levels of labels are numbered from the axis out, beginning with 1.</p> <p><i>labelIndex</i> Integer. An integer that is currently unused.</p> <p><i>mouseFlags</i> Integer. An integer that indicates if a key is held down when the mouse button is clicked.</p> <p><i>cancel</i> Integer. An integer that is not used at this time.</p>	<p>Occurs when the user double clicks on an axis label.</p> <p>The event handler determines which axis label is activated and sets <i>axisId</i> to <i>VtChAxisId</i> constant.</p> <p>The event handler determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to <i>VtChMouseFlag</i> constant.</p>
OnFootnoteActivated(&MOUSEFLAGS As number, &CANCEL As number)	<p><i>mouseFlags</i> Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings.</p> <p><i>cancel</i> This argument is not used at this time.</p>	<p>Occurs when the user double clicks the chart footnote.</p> <p>The event handler determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to <i>VtChMouseFlag</i> constant.</p>
OnFootnoteSelected(&MOUSEFLAGS As number, &CANCEL As number)	<p><i>mouseFlags</i> Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings.</p> <p><i>cancel</i> This argument is not used at this time.</p>	<p>Occurs when the user clicks the chart footnote.</p> <p>The event handler determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to <i>VtChMouseFlag</i> constant.</p>
OnTitleSelected(&MOUSEFLAGS As number, &CANCEL As number)	<p><i>mouseFlags</i> Integer. Indicates whether a key is held down when the mouse</p>	<p>Occurs when the user clicks the chart title.</p> <p>The event handler</p>

	button is clicked, as described in Settings. <i>cancel</i> Integer. This argument is not used at this time.	determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to VtChMouseFlag constant.
OnTitleUpdated(&UPDATE FLAGS As number)	<i>updateFlags</i> Provides information about the update of the title, as described in Settings.	Occurs when the chart title has changed. <i>The updateFlags is set to VtChUpdateFlags constant.</i>
OnPointLabelUpdated(&SERIES As number, &DATAPOINT As number, &UPDATEFLAGS As number)	<i>series</i> Integer. Identifies the series containing the data point. Series are numbered in the order that their columns appear in the data grid, beginning with 1. <i>dataPoint</i> Integer. Identifies the data point's position in the series. Points are numbered in the order that their rows appear in the data grid, beginning with 1. <i>updateFlags</i> Integer. Provides information about the update of the data point label, as described in Settings.	Occurs when a data point label has changed. <i>The updateFlags is set to VtChUpdateFlags constant.</i>
OnAxisUpdated(&AXISID As number, &AXISINDEX As number, &UPDATEFLAGS As number)	<i>axisId</i> Integer. An integer that identifies a specific axis. <i>axisIndex</i> Integer. An integer reserved for future use. For this version of Chart control, 1 is the only valid value for this argument. <i>updateFlags</i> Integer. An integer that provides information about the update of the axis.	Occurs when an axis has changed. The event handler determines which axis is updated and sets <i>axisId</i> to VtChAxisId constant. <i>The updateFlags is set to VtChUpdateFlags constant.</i>
OnDonePainting()		Occurs immediately after the chart repaints or redraws.
OnChartUpdated(&UPDATE FLAGS As number)	<i>updateFlags</i> Integer. Provides information about the update of the chart, as described in Settings.	Occurs when the chart has changed. <i>The updateFlags is set to VtChUpdateFlags constant.</i>
OnPlotSelected(&MOUSEF	<i>mouseFlags</i> Integer.	Occurs when the user clicks

LAGS As number, &CANCEL As number)	Indicates whether a key is held down when the mouse button is clicked, as described in Settings. <i>cancel</i> Integer. This argument is not used at this time.	the chart plot. The event handler determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to VtChMouseFlag constant.
OnLegendSelected(&MOUSE EFLAGS As number, &CANCEL As number)	<i>mouseFlags</i> Integer. Indicates whether a key is held down when the mouse button is clicked. <i>cancel</i> Integer. This argument is not used at this time.	Occurs when the user clicks on the chart legend. The event handler determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to VtChMouseFlag constant.
OnPointLabelSelected(&SERIES As number, &DATAPOINT As number, &MOUSEFLAGS As number, &CANCEL As number)	<i>series</i> Integer. Identifies the series containing the data point. Series are numbered in the order that their columns appear in the data grid, beginning with 1. <i>dataPoint</i> Integer. Identifies the data point's position in the series. Points are numbered in the order that their rows appear in the data grid, beginning with 1. <i>mouseFlags</i> Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings. <i>cancel</i> This argument is not used at this time.	Occurs when the user clicks a data point label. The event handler determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to VtChMouseFlag constant.
OnSeriesActivated(&SERIES As number, &MOUSEFLAGS As number, &CANCEL As number)	<i>series</i> Integer. Identifies the series containing the data point. Series are numbered in the order that their columns appear in the data grid, beginning with 1. <i>mouseFlags</i> Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings. <i>cancel</i> Integer. This argument is not used at this time.	Occurs when the user double clicks a chart series. You can replace the standard user interface by canceling the event and displaying your own dialog box. The event handler determines if a key is held down when the mouse button is clicked and sets <i>mouseFlags</i> to VtChMouseFlag constant.

OnOLEGiveFeedback(&EFFECT As number, &DEFAULTCURSORS As boolean)		OLEGiveFeedback event.
OnOLECompleteDrag(&EFFECT As number)		OLECompleteDrag event.
OnOLEStartDrag(&DATA As any, &ALLOWEDEFFECTS As number)		OLEStartDrag event
OnOLESetData(&DATA As any, &DATAFORMAT As number)		OLESetData event
OnOLEDragOver(&DATA As any, &EFFECT As number, &BUTTON As number, &SHIFT As number, &X As number, &Y As number, &STATE As number)		OLEDragOver event
OnOLEDragDrop(&DATA As any, &EFFECT As number, &BUTTON As number, &SHIFT As number, &X As number, &Y As number)		OLEDragDrop event

Declaring a Chart Object

Charts are declared using the Object data type. For example,

```
Local Object &CHART;
```

Scope of a Chart Object

All chart objects, that is, charts, axis, plots, etc., can be instantiated from PeopleCode or from a Visual Basic program.

ActiveX controls are not usable with the Web Client or PeopleSoft Internet Architecture.

This object can be used anywhere you have PeopleCode. However, as a chart control is a control on a page, you will generally only use this object in PeopleCode programs that are associated with an online process, and not in an Application Engine program, a message subscription, a Business Component, and so on.

You can't access a chart control until after the Component Processor has loaded the page. This means you can't use the **GetControl** function in an event prior to the Activate Event.



For more information, see PeopleCode and the Component Processor.

Chart Properties

ActiveSeriesCount

Returns the number of series that appear on a chart based on the number of columns in the DataGrid Object object and the type of chart being drawn.

This property is read-only.

AllowDithering

Returns or sets a value that determines whether to disable color dithering for charts on 8-bit color monitors in order to enable use of chart control's own color palette and enhance the chart display.

The valid values are:

True: Color dithering is allowed.

False: (Default) Chart control's color palette is used for enhanced color matching and display.

This property is read-write.

AllowDynamicRotation

Returns or sets a value that indicates whether users can interactively rotate three-dimensional charts by holding down the control key to display the rotation cursor.

The valid values are:

True: (Default) The user can interactively rotate the chart with the cursor.

False: The user cannot interactively rotate the chart with the cursor.

This property is read-write.

AllowSelections

Returns or sets a value that indicates whether users can select chart objects.

The valid values are:

True: (Default) The user can interactively select chart objects.

False: The user cannot select chart objects.

This property is read-write.

AllowSeriesSelection

Returns or sets a value that indicates whether a series is selected when a user clicks on an individual chart data point.

The valid values are:

True: (Default) Users can select a series by clicking a data point.

False: Clicking a data point selects only that data point, not the entire series.

This property is read-write.

AutoIncrement

Returns or sets a value that determines whether the current data point row and column is automatically incremented after the Data property is used to enter or update data in the data grid.

The valid values are:

True: When the Data property is changed, the Row property updates to the next row in the column. If you are at the end of a column, the Column property increments to the next column.

False: (Default) The current data point is not incremented.

This property is read-write.

Backdrop

Returns a reference to a BackDrop Object object that describes the shadow, pattern, or picture behind a chart or chart element.

This property is read-only.

BorderStyle

Returns or sets the border style to be placed around the chart.

Valid values are defined in VtBorderStyle.

Though the constants are also shown, this property takes and returns a number.

This property is read-write.

Chart3d

Returns a value that specifies whether a chart is three dimensional.

The valid values are:

True: The chart is a three-dimensional chart.

False: The chart is not a three-dimensional chart.

This property is read-only.

Example

In the following example, if the chart is 3D, every time the user double clicks, the chart rotates 15 degrees:

```
Function OnDbClick()  
  
    &MYCHART = GetControl();  
  
    If &MYCHART.chart3d Then  
  
        &ROTATE = &MYCHART.plot.view3d.rotation;  
  
        &MYCHART.plot.view3d.rotation = &ROTATE + 15;  
  
    End-If;  
  
End-Function;
```

ChartType

Returns or sets the type of chart used to plot the data in the data grid.

Valid values are defined in VtChChartType.

This property is read-write.

Column

Returns or sets the current data column (series) in the data grid.

You must select a column before you can use other properties to change the column's corresponding chart series or any data point within the series.

This property is read-write.

Example

The following loops through every column (series) and row of a chart.

```
&CHART = GetControl();
```

```

For &I = 1 To &CHART.ColumnCount

    &CHART.column = &I;

    For &J = 1 to &CHART.RowCount

        &CHART.row = &J;

        /* do processing */

    End-For;

End-For;

```

ColumnCount

Returns or sets the number of columns in the current data grid associated with a chart.

This property is read-write

Example

The following example sets the number of columns (series) for a chart equal to the number of fields in a record.

```

&REC = GetRecord(RECORD.CHART_DATA);

&CHART = GetControl();

&CHART.ColumnCount = &REC.FieldCount;

```

ColumnLabel

Returns or sets the label text associated with a column identified by the Column property. This property takes a string value.

This property is read-write.

ColumnLabelCount

Returns or sets the number of levels of labels on the columns in the data grid associated with a chart.

Column label levels are numbered from bottom to top, beginning at 1. Levels are added or subtracted from the top.

This property is read-write.

ColumnLabelIndex

Returns or sets a specific level of column labels associated with a chart.

To set a label on a column with more than one level of labels, or to return the current value for a label, you must first identify which level you want to affect. Column label levels are numbered from bottom to top, beginning at 1.

This property is read-write.

Data

Returns or sets a value of a specific data point in the data grid identified by Column and Row in the data grid of a chart. If the current data point already contains a value, it is replaced by the new value. The chart is redrawn to reflect the new value for the current data point.

This property takes a number value.

This property is read-write.

Example

The following example loads a simple chart (single series) with data:

```
For &I = &MAX To 1 Step - 1

    &VAL = FetchValue(SCROLL.JOB_VW, &I, JOB_VW.ANNUAL_RT);

    &DATE = FetchValue(SCROLL.JOB_VW, &I, JOB_VW.EFFDT);

    &CHART.Row = &MAX - &I + 1;

    &CHART.data = &VAL;

    &CHART.rowlabel = String(&DATE);

End-For;
```

DataGrid

Returns a reference to a DataGrid object that describes the data grid associated with a chart.

This property is read-only.

Example

```
&MYCHART = GetControl();

&DATA_GRID = &MYCHART.DataGrid;
```

DoSetCursor

Returns or sets a value that indicates whether the cursor can be set by a chart. The **DoSetCursor** property determines whether the application can control what the mouse pointer looks like. The valid values are:

True (Default): The application can control the mouse pointer appearance.

False: The application cannot control the mouse pointer appearance.

This property is read-write.

DrawMode

Returns or sets a value that determines when and how a chart is repainted.

The valid values are:

<i>Numeric value</i>	<i>Constant</i>	<i>Description</i>
0	VtChDrawModeDraw	Draws directly to the display device.
1	VtChDrawModeBlit	Blits an offscreen drawing to the display device.



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

Enabled

Returns or sets a value that determines whether the chart control can respond to user-generated events.

The **Enabled** property allows the controls to be enabled or disabled at runtime. For example, you can disable objects that don't apply to the current state of the application. You can also disable a control used purely for display purposes, such as a text box that provides read-only information.

The valid values are:

True (default) Allow chart to respond to events

False: Prevent chart from responding to events.

This property is read-write.

Footnote

Returns a reference to a Footnote Object object that provides information about the descriptive text used to annotate a chart.

This property is read-only.

FootnoteText

Returns or sets the text used as the chart footnote. This property takes a string value. The same results can be achieved by using the Text property of the Footnote Object object.

This property is read-write.

Legend

Returns a reference to a Legend Object object that contains information about the appearance and behavior of the graphical key and accompanying text that describes the chart series.

This property is read-only.

MousePointer

Returns or sets a value indicating the type of mouse pointer, defined by VtMousePointer, displayed when the mouse is over a particular part of an object at runtime.

This property is read-write.



You must use the numeric value, not the constant, in your PeopleCode program.

OLEDragMode

Returns or sets whether this control can act as an OLE drag/drop source, and whether this process is started automatically or under programmatic control.

Valid values are:

Numeric value	Constant
0	ccOLEDragManual
1	ccOLEDragAutomatic



You must use the numeric value, not the constant, in your PeopleCode program.

When **OLEDragMode** is set to **Manual**, you must call the OLEDrag method to start dragging, which then triggers the **OLEStartDrag** event.

When **OLEDragMode** is set to **Automatic**, the source component fills the DataGrid Object object with the data it contains and sets the *effects* parameter before initiating the OLEStartDrag event (as well as the OLESetData and other source-level OLE drag/drop events) when the user attempts to drag out of the control. This gives you control over the drag/drop operation and

allows you to intercede by adding other formats, or by overriding or disabling the automatic data and formats.

If the source's **OLEDragMode** property is set to **Automatic**, and no data is loaded in the **OLEStartDrag** event, or *aftereffects* is set to **0**, then the OLE drag/drop operation does not occur.



If the **DragMode** property of a control is set to **Automatic**, the setting of **OLEDragMode** is ignored, because regular Visual Basic drag and drop events take precedence.

This property is read-write.

OLEDropMode

Returns or sets whether this control can act as an OLE drop target. Valid values are:

Numeric value	Constant
0	ccOLEDropNone
1	ccOLEDropManual
2	ccOLEDropAutomatic



You must use the numeric value, not the constant, in your PeopleCode program. Also, the target component inspects what is being dragged over it in order to determine which events to trigger. There is no collision of components or confusion about which events are fired, because only one type of object can be dragged at a time.

This property is read-write.

Plot

Returns a reference to a Plot Object object that describes the area upon which a chart is displayed.

This property is read-only.

RandomFill

Indicates whether the data for a chart data grid was randomly generated.

The valid values are:

True: (default) Random data is used to draw the chart.

False: No random data is generated. The user provides the data for the chart.

This property is read-write.

Repaint

Returns or sets a value that determines if the Chart control is repainted after a change is made to the chart.

The valid values are:

True: (default) Refreshes the control.

False: Does not allow the control to repaint when a change is made to the chart. This is useful when several operations are performed on the chart and you do not want the chart to continually repaint during the process.

This property is read-write.

Row

Returns or sets the current data row in the data grid. This property takes a number value.

This property is read-write.

Example

The following function takes the values passed in from an array to set the datapoints for a series.

```
Function AssignData (&ARRAY as array, &PAGENAME as string, &CHARTNAME as
string);

    &CHART = GetControl(PAGE.&PAGENAME, &CHARTNAME);

    &CHART.RowCount = &ARRAY.Len;

    For &I = 1 to &CHART.RowCount;

        &CHART.Row = &I;

        &CHART.Data = &ARRAY.Shift();

    End-For;

    /* do other processing */

End-Function;
```

RowCount

Returns or sets the number of rows in the current data grid associated with a chart. This property takes a number value.

This property is read-write.

RowLabel

Returns or sets a data label associated with the row identified by the Row property. This property takes a string value.

This property is read-write.

RowLabelCount

Returns or sets the number of levels of labels on the rows in a data grid associated with a chart. This property takes a number value.

This property is read-write.

RowLabelIndex

Returns or sets a value that specifies a level of row labels. This property takes a number value.

To set a label on a row with more than one level of labels, or to return the current value for a label, you must first identify which level you want to affect. Row label levels are numbered from right to left, beginning at 1.

This property is read-write.

SeriesColumn

Returns or sets the column position in which the current series is displayed. This property takes a number value.

You can use this property to reorder series. If two series are assigned the same position, they are stacked.

This property is read-write.

SeriesType

Returns or sets the series type, as defined by `VtChSeriesType`, used to display the current series identified in the Column property. This property takes a number value.

You must select the series to change using the Column property before using the **SeriesType** property.

This property is read-write.



You must use the numeric value, not the constant, in your PeopleCode program.

ShowLegend

Returns or sets a value that indicates whether a legend is visible for a chart. The default legend location is to the right side of the chart.

The valid values are:

True: The legend appears on the chart in the position indicated by the Location Object object.

False: (default) The legend is not displayed on the chart.

This property is read-write.

Stacking

Returns or sets a value that determines whether all the series in the chart are stacked.

The valid values are:

True: All chart series are stacked.

False: (default) Chart series are not stacked.

This property is read-write.

TextLengthType

Returns or sets how text is drawn, defined by VtTextLengthType, to optimize the appearance either on the screen or on the printed page. This property takes a number value.

This property is read-write.



You must use the numeric value, not the constant, in your PeopleCode program.

Title

Returns a reference to a Title Object object that describes the text used to title a chart.

This property is read-write.

TitleText

Returns or sets the text displayed as the chart title. This property takes a string value.

This property provides a simple means to set or return the chart title. This property is functionally identical to using **Chart.Title.Text**.

This property is read-write.

Chart Methods

EditCopy

Syntax

```
EditCopy()
```

Description

Copies a picture of the current chart to the clipboard in Windows metafile format. It also copies the data being used to create the chart to the clipboard.

This method allows you to paste the chart's data or a picture of the chart itself into another application. Since both the data and the picture of the chart are stored on the clipboard, what gets pasted into the new application varies depending on the type of application. For example, if you execute the chart's **EditCopy** method in your code and then go to an Excel spreadsheet and select **Edit Paste**, the chart data set is placed in the spreadsheet. To insert the picture of the chart into the spreadsheet, select **Edit Paste Special** and select the **Picture** type.

EditPaste

Syntax

```
EditPaste()
```

Description

Pastes a Windows metafile graphic or tab-delimited text from the clipboard into the current selection on a chart.

The chart can accept several types of information from the clipboard, depending on the currently selected chart element when **EditPaste** is called. If the entire chart is selected, the chart looks for data on the clipboard and attempts to use this new data to redraw the chart. If an item that can accept a picture, such as a bar or chart backdrop is selected, the chart looks for a metafile on the clipboard. If it finds a metafile, it uses that metafile to fill the selected object.

Layout

Syntax

```
Layout()
```

Description

Lays out a chart, forcing recalculation of automatic values.

A chart is laid out the first time it is drawn. When any chart settings change, the chart is again laid out at the next draw. There are a number of settings the chart calculates, such as the axis minimum and maximum values, based on the chart type or some other setting. These values are not determined until the chart is laid out. If you attempt to "get" these automatic values before the chart is properly laid out, they will not reflect the new values.

OLEDrag

Syntax

```
OLEDrag()
```

Description

Causes a component to initiate an OLE drag/drop operation.

When the **OLEDrag** method is called, the component's OLEStartDrag event occurs, allowing it to supply data to a target component.

Refresh

Syntax

```
Refresh()
```

Description

Forces a complete repaint of the chart control.

Use the **Refresh** method when you want to update the data structures of a data control.

Generally, painting the control is handled automatically while no events are occurring. However, there may be situations where you want the control updated immediately. For example, if you use the control to show the current status of the data structure, you can use **Refresh** to update the control whenever a change is made to the data structure.

SelectPart

Syntax

```
SelectPart(part , index1, index2, index3, index4)
```

Description

Selects the specified chart part.

Parameters

<i>part</i>	Specifies the chart element. This parameter is required. Valid constants are <code>VtChPartType</code> .
<i>index1</i>	If <i>part</i> refers to a series or a data point, this argument specifies which series. Series are numbered in the order their corresponding columns appear in the data grid from left to right, beginning with 1. If <i>part</i> refers to an axis or axis label, this argument identifies the axis type with a <code>VtChAxisId</code> constant.
<i>Index2</i>	If <i>part</i> refers to a data point, this argument specifies which data point in the series identified by <i>index1</i> . This parameter takes a number value.
<i>index3</i>	If <i>part</i> refers to an axis label, this argument refers to the level of the label. Axis label levels are numbered from the axis out, beginning with 1. This parameter takes a number value. If <i>part</i> is not an axis label, the argument is unused.
<i>index4</i>	This argument is unused at this time.

Returns

None.

Example

The following example selected the third series in a chart. Note that there are no optional parameters: even though the type of *part* being selected, the other parameters must also be specified, even if unused.

```
&CHART = GetControl();  
  
&CHART.selectpart(5, 3, 0, 0, 0);
```

ToDefaults

Syntax

```
ToDefaults()
```

Description

Returns the chart to its initial settings.



This method will not only reset the values, but reset the chart type as well to a 2D bar chart.

Axis Object

The **Axis** object is an axis on a chart.

An axis object is returned from the Axis method of a Plot Object object

Three axes are available: *x*, *y*, and *z*. The *z* axis is visible only when the chart is a 3D chart.

Axis Properties

AxisGrid

Returns a reference to an AxisGrid Object object that describes the planar area surrounding a chart axis.

This property is read-only.

AxisScale

Returns a reference to an AxisScale Object object that describes how chart values are plotted on an axis.

This property is read-only.

AxisTitle

Returns a reference to an AxisTitle Object object associated with the axis of a chart.

This property is read-only.

CategoryScale

Returns a reference to a CategoryScale Object object that describes the scale information for a category axis.

This property is read-only.

Intersection

Returns a reference to an Intersection Object object that describes the point at which an axis intersects another axis on a chart.

This property is read-only.

LabelLevelCount

Returns the number of levels of labels for a given axis. This property take a number value.

This property is read-only.

Labels

Returns a reference to a Labels Collection collection.

This property is read-only.

Pen

Returns or sets a reference to a Pen Object object that describes the color and pattern of lines or edges on chart elements.

This property is read-only.

Tick

Returns a reference to a Tick Object object that describes a marker indicating a division along a chart axis.

This property is read-only.

ValueScale

Returns a reference to a ValueScale Object object that describes the scale used to display a value axis.

This property is read-only.

AxisGrid Object

The **AxisGrid** object describes the planar area surrounding a chart axis.

An AxisGrid object is returned from the AxisGrid property of an Axis Object object.

AxisGrid Properties

MinorPen

Returns a reference to a Pen Object object that describes the appearance of the minor axis grid lines.

This property is read-only.

MajorPen

Returns a reference to a Pen Object object that describes the appearance of the major axis grid lines.

This property is read-only.

AxisScale Object

The **AxisScale** object controls how chart values are plotted on an axis.

An **AxisScale** object is returned from the AxisScale property of an Axis Object object.

AxisScale Properties

Hide

Returns or sets a value that determines whether the axis on a chart is hidden.

The valid values are:

True: The axis scale, line, ticks and title are hidden.

False: (default) The axis appears on the chart.

This property is read-write.

LogBase

Returns or sets the logarithm base used to plot chart values on a logarithmic axis. This property takes a number value.

The default base is 10. The valid range is 2 to 100.

The axis type is controlled by the **Type** property.

This property is read-write.

PercentBasis

Returns or sets the type, defined by **VtChPercentAxisBasis**, of percentage used to plot chart values on a percent axis. This property takes a number value.

VtChPercentAxisBasis provides methods of displaying percentage axes. The valid values are:

<i>Numeric value</i>	<i>Constant</i>	<i>Description</i>
0	VtChPercentAxisBasisMa	The largest value in the chart is considered 100

	xChart	percent and all other values on the chart are displayed as percentages of that value.
1	VtChPercentAxisBasisMaximumRow	The largest value in each row is considered 100 percent and all other values in that row are displayed as percentages of that value.
2	VtChPercentAxisBasisMaximumColumn	The largest value in each series is considered 100 percent and all other values in that series are displayed as percentages of that value.
3	VtChPercentAxisBasisSumChart	All values in the chart are added together, and that value is considered 100 percent. All other values are displayed as percentages of that value.
4	VtChPercentAxisBasisSumRow	All values in each row are added together and the total value for each row is considered 100 percent. All other values in that same row are displayed as percentages of that value. This is the basis for 100 percent stacked charts.
5	VtChPercentAxisBasisSumColumn	All values in each series are added together to give a total value for each series. All values are displayed as a percentage of their series total value.

This property is read-write.

Type

Returns or sets the scale type, defined by **VtChScaleType**, of an axis. This property takes a number value.

VtChScaleType provides methods for plotting chart values and displaying the chart scale.

Number value	Constant	Description
0	VtChScaleTypeLinear	Chart values are plotted in a

		linear scale with values ranging from the minimum to the maximum chart range value.
1	VtChScaleTypeLogarithmic	Chart values are plotted in a logarithmic scale with values based on a specific log scale set with the logBase argument of this function.
2	VtChScaleTypePercent	Chart values are plotted in a linear scale with values based on the percentages of the chart range values.

This property is read-write.

AxisTitle Object

The **AxisTitle** object describes an axis title on a chart.

An **AxisTitle** object is returned from the AxisTitle property on an Axis Object object.

AxisTitle Properties

Backdrop

Returns a reference to a BackDrop Object object that describes the shadow, pattern, or picture behind a chart or chart element.

This property is read-only.

Font

Returns or sets a reference to a standard Font Object object that describes the font used to display text on the chart.

This property is read-write.

Text

Returns or sets the text used to display a chart element such as an axis title, data point label, footnote, or chart title. This property takes a string value.

The **Text** property is the default property for each of the objects to which it applies.

This property is read-write.

TextLayout

Returns a reference to a TextLayout Object object that describes text positioning and orientation.

This property is read-only.

TextLength

Returns the length of the axis title. This property takes a number value.

This property is read-only.

Visible

Returns or sets a value that determines whether a chart element is displayed.

The valid values are:

True (default): The chart, axis title, label, or marker are displayed.

False: The elements are hidden.

This property is read-write.

VtFont

Returns a reference to a VtFont Object object that describes the font used to display chart text.

This property is read-only.

Backdrop Object

The **Backdrop** object describes a shadow or pattern behind a chart element.

A **Backdrop** object is returned from the **Backdrop** property of a Chart Properties object or an AxisTitle Object object.

Backdrop Properties

Fill

Returns a reference to a Fill Object object that describes the type and appearance of a chart object's backdrop.

This property is read-only.

Frame

Returns a reference to a Frame Object object that describes the appearance of the frame around a chart element.

This property is read-only.

Shadow

Returns a reference to a Shadow Object object that describes the appearance of a shadow on chart elements.

This property is read-only.

Brush Object

The **Brush** object describes the fill type used to display a chart element.

A **Brush** object is returned from the **Brush** property of the following objects:

- DataPoint Object
- Fill Object
- Plotbase Object
- Shadow Object
- Wall Object

Brush Properties

FillColor

Returns a reference to a VtColor Object object that describes the color used to fill a chart element.

This property is read-only.

Index

Returns or sets the pattern or hatch used in the brush if its Style property is set to **VtBrushStylePattern** or **VtBrushStyleHatch**. This property takes a number value.

This property is read-write.

PatternColor

Returns a reference to a VtColor Object object that describes the pattern color used to fill a chart element.

This property is read-only.

Style

Returns or sets the style used to draw certain chart elements.

For the **Brush** object, a **VtBrushStyle** constant describing the brush pattern. The valid values are:

<i>Number value</i>	<i>Constant</i>	<i>Description</i>
0	VtBrushStyleNull	No brush (background shows through)
1	VtBrushStyleSolid	Solid color brush
2	VtBrushStylePattern	Bitmap patterned brush
4	VtBrushStyleHatched	Hatched brush

This property is read-write.

CategoryScale Object

The **CategoryScale** object describes the scale for a category axis.

A **CategoryScale** object is returned from the CategoryScale property of the Axis Object object.

CategoryScale Properties

Auto

Returns or sets a value that indicates whether the axis is automatically scaled.

The valid values are:

True: The axis is automatically scaled based on the data being charted.

False: The axis is not automatically scaled. Values in DivisionsPerLabel and DivisionsPerTick are used to determine the scale.

This property is read-write.

DivisionsPerLabel

Returns or sets the number of divisions to skip between labels. This property takes a number value.

If this property is set, the object's Auto property is automatically set to False.

This property is read-write.

DivisionsPerTick

Returns or sets the number of divisions to skip between tick marks. This property takes a number value.

If this property is set, the object's Auto property is automatically set to False.

This property is read-write.

LabelTick

Returns or sets a value that indicates whether category axis labels are centered on an axis tick mark.

The valid values are:

True: The labels are centered on a tick mark.

False: The labels are centered between two tick marks.

If this property is set, the object's Auto property is automatically set to False.

This property is read-write.

Coor Object

The **Coor** object describes a floating x and y coordinate pair for a chart.

A **Coor** object is returned from the following:

- The Offset property of a Shadow Object object
- The Min and Max properties of a Rect Object object.

Coor Properties

X

Returns or sets the x value in a floating coordinate pair for a chart.

This property is read-write.

Y

Returns or sets the y value in a floating coordinate pair for a chart.

This property is read-write.

Coor Method

Set

Syntax

Set (*X*, *Y*)

Description

Sets the x and y coordinate values for a chart.

Parameters

<i>X</i>	Identifies the x value of the coordinate.
<i>Y</i>	Identifies the y value of the coordinate.

DataGrid Object

The **DataGrid** object represents a virtual matrix containing labels and data points for the **Chart** control. The **DataGrid** object is configured as rows and columns. You can add and subtract rows, columns, and labels to this matrix to change the appearance of the chart.

A **DataGrid** object is returned by the DataGrid property on a Chart Properties object.

DataGrid Properties

ColumnCount

Returns or sets the number of columns in the current data grid associated with a chart. This property takes a number value.

This property is read-write.

ColumnLabelCount

Returns or sets the number of levels of labels on the columns in the data grid associated with a chart. This property takes a number value.

Column label levels are numbered from bottom to top, beginning at 1. Levels are added or subtracted from the top.

This property is read-write.

RowCount

Returns or sets how many rows there are in each column of a data grid associated with a chart. This property takes a number value.

This property is read-write.

RowLabelCount

Returns or sets the number of levels of labels on the rows in a data grid associated with a chart. This property takes a number value.

Set this property to add or delete levels of labels from data grid rows. Row label levels are numbered from right to left, beginning at 1. Levels are added or subtracted from the left.

This property is read-write.

DataGrid Methods

ColumnLabel

Syntax

```
ColumnLabel (Column, LabelIndex)
```

Description

Returns the label on a data column in the grid associated with a chart.



This method will only return the specified label. If you want to set the label, use the ColumnLabel property on the chart object instead.

Parameters

<i>column</i>	Identifies a specific data column. Columns are numbered from left to right beginning with 1. Any columns containing labels are not counted as data columns. This property takes a number value.
<i>labelIndex</i>	Identifies a specific label. If more than one level of column labels exist for the column, you must identify one of them. Column labels are numbered from bottom to top beginning at 1. If you don't have more than one level of column labels, specify a 1 for this parameter.

Returns

The column label text as a string.

Example

The following code is in the OnSeriesSelected event, which passes in the number of the series (&SERIES) as a number. It returns the column (series) label as a string.

```
&CHART = GetControl();  
  
&DGRID = &CHART.datagrid;  
  
&STRING = &DGRID.columnlabel(&SERIES, 1);
```

Related Topics

DeleteColumnLabels, ColumnLabel property

CompositeColumnLabel

Syntax

```
CompositeColumnLabel (Column)
```

Description

Returns the multilevel label string that identifies a column in the data grid associated with a chart.

Parameter

<i>column</i>	Identifies a specific data column. Columns are numbered from left to right beginning with 1. Any columns containing labels are not counted as data columns. This parameter takes a number value.
---------------	--

Returns

A string value.

Related Topics

CompositeRowLabel, DeleteColumnLabels

CompositeRowLabel

Syntax

```
CompositeRowLabel (Row)
```

Description

Returns the multilevel label string that identifies a row in the data grid associated with a chart.

Parameter

<i>row</i>	Identifies a specific data row. Rows are numbered from top to bottom beginning with 1. Any rows containing labels are not counted as data rows. This parameter takes a number value.
------------	--

Returns

A string value.

Related Topics

CompositeColumnLabel, DeleteRowLabels

DeleteColumnLabels

Syntax

```
DeleteColumnLabels (LabelIndex, Count)
```

Description

Deletes levels of labels from the data columns in a data grid associated with a chart.

Parameters

<i>labelIndex</i>	Identifies the number of the first level of labels you want to delete. Column label levels are numbered bottom to top, beginning with 1. This parameter takes a number value.
-------------------	---

<i>count</i>	Specifies the number of label levels you want to delete. The number of columns being deleted is calculated from the column identified in <i>labelIndex</i> up. This parameter takes a number value.
--------------	---

Returns

None.

RelatedTopics

ColumnLabel property

DeleteColumns

Syntax

```
DeleteColumns(Column, Count)
```

Description

Deletes columns of data and their associated labels from the data grid associated with a chart. This will delete not only the column, but any labels associated with the column.

Parameters

<i>Column</i>	Identifies a specific data column. Columns are numbered from left to right beginning with 1. This parameter takes a number value.
<i>Count</i>	Specifies the number of columns you want to delete. This parameter takes a number value.

Returns

None.

Example

The following example will delete the series the user has selected.

```
Function OnSeriesSelected(&SERIES As number, &MOUSEFLAGS As number, &CANCEL As number)
```

```
    &CHART = GetControl();  
    &DGRID = &CHART.datagrid;  
    &DGRID.deletecolumns(&SERIES, 1);
```

```
End-Function;
```

Related Topics

DeleteRows

DeleteRowLabels

Syntax

```
DeleteRowLabels(LabelIndex, Count)
```

Description

Deletes levels of labels from the data rows in a data grid associated with a chart.

Parameters

<i>labelIndex</i>	Identifies the number of the first level of labels you want to delete. Row labels are numbered right to left, beginning with 1. This parameter takes a number value.
<i>count</i>	Specifies the number of label levels you want to delete. Row labels are deleted from the row identified by <i>labelIndex</i> to the left. This parameter takes a number value.

Returns

None.

Related Topics

CompositeRowLabel

DeleteRows

Syntax

```
DeleteRows(Row, Count)
```

Description

Deletes rows of data and their associated labels from the data grid associated with a chart.

Parameters

<i>row</i>	Identifies a specific data row. Rows are numbered from top to bottom beginning with 1. This parameter takes a number value.
<i>count</i>	Specifies the number of rows you want to delete. This parameter takes a number value.

Returns

None.

Example

```
&DATAGRID.DeleteRows(&ROW, 1);
```

Related Topics

RowCount

GetData

Syntax

```
GetData(Row, Column, &DataPoint, nullFlag)
```

Description

Gets the value for a specific data point in the data grid associated with a chart. This value is returned in the *&DataPoint* variable. You must declare this variable as type Number, otherwise you'll receive a runtime error.

Parameters

<i>row</i>	Identifies the row containing the data point value. This parameter takes a number value.
<i>column</i>	Identifies the column containing the data point value. This parameter takes a number value.
<i>&DataPoint</i>	The data point value. This parameter takes a number variable. The value of the data point is returned in this parameter, so you must use a variable for the parameter. You must declare this variable as type Number, otherwise you'll receive a runtime error.
<i>nullFlag</i>	Indicates whether or not the data point value is a null. You must specify a number for this parameter.

Returns

The value for a specific data point in the data grid associated with a chart. This value is returned in the *&DataPoint* variable.

Example

The following code returns the value of all the data points for a series:

```
Local number &NEWVALUE;
```



```

Function OnSeriesSelected(&SERIES As number, &MOUSEFLAGS As number, &CANCEL As
number)

    &CHART = GetControl();

    &DATAGRID = &CHART.datagrid;

    /* rows equals the number of datapoints in a series */

    &ROWS = &DATAGRID.RowCount;

    &VALUE = 0;

    For &I = 1 To &ROWS

        &DATAGRID.getdata(&SERIES, &I, &NEWVALUE, 0);

        /* do processing with &NEWVALUE */

    End-For;

End-Function;

```

Related Topics

SetData, Data

InitializeLabels

Syntax

```
InitializeLabels()
```

Description

Assigns each label in the first level of data grid labels a unique identifier.

MoveData

Syntax

```
MoveData(Top, Left, Bottom, Right, OverOffset, DownOffset)
```

Description

Moves a range of data within a data grid associated with a chart.

Parameters

<i>top</i>	Identifies the first row in the range to move.
<i>left</i>	Identifies the first column in the range to move.
<i>bottom</i>	Identifies the last row in the range to move.
<i>right</i>	Identifies the last column in the range to move.
<i>overOffset</i>	Identifies the horizontal direction data should be moved. A positive value moves data to the right; a negative value moves data to the left.
<i>downOffset</i>	Identifies the vertical direction data should be moved. A positive value moves data down, a negative value moves data up.

Returns

None.

RandomDataFill

Syntax

```
RandomDataFill()
```

Description

Fills the data grid associated with a specific chart with randomly generated data.

Example

```
&CHART.RandomDataFill();
```

Related Topics

RandomFillColumns, RandomFillRows

RandomFillColumns

Syntax

```
RandomFillColumns(Column, Count)
```

Description

Fills a number of data grid columns associated with a chart with random values.

Parameters

<i>column</i>	Identifies the first column you wish to fill. Columns are numbered from left to right beginning with 1. This parameter takes a number value.
<i>count</i>	Specifies the number of columns you want to fill with random data. This parameter takes a number value.

Returns

None.

Example

The following example fills all but the first column with random data.

```
Function OnClick()  
  
    &CHART = GetControl();  
  
    &DATAGRID = &CHART.DataGrid;  
  
    &COLS = &DATAGRID.columncount;  
  
    &DATAGRID.RandomFillColumns(2, &COLS);  
  
End-Function;
```

Related Topics

RandomDataFill, RandomFillRows

RandomFillRows

Syntax

```
RandomFillRows (Row, Count)
```

Description

Fills a number of data grid rows associated with a chart with random values.

Parameters

<i>row</i>	Identifies the first row you wish to fill. Rows are numbered from top to bottom beginning with 1. This parameter takes a number value.
<i>count</i>	Specifies the number of rows you want to fill with random data. This parameter takes a number value.

Returns

None.

Example

The following example fills all but the first row with random data.

```
Function OnClick()

    &CHART = GetControl();

    &DATAGRID = &CHART.DataGrid;

    &ROWS = &DATAGRID.rowcount;

    &DATAGRID.RandomFillRows(2, &ROWS);

End-Function;
```

Related Topics

RandomFillColumns, RandomDataFill

SetData

Syntax

```
SetData(Row, Column, DataPoint, nullFlag)
```

Description

Sets the value for a specific data point in the data grid associated with a chart.

Parameters

<i>row</i>	Identifies the row containing the data point value. This parameter takes a number value.
<i>column</i>	Identifies the column containing the data point value. This parameter takes a number value.
<i>DataPoint</i>	The data point value.
<i>nullFlag</i>	Indicates whether or not the data point value is a null. You must specify a number for this parameter.

Returns

None.

Example

The following initializes all data in the datagrid to null values:

```

&CHART = GetControl();

&ROWS = &CHART.datagrid.RowCount;

&COLUMNS = &CHART.DataGrid.columnCount;

For &I = 1 to &ROWS
    For &J = 1 to &COLUMNS
        &CHART.DataGrid.SetData(&I, &J, 0, 1);
    End-For;
End-For;

```

SetSize

Syntax

```
SetSize(RowLabelCount, ColumnLabelCount, DataRowCount, DataColumnCount)
```

Description

Resizes the number of data columns and rows, as well as the number of levels of column labels and row labels of a data grid associated with a chart at one time.

This method can be used in place of RowCount, ColumnCount, RowLabelCount and ColumnLabelCount.



If you reduce the size of the data grid, data in deleted rows or columns is destroyed.

Parameters

<i>rowLabelCount</i>	Returns or sets the number of levels of row labels you want on the data grid. This parameter takes a number value.
<i>columnLabelCount</i>	Returns or sets the number of levels of column labels you want on the data grid. This parameter takes a number value.
<i>dataRowCount</i>	Returns or sets the number of data rows you want on the data grid. This parameter takes a number value.
<i>dataColumnCount</i>	Returns or sets the number of data columns you want on the data grid. This parameter takes a number value.

Returns

None.

Example

The following code sets the size of the data grid based on the size of an associated grid:

```

&CHART = GetControl();

&DGRID = &CHART.datagrid;

&DGRID.SetSize(1, 1, Int(&RANGE), Int(&DATA_GRID.ColumnCount);

```

Related Topics

GetData, SetData

DataGrid Special Property

The following property can be used to get a value. However, it *cannot* be used to set a value. To set a value, you will have to use the ObjectSetProperty built-in PeopleCode function, and the following syntax:

```

ObjectSetProperty(&Object, "PropertyName", &NewValue, &IndexValue1 [,
&IndexValue2. . .]);

```

The following would set the RowLabel with indexes &Data_Grid_Label and 2 to the string (&FY):

```

ObjectSetProperty(&Data_Grid, "RowLabel", String(&FY), &Data_Grid_Label, 2);

```

RowLabel

Syntax

```

RowLabel (Row, LabelIndex)

```

Description

Returns or sets a specific row label in the current data grid associated with a chart.

Parameters

<i>row</i>	Specifies a row. Rows are numbered from top to bottom beginning at 1. This parameter takes a number value.
------------	--

labelIndex

Specifies a specific level of row labels. Row labels are numbered from left to right beginning at 1. This parameter takes a number value.

Returns

The row label as a string.

Example

```
&STRING = &DGRID.RowLabel (&SERIES, &ROW) ;
```

Related Topics

RowLabel

DataPoint Object

The **DataPoint** object describes the attributes of an individual data point on a chart.

A **DataPoint** object is returned from the Item method of a DataPoints Object collection.

DataPoint Properties

Brush

Returns a reference to a Brush Object object that describes the fill type used to display a chart element.

This property is read-only.

DataPointLabel

Returns a reference to a DataPointLabel Object object that describes a label on an individual chart data point.

This property is read-only.

EdgePen

Returns the Pen Object object used to draw the edge of the data point on a chart.

This property is read-only.

Marker

Returns a reference to a Marker Object object that describes the icon used to identify a data point on a chart.

This property is read-only.

Offset

Returns or sets the distance that a chart element is offset or pulled away from its default location. This property takes a number value.

For the **DataPoint** object, this is an integer describing the offset distance. Offset is measured in inches or centimeters depending upon your default Windows settings.

This property is read-write.

DataPoint Methods

ResetCustom

Syntax

```
ResetCustom()
```

Description

Resets any custom attributes placed on a data point to the series default.

Related Topics

ToDefaults

Select

Syntax

```
Select()
```

Description

Selects an individual data point.

Related Topics

SetData, GetData, Data

DataPointLabel Object

The **DataPointLabel** object describes the label for a data point on a chart.

A **DataPointLabel** object is returned from the DataPointLabel property on a DataPoint Object object.

DataPointLabel Properties

Backdrop

Returns a reference to a BackDrop Object object that describes the shadow, pattern, or picture behind a chart or chart element.

This property is read-write.

Component

Returns or sets the type of label to be used to identify the data point.

The **VtChLabelComponent** constants provide options for displaying chart labels. The valid values are:

<i>Number value</i>	<i>Constant</i>	<i>Description</i>
1	VtChLabelComponentValue	The value of the data point appears in the label. Data points in XY, Polar, and Bubble charts actually have two or three values. The default label for these chart types display all values in a standard format. You can customize this format to highlight an individual data value.
2	VtChLabelComponentPercent	The value of the data point is displayed in the label as a percentage of the total value of the series.
4	VtChLabelComponentSeriesName	The series name is used to label the data point. This name is taken from the label associated with the column in the data grid.
8	VtChLabelComponentPointName	The data point name is used

<i>Number value</i>	<i>Constant</i>	<i>Description</i>
	me	to label the data point.



You must check for the numeric value, not the constant, in your PeopleCode program.

This property is read-only.

Custom

Returns or sets a value that determines if custom text is used to label a data point on a chart.

The valid values are:

True: The label contains custom text.

False: (default) Information specified by the **DataPointLabel** object's Component property is used to label the data point.

This property is read-write.

Font

Returns or sets a reference to a standard Font Object object that describes the font used to display text on the chart.

This property is read-write.

LineStyle

Returns or sets the type of line used to connect a data point to a label on a chart.

The **VtChLabelLineStyle** constants provide options for displaying lines connecting a label and series. The valid values are:

<i>Number value</i>	<i>Constant</i>	<i>Description</i>
0	VtChLabelLineStyleNone	No line connects the label and series.
1	VtChLabelLineStyleStraight	A straight line connects the label and series.
2	VtChLabelLineStyleBent	A bent line connects the label and series.



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

LocationType

Returns or sets the standard position used to display a chart element. This property takes a number value.

VtChLocationType provides location options for chart elements.

This property is read-write.

Offset

Returns the distance that a chart element is offset or pulled away from its default location. This property takes a number value.

For the DataPointLabel and Shadow objects, this is a reference to an LCoor Object object that describe the x and y values of the offset.

For the DataPointLabel object, this property indicates the distance that a data point label is offset or pulled away from one of the predefined (standard) label positions. The offset is added to the position calculated for the point based on the DataPointLabel object's LocationType setting.

This property is read-only.

PercentFormat

Returns or sets a string that describes the format used to display the label as a percent. This property takes a string value.

Use the DataPointLabel object's Component property to change the label type.

This property is read-write.

Text

Returns or sets the text used to display a chart element such as an axis title, data point label, footnote, or chart title. This property takes a string value.

This property is read-write.

TextLayout

Returns a reference to a TextLayout Object object that describes text positioning and orientation.

This property is read-only.

TextLength

Returns or sets the number of characters in the text of a chart axis title, data point label, footnote, or chart title. This property takes a number value.

This property is read-write.

ValueFormat

Returns or sets the format used to display the label as a value. This property takes a number value.

Use the DataPointLabel object's Component property to change the label type.

This property is read-write.

VtFont

Returns a reference to a VtFont Object object that describes the font used to display chart text.

This property is read-only.

DataPointLabel Methods

ResetCustomLabel

Syntax

```
ResetCustomLabel ()
```

Description

Resets any custom attributes placed on a data point label in a chart to the series default.

Related Topics

ToDefaults

Select

Syntax

```
Select ()
```

Description

Selects a data point label.

Related Topics

Label Object object

DataPoints Object

The **DataPoints** collection contains a group of chart data points.

A **DataPoints** collection is returned from the DataPoints property on a Series Object object.

DataPoints Properties

Item

Syntax

`Item (Index)`

Description

Returns a reference to a **Datapoint** object pointed by Index in the collection object.

This property is read-only.

Count

Syntax

`Count ()`

Description

Returns the number of items in the collection.

This property is read-only.

Fill Object

The **Fill** object describes the type and appearance of an object's backdrop in a chart.

A **Fill** object is returned by the Fill property on a BackDrop Object object.

Fill Properties

Brush

Returns a reference to a Brush Object Object that describes the fill type used to display a chart element.

This property is read-only.

Style

Returns or sets the style used to draw certain chart elements.

For the **Fill** object, a **VtFillStyle** constant that describes the style of fill. A fill can have a brush, which is a solid color or patterned fill. The valid values are:

<i>Number value</i>	<i>Constant</i>	<i>Description</i>
0	VtFillStyleNull	No fill (background shows through)
1	VtFillStyleBrush	A solid color or pattern fill
2	VtFillStyleGradient	



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

Font Object

The **Font** object contains font attributes (font name, font size, color, and so on) for a control.

A font object is returned from the **Font** property of the following objects:

- AxisTitle Object
- DataPointLabel Object
- Footnote Object
- Label Object
- Legend Object
- Title Object

In addition to the Font object, many objects return a VtFont object.



For more information, see VtFont Object.

Font Object Properties

Bold

Indicates whether the font is boldfaced. False is the default.

This property is read-write.

Charset

The character set used in the font. Valid values are:

<i>Number value</i>	<i>Description</i>
0	ANSI_CHARSET
1	DEFAULT_CHARSET
2	SYMBOL_CHARSET



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

Italic

Indicates whether the font is italicized. False is the default.

This property is read-write.

Name

The name of the font, for example, Arial. This property takes a string value.

This property is read-write.

Size

The point size of the font. This property takes a number value.

This property is read-write.

Strikethrough

Indicates whether the font is strikethrough. False is the default.

This property is read-write.

Underline

Indicates whether the font is underlined. False is the default.

This property is read-write.

Weight

The boldness of the font. This property takes a number value.

This property is read-write.

Footnote Object

The **Footnote** object describes the descriptive text that appears beneath a chart.

A **Footnote** object is returned from the Footnote property on a chart object.

Footnote Properties**Backdrop**

Returns a reference to a BackDrop Object object that describes the shadow, pattern, or picture behind a chart or chart element.

This property is read-only.

Font

Returns or sets a reference to a standard Font Object object that describes the font used to display text on the chart.

This property is read-write.

Location

Returns a reference to a Location Object object that describes the position of textual chart elements.

This property is read-only.

Text

Returns or sets the text used to display a chart element such as an axis title, data point label, footnote, or chart title. This property takes a string value.

The **Text** property is the default property for each of the objects to which it applies.

This property is read-write.

TextLayout

Returns a reference to a TextLayout Object object that describes text positioning and orientation.

This property is read-only.

TextLength

Returns or sets the number of characters in the text of a chart axis title, data point label, footnote, or chart title. This property takes a number value.

This property is read-write.

VtFont

Returns a reference to a VtFont Object object that describes the font used to display chart text.

This property is read-only.

Footnote Method

Select

Syntax

```
Select ( )
```

Description

Selects the specified chart element.

Frame Object

The **Frame** object holds information about the appearance of the frame around a chart element.

A **Frame** object is returned from the Frame property on a BackDrop Object object.

Frame Properties

FrameColor

Returns a reference to a VtColor Object object that specifies the color used to frame a chart element.

This property is read-only.

SpaceColor

Returns a reference to a VtColor Object object that specifies the color used fill the space between double frames around a chart element.

This property is read-only.

Style

Returns or sets the style used to draw certain chart elements.

A **VtFrameStyle** constant that describes the type of frame. The valid values are:

<i>Number value</i>	<i>Constant</i>	<i>Description</i>
0	VtFrameStyleNull	No frame.
1	VtFrameStyleSingleLine	A single line encloses the backdrop.
2	VtFrameStyleDoubleLine	Two equal width lines enclose the backdrop.
3	VtFrameStyleThickInner	A thick inner line and a thin outer line enclose the backdrop.
4	VtFrameStyleThickOuter	A thin inner line and a thick outer line enclose the backdrop.



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

Width

Returns or sets the width of a chart element, in points.

This property is read-write.

Intersection Object

The **Intersection** object describes the point at which an axis intersects an intersecting axis on a chart.

An **Intersection** object is returned from the Intersection property on an Axis Object object.

Intersection Properties

Auto

Returns or sets a value that determines whether or not the Intersection object uses the value of the Point property to position the axis.

The valid values are:

True: (default) The axis is positioned in its standard location.

False: The intersecting axis is positioned at the value indicated by Point.

This property is read-write.

AxisID

Returns a specific axis that intersects with the current axis. This property takes a number value.

The returning value is equal to one of constants in VtChAxisId.

This property is read-only.

Index

Returns which axis intersects another axis when there is more than one axis with the same index. The return value is a number that specifies the index of the intersecting axis. Currently, 1 is the only valid value for this argument.

This property is read-only.

LabelsInsidePlot

Returns or sets a value that determines whether to leave the axis labels at the normal location or move them with the axis to the new intersection point.

If this property is set, then the Intersection object's Auto property is automatically set to False.

The valid values are:

True: (default) The axis labels remain at the normal location.

False: The labels move inside the plot to the new intersection point.

This property is read-write.

Point

Returns or sets the point where the current axis intersects with another axis. This property takes a number value.

If this property is set, then the Intersection object's Auto property is automatically set to False.

This property is read-write.

Label Object

The **Label** object describes a specific chart axis label.

A **Label** object is returned from the Item method on a Labels Collection collection.

Label Properties

Auto

Returns or sets a value that determines whether axis labels are automatically rotated to improve the chart layout.

The valid values are:

True: (Default) The labels may be rotated.

False: The labels are not rotated. Long labels may not display properly.

This property is read-write.

Backdrop

Returns a reference to a BackDrop Object object that describes the shadow, pattern, or picture behind a chart or chart element.

This property is read-only.

Font

Returns or sets a reference to a standard Font Object object that describes the font used to display text on the chart.

This property is read-write.

Format

Returns or sets the characters that define the format used to display the axis label. This property takes a number value.

This property is read-write.

FormatLength

Returns the length of the format string. This property takes a number value.

This property is read-only.

Standing

Returns or sets a value that specifies whether axis labels are displayed horizontally in the x or z plane or vertically on the text baseline in the y plane.

The valid values are:

True: The axis labels are displayed vertically on the text baseline in the y plane.

False: (Default) The axis labels are displayed horizontally in the x or z plane.

This property is read-write.

TextLayout

Returns a reference to a TextLayout Object object that describes text positioning and orientation.

This property is read-only.

VtFont

Returns a reference to a VtFont Object object that describes the font used to display chart text.

This property is read-only.

Labels Collection

The **Labels** collection contains a group of chart axis labels, that is, label objects.

A **labels** collection is returned by the Labels property on an Axis Object object.

Labels Collection Properties

Count

Returns the number of objects in a collection.

This property is read-only.

Item

Syntax

`Item(Item)`

Description

Returns a reference to an object within a collection that describes a chart element.

This property is read-only.

LCoor Object

The **LCoor** object describes a long integer x and y coordinate pair.

A **LCoor** object is returned by the Offset property of a DataPointLabel Object object.

LCoor Properties

X

Returns or sets the x value in a floating coordinate pair for a chart.

This property is read-write.

Y

Returns or sets the y value in a floating coordinate pair for a chart.

This property is read-write.

LCoor Method

Set

Syntax

`Set (X, Y)`

Description

Sets the x and y coordinate values for a chart.

Parameters

<i>X</i>	Identifies the x value of the coordinate.
<i>Y</i>	Identifies the y value of the coordinate.

Legend Object

The **Legend** object represents the graphical key and accompanying text that describes a chart series.

A **Legend** object is returned from the Legend property on a chart object.

Legend Properties

Backdrop

Returns a reference to a BackDrop Object object that describes the shadow, pattern, or picture behind a chart or chart element.

This property is read-only.

Font

Returns or sets a reference to a standard Font Object object that describes the font used to display text on the chart.

This property is read-write.

Location

Returns a reference to a Location Object object that describes the position of textual chart elements.

This property is read-only.

TextLayout

Returns a reference to a TextLayout Object object that describes text positioning and orientation.

This property is read-only.

VtFont

Returns a reference to a VtFont Object object that describes the font used to display chart text.

This property is read-only.

Legend Method

Select

Syntax

```
Select ()
```

Description

Selects the specified chart element.

Light Object

The **Light** object represents the light source illuminating a three-dimensional chart.

A **Light** object is returned by the Light property on a Plot Object object.

Light Properties

AmbientIntensity

Returns or sets the percentage of ambient light illuminating a three-dimensional chart. Valid values are 0 to 1. If set to 1, all sides of the chart elements are fully illuminated no matter what light sources are turned on.

If set at 0, there is no contribution from ambient light; only the sides of the chart elements facing active light sources are illuminated.

This property is read-write.

EdgeIntensity

Returns or sets the intensity of light used to draw the edges of objects in a three-dimensional chart. Valid values are 0 to 1.0. An intensity of 0 turns edges off, drawing the edges as black lines; and an intensity of 1 fully illuminates the edges using the element's pen color.

If this property is set, then the Light object's EdgeVisible property is automatically set to True.

This property is read-write.

EdgeVisible

Returns or sets a value that determines whether edges are displayed on the elements in a three-dimensional chart.

If the EdgeIntensity property is set, then this property is automatically set to True.

The valid values are:

True Edges are visible.

False Edges are not displayed on elements in the three-dimensional chart.

This property is read-write.

LightSources

Returns a reference to a Lightsources Collection collection that describe all light sources used to illuminate a three-dimensional chart.

This property is read-only.

Lightsource Object

The **Lightsource** object represents the light source used to illuminate elements in a three-dimensional chart.

A **Lightsource** object is returned from the Item method on a Lightsources Collection collection.

Lightsource Properties

Intensity

Returns or sets the strength of the light coming from the light source.

If the intensity is set to 100 percent (1), chart surfaces facing the light source are fully illuminated. If the light is set at 50 percent (.5), these surfaces receive 50 percent illumination from this light. Valid range is 0 to 1.

Intensity is the default property of the **LightSource** object.

This property is read-write.

X

Returns or sets the x value in a floating coordinate pair for a chart. This property takes a number value.

This property is read-write.

Y

Returns or sets the y value in a floating coordinate pair for a chart. This property takes a number value.

This property is read-write.

Z

Returns or sets the z value in a coordinate location. This property takes a number value.

This property is read-write.

Lightsource Method

Set

Syntax

```
Set (X, Y, Z, Intensity)
```

Description

Sets the x, y, and z coordinates and the intensity for the LightSource object location.

Parameters

<i>X</i>	Identifies the x value of the light source location. This parameter takes a number value.
<i>Y</i>	Identifies the y value of the light source location. This parameter takes a number value.
<i>Z</i>	Identifies the z value of the light source location. This parameter takes a number value.

Intensity

Indicates the light source intensity. This parameter takes a number value.

Lightsources Collection

The **Lightsources** collection describes a group of **LightSource** objects in a chart.

A **Lightsources** collection is returned from the `LightSources` property on a `Light Object` object.

Lightsources Properties

Count

Returns the number of objects in a collection.

This property is read-only.

Item

Syntax

`Item(Index)`

Description

Returns a reference to an object within a collection that describes a chart element.

This property is read-only.

Lightsources Methods

Add

Syntax

`Add(X, Y, Z, Intensity)`

Description

Adds a **LightSource** object to the **LightSources** collection.

Setting x, y and z to 0 generates a runtime error.

Parameters

<i>X</i>	Identifies the x value of the light source location. This parameter takes a number value.
<i>Y</i>	Identifies the y value of the light source location. This parameter takes a number value.
<i>Z</i>	Identifies the z value of the light source location. This parameter takes a number value.
<i>Intensity</i>	Indicates the light source intensity. This parameter takes a number value.

Remove

Syntax

Remove (*Index*)

Description

Removes a **LightSource** from the **LightSources** collection.

Parameter

<i>Index</i>	A specific light source by position in the list of light sources. This parameter takes a number value.
--------------	--

Location Object

The **Location** object represents the current position of a textual chart element such as the title, legend, or footnote.

A **Location** object is returned from the **location** property of the following objects:

- Footnote Object
- Legend Object
- Title Object

Location Properties

LocationType

Returns or sets the standard position used to display a chart element. This property takes a number value. **VtChLocationType** provides location options for chart elements.

This property is read-write.

Rect

Returns a reference to a Rect Object object that defines a coordinate location.

This property is read-only.

Visible

Returns or sets a value that determines whether a chart element is displayed.

The valid values are:

True The footnote, legend, or title is displayed.

False The elements are hidden.

This property is read-write.

Marker Object

The **Marker** object describes a marker that identifies a data point on a chart.

A **Marker** object is returned by the Marker property on the DataPoint Object object.

Marker Properties

FillColor

Returns a reference to a VtColor Object object that describes the color used to fill a chart element.

This property is read-only.

Pen

Returns a reference to a Pen Object object that describes the color and pattern of lines or edges on chart elements.

This property is read-only.

Size

Returns or sets the size of a chart element in points. This parameter takes a float value.

This property is read-write.

Style

Returns or sets the style used to draw certain chart elements. This property takes a number value.

For the **Marker** object, a **VtMarkerStyle** constant lists the marker type. The valid values are:

Number value	Constant	Description
0	VtMarkerStyleDash	Dash marker
1	VtMarkerStylePlus	Plus marker
2	VtMarkerStyleX	X marker
3	VtMarkerStyleStar	Star marker
4	VtMarkerStyleCircle	Circle marker
5	VtMarkerStyleSquare	Square marker
6	VtMarkerStyleDiamond	Diamond marker
7	VtMarkerStyleUpTriangle	Triangle marker
8	VtMarkerStyleDownTriangle	Down triangle marker
9	VtMarkerStyleFilledCircle	Filled circle marker
10	VtMarkerStyleFilledSquare	Filled square marker
11	VtMarkerStyleFilledDiamond	Filled diamond marker
12	VtMarkerStyleFilledUpTriangle	Filled triangle marker
13	VtMarkerStyleFilledDownTriangle	Filled down triangle marker
14	VtMarkerStyle3dBall	Three-dimensional ball marker
15	VtMarkerStyleNull	



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-only.

Visible

Returns or sets a value that determines whether a chart element is displayed.

The valid values are:

True The marker is displayed.

False The element is hidden.

This property is read-write.

Pen Object

The **Pen** object describes the color and pattern of lines or edges on a chart.

A **Pen** object is returned from the **Pen** property of the following objects:

- Axis Object
- Marker Object
- Plotbase Object
- Series Object

A **Pen** object is also returned from the GuideLinePen property on a Series Object object.

Pen Properties

Cap

Returns or sets a value that determines how line ends are capped.

VtPenCap provides methods for displaying line endings. The valid values are:

<i>Number value</i>	<i>Constant</i>	<i>Description</i>
0	VtPenCapButt	The line is squared off at the endpoint.
1	VtPenCapRound	A semicircle with the diameter of the line thickness is drawn at the end of the line.
2	VtPenCapSquare	The line continues beyond the endpoint for a distance equal to half the line thickness and is squared off.



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

Join

Returns or sets a value that determines how line segments are formed.

The **VtPenJoin** constants provide options for joining line segments in a series. The valid values are:

Number value	Constant	Description
0	VtPenJoinMiter	The outer edges of the two lines are extended until they meet.
1	VtPenJoinRound	A circular arc is drawn around the point where the two lines meet.
2	VtPenJoinBevel	The notch between the ends of two joining lines is filled.



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

Limit

Returns or sets the joint limit, in points, of the line. This property takes a number value.

A joint limit as a multiple of the line width. If two lines meet at a sharp angle, a mitered join results in a corner point that extends beyond the actual corner. If the distance from the inner join point to the outer join point exceeds the value in this variable, the join automatically changes to a bevel.

This property is read-write.

Style

Returns or sets the style used to draw certain chart elements.

The **VtPenStyle** constant that describes the style of pen.

This property is read-write.

VtColor

Returns a reference to a VtColor Object object that describes a drawing color in a chart.

This property is read-only.

Width

Returns or sets the width of a chart element, in points.

This property is read-write.

Plot Object

The **Plot** object represents the area upon which a chart is displayed.

A **Plot** object is returned from the Plot property on a chart object.

The **Plot** object allows you to program the following objects:

- Axis Object object—represents the *x*, *y*, and *z* axis of the chart. The *z* axis is only visible on 3D charts.
- BackDrop Object object—the area behind the axes.
- Location Object object—the ambient and edge light of the plot.
- Rect Object object—the location of the plot.
- Plotbase Object object—the appearance of the area beneath a chart.
- SeriesCollection Collection object—the collection of series sets.
- View3D Object object—the elevation and rotation of the 3D image.
- Wall Object object—the area behind the plot.
- Weighting Object object—the size of a pie in relation to other pies in the same chart.

Plot Properties

AngleUnit

Returns or sets the unit of measure used for all chart angles.

A **VtAngleUnits** constant describing the unit of measure. The valid values are:

Number value	Constant	Description
0	VtAngleUnitsDegrees	Chart angles are measured in degrees.
1	VtAngleUnitsRadians	Chart angles are measured in radians.
2	VtAngleUnitsGrads	Chart angles are measured in grads.



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

AutoLayout

Returns or sets a value that determines whether or not a Plot object is in manual or automatic layout mode.

The valid values are:

True (default) The Plot object automatically determines the proper size and position of the plot based on the size and position of other elements.

False The coordinates specified by Plot object's LocationRect property are used to position the plot

This property is read-write.

Backdrop

Returns a reference to a BackDrop Object object that describes the shadow, pattern, or picture behind a chart or chart element.

This property is read-only.

BarGap

Returns or sets the spacing of two-dimensional bars or clustered three-dimensional bars within a category. This property takes a number value.

This is measured as a percentage of the bar width. A value of 0 results in the bars touching. A value of 100 means the gap between the bars is as wide as the bars.

This property is read-write.

Clockwise

Returns or sets a value that specifies whether pie charts are drawn in a clockwise direction.

The valid values are:

True (default) Pie charts are drawn in a clockwise direction.

False The charts are drawn in a counterclockwise direction.

This property is read-write.

DataSeriesInRow

Returns or sets a value that indicates whether series data is being read from a row or a column in a data grid associated with a chart.

The valid values are:

True Series data is being read from a row in a data grid.

False Series data is being read from a column.

This property is read-write.

DefaultPercentBasis

Returns the default axis percentage basis for the chart. This property takes a number value.

The return value is an integer that specifies the default axis percentage basis.

This property is read-only.

DepthToHeightRatio

Returns or sets the percentage of the chart height to be used as the chart depth. This property takes a number value.

This property is read-write.

Light

Returns a reference to a Light Object object that provides information about the light illuminating a three-dimensional chart.

This property is read-only.

LocationRect

Returns a reference to a Rect Object object that specifies the location of the chart plot using x and y coordinates.

The values of this property are used to position the plot if AutoLayout is False.

If this property is set, then the **AutoLayout** property is automatically set to False.

This property is read-only.

PlotBase

Returns a reference to a Plotbase Object object that describes the appearance of the area beneath a chart.

This property is read-only.

Projection

Returns or sets the type of projection used to display the chart.

VtProjectionType provides viewpoint and perspective options for displaying and viewing a chart. The valid values are:

<i>Number value</i>	<i>Constant</i>	<i>Description</i>
0	VtProjectionTypePerspective	This provides the most realistic three-dimensional appearance. Objects farther away from you converge toward a vanishing point. This is the default projection.
1	VtProjectionTypeOblique	This is sometimes referred to as 2.5 dimensional. The chart does have depth, but the xy plane does not change when the chart is rotated or elevated.
2	VtProjectionTypeOrthogonal	Perspective is not applied in this three-dimensional view. The major advantage of using this type of projection is that vertical lines remain vertical, making some charts easier to read.
3	VtProjectionTypeFrontal	
4	VtProjectionTypeOverhead	



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

SeriesCollection

Returns a reference to a SeriesCollection Collection collection that provides information about the series that make up a chart.

This property is read-only.

Sort

Returns or sets the type of sort order used in a pie chart.

The **VtSortType** constants provide options for sorting pie charts. The valid values are:

Number value	Constant	Description
0	VtSortTypeNone	Pie slices are drawn in the order the data appears in the data grid.
1	VtSortTypeAscending	Pie slices are drawn, in order, from the smallest to the largest slice, starting at the defined starting angle and in the defined plot direction.
2	VtSortTypeDescending	Pie slices are drawn, in order, from the largest to the smallest slice, starting at the defined starting angle and in the defined plot direction.



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

StartingAngle

Returns or sets the position where you want to start drawing pie charts. This property takes a number value.

This angle can be measured in degrees, radians, or grads, depending on the current **AngleUnit** setting.

A value of 0 degrees indicates the 3 o'clock position. Setting the starting angle to 90 degrees moves the starting position to 12 o'clock if the **Clockwise** property is set to counterclockwise, or to 6 o'clock if it's set to clockwise. Valid values range from -360 to 360 degrees.

This property is read-write.

SubPlotLabelPosition

Returns or sets the position used to display a label on each pie in a chart.

The **VtSubPlotLabelLocationType** constants provide methods for displaying the subplot label. The valid values are:

Number value	Constant	Description
0	VtChSubPlotLabelLocationTypeNone	No subplot label is displayed.
1	VtChSubPlotLabelLocationTypeAbove	The subplot label is displayed above the pie.
2	VtChSubPlotLabelLocationTypeBelow	The subplot label is displayed below the pie.
3	VtChSubPlotLabelLocationTypeCenter	The subplot label is centered on the pie.



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

UniformAxis

Returns or sets a value that specifies whether the unit scale for all value axes in a chart is uniform.

The valid values are:

True The unit scale for all value axes is uniform.

False The unit scale is not uniform. The unit scale is determined by the plot size and positioning set according to the **AutoLayout** or **LocationRect** property. If **AutoLayout** is **True**, the plot size and position are based on the size and position of other automatically laid out elements. If **False**, the coordinates specified by **LocationRect** are used to position the plot and determine the axes unit scale.

This property is read-write.

View3d

Returns a reference to a View3D Object object that describes the physical orientation of a three-dimensional chart.

This property is read-only.

Wall

Returns a reference to a Wall Object object that describes the planar area depicting the y axes on a three-dimensional chart.

This property is read-only.

Weighting

Returns a reference to a Weighting Object object that describes the size of a pie in relation to other pies in the same chart.

This property is read-only.

WidthToHeightRatio

Returns or sets the percentage of the chart height to be used as the chart width. This property takes a number value.

This property is read-write.

XGap

Returns or sets the spacing of bars between divisions on the x axis. This space is measured as a percentage of the bar width. This property takes a number value.

A value of 0 results in the series of bars touching.

This property is read-write.

ZGap

Returns or sets the spacing of three-dimensional bars between divisions on the z axis. This space is measured as a percentage of the bar depth. This property takes a number value.

A value of 0 results in the series of bars touching along the z axis.

This property is read-write.

Additional Plot Property

Axis

Syntax

```
Axis(axisID, [&Index])
```

Description

Returns a reference to an Axis Object object that describes an axis on a chart.

AxisID A constant that identifies a specific axis. Valid values are:

Number value	Constant	Description
0	VtChAxisIdX	Identifies the x axis.
1	VtChAxisIdY	Identifies the y axis.
2	VtChAxisIdY2	Identifies the secondary y axis.
3	VtChAxisIdZ	Identifies the z axis.



You must use the numeric value, not the constant, in your PeopleCode program.

index Reserved for future use. Identifies the specific axis when there is more than one axis with the same axisID.

Plotbase Object

The **Plotbase** object describes the area beneath a chart.

A **Plotbase** object is returned from the PlotBase property on a Plot Object object.

Plotbase Properties

BaseHeight

Returns or sets the height of the three-dimensional chart base in points. This property takes a number value.

This property is read-write.

Brush

Returns a reference to a Brush Object object that describes the fill type used to display a chart element.

This property is read-only.

Pen

Returns or sets a reference to a Pen Object object that describes the color and pattern of lines or edges on chart elements.

This property is read-only.

Rect Object

The **Rect** object defines a coordinate location.

A **Rect** object is returned from the **Rect** property on the following objects:

- Location Properties
- Plot Object

Rect Properties

Min

Returns a reference to a Coor Object object that specifies the starting corner of a rectangle.

This property is read-only.

Max

Returns a reference to a Coor Object object that specifies the ending corner of a rectangle.

This property is read-only.

Series Object

The **Series** object represents a group of data points on a chart.

A **Series** object is returned from the Item property on a SeriesCollection Collection collection.

Series Properties

DataPoints

Returns a reference to a DataPoints Object Collection that describes the data points within a chart series.

This property is read-only.

GuideLinePen

Returns a reference to a Pen Object object that describes the pattern of line and color used to display guidelines.

Setting this property automatically sets the ShowLine property to True.

This property is read-only.

LegendText

Returns or sets the text that identifies the series in the legend of a chart. This property takes a string value.

This property is read-write.

Pen

Returns a reference to a Pen Object object that describes the color and pattern of lines or edges on chart elements.

This property is read-only.

Position

Returns a reference to a SeriesPosition Object object that describes the location of one series in relation to other chart series.

This property is read-only.

SecondaryAxis

Returns or sets a value that determines whether the series is charted on the secondary axis.

The valid values are:

True The series is charted on the secondary axis.

False (Default) The series is not charted on the secondary axis.

This property is read-write.

SeriesMarker

Returns a reference to a SeriesMarker Object object that describes a marker that identifies all data points within one series on a chart.

This property is read-only.

SeriesType

Returns or sets the type used to display the current series. This property takes a number value.

You must select the series to change using the Column property before using the **SeriesType** property.

The VtChSeriesType constants provide options for types of series.

This property is read-write.

ShowLine

Returns or sets a value that determines whether the lines connecting data points on a chart are visible.

The valid values are:

True (Default) The lines connecting data points appear on the chart.

False The data point lines do not appear.

This property is read-write.

StatLine

Returns a reference to a StatLine Object object that describes how statistic lines are displayed on a chart.

This property is read-only.

Series Special Properties

The following properties cannot be used to set a value. To set a value, you will have to use the ObjectSetProperty built-in PeopleCode function.

```
ObjectSetProperty(&MYCHART, &MYSERIES.ShowGuideLine(AxisID), True);
```

Or

```
ObjectSetProperty(&MYCHART, &MYSERIES.TypeByChartType(chtype), SeriesType);
```

ShowGuideLine

Syntax

```
ShowGuideLine([axisID], [, Index]) = [DisplayGuidelines]
```

Description

Returns a value that determines whether or not the connecting data point lines on a chart are displayed for a series.

Parameters

<i>axisId</i>	A <code>VtChAxisId</code> constant describing the series axis you want to set this property for. This property takes a number value.
<i>index</i>	An integer reserved for future use. For this version of Chart control, 1 is the only valid value for this argument.
<i>DisplayGuidelines</i>	A boolean value indicating whether series guidelines are displayed or not. The valid values are: True: The series guidelines are displayed. False: (Default) The series guidelines aren't displayed.

TypeByChartType

Syntax

```
TypeByChartType(chtype) = [SeriesType]
```

Description

Returns the series type used to draw a series if the chart type is set to *chType*. This method allows you to get the series type information based on a specified chart type without actually setting the chart type.

Parameters

<i>chtype</i>	A <code>VtChChartType</code> constant describing the chart type.
<i>SeriesType</i>	Returns a <code>VtChSeriesType</code> constant.

Series Method

Select

Syntax

```
Select()
```

Description

Selects the specified chart element.

SeriesCollection Collection

The **Seriescollection** describes a collection of chart series. Series are identified in the order of data grid columns, beginning with 1.

A **Seriescollection** is returned by the SeriesCollection property on a Plot Object object.

SeriesCollection Property

Item

Syntax

```
Item(Item)
```

Description

Returns a reference to an object within a collection that describes a chart element.

This property is read-only.

SeriesCollection Method

Count

Returns the number of objects in a collection.

SeriesMarker Object

The **SeriesMarker** object describes a marker that identifies all data points within one series on a chart.

A **SeriesMarker** object is returned from the SeriesMarker property on a Series Object object.

SeriesMarker Properties

Auto

Returns or sets a value that determines if the **SeriesMarker** object assigns the next available marker to all data points in the series.

Set this property to False if you wish to change the series marker type.

This property is automatically set to False if the Marker property of the DataPoint Object object is set.

The valid values are:

True (Default) The **SeriesMarker** object assigns the marker.

False You can assign a custom marker.

This property is read-write.

Show

Returns or sets a value that determines whether series markers are displayed on a chart.

The valid values are:

True Series markers are displayed.

False (default) Series markers are not displayed.

This property is read-write.

SeriesPosition Object

The **SeriesPosition** object describes the location where a chart series is drawn in relation to other series. If all series have the same order (position), they are stacked.

A **SeriesPosition** object is returned from the Position property on a Series Object object.

SeriesPosition Properties

Excluded

Returns or sets a value that determines whether a series is included on the chart.

The valid values are:

True The chart is drawn without including the series.

False (Default) The series is included when the chart is drawn. A series may be included in a chart, but still not display because it is Hidden.

This property is read-write.

Hidden

Returns or sets a value that determines whether a series is displayed on the chart.

The valid values are:

True The chart is drawn without displaying the series. However, any space allocated for the series still exists.

False (default) The series is displayed.

This property is read-write.

Order

Returns or sets the position of the series in the chart. If the position in order matches another series, the series are stacked. This property takes a number value.

This property is read-write.

StackOrder

Returns or sets in what position the current series is drawn if it is stacked with other series. This property takes a number value.

The value of this property specifies the order of the series if stacked with other series. Lower stack orders are on the bottom of the stack.

This property is read-write.

Shadow Object

The **Shadow** object holds information about the appearance of a shadow on a chart element.

A **Shadow** object is returned from the Shadow property of a BackDrop Object object.

Shadow Properties

Brush

Returns a reference to a Brush Object object that describes the fill type used to display a chart element.

This property is read-only.

Offset

Returns the distance that a chart element is offset or pulled away from its default location.

For the **Shadow** object, this is a reference to a Coor Object object that describe the x and y values of the offset.

This property is read-only.

Style

Returns or sets the style used to draw certain chart elements.

A **VtShadowStyle** constant used to describe the shadow type. The valid values are:

<i>Number value</i>	<i>Constant</i>	<i>Description</i>
0	VtShadowStyleNull	No shadow.
1	VtShadowStyleDrop	Drop shadow.



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

StatLine Object

The **StatLine** object describes how statistic lines are displayed on a chart.

A **Statline** object is returned from the StatLine property on a Series Object object.

StatLine Properties

Flag

Returns or sets which statistic lines are being displayed for a series.

The VtChStats constants provide methods of displaying statistic lines on a chart.

If more than one statistics line is displayed, the constants are combined with an OR operator.

This property is read-write.

VtColor

Returns a reference to a `VtColor Object` object that describes a drawing color in a chart.

This property is read-only.

Width

Returns or sets the width of a chart element, in points. This property takes a number value.

This property is read-write.

StatLine Special Property

Use the `ObjectSetProperty` built-in function to set the `Style` property:

```
ObjectSetProperty(&MYCHART, &MYSERIES.Style(type), Style);
```

Style

Syntax

```
Style(type) = [style]
```

Description

Returns the line type used to display the statistic line.

Parameters

<i>type</i>	A VtChStats constant used to describe the line type.
-------------	--

<i>style</i>	A VtPenStyle constant used to describe the stat line style.
--------------	---

This property is read-only.

TextLayout Object

The **TextLayout** object represents text positioning and orientation.

A **TextLayout** object is returned by the **TextLayout** property on the following objects:

- AxisTitle Object
- DataPointLabel Object
- Footnote Object
- Label Object
- Legend Object
- Title Object

TextLayout Properties

HorzAlignment

Returns or sets the method of horizontal alignment of text.

The **VtHorizontalAlignment** constants provide options for text alignment. The valid values are:

<i>Number value</i>	<i>Constant</i>	<i>Description</i>
0	VtHorizontalAlignmentLeft	All lines of text are aligned on the left margin.
1	VtHorizontalAlignmentRight	All lines of text are aligned on the right margin.
2	VtHorizontalAlignmentCenter	All lines of text are centered horizontally.
3	VtHorizontalAlignmentFill	
4	VtHorizontalAlignmentFlush	



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

Orientation

Returns or sets the method of orientation for text.

The **VtOrientation** constants provide options for positioning text. The valid values are:

Number value	Constant	Description
0	VtOrientationHorizontal	The text is displayed horizontally.
1	VtOrientationVertical	The letters of the text are drawn one on top of each other from the top down.
2	VtOrientationUp	The text is rotated to read from bottom to top.
3	VtOrientationDown	The text is rotated to read from top to bottom.



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

VertAlignment

Returns or sets the method used to vertically align text.

The **VtVerticalAlignment** constants provide methods of vertically aligning text. The valid values are:

Number value	Constant	Description
0	VtVerticalAlignmentTop	All lines of text are aligned at the top margin.
1	VtVerticalAlignmentBottom	All lines of text are aligned at the bottom margin.
2	VtVerticalAlignmentCenter	All lines of text are centered vertically.



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

WordWrap

Returns or sets a value that determines whether text wraps.

The valid values are:

True Text wraps.

False (Default) Text does not wrap.

This property is read-write.

Tick Object

The **Tick** object describes a marker indicating a division along a chart axis.

A **Tick** object is returned by the **Tick** property on an **Axis Object** object.

Tick Properties

Length

Returns or sets the length of axis tick marks, measured in points.

This property is read-write.

Style

Returns or sets the style used to draw certain chart elements.

A **VtChAxisTickStyle** constant used to describe the axis tick position. The valid values are:

<i>Number value</i>	<i>Constant</i>	<i>Description</i>
0	VtChAxisTickStyleNone	No tick marks are displayed on the axis.
1	VtChAxisTickStyleCenter	Tick marks are centered across the axis.
2	VtChAxisTickStyleInside	Tick marks are displayed inside the axis.
3	VtChAxisTickStyleOutside	Tick marks are displayed outside the axis.



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

Title Object

The **Title** object represents the text identifying the chart.

A **Title** object is returned by the Title property on a Chart object.

Title Properties

Backdrop

Returns a reference to a BackDrop Object object that describes the shadow, pattern, or picture behind a chart or chart element.

This property is read-only.

Font

Returns or sets a reference to a standard Font Object object that describes the font used to display text on the chart.

This property is read-write.

Location

Returns a reference to a Location Object object that describes the position of textual chart elements.

This property is read-only.

Text

Returns or sets the text used to display a chart element such as an axis title, data point label, footnote, or chart title. This property returns a string value.

The **Text** property is the default property for each of the objects to which it applies.

This property is read-write.

TextLayout

Returns a reference to a TextLayout Object object that describes text positioning and orientation.

This property is read-only.

TextLength

Returns or sets the number of characters in the text of a chart axis title, data point label, footnote, or chart title.

This property takes a number value, that is, the number of characters in the text.

This property is read-write.

VtFont

Returns a reference to a VtFont Object object that describes the font used to display chart text.

This property is read-only.

Title Method

Select

Syntax

```
Select()
```

Description

Selects the specified chart element.

ValueScale Object

The **ValueScale** object describes the scale used to display a value axis.

A **ValueScale** object is returned by the ValueScale property on an Axis Object object.

ValueScale Properties

Auto

Returns or sets a value that determines whether automatic scaling is used to draw the value axis.

The valid values are:

True The scale is automatically set based on the data being charted.

False The values in the Minimum, Maximum, MajorDivision and MinorDivision properties are used to scale the axis.

This property is read-write.

MajorDivision

Returns or sets the number of major divisions displayed on the axis. This property takes a number value.

If this property is set, then the ValueScale object's Auto property is automatically set to False.

This property is read-write.

Example

The following example sets the ValueScale properties for the X axis:

```
&CONTROL = GetControl();

&PLOT = &CHART.Plot;

&&Y_AXIS = &PLOT.Axis(1, 1);

&X_AXIS = &PLOT.Axis(0, 1);

&CONTROL.Plot.AutoLayout = False;

&CONTROL.Plot.UniformAxis = False;

&CONTROL.Plot.WidthToHeightRatio = 3;

&Y_AXIS.ValueScale.Auto = False;

&Y_AXIS.ValueScale.Maximum = Truncate(&MAX_YC_RATE + (&MAX_YC_RATE / 10),
&TRUNC_DIGITS);

&Y_AXIS.ValueScale.Minimum = Truncate(&MIN_YC_RATE - (&MIN_YC_RATE / 10),
&TRUNC_DIGITS);

&Y_AXIS.ValueScale.MajorDivision = 10;

&Y_AXIS.ValueScale.MinorDivision = 1;
```

Maximum

Returns or sets the highest or ending value on the chart value axis. This property takes a number value.

If this property is set, then the ValueScale object's Auto property is automatically set to False.

The Maximum property should be set before the **Minimum** property to avoid a chart display error.

This property is read-write.

Example

The following code example sets the values for the X Axis:

```
&CONTROL = GetControl();

&PLOT = &CHART.Plot;

&&Y_AXIS = &PLOT.Axis(1, 1);

&X_AXIS = &PLOT.Axis(0, 1);


&CONTROL.Plot.AutoLayout = False;

&CONTROL.Plot.UniformAxis = False;

&CONTROL.Plot.WidthToHeightRatio = 3;


&X_AXIS.CategoryScale.Auto = False;

&X_AXIS.ValueScale.Auto = False;

&X_AXIS.ValueScale.Maximum = Int(&DOMAIN_END) + 1;

&X_AXIS.ValueScale.Minimum = 0;

&X_AXIS.ValueScale.MajorDivision = 10;

&X_AXIS.ValueScale.MinorDivision = 1;
```

Minimum

Returns or sets the lowest or beginning value on the chart value axis. This property takes a number value.

If this property is set, then the ValueScale object's Auto property is automatically set to False.

The Maximum property should be set before the **Minimum** property to avoid a chart display error.

This property is read-write.

MinorDivision

Returns or sets the number of minor divisions displayed on the axis. This property takes a number value.

If this property is set, then the ValueScale object's Auto property is automatically set to False.

This property is read-write.

View3D Object

The **View3D** object represents the physical orientation of a three-dimensional chart.

A **View3D** object is returned from the View3d property on a Plot Object object.

View3D Properties

Elevation

Returns or sets a value that describes the degree of elevation from which a three-dimensional chart is viewed. This property takes a number value.

Elevation can be any number from 0 to 90 degrees. If you set the elevation to 90 degrees, you look directly down onto the top of the chart. If you set the elevation to 0, you look directly at the side of the chart. The default elevation is 30 degrees.

By default, degrees are used to measure elevation. However, these settings use the current settings for the AngleUnit property on the Plot Object object. The other options are: Grads and Radians.

This property is read-write.

Rotation

Returns or sets a value that describes the degree of rotation from which a three-dimensional chart is viewed. This property takes a number value.

Rotation can range from 0 to 360 degrees. By default, degrees are used to measure rotation. However, these settings use the current settings for the AngleUnit property on the Plot Object object. The other options are: Grads and Radians.

This property is read-write.

Example

The following example allows the user to rotate a 3D chart 15 degrees every time they double-click on it.

```
Function OnDbClick()
```

```

&MYCHART = GetControl();

If &MYCHART.Chart3D Then

    &ROTATE = &MYCHART.plot.view3d.rotation;

    &MYCHART.plot.view3d.rotation = &ROTATE + 15;

End-If;

End-Function;

```

View3D Method

Set

Syntax

Set (*Rotation*, *Elevation*)

Description

Sets the rotation and degree of elevation for a three-dimensional chart.

Parameters

<i>Rotation</i>	The degree of rotation. Rotation can range from 0 to 360 degrees. By default, degrees are used to measure rotation. However, these settings use the current settings for the AngleUnit property on the Plot Object object. The other options are: Grads and Radians.
<i>Elevation</i>	The degree of elevation. Elevation can be any number from 0 to 90 degrees.

If you set the elevation to 90 degrees, you look directly down onto the top of the chart. If you set the elevation to 0, you look directly at the side of the chart. The default elevation is 30 degrees.

By default, degrees are used to measure elevation. However, these settings use the current settings for the AngleUnit property on the Plot Object object. The other options are: Grads and Radians.

VtColor Object

The **VtColor** object describes a drawing color in a chart.

A **VtColor** object is returned from the following:

- FillColor and PatternColor properties of a Brush Object object.

- **VtColor** property of the StatLine Object, Pen Object and VtFont Object objects.

VtColor Properties

Automatic

Returns or sets a value that determines whether the color is calculated automatically. This is only used for edge pens on chart elements.

The valid values are:

True: Color automatically picks up the brush color used on the chart series.

False: The color is determined based on the settings of Value.

This property is read-write.

Blue

Returns or sets the blue component of the RGB value in a chart. This property takes a number value.

RGB specifies the relative intensity of red, green, and blue to cause a specific color to be displayed. The valid range for a normal RGB color is 0 to 16,777,215. The value for any argument to RGB that exceeds 255 is assumed to be 255.

This property is read-write.

Green

Returns or sets the green component of the RGB value in a chart. This property takes a number value.

RGB specifies the relative intensity of red, green, and blue to cause a specific color to be displayed. The valid range for a normal RGB color is 0 to 16,777,215. The value for any argument to RGB that exceeds 255 is assumed to be 255.

This property is read-write.

Red

Returns or sets the red component of the RGB value in a chart. This property takes a number value.

RGB specifies the relative intensity of red, green, and blue to cause a specific color to be displayed. The valid range for a normal RGB color is 0 to 16,777,215. The value for any argument to RGB that exceeds 255 is assumed to be 255.

This property is read-write.

VtColor Method

Set

Syntax

Set (*Red*, *Green*, *Blue*)

Description

Sets the red, green and blue values of the Color object.

Parameters

<i>Red</i>	The value for the red component of color. This property takes a number value.
<i>Green</i>	The value for the red component of color. This property takes a number value.
<i>Blue</i>	The value for the red component of color. This property takes a number value.

RGB specifies the relative intensity of red, green, and blue to cause a specific color to be displayed. The valid range for a normal RGB color is 0 to 16,777,215. The value for any argument to RGB that exceeds 255 is assumed to be 255.

VtFont Object

The **VtFont** object describes the font used to display chart text.

A **VtFont** object is returned from the VtFont property of the following objects:

- AxisTitle Object
- DataPointLabel Object
- Footnote Object
- Label Object
- Legend Object
- Title Object

VtFont Properties

Effect

Returns or sets the font effects in a chart.

A **VtFontEffect** constant describing the font effect. The valid values are:

<i>Number value</i>	<i>Constant</i>	<i>Description</i>
256	VtFontEffectStrikeThrough	Applies the strike-through attribute to the font.
512	VtFontEffectUnderline	Applies the underscore attribute to the font.



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

Name

Returns or sets the name of the font. This is the default property of the **VtFont** object. This property takes a string value.

This property is read-write.

Size

Returns or sets the size of a chart element in points. This property takes a number value.

This property is read-write.

Style

Returns or sets the style used to draw certain chart elements

A **VtFontStyle** constant describing the style of font. The valid values are:

<i>Number value</i>	<i>Constant</i>	<i>Description</i>
1	VtFontStyleBold	Applies the bold attribute to the font.
2	VtFontStyleItalic	Applies the italic attribute to the font.

<i>Number value</i>	<i>Constant</i>	<i>Description</i>
4	VtFontStyleOutline	Applies the outline attribute to the font.



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

VtColor

Returns a reference to a **VtColor** object that describes a drawing color in a chart.

This property is read-only.

Wall Object

The **Wall** object represents a planar area depicting the **y** axes on a three-dimensional chart.

A **Wall** object is returned from the Wall property on a Plot Object object.

Wall Properties

Brush

Returns a reference to a Brush Object object that describes the fill type used to display a chart element.

This property is read-only.

Pen

Returns or sets a reference to a Pen Object object that describes the color and pattern of lines or edges on chart elements.

This property is read-only.

Width

Returns or sets the width of a chart element, in points. This property takes a number value.

This property is read-write.

Weighting Object

The **Weighting** object represents the size of a pie in relation to other pies in the same chart.

A **Weighting** object is returned from the Weighting property on a Plot Object object.

Weighting Properties

Basis

Returns or sets the type of weighting used to determine pie size on a chart.

A **VtChPieWeightBasis** constant that identifies the weighting type. The valid values are:

<i>Number value</i>	<i>Constant</i>	<i>Description</i>
0	VtChPieWeightBasisNone	All pies are drawn the same size.
1	VtChPieWeightBasisTotal	The slice values in each pie are totalled and the pie with the highest total identified. The size of each pie in the chart is determined by the ratio of its total value compared to the largest pie.
2	VtChPieWeightBasisSeries	The first column of data in the data grid holds the relative size index. In other words, if you have 5 categories, you can control the size of the pies representing each category by using the first column of the data grid to number the rows 1 through 5. The size of the pie is determined by the ratio of its first column value and the largest value in the first column. The pie containing the 1 is the largest pie; the one containing the 5 the smallest. It is most common to exclude this first column of data so that the values are not drawn as a pie slice.



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

Style

Returns or sets the style used to draw certain chart elements.

A **VtChPieWeightStyle** constant that identifies the weighting factor method. The valid values are:

Number value	Constant	Description
0	VtChPieWeightStyleArea	The area of the individual pies changes, <i>based on their weighting</i> .
1	VtChPieWeightStyleDiameter	The diameter of the individual pies changes, based on their weighting.



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

Weighting Method

Set

Syntax

Set (*Basis*, *Style*)

Description

Sets the basis and style of the Weighting object.

Valid values for the **Basis** and **Style** parameters are listed under the Basis and Style properties, respectively.

CHAPTER 9

TreeView

A **TreeView** control displays a hierarchical list of **Node** objects, each of which consists of a label and an optional bitmap. A **TreeView** is typically used to display the headings in a document, the entries in an index, the files and directories on a disk, or any other kind of information that might usefully be displayed as a hierarchy.

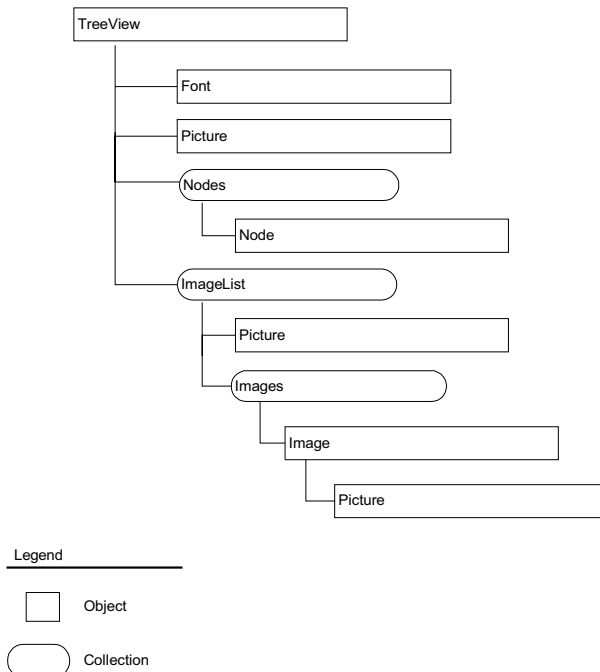


The TreeView ActiveX control is only available in Windows client. It does *not* work in the PeopleSoft Internet Architecture. If you want to display data using a tree in the PeopleSoft Internet Architecture, use the GenerateTree function. For more information see Using the GenerateTree Function.

From the TreeView control, you instantiate all the other objects you will need to work with (this is similar to how the session object works in PeopleCode, or the Tree API. You need to instantiate the high-level object using a built-in PeopleCode function, then you can instantiate all the other objects you need off the high-level object.)

From the TreeView control you can instantiate both objects and collections. A *collection* is just a group of like things. For example, you can have a collection of all the nodes in your tree, or all the images.

The following flow chart shows the different types of objects and collections instantiated from the TreeView control.



Objects and collections instantiated from the Tree View

The **TreeView** control uses the **ImageList** control to store the bitmaps and icons that are displayed in **Node** objects. You must put an **ImageList** control on the same page as the **TreeView** control in order to use **ImageLists**. You can add images to the **ImageList** control at either design time or runtime.



You can only associate a **TreeView** control with an **ImageList** control at runtime, not at design time. See [Associating a TreeView with an ImageList](#).

A **TreeView** control can use only one **ImageList** at a time. This means that every item in the **TreeView** control will have an equal-sized image next to it when the **TreeView** control's **Style** property is set to a style which displays images.



For more information about the **ImageList** Control, see [ImageList](#).



For examples of PeopleCode programs manipulating the **TreeView** ActiveX control, see [TreeView Examples](#).

Associating a TreeView with an ImageList

You can only associate a TreeView control with an ImageList control at runtime. You can only associate the two controls after *both* have been instantiated. Use the ImageList property on the TreeView control to associate the controls. You can add, change or delete images to the ImageList either before or after being associated with a TreeView control.

The following example sets up the image list and adds images to it before associating the ImageList with the TreeView. You

```
&TREE = GetControl();

&NODES = &TREE.Nodes;

/* Set up image list */

&IMAGELIST = GetControl(%Page, "IMAGES");

/* Associate TreeView with ImageList */

&TREE.imagelist = &IMAGELIST;

/* add images here */

&LISTIMAGES = &IMAGELIST.ListImages;

&LISTIMAGES.add(1, "key1", IMAGE.CLOSED_FOLDER);

&LISTIMAGES.add(2, "key2", IMAGE.OPEN_FOLDER);
```

Setting Indexed Properties

Some of the objects associated with the chart object have special properties (called indexed properties) that can't be set using the native property. These properties can only be set using the PeopleCode ObjectSetProperty built-in function. These exceptions are marked in the documentation.

For example, the Item property on some collections is considered a special property. You can use the native property to get an item in a collection:

```
&MYOBJECT = &MYCOLLECTION.Item(&Index);
```

However, you can't use the Item property to set an item. Instead, use the following code:

```
ObjectSetProperty(&MYCHART, &MYCOLLECTION.Item(&Index), &MYOBJECT);
```

TreeView Events

As the TreeView control is a high-level control that you can draw on a page in the Application Designer, it has events associated with it.

PeopleCode Event	Argument Description	General Description
OnBeforeLabelEdit(&CANCEL As number)	<i>&cancel</i> An integer that determines if the operation is canceled. Any nonzero integer cancels the operation. The default is 0.	<p><i>cancel</i> An integer that determines if the operation is canceled. Any nonzero integer cancels the operation. The default is 0.</p> <p>Occurs when a user attempts to edit the label of the currently selected Node object.</p> <p>Both the AfterLabelEdit and the BeforeLabelEdit events are generated only if the LabelEdit property is set to 0 (Automatic), or if the StartLabelEdit method is invoked.</p> <p>The BeforeLabelEdit event occurs after the standard Click event.</p> <p>To begin editing a label, the user must first click the object to select it, and click it a second time to begin the operation. The BeforeLabelEdit event occurs after the second click.</p> <p>To determine which object's label is being edited, use the SelectedItem property. The following example</p>

		<p>checks the index of a selected Node before allowing an edit. If the index is 1, the operation is cancelled.</p> <p>Function OnBeforeLabelEdit(&CANCEL As number)</p> <p>If GetControl().SelectedItem .Index = 1 Then</p> <p>...</p> <p>End-if</p> <p>End-Function;</p>
OnAfterLabelEdit(&CANCEL As number, &NEWSTRING As string)	<p><i>&cancel</i> An integer that determines if the label editing operation is canceled. Any nonzero integer cancels the operation. Boolean values are also accepted.</p> <p><i>&newstring</i> The string the user entered, or empty string if the user canceled the operation.</p>	<p>Occurs after a user edits the label of the currently selected Node object.</p> <p>The AfterLabelEdit event is generated after the user finishes the editing operation, which occurs when the user clicks on another Node or presses the ENTER key.</p> <p>To cancel a label editing operation, set <i>cancel</i> to any nonzero number or to True. If a label editing operation is canceled, the previously existing label is restored.</p> <p>The <i>newstring</i> argument can be used to test for a condition before canceling an operation.</p>
OnCollapse(&NODE As any)	<i>&node</i> A reference to the clicked Node object.	Generated when any Node object in a TreeView

		<p>control is collapsed.</p> <p>The Collapse event occurs before the standard Click event.</p> <p>There are three methods of collapsing a Node: by setting the Node object's Expanded property to False, by double-clicking a Node object, and by clicking a plus/minus image when the TreeView control's Style property is set to a style that includes plus/minus images. All of these methods generate the Collapse event.</p> <p>The event passes a reference to the collapsed Node object. The reference can validate an action, as in the following example:</p> <pre> Function OnCollapse((&NODE As any) If &NODE.Index = 1 Then &NODE.Expanded = True; /* Expand the node again */ End-If; End-Function; </pre>
OnExpand(&NODE As any)	<i>&node</i> A reference to the expanded Node object.	Occurs when a Node object in a TreeView control is expanded, that is, when its child nodes become visible.

		<p>The Expand event occurs after the Click and DblClick events.</p> <p>The Expand event is generated in three ways: when the user double-clicks a Node object that has child nodes; when the Expanded property for a Node object is set to True; and when the plus/minus image is clicked. Use the Expand event to validate an object, as in the following example:</p> <pre> Function Expand((&NODE As any) If &Node.Index != 1 Then &Node.Expanded = False; /* Prevent expand. */ End-If; End-function; </pre>
OnNodeCheck(&NODE As any)	<i>&Node</i> Returns a reference to the checked Node object.	Occurs when the Checked property of a Node object equals True and the Node object is checked or unchecked.
OnNodeClick(&NODE As any)	<i>&node</i> A reference to the clicked Node object.	<p>Occurs when a Node object is clicked.</p> <p>The standard Click event is generated when the user clicks any part of the TreeView control outside a node object. The NodeClick event is generated when the user clicks a particular Node</p>

		<p>object; the NodeClick event also returns a reference to a particular Node object which can be used to validate the Node before further action is taken.</p> <p>The NodeClick event occurs before the standard Click event.</p>
OnKeyDown(&KEYCODE As number, &SHIFT As number)		Occurs when the user presses a key while an object has the focus.
OnKeyUp(&KEYCODE As number, &SHIFT As number)		Occurs when the user releases a key while an object has the focus.
OnKeyPress(&KEYASCII As number)		Occurs when the user presses and releases an ANSI key.
OnMouseDown(&BUTTON As number, &SHIFT As number, &X As any, &Y As any)		Occurs when the user presses the mouse button while an object has the focus.
OnMouseUp(&BUTTON As number, &SHIFT As number, &X As any, &Y As any)		Occurs when the user releases the mouse button while an object has the focus.
OnMouseMove(&BUTTON As number, &SHIFT As number, &X As any, &Y As any)		Occurs when the user moves the mouse.
OnClick()		Occurs when the user presses and then releases a mouse button over an object
OnDbClick()		Occurs when you press and release a mouse button and then press and release it again over an object.
OnOLEGiveFeedback(&EFFECT As number, &DEFAULTCURSORS As boolean)		OLEGiveFeedback event.

OnOLECompleteDrag(&EFFECT As number)		OLECompleteDrag event.
OnOLEStartDrag(&DATA As any, &ALLOWEDEFFECTS As number)		OLEStartDrag event
OnOLESetData(&DATA As any, &DATAFORMAT As number)		OLESetData event
OnOLEDragOver(&DATA As any, &EFFECT As number, &BUTTON As number, &SHIFT As number, &X As number, &Y As number, &STATE As number)		OLEDragOver event
OnOLEDragDrop(&DATA As any, &EFFECT As number, &BUTTON As number, &SHIFT As number, &X As number, &Y As number)		OLEDragDrop event

Declaring a TreeView Object

TreeViews are declared using the Object data type. For example,

```
Local Object &TREE;
```

Scope of a TreeView Object

All chart objects, that is, charts, axis, plots, and so on, can be instantiated from PeopleCode or from a Visual Basic program.

ActiveX controls are not usable with either the Web Client or PeopleSoft Internet Architecture.

This object can be used anywhere you have PeopleCode. However, as a TreeView control is a control on a page, you will generally only use this object in PeopleCode programs that are associated with an online process, and not in an Application Engine program, a message subscription, a Business Component, and so on.

You can't access a TreeView control until after the Component Processor has loaded the page. This means you can't use the **GetControl** function in an event prior to the Activate Event.



For more information, see PeopleCode and the Component Processor.

TreeView Properties

Appearance

Controls the appearance of the TreeView. The valid values are:

Number value	Constant	Description
0	ccFlat	TreeView is 2 dimensional
1	cc3D	TreeView is 3 dimensional



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

BorderStyle

Controls the appearance of the border around the TreeView control. The valid values are:

Number value	Constant	Description
0	ccNone	TreeView has no border.
1	ccFixedSingle	TreeView has a single-line border.



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

Checkboxes

Specifies whether a checkbox displays next to every node or not. The valid values are:

False (default)

True



If the style of tree you've specified displays plus or minus signs next to closed or open nodes, this property has no effect: even if you set it to True, checkboxes won't be displayed.

This property is read-write.

DropHighlight

Returns a reference to a Node Object object that is highlighted with the system highlight color when the cursor moves over it.

For example,

```
Function OnOLEDragOver(&SOURCE As object, &X As number, &Y As number,
&STATE As number)
```

```
    &THIS = GetControl();
```

```
    &THIS.DropHighlight = &THIS.HitTest(X,Y)
```

```
End-Function;
```

```
Function onOLEDragDrop(&SOURCE As object, &X As number, &Y As number)
```

```
    &Caption = GetControl().DropHighlight.Text;
```

```
    GetControl().DropHighlight = Nothing /* must be done to release the
highlight effect */
```

```
End-Function;
```

This property is read-only.

Enabled

Specifies whether the user can edit the TreeView. This property takes a boolean value.

This property is read-write.

Font

Returns a reference to a Font Object object.

This property is read-only.

FullRowSelect

This property determines whether an entire row can be selected or not. The valid values are:

False (default)

True

This property is read-write.

HideSelection

This property specifies whether the selection is hidden or not. The valid values are:

True (default)

False

This property is read-write.

HotTracking

This property determines whether mouse-sensitive highlighting is enabled. The valid values are:

True

False (default)

This property is read-write.

ImageList

Sets a reference to an ImageList object (a collection of **Images**)



You can't associate an ImageList object with a TreeView control at design time. You can only do it at runtime.

This property is read-write.

Example

```
Local Object &TREE, &IMAGELIST;

&TREE = GetControl();

&IMAGELIST = GetControl(%Page, "IMAGES");

&TREE.ImageList = &IMAGELIST;
```



For an example program, see TreeView and ImageList.

Indentation

This property sets the width of the indentation of objects in a control. This property takes a number value. Default is 32.

This property is read-write.

LabelEdit

The valid values are:

Number value	Constant
0	TvwAutomatic
1	TvwManual



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

LineStyle

The valid values are:

Number value	Constant
0	TvwTreeLines
1	TvwRootLines



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

MousePointer

This property sets or returns the appearance of the mouse pointer.

Number value	Constant
0	CcDefault
1	CcArrow
2	CcCross
3	CciBeam
4	CcIcon
5	CcSize

Number value	Constant
6	CcSizeNESW
7	CcSizeNS
8	CcSizeNWSE
9	CcSizeEW
10	CcUpArrow
11	CcHourglass
12	CcNoDrop
13	CcArrowHourglass
14	CcArrowQuestion
15	CcSizeAll
99	CcCustom



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

Nodes

Returns a reference to a Nodes Collection collection.

This property is read-only.

OLEDragMode

Returns or sets whether the component or the programmer handles an OLE drag/drop operation.

Valid values are:

Number value	Constant
0	CcOLEDragManual
1	CcOLEDragAutomatic



You must use the numeric value, not the constant, in your PeopleCode program.

When **OLEDragMode** is set to **Manual**, you must call the **OLEDrag** method to start dragging, which then triggers the **OLEStartDrag** event.

When **OLEDragMode** is set to **Automatic**, the source component fills the **DataObject** object with the data it contains and sets the *effects* parameter before initiating the **OLEStartDrag** event (as well as the **OLESetData** and other source-level OLE drag/drop events) when the user attempts to drag out of the control. This gives you control over the drag/drop operation and allows you to intercede by adding other formats, or by overriding or disabling the automatic data and formats using the **Clear** or **SetData** methods.

If the source's **OLEDragMode** property is set to **Automatic**, and no data is loaded in the **OLEStartDrag** event, or *aftereffects* is set to **0**, then the OLE drag/drop operation does not occur.



If the **DragMode** property of a control is set to **Automatic**, the setting of **OLEDragMode** is ignored, because regular Visual Basic drag and drop events take precedence.

This property is read-write.

OLEDropMode

Returns or sets how a target component handles drop operations.

Valid values are:

Number value	Constant
0	CcOLEDropNone
1	CcOLEDropAutomatic



You must use the numeric value, not the constant, in your PeopleCode program. **Also**, The target component inspects what is being dragged over it in order to determine which events to trigger. There is no collision of components or confusion about which events are fired, because only one type of object can be dragged at a time.

This property is read-write.

PathSeparator

This property takes a string value.

This property is read-write.

Scroll

This property specifies if scrollbars are displayed. This property takes a boolean value. False is the default. However, if your nodes are expanded beyond the end of the control on the page, scrollbars will automatically be displayed.

This property is read-write.

SelectedItem

Returns a reference to a Node Object object (a single node)

This property is read-only.

SingleSel

This property specifies if an item is expanded when selected. This property takes a boolean value. False is the default.

Sorted

This property does the following:

- determines whether the child nodes of a **Node** object are sorted alphabetically.
- determines whether the root level nodes of a **TreeView** control are sorted alphabetically.

This property takes a boolean value. False is the default.

Setting the **Sorted** property to **True** sorts the current **Nodes** collection only. When you add new **Node** objects to a **TreeView** control, you must set the **Sorted** property to **True** again to sort the added **Node** objects.

This property is read-write.

Style

This property specifies the style of the TreeView. The valid values are:

Number value	Constant
0	TvwTextOnly
1	TvwPictureText
2	TvwPlusMinusText
3	TvwPlusPictureText
4	TvwTreelinesText
5	TvwTreelinesPictureText

Number value	Constant
6	TvwTreelinesPlusMinusText
7	TvwTreelinesPlusMinusPictureText



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

TreeView Methods

GetVisibleCount

Syntax

```
GetVisibleCount()
```

Description

Retrieves the number of Node Object objects that can be fully visible in the tree view control. The number of **Node** objects is determined by how many lines can fit in a window. The total number of lines possible is determined by the height of the control and the **Size** property of the Font Object object. The count includes the partially visible item at the bottom of the list.

This number may greater than the number of nodes currently in the control.

HitTest

Syntax

```
HitTest(X, Y)
```

Description

Returns a reference to Node Object object located at the coordinates of x and y. Most often used with drag-and-drop operations to determine if a drop target item is available at the present location. Both *X* and *Y* take a number (decimal) value.

StartLabelEdit

Syntax

```
StartLabelEdit()
```

Description

Enables a user to edit a label. Used with the AfterLabelEdit and BeforeLabelEdit events.

Font Object

The **Font** object contains font attributes (font name, font size, color, and so on) for the TreeView control.

A font object is returned from the Font property, used with the TreeView object.

```
&MYTREE = GetControl();  
  
&MYFONT = &MYTREE.Font;
```

Font Properties

Bold

Indicates whether the font is boldfaced. False is the default.

This property is read-write.

Charset

The character set used in the font. Valid values are:

<i>Number value</i>	<i>Description</i>
0	ANSI_CHARSET
1	DEFAULT_CHARSET
2	SYMBOL_CHARSET



You must use the numeric value, not the constant, in your PeopleCode program.

This property is read-write.

Italic

Indicates whether the font is italicized. False is the default.

This property is read-write.

Name

The name of the font, for example, Arial. This property takes a string value.

This property is read-write.

Size

The point size of the font. This property takes a number value.

This property is read-write.

Strikethrough

Indicates whether the font is strikethrough. False is the default.

This property is read-write.

Underline

Indicates whether the font is underlined. False is the default.

This property is read-write.

Weight

The boldness of the font. This property takes a number value.

This property is read-write.

Node Object

A **Node** object is an item in a **TreeView** control that can contain images and text.

A **Nodes** collection contains one or more **Node** objects.

A **Node** object is returned from the following TreeView object properties:

- DropHighlight
- SelectedItem

```
&MYTREE = GetControl();
```

```
&MYNODE = &MYTREE.SelectedItem;
```

A **node** object is returned from the following **Nodes** collection properties:

- Add

- Item

```
&TREE = GetControl();

&NODECOL = &TREE.Nodes;

For &I = 1 to &NODECOL.Count;

    &MYNODE = &NODECOL.Item(&I);

    /* do some processing */

End-For;
```

Nodes can contain both text and pictures. However, to use pictures, you must associate an ImageList with a TreeView control using the **ImageList** property. You can only associate an ImageList with a TreeView control at runtime.



For more information, see Associating a TreeView with an ImageList.

Pictures can change depending on the state of the node; for example, a selected node can have a different picture from an unselected node if you set the **SelectedImage** property to an image from the associated **ImageList**.

After creating a **TreeView** control, you can add, remove, arrange, and otherwise manipulate **Node** objects by setting properties and invoking methods.

You can navigate through a tree in code by retrieving a reference to **Node** objects using **Root**, **Parent**, **Child**, **FirstSibling**, **Next**, **Previous**, and **LastSibling** properties. The **Child**, **FirstSibling**, **LastSibling**, **Previous**, **Parent**, **Next**, and **Root** properties all return a reference to another **Node** object.

Several styles are available which alter the appearance of the control. **Node** objects can appear in one of eight combinations of text, bitmaps, lines, and plus/minus signs.

Using the Index Property

Here is a clarification regarding Index numbers on tree nodes:

Every node created by TreeView has a unique index number assigned to it. For example, the following code will give you the index number assigned to a particular tree node:

```
&INDX_VAL = GetControl("PC_TREEDEMO_PNL2", "ACTIVEX1").SELECTEDITEM.INDEX;
```

The Index number is dynamically created at the time you create the tree. When you add additional nodes to an existing tree, the index number begins incrementing from the very last index number on the existing tree.

For example, if your last node on the tree has an index number of 78, the newly inserted nodes will have index numbers 79, 80, 81, etc. It doesn't matter where on the tree you insert the node.

If you re-build the tree (re-load the page), with the new nodes already a part of the data collection, the index numbers will change. This means what was index 79, 80, 81 and so on, is now 24, 25, 26, and so on.

Using index numbers to navigate and position up and down a tree is very essential in managing a tree. Just remember the index numbers always change (especially if your tree grows).

Using the Key Property

If you need to reference nodes on a tree, instead of using Index (since these change), you can use the **Key** property.

When creating a tree, assign your own unique ID (it must be a string value) to the Key property of the node collection. For example:

```
GetControl("PC_TREEDEMO_PNL2", "ACTIVEX1").NODES.ITEM(&PARENT_NODE_IDX).KEY =  
"PS" | EMPLID;
```



In the previous example, you must put at least one alphabetic character in front of the EMPLID assignment. If EMPLID is all numeric, you are not allowed to assign that value to the **Key** property, even if you do String(EMPLID).

Using this example, you can now use the **Key** property to point your way back to the data base.

Node Properties

Bold

Specifies whether the node is displayed in **bold** font or not. This property takes a boolean value: False is the default.

This property is read-write.

Checked

If the tree is displayed with checkboxes, this property specifies whether the node has a check in the checkbox or not. This property is only valid when the Checkboxes property is set to True.

This property is read-write.

Child

Returns a reference to the first child of a Node Object object in a **TreeView** control.

This property is read-only.

Children

Returns the number of child Node Object objects contained in a **Node** object. This property takes a number value.

The **Children** property can be used to check if a **Node** object has any children before performing an operation that affects the children.

This property is read-only.

Expanded

Specifies whether the node is expanded or not. This property takes a boolean value: False is the default.

This property is read-write.

ExpandedImage

The Index or Key of an image in an ImageList control used when the Node is expanded. Can be of type Any (number or string).

This property is read-write.

FirstSibling

Returns a reference to the first sibling of a **Node** object in a **TreeView** control.

The first sibling is the **Node** that appears in the first position in one level of a hierarchy of nodes. Which **Node** actually appears in the first position depends on whether or not the **Node** objects at that level are sorted, which is determined by the Sorted property.

This property is read-only.

FullPath

Returns the fully qualified path of the referenced Node object in a TreeView control. This property takes a string value. When you assign this property to a string variable, the string is set to the FullPath of the node with the specified index.

This property is read-write.

Image

The index or key of a ListImage object to be used. Can be of type Any (number or string).

This property is read-write.

Index

This property specifies the position of the node in the node collection. This property takes a number value.

This property is read-write.

Key

This property returns or sets a string that uniquely identifies a member in a collection. The string value must contain at least one letter.

If the string is not unique, an error will occur.

You can set the **Key** property when you use the Add method to add an object to a collection.

If you expect the **Index** property to change dynamically, refer to objects in a collection using the **Key** property.



You need a special construction for the Key property to work correctly. You must concatenate the value to the string property.

For example:

```
&PARENT.KEY = String(&I) | "MyKey";
```

Or

```
&NODE.KEY = "Key" | String(&I);
```

This property is read-write.

LastSibling

Returns a reference to the last **Node** object in a hierarchy level.

This property is read-only.

Next

Returns a reference to the next sibling **Node** of a TreeView control's Node object.

This property is read-only.

Parent

Returns or sets the parent object of a **Node** object.

This property is read-only.

Previous

Returns a reference to the previous sibling of a **Node** object.

This property is read-only.

Root

Returns a reference to the root **Node** object of a selected Node

This property is read-only.

Selected

Specifies whether a node is selected or not. This property takes a boolean value: False is the default.

This property is read-write.

SelectedImage

Returns or sets the Index or Key of an image in an ImageList control which is displayed when a Node object is selected. Can be of type Any (number or string).

This property is read-write.

Sorted

Specifies whether the nodes are displayed in alphabetic order. This property takes a boolean value: the default is False.

This property is read-write.

Tag

Stores any extra data needed for your program. This property takes a string value.

This property is read-write.

Text

Returns or sets the text contained in an object. This property takes a string value.

This property is read-write.

Visible

Specifies whether the node is visible or not. This property takes a boolean value: True is the default.

This property is read-write.

Node Methods

CreateDragImage

Syntax

```
CreateDragImage()
```

Description

Returns a reference to a Picture Object object. Creates a composite image from an icon and a caption for use in drag and drop operations.

EnsureVisible

Syntax

```
EnsureVisible()
```

Description

Ensures a Node object is visible, scrolling or expanding the control if necessary. Takes a Boolean value.

Nodes Collection

A **nodes** object is a collection of **node** objects.

A **Nodes** object is returned from the TreeView object's Nodes property:

```
&TREE = GetControl();  
  
&NODECOL = &TREE.Nodes;
```

The following code will allow you to build a simple tree using a TreeView control on your page. This code is in the PSControlInit event.

```
Local object &THIS;  
  
Local object &NODES;
```

```

Function PSControlInit()

    &THIS = GetControl();

    &NODES = &THIS.Nodes;

    &PARENT = &NODES.ADD();

    &PARENT.TEXT = "ROOT";

    &INDEX = &PARENT.INDEX;

    &CHILD = &NODES.ADD(&INDEX, 4);

    &CHILD.TEXT = "Child1";

    &CHILD = &NODES.ADD(&INDEX, 4);

    &CHILD.TEXT = "Child2";

    &INDEX = &CHILD.INDEX;

    &CHILD = &NODES.ADD(&INDEX, 4);

    &CHILD.TEXT = "GrandChild1";

    &CHILD = &NODES.ADD(&INDEX, 4);

    &CHILD.TEXT = "GrandChild2";

End-Function;

```

Nodes Property

Count

Returns the number of Node Object objects in the Nodes collection.

This property is read-only.

Nodes Methods

Add

Syntax

```
Add([relative, ] [relationship, ] [key, ] text [, image] [ , selectedimage])
```

Description

Returns a reference to the new Node Object object.

Parameters

Relative

The index number (Node.Index) or key (string) of a pre-existing Node object. This parameter is optional for a root level node. Can be used with data type Any.

Relationship

This parameter is optional for any root level node. The valid values are:

Number value	Constant	Description
0	tvwFirst	First. The Node is placed before all other nodes at the same level of the node named in relative.
1	tvwLast	Last. The Node is placed after all other nodes at the same level of the node named in relative. Any Node added subsequently may be placed after one added as Last.
2	tvwNext	(Default) Next. The Node is placed after the node named in relative.
3	tvwPrevious	Previous. The Node is placed before the node named in relative.
4	tvwChild	Child. The Node becomes a child node of the node named in relative.



You must use the numeric value, not the constant, in your PeopleCode program.

Key

A unique string that can be used to retrieve the **Node** with the **Item** method. This parameter is optional.

Text

The string that appears in the Node. **This parameter is required.**

Image The index of an image in an associated **ImageList** control. This parameter is optional.

Selectedimage The index of an image in an associated **ImageList** control that is shown when the **Node** is selected. This parameter is optional.

Clear

Syntax

```
Clear()
```

Description

Removes all objects in a collection. If you only want to remove a single node from the collection, use the **Remove** method.

Item

Syntax

```
Item(Index)
```

Description

Returns a reference to a **Node** object. The *index* parameter must be equal to or less than the number of nodes in the collection.

Remove

Syntax

```
Remove(Index)
```

Description

Removes the node object specified in the collection by *index*. If you want to remove all the nodes in a collection, use the **Clear** method.

Picture Object

The **Picture** object enables you to manipulate bitmaps, icons, metafiles, enhanced metafiles, GIF, and JPEG images assigned to objects having a Picture property.

A **Picture** object is returned from the following:

- the **ExtractIcon** method on an Image Object object.

- the Overlay method on an ImageList object.
- the Picture property on an Image Object object.

Picture Properties

Height

Return the height of the picture in HiMetric units. This property is read-only.

Type

Returns the type of the picture. Valid values are:

<i>Number value</i>	<i>Description</i>
0	Picture is empty
1	Bitmap
2	Metafile
3	Icon
4	Enhanced metafile

This property is read-only.

Width

Return the height of the picture in HiMetric units. This property is read-only.

ImageList

The **TreeView** control uses the **ImageList** control to store the bitmaps and icons that are displayed in **Node** objects. You must put an ImageList control on the same page as the TreeView control in order to use ImageLists. The ImageList control does not display at runtime.

You can either add images to the ImageList control at design time, using the property sheet provided by the vendor, or you can add them in at runtime. Images that are added at runtime must already have been created as image components in the Application Designer. You can only associate a TreeView control with an ImageList control at runtime, not at design time.



For more information about associating a TreeView with an ImageList, see [Associating a TreeView with an ImageList](#).



For an example program, see TreeView and ImageList.

Declaring ImageList Objects

ImageLists, Images and Image objects are declared using the Object data type. For example,

```
Local Object &IMAGELIST;
```

```
Local Object &IMAGES;
```

Scope of ImageList Objects

ImageList, Images, and Image objects can be instantiated from PeopleCode or from a Visual Basic program.

ActiveX controls are not usable with either the Web Client or PeopleSoft Internet Architecture.

This object can be used anywhere you have PeopleCode. However, these controls are on a page, you will generally only use them in PeopleCode programs that are associated with an online process, and not in an Application Engine program, a message subscription, a Business Component, and so on.

You can't access an ImageList control until after the Component Processor has loaded the page. This means you can't use the **GetControl** function in an event prior to the Activate Event.



For more information, see PeopleCode and the Component Processor.

You can get an ImageList object using the GetControl function:

```
&ILIST = GetControl(%Page, "IMAGES");
```

The ImageList control only has the following events associated with it:

- PSControlInit Event
- PSLostFocus Event

ImageList Properties

ImageHeight

This property specifies the height of the image. This property takes a number value.

This property is read-only.

ImageWidth

This property specifies the width of the image. This property takes a number value.

This property is read-only.

ListImages

Returns a reference to the Images Collection collection.

This property is read-only.

MaskColor

Numeric color value. This property takes a number value.

UseMaskColor

This property specifies whether to use a mask color with the image. This property takes a boolean value: the default is False.

ImageList Method

Overlay

Syntax

```
Overlay(index1, index2)
```

Description

Draws one image from a ListImages collection over another. Returns a reference to the new object as a Picture Object object.

<i>index1</i>	An integer (Index property) or unique string (Key property) that specifies the image to be overlaid.
<i>index2</i>	An integer (Index property) or unique string (Key property) that specifies the image to be drawn over the object specified in <i>index1</i> .

The color of the image that matches the MaskColor property is made transparent. If no color matches, the image is drawn opaquely over the other image.

Images Collection

An **Images** collection is returned from the ListImages property on an ImageList object:

```
Local Object &ILIST, &IMAGES;  
  
&ILIST = GetControl();  
  
&IMAGES = &ILIST.ListImages;
```

Images Properties

Count

Returns the number of objects in the collection.

This property is read-only.

Images Methods

Add

Syntax

```
Add([index, ] [key,] Picture)
```

Description

Adds a new image to the collection. Returns a reference to the new **Image** object.

Index The index number you want the new image to have.

Key Returns or sets a string that uniquely identifies a member in a collection. If the string is not unique, an error will occur. You can set the **Key** property when you use the **Add** method to add an object to a collection. If you expect the **Index** property to change dynamically, refer to objects in a collection using the **Key** property.

Picture The object you want to add. This must be a reference to an already instantiated picture object.

Returns

None.

Example

The following example adds two images (created in Application Designer as image components) to an image list.

```
/* Set up image list */

&IMAGELIST = GetControl(%Page, "IMAGES");

&LISTIMAGES = &IMAGELIST.ListImages;


/* add images here */

&LISTIMAGES.add(1, "key1", IMAGE.CLOSED_FOLDER);

&LISTIMAGES.add(2, "key2", IMAGE.OPEN_FOLDER);
```

Clear

Syntax

```
Clear()
```

Description

Removes all objects in a collection. If you only want to remove a single node from the collection, use the Remove method.

Item

Syntax

```
Item(Index)
```

Description

Returns a reference to an Image Object object. The *index* parameter must be equal to or less than the number of nodes in the collection.

Remove

Syntax

```
Remove(Index)
```

Description

Removes the **Image** object specified in the collection by *index*. If you want to remove all the images in a collection, use the Clear method.

Image Object

An **Image** is an object that associates a picture with some key text that allows it to be used in an Images collection object.

An **Image** object is returned from the following Images Collection collection methods:

- Add
- Item

```
Local Object &ILIST, &IMAGES;  
  
&ILIST = GetControl();  
  
&IMAGES = &ILIST.ListImages;  
  
For &I = 1 to &IMAGES.Count  
    &MYIMAGE = &IMAGES.Index(&I);  
    /* do some processing */  
End-For;
```

Image Properties

Index

This property returns the position index of the image in the images collection. This property takes a number value.

This property is read-write.

Key

This property returns a string that uniquely identifies a member in a collection.

If you expect the **Index** property to change dynamically, refer to objects in a collection using the **Key** property.

This property is read-write.

Picture

Returns a reference to a Picture Object object.

This property is read-only.

Tag

Additional information about the object. This property takes a string value.

This property is read-write.

Image Methods

ExtractIcon

Syntax

```
ExtractIcon()
```

Description

Returns a reference to the Picture Object object associated with this image.

Index

Syntax

```
Index(index)
```

Description

Returns a reference to an **Image** object. The *index* parameter must be equal to or less than the number of nodes in the collection.

TreeView Examples

The following examples use PeopleCode to load the TreeView ActiveX control with data and to manipulate it.

TreeView and Tree API

The following sample program will load a TreeView control with data from the first tree in the list of all trees created with Tree Manager.



The following is just sample code to load the tree. A more realistic example would **Close** the tree when processing was finished. This code will produce an error every other time it is run because the tree is never closed.

```
Function PSControlInit()
```

```

&SESSION = GetSession();

&SESSION.Connect(1, "EXISTING", "", "", 0);

&VC_TREECOLL = &SESSION.FindTree("", "", "", "", "");

&VC_TREE = &VC_TREECOLL.First;

/* Get the Tree Name, Eff Date, etc. from tree */

&SETID = &VC_TREE.KeySetId;

&USERKEY = &VC_TREE.KeyUserKeyValue;

&TREENAME = &VC_TREE.KeyName;

&EFFDT = &VC_TREE.KeyEffDt;

&BRANCHNAME = &VC_TREE.KeyBranchName;

&UPDATE = False;

&VC_TREE.Open(&SETID, &USERKEYVALUE, &TREENAME, &EFFDT, &BRANCHNAME,
&UPDATE);

/* Get root Node */

&SRC_PARENT = &VC_TREE.FindRoot();

&PARENT_NAME = &SRC_PARENT.NAME;

/* Get ActiveX Control of TreeView and add the Parent (ROOT) node with the
same NAME and Push the Node onto a stack */

&TREEVIEW1 = GetControl();

&TREEVIEW1.STYLE = 7;

&TREEVIEW1.font.bold = False;

&TREEVIEW1.font.size = 14;

&DESC_PARENT = &TREEVIEW1.nodes.Add();

&DESC_PARENT.TEXT = &PARENT_NAME;

```

```

&DESC_PARENT.EXPANDED = True;

&INDX = &DESC_PARENT.INDEX;

&SRC_STACK = CreateArray(&SRC_PARENT);

&INDX_STACK = CreateArray(&INDX);

/* Loop through the Tree and process any Node still in the stack */

While &SRC_STACK.Len > 0

    /* Pop each Node as the Stack and process it's Children */

    &SRC_PARENT = &SRC_STACK.Pop();

    &INDX = &INDX_STACK.Pop();

    /* process the leaves */

    If &SRC_PARENT.HasChildLeaves Then

        &SRC_CHILD_LEAF = &SRC_PARENT.FirstChildLeaf;

        &DESC_CHILD_LEAF = GetControl().NODES.ADD(&INDX, 4);

        &RANGE_FROM = &SRC_CHILD_LEAF.RANGEFROM;

        &RANGE_TO = &SRC_CHILD_LEAF.RANGETO;

        If &RANGE_FROM = &RANGE_TO Then

            &LEAF_DESCR = &RANGE_FROM;

        Else

            &LEAF_DESCR = &RANGE_FROM | "-" | &RANGE_TO;

        End-If;

        &DESC_CHILD_LEAF.TEXT = &LEAF_DESCR;

        &DESC_CHILD_LEAF.EXPANDED = False;

    While &SRC_CHILD_LEAF.HasNextSib

        &SRC_CHILD_LEAF = &SRC_CHILD_LEAF.NextSib;

```

```

&DESC_CHILD_LEAF = GetControl().NODES.ADD(&INDX, 4);

&RANGE_FROM = &SRC_CHILD_LEAF.RANGEFROM;

&RANGE_TO = &SRC_CHILD_LEAF.RANGETO;

If &RANGE_FROM = &RANGE_TO Then

    &LEAF_DESCR = &RANGE_FROM;

Else

    &LEAF_DESCR = &RANGE_FROM | "-" | &RANGE_TO;

End-If;

&DESC_CHILD_LEAF.TEXT = &LEAF_DESCR;

&DESC_CHILD_LEAF.EXPANDED = False;

End-While;

End-If;

/* process the nodes */

If &SRC_PARENT.HasChildNodes Then

    &SRC_CHILD = &SRC_PARENT.FirstChildNode;

    &DESC_CHILD = GetControl().NODES.ADD(&INDX, 4);

    &DESC_CHILD.TEXT = &SRC_CHILD.NAME;

    &DESC_CHILD.EXPANDED = False;

    &SRC_STACK.Push(&SRC_CHILD);

    &T_INDX = &DESC_CHILD.INDEX;

    &INDX_STACK.Push(&T_INDX);

    While &SRC_CHILD.HasNextSib

        &SRC_CHILD = &SRC_CHILD.NextSib;

        &DESC_CHILD = GetControl().NODES.ADD(&INDX, 4);

        &DESC_CHILD.TEXT = &SRC_CHILD.NAME;

```

```

        &DESC_CHILD.EXPANDED = False;

        &SRC_STACK.Push(&SRC_CHILD);

        &T_INDX = &DESC_CHILD.INDEX;

        &INDX_STACK.Push(&T_INDX);

    End-While;

End-If;

End-While;

End-Function;

```

TreeView and ImageList

The following code example sets up a generic Tree View control. The OPEN_FOLDER and CLOSED_FOLDER are image components already existing in the Application Designer.

```

Local object &THIS;

Local object &NODES;

Function PSControlInit()

    &TREE = GetControl();

    &NODES = &TREE.Nodes;

    /* Set up image list */

    &IMAGELIST = GetControl(%Page, "IMAGES");

    &LISTIMAGES = &IMAGELIST.ListImages;

    /* add images here */

    &LISTIMAGES.add(1, "key1", IMAGE.CLOSED_FOLDER);

    &LISTIMAGES.add(2, "key2", IMAGE.OPEN_FOLDER);

```

```
&TREE.imagelist = &IMAGELIST;

/* set up the parent node */

&PARENT = &NODES.ADD();

&PARENT.TEXT = "ROOT";

&PARENT.image = 1;

&PARENT.expandedimage = 2;

&INDEX = &PARENT.INDEX;

/* set up child nodes */

&CHILD = &NODES.ADD(&INDEX, 4);

&CHILD.TEXT = "Child1";

&CHILD.image = 1;

&CHILD.expandedimage = 2;

&CHILD = &NODES.ADD(&INDEX, 4);

&CHILD.TEXT = "Child2";

&CHILD.image = 1;

&CHILD.expandedimage = 2;

&INDEX = &CHILD.INDEX;

&CHILD = &NODES.ADD(&INDEX, 4);

&CHILD.TEXT = "GrandChild1";

&CHILD.image = 1;

&CHILD.expandedimage = 2;

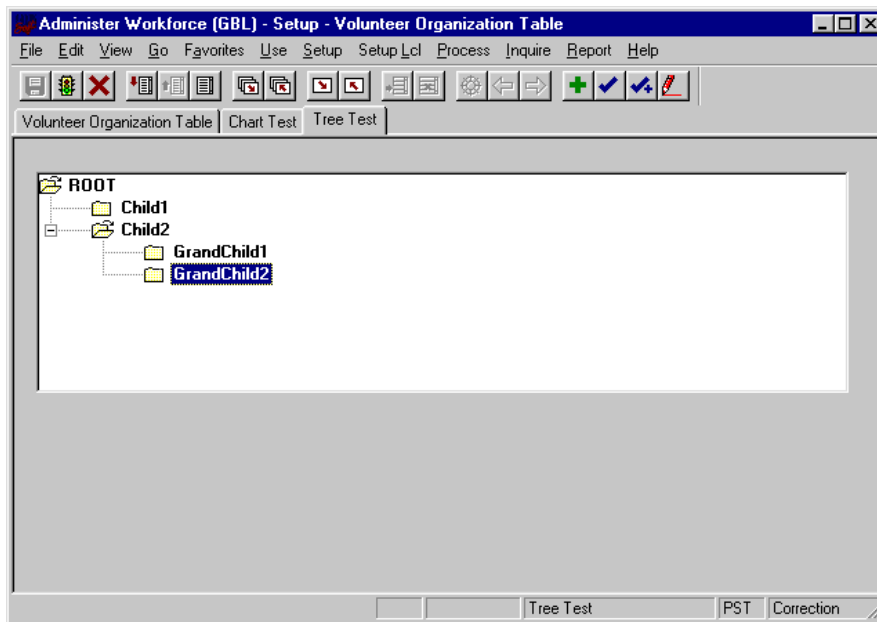
&CHILD = &NODES.ADD(&INDEX, 4);

&CHILD.TEXT = "GrandChild2";

&CHILD.image = 1;

&CHILD.expandedimage = 2;

End-Function;
```

TreeView on page produced by previous example code

TreeView and Grid

The following example uses PeopleCode to map data from a Grid to a TreeView control.

```
Function PSControlInit()

    &MAXCOUNT = ActiveRowCount(SCROLL.PC_TREEDEMO);      For &I = 1 To &MAXCOUNT
    &FVAL = FetchValue(SCROLL.PC_TREEDEMO, &I, PC_TREEDEMO.PC_RELATION_FLD);

        If &FVAL = "SLB" Then                                &PARENT = GetControl("PC_TREEDEMO_PNL2",
"ACTIVEX1").NODES.ADD();                                &PARENT.TEXT = FetchValue(SCROLL.PC_TREEDEMO,
&I, PC_TREEDEMO.FIELDNAME);                                &PARENT_NODE_IDX = &PARENT.INDEX;

            Else                If &FVAL = "CHLD" Then                &CHILD =
GetControl("PC_TREEDEMO_PNL2", "ACTIVEX1").NODES.ADD(&PARENT_NODE_IDX, 4);
&CHILD.TEXT = FetchValue(SCROLL.PC_TREEDEMO, &I, PC_TREEDEMO.FIELDNAME);
&CHILD_NODE_IDX = &CHILD.INDEX;                                &CHILD.EXPANDED = True;

                Else                &CHILD = GetControl("PC_TREEDEMO_PNL2",
"ACTIVEX1").NODES.ADD(&CHILD_NODE_IDX, 4);                                &CHILD.TEXT =
FetchValue(SCROLL.PC_TREEDEMO, &I, PC_TREEDEMO.FIELDNAME);
&CHILD.EXPANDED = True;                                End-If;

        End-If;

    End-For;    End-Function;
```

Chart, TreeView, and ImageList Cheat Sheets

The following tables list the Chart, Treeview, and ImageList properties and methods, and all the objects instantiated from these objects, plus their properties and methods.

A property marked with an asterisks (*) is the default property for that object.

Properties that are read-only are marked with RO (read-only). If a property isn't marked RO, it's read-write.

Chart Control Objects, Properties and Methods

<i>Object Name</i>	<i>Properties and Methods</i>	<i>Returns</i>
Chart	Properties	
	ActiveSeriesCount	Number, RO
	AllowDithering	Boolean
	AllowDynamicRotation	Boolean
	AllowSelections	Boolean
	AllowSeriesSelection	Boolean
	AutoIncrement	Boolean
	Backdrop	Backdrop Object, RO
	BorderStyle	Number
	Chart3d	Boolean, RO
	ChartType	Number
	Column	Number
	ColumnCount	Number
	ColumnLabel	String
	ColumnLabelCount	Number
	ColumnLabelIndex	Number
	Data	Number
	DataGrid	DataGrid object, RO
	DoSetCursor	Boolean
	DrawMode	Number
	Enabled	Boolean
	Footnote	Footnote Object, RO
	FootnoteText	String

	Legend	Legend Object, RO
	MousePointer	Number
	OLEDragMode	Number
	OLEDropMode	Number
	Plot	Plot object, RO
	RandomFill	Boolean
	Repaint	Boolean
	Row	Number
	RowCount	Number
	RowLabel	String
	RowLabelCount	Number
	RowLabelIndex	Number
	SeriesColumn	Number
	SeriesType	Number
	ShowLegend	Boolean
	Stacking	Boolean
	TextLengthType	Number
	Title	Title object
	TitleText	String
Chart	Methods	
	EditCopy()	
	EditPaste()	
	Layout()	
	OLEDrag()	
	Refresh()	
	SelectPart(<i>part, index1, index2, index3, index4</i>)	
	ToDefaults()	
Axis	Properties	
	AxisGrid	AxisGrid Object, RO
	AxisScale	AxisScale Object, RO
	AxisTitle	AxisTitle object, RO
	CategoryScale	CategoryScale object, RO

	Intersection	Intersection object, RO
	LabelLevelCount	Number, RO
	Labels	Labels object, RO
	Pen	Pen object, RO
	Tick	Tick object, RO
	ValueScale	ValueScale object, RO
AxisGrid	Properties	
	MinorPen	Pen Object, RO
	MajorPen	Pen object, RO
AxisScale	Properties	
	Hide	Boolean
	LogBase	Number
	PercentBasis	Number
	Type	Number
AxisTitle	Properties	
	Backdrop	Backdrop object, RO
	Font	Font object
	Text*	String
	TextLayout	TextLayout object, RO
	TextLength	Number, RO
	Visible	Boolean
	VtFont	VtFont object, RO
BackDrop	Properties	
	Fill	Fill Object, RO
	Frame	Frame object, RO
	Shadow	Shadow object, RO
Brush	Properties	
	FillColor	VtColor object, RO
	Index	Number
	PatternColor	VtColor object, RO
	Style	Number
CategoryScale	Properties	
	Auto	Boolean

	DivisionsPerLabel	Number
	DivisionsPerTick	Number
	LabelTick	Boolean
Coor	Properties	
	X	Number
	Y	Number
Coor	Method	
	Set(<i>X</i> , <i>Y</i>)	
DataGrid	Properties	
	ColumnCount	Number
	ColumnLabelCount	Number
	RowCount	Number
	RowLabelCount	Number
DataGrid	Special Property: can only be used with ObjectSetProperty built-in function.	
	RowLabel(<i>row</i> , <i>labelIndex</i>)	String
DataGrid	Methods	
	ColumnLabel(<i>Column</i> , <i>LabelIndex</i>)	
	CompositeColumnLabel(<i>Column</i>)	
	CompositeRowLabel(<i>Row</i>)	
	DeleteColumnLabels(<i>LabelIndex</i> , <i>Count</i>)	
	DeleteColumns(<i>Column</i> , <i>Count</i>)	
	DeleteRowLabels(<i>LabelIndex</i> , <i>Count</i>)	
	DeleteRows(<i>Row</i> , <i>Count</i>)	
	GetData(<i>Row</i> , <i>Column</i> , & <i>DataPoint</i> , <i>nullFlag</i>)	Variable
	InitializeLabels()	
	MoveData(<i>Top</i> , <i>Left</i> , <i>Bottom</i> , <i>Right</i> , <i>OverOffset</i> ,	

	<i>DownOffset)</i>	
	RandomDataFill()	
	RandomFillColumns(<i>Column</i> <i>, Count</i>)	
	RandomFillRows(<i>Row</i> , <i>Count</i>)	
	SetData(<i>Row</i> , <i>Column</i> , <i>DataPoint</i> , <i>nullFlag</i>)	
	SetSize(<i>RowLabelCount</i> , <i>ColumnLabelCount</i> , <i>DataRowCount</i> , <i>DataColumnCount</i>)	
DataPoint	Properties	
	Brush	Brush object, RO
	DataPointLabel	DataPointLabel object, RO
	EdgePen	Pen object, RO
	Marker	Marker object, RO
	Offset	Number
DataPoint	Methods	
	ResetCustom()	
	Select()	DataPoint
DataPointLabel	Properties	
	Backdrop	Backdrop object
	Component	Number, RO
	Custom	Boolean
	Font	Font object
	LineStyle	Number
	LocationType	Number
	Offset	LCoor object, RO
	PercentFormat	String
	Text*	String
	TextLayout	TextLayout object, RO
	TextLength	Number
	ValueFormat	Number
	VtFont	VtFont, RO

DataPointLabel	Methods	
	ResetCustomLabel()	
	Select()	DataPointLabel
DataPoints	Properties	
	Item(<i>Index</i>)	DataPoint object, RO
	Count()	Number, RO
Fill	Properties	
	Brush	Brush object, RO
	Style	Number
Font	Properties	
	Bold	Boolean
	Charset	Number
	Italic	Boolean
	Name	String
	Size	Number
	Strikethrough	Boolean
	Underline	Boolean
	Weight	Number
Footnote	Properties	
	Backdrop	Backdrop object, RO
	Font	Font object
	Location	Location object, RO
	Text*	String
	TextLayout	TextLayout object, RO
	TextLength	Number
	VtFont	VtFont object, RO
Footnote	Method	
	Select()	Footnote object
Frame	Properties	
	FrameColor	VtColor object, RO
	SpaceColor	VtColor object, RO
	Style	Number
	Width	Number

Intersection	Properties	
	Auto	Boolean
	AxisID	Number, RO
	Index	Number, RO
	LabelsInsidePlot	Boolean
	Point	Number
Label	Properties	
	Auto	Boolean
	Backdrop	Backdrop object, RO
	Font	Font object
	Format	Number
	FormatLength	String, RO
	Standing	Boolean
	TextLayout	TextLayout object, RO
	VtFont	VtFont object, RO
Labels (collection)	Properties	
	Count	Number, RO
	Item(<i>Item</i>)	Label object, RO
LCoor	Properties	
	X	Number
	Y	Number
LCoor	Method	
	Set(<i>X</i> , <i>Y</i>)	
Legend	Properties	
	Backdrop	Backdrop object, RO
	Font	Font object
	Location	Location object, RO
	TextLayout	TextLayout object, RO
	VtFont	VtFont object, RO
Legend	Method	
	Select()	Legend object
Light	Properties	
	AmbientIntensity	Number

	EdgeIntensity	Number
	EdgeVisible	Boolean
	LightSources	LightSources object, RO
Lightsource	Properties	
	Intensity	Number
	X	Number
	Y	Number
	Z	Number
Lightsource	Method	
	Set(<i>X, Y, Z, Intensity</i>)	
Lightsources (collection)	Properties	
	Count	Number, RO
	Item(<i>Index</i>)	LightSource object, RO
Lightsources (collection)	Methods	
	Add(<i>X, Y, Z, Intensity</i>);	
	Remove(<i>Index</i>)	
Location	Properties	
	LocationType	Number
	Rect	Rect object, RO
	Visible	Boolean
Marker	Properties	
	FillColor	VtColor object, RO
	Pen	Pen object, RO
	Size	Number
	Style	Number, RO
	Visible	Boolean
Pen	Properties	
	Cap	Number
	Join	Number
	Limit	Number
	Style	Number
	VtColor	VtColor object, RO
	Width	Number

Plot	Properties	
	AngleUnit	Number
	AutoLayout	Boolean
	Backdrop	Backdrop object, RO
	BarGap	Number
	Clockwise	Boolean
	DataSeriesInRow	Boolean
	DefaultPercentBasis	Number, RO
	DepthToHeightRatio	Number
	Light	Light object, RO
	LocationRect	Rect object, RO
	PlotBase	Plotbase object, RO
	Projection	Number
	SeriesCollection	SeriesCollection object, RO
	Sort	Number
	StartingAngle	Number
	SubPlotLabelPosition	Number
	UniformAxis	Boolean
	View3d	View3D object, RO
	Wall	Wall object, RO
	Weighting	Weighting object, RO
	WidthToHeightRatio	Number
	XGap	Number
	ZGap	Number
Plot	Property (Additional)	
	Axis(<i>axisID</i> , [<i>&Index</i>]);	Axis object
Plotbase	Properties	
	BaseHeight	Number
	Brush	Brush object, RO
	Pen	Pen object, RO
Rect	Properties	
	Min	Coor object, RO
	Max	Coor object, RO

Series	Properteis	
	DataPoints	Datapoints Collection, RO
	GuideLinePen	Pen object, RO
	LegendText	String
	Pen	Pen object, RO
	Position	SeriesPostiion object, RO
	SecondaryAxis	Boolean
	SeriesMarker	SeriesMarker object, RO
	SeriesType	SeriesType object
	ShowLine	Boolean
	StatLine	StatLine object, RO
Series	Special Properties: can only be used with ObjectSetProperty built-in function.	
	ShowGuideLine([<i>axisID</i>], [, <i>Index</i>]) = [<i>boolean</i>]	Boolean
	TypeByChartType(<i>chtype</i>) = [<i>SeriesType</i>]	Number
Series	Method	
	Select()	Series object, RO
SeriesCollection	Property	
	Item(<i>Item</i>)	Series Object, RO
SeriesCollection	Method	
	Count	Number
SeriesMarker	Properties	
	Auto	Boolean
	Show	Boolean
SeriesPosition	Properties	
	Excluded	Boolean
	Hidden	Boolean
	Order	Number
	StackOrder	Number
Shadow	Properties	

	Brush	Brush object, RO
	Offset	Number, RO
	Style	Number
StatLine	Properties	
	Flag	Number
	VtColor	VtColor Object, RO
	Width	Number
StatLine	Special Property: can only be used with ObjectSetProperty built-in function.	
	Style(<i>type</i>) = [<i>style</i>]	Number
TextLayout	Properties	
	HorzAlignment	Number
	Orientation	Number
	VertAlignment	Number
	WordWrap	Boolean
Tick	Properties	
	Length	Number
	Style	Number
Title	Properties	
	Backdrop	Backdrop object, RO
	Font	Font object
	Location	Location object, RO
	Text*	String
	TextLayout	TextLayout object, RO
	TextLength	Number
	VtFont	VtFont object, RO
Title	Method	
	Select()	Title object
ValueScale	Properties	
	Auto	Boolean
	MajorDivision	Number
	Maximum	Number

	Minimum	Number
	MinorDivision	Number
View3D	Properties	
	Elevation	Number
	Rotation	Number
View3d	Method	
	<i>Set(Rotation, Elevation)</i>	
VtColor	Properties	
	Automatic	Boolean
	Blue	Number
	Green	Number
	Red	Number
VtColor	Method	
	<i>Set(Red, Green, Blue)</i>	
VtFont	Properties	
	Effect	Number
	Name	String
	Size	Number
	Style	Number
	VtColor	VtColor object, RO
Wall	Properties	
	Brush	Brush object, RO
	Pen	Pen object, RO
	Width	Number
Weighting	Properties	
	Basis	Number
	Style	Number
Weighting	Method	
	<i>Set(Basis, Style)</i>	

TreeView Objects, Properties and Methods

Object Name	Properties and Methods	Returns
TreeView	Properties	
	Appearance	Number
	BorderStyle	Number
	Checkboxes	Boolean
	DropHighlight	Node object, RO
	Enabled	Boolean
	Font	Font object, RO
	FullRowSelect	Boolean
	HideSelection	Boolean
	HotTracking	Boolean
	ImageList;	ImageList object
	Indentation	Number
	LabelEdit	Number
	LineStyle	Number
	MousePointer	Number
	Nodes	Nodes object, RO
	OLEDragMode	Number
	OLEDropMode	Number
	PathSeparator	String
	Scroll	Boolean
	SelectedItem	Node object, RO
	SingleSel	Boolean
	Sorted	Boolean
	Style	Number
TreeView	Methods	
	GetVisibleCount()	Number
	HitTest(<i>X</i> , <i>Y</i>)	Node Object
	StartLabelEdit()	
Font	Properties	
	Bold	Boolean

	Charset	Number
	Italic	Boolean
	Name	String
	Size	Number
	Strikethrough	Boolean
	Underline	Boolean
	Weight	Number
Node	Properties	
	Bold	Boolean
	Checked	Boolean
	Child	Node object, RO
	Children	Node object, RO
	Expanded	Boolean
	ExpandedImage	Number
	FirstSibling	Node object, RO
	FullPath	String
	Image	Number
	Index	Number
	Key	String
	LastSibling	Node object, RO
	Next	Node object, RO
	Parent	Node object, RO
	Previous	Node object, RO
	Root	Node object, RO
	Selected	Boolean
	SelectedImage	Number
	Sorted	Boolean
	Tag	String
	Text*	String
	Visible	Boolean
Node	Methods	
	CreateDragImage()	Picture Object
	EnsureVisible()	Boolean

Nodes (collection)	Properties	
	Count	Number
Nodes (collection)	Methods	
	Add([<i>relative</i> ,] [<i>relationship</i> ,] [key,] [text,] [<i>image</i> ,] [<i>selectedimage</i>])	Node object
	Clear()	
	Item(<i>Index</i>)	Node object
	Remove (<i>Index</i>)	
Picture	Properties	
	Height	Number
	Type	Number
	Width	Number

ImageList Objects, Properties and Methods

Object Name	Properties and Methods	Returns
ImageList	Properties	
	ImageHeight	Number, RO
	ImageWidth	Number, RO
	ListImages	Images object
	MaskColor	Number
	UseMaskColor	Boolean
ImageList	Method	
	Overlay(<i>index1</i> , <i>index2</i>)	
Images (collection)	Properties	
	Count	Number
Images (collection)	Methods	
	Add([<i>index</i> ,] [key,] <i>Picture</i>)	Image object
	Clear()	
	Item(<i>Index</i>)	Image object
	Remove (<i>Index</i>)	

Image	Properties	
	Index	Number
	Key	String
	Picture	Picture object, RO
	Tag	String
Image	Methods	
	ExtractIcon()	Picture object
	Index(<i>Index</i>)	Image object

CHAPTER 10

Meta-SQL

Meta-SQL functions expand to platform-specific SQL substrings. They are used in functions that pass SQL strings, that is,

- SQLExec
- the scroll buffer functions (ScrollSelect and its relatives)
- in Application Designer to construct dynamic views
- with some Rowset Class methods (Select, SelectNew, Fill, and so on.)
- with SQL Class
- in Application Engine
- with some ProcessRequest Class methods (Insert, Update, and so on.)
- with COBOL

Date Considerations

You can avoid confusion when using meta-SQL such as %Datein and %Dateout if you keep in mind that the "in" functions are used in the WHERE subclause of a SQL query and "out" functions are used in the SELECT (main) clause of the query. For example:

```
select emplid, %dateout(effdt) from ps_car_alloc a where car_id = ' ' |
&REGISTRATION_NO | ' ' and plan_type = ' ' | &PLAN_TYPE | ' ' and a.effdt = (select
max (b.effdt) from ps_car_alloc b where a.emplid=b.emplid and b.effdt <=
%currentdatein) and start_dt <= %currentdatein and (end_dt is null or end_dt >=
%currentdatein)";
```

Use of Date, Datetime, or Time Wrappers with an Application Engine Program

It has always been the expectation in PeopleCode that you should use the date or time wrappers (%Datein, %TimeOut, and so on) when selecting date or time columns into memory. The reason for this is simple: different database platforms use different internal formats for these data types. Those different formats range from 1900-01-01 to 01-JAN-1900. DateTime (timestamp) formats are an even bigger deal.

In PeopleCode (SQLExecs and the like), the developer must use both an "Out" wrapper when selecting a date/time value into memory, as well as an "In" wrapper when referencing the value as a bind variable.

In an Application Engine program, things are different. When you populate a date/time state field in a %Select, you still have to use an "Out" wrapper to get the value into the standard format. But when you reference this state field in a %Bind, Application Engine automatically provides the "In" wrapper around the substituted literal or bind marker (the latter if reuse is in effect).

Actually, Application Engine does a bit more: if you use the %Bind(date) in the select list of another %Select statement, to load the value into another date field, Application Engine doesn't provide any wrapper (since you are selecting a value that is already in the standard format, you don't need to use any wrapper.)

Date and Time Out Wrappers for Static and Dynamic Views

Dynamic views containing date, time, or datetime fields *must* be wrapped with the appropriate Meta-SQL. PeopleTools uses the SQL directly from the view definition (View Text) and doesn't generate anything, so no Meta-SQL wrapping is done.

For a normal view (that is, a static view) PeopleTools builds the SELECT statement and subsequently wraps Date, Time and DateTime columns with the appropriate Meta-SQL. Normal Views should *not* contain Meta-SQL (otherwise the system adds the Meta-SQL to a field that has already been passed through Meta-SQL and the statement fails).

Using {DateTimein-prefix} in SQR

In SQR, if you are using {DateTimein-prefix} , and so on, you need to do the following:

- For string or let statements when using dynamic SQL, you need to use the following:

```
{DYN-Date***in/out-prefix/suffix}
```

- For SQL statements, you need to use the regular SQL, as follows:

```
{Date*** in/out-prefix/suffix}
```

Meta-SQL Placement Considerations

Not all meta-SQL constructs can be used by all programs. Some meta-SQL constructs can be used in Application Engine. Other constructs can only be used as part of a SQL statement in a record view. The following table contains a list of all the available meta-SQL constructs. An X in the appropriate column means that you can use that construct in that type of program.

If a meta-SQL construct is supported in PeopleCode, it is supported in all types of PeopleCode programs, that is, in Application Engine PeopleCode programs (actions), Application Message PeopleCode programs, and so on.



"Used in PeopleCode" does *not* mean that you can use a meta-SQL statement like a function. It means that you can use the meta-SQL statement in **SQLExec**, the **Select** method, the **Fill** method, and so on.



You should only place meta-SQL in a Dynamic View, not in a SQL view.

The meta-SQL constructs that are only available for Application Engine are fully described in Application Engine Meta-SQL.

Name	Used in PeopleCode	Used in Application Engine	Used in COBOL	Used in Dynamic View
%Abs	X	X		X
%Bind		X		
%ClearCursor		X		
%Abs	X	X		X
%CurrentDateIn	X	X	X	X
%CurrentDateOut	X	X	X	X
%CurrentDateTimeIn	X	X	X	X
%CurrentDateTimeOut	X	X	X	X
%CurrentTimeIn	X	X	X	X
%CurrentTimeOut	X	X	X	X
%DateAdd	X	X	X	X
%DateDiff	X	X	X	X
%DateIn	X	X	X	X
%DateOut	X	X	X	X
%DateTimeDiff	X	X	X	X
%DateTimeIn	X	X	X	X
%DateTimeOut	X	X	X	X
%DecDiv	X	X	X	X
%DecMult	X	X	X	X
%Delete(:num)	X			
%DTTM	X	X		X
%Execute		X		

%ExecuteEdits		X		
%FirstRows	X	X		
%Insert	X			
%InsertSelect	X	X		X
%InsertValues	X			X
%Join	X	X		X
%KeyEqual	X			X
%KeyEqualNoEffDt	X			X
%Like	X	X		X
%LikeExact	X	X		X
%List		X		
%ListBind		X		
%ListEqual		X		
%Next and %Previous		X		
%OldKeyEqual	X			X
%OPRCLAUSE				X
%Round	X	X	X	X
%RoundCurrency		X		
%Select		X		
%SelectAll	X			
%SelectByKey	X			
%SelectByKeyEffDt	X			
%SelectDistinct	X			
%SelectInit		X		
%SQL	X	X		X
%Substring	X	X	X	X
%SUBREC				X
%Table	X	X		X
%TextIn	X	X		X
%TimeIn	X	X	X	X
%TimeOut	X	X	X	X
%TrimSubstr	X	X	X	X
%Truncate	X	X	X	X

%TruncateTable	X	X	X	X
%Update	X			
%UpdatePairs	X			X
%UpdateStats		X		
%Upper	X	X		X

Meta-SQL Reference

Throughout this section, we use typographical conventions to distinguish between different elements of the PeopleCode language, such as bold to indicate function names, italics for arguments, and so on.



For more information, see *Typographical Conventions and Visual Cues*.



The parameter *rename* refers to a *record* name, *not a table name*. If you specify a table name (for example, PS_ST_OPTION_PARMS) you'll receive a SQL error. You should use the record name (for example, ST_OPTION_PARMS) instead. Also, don't use quotation marks around a record name.

%Abs

Syntax

%Abs (*x*)

Description

%Abs returns a decimal value equal to the absolute value of a number *x*.



This meta-SQL is not implemented for COBOL.

Example

```
SELECT INVENTORY_CODE FROM INVENTORY_TABLE WHERE %ABS (NEW_AMOUNT - OLD_AMOUNT) >
SOME_ALLOWED_VALUE
```

%Concat

Syntax

string1 %Concat *string2*

Description

At runtime, the **%Concat** function will be replaced by the string concatenation operator appropriate for the RDBMS being executed on. For example, on DB2, the %Concat function is replaced with CONCAT, while on Sybase it's replaced with a "+".

This function is supported with the same limitations as the native concatenation operator on the RDBMS the function is being executed on. For example, some platforms will allow you to concatenate a string with a numeric value; others will flag this as an error. PeopleTools makes no attempt to check or convert the data types of either of the operands.



This meta-SQL is not implemented for COBOL.

Examples

```
SELECT 'A' %Concat 'B' FROM PS_INSTALLATION. . .
```

```
SELECT LAST_NAME %Concat ', ' %Concat FIRST_NAME FROM PS_EMPLOYEE
```

%CurrentDateIn

%CurrentDateIn expands to a platform-specific SQL substring representing the current date in the WHERE clause of a SQL SELECT or UPDATE statement, or when the current date is passed in an INSERT statement.

%CurrentDateOut

%CurrentDateOut expands to platform-specific SQL for the current date in the SELECT clause of an SQL query.

%CurrentDateTimeIn

%CurrentDateTimeIn expands to a platform-specific SQL substring representing the current datetime in the WHERE clause of a SQL SELECT or UPDATE statement, or when the current date time is passed in an INSERT statement.

%CurrentDateTimeOut

%CurrentDateTimeOut expands to platform-specific SQL for the current datetime in the SELECT clause of an SQL query.

%CurrentTimeIn

%CurrentTimeIn expands to a platform-specific SQL substring representing the current time in the WHERE clause of a SQL SELECT or UPDATE statement, or when the current time is passed in an INSERT statement.

%CurrentTimeOut

%CurrentTimeOut expands to platform-specific SQL for the current time in the SELECT clause of an SQL query.

%DateAdd

Syntax

```
%DateAdd(date_from, add_days)
```

Description

%DateAdd returns a date by adding *add_days* to *date_from*. *add_days* can be negative.

Examples

```
&ANSWER = %DateAdd(%DateIn('1997-02-10'), 5);

(&ANSWER = 1997-02-15)

&ANSWER = %DateAdd(%DateIn('1997-02-10'), -5);

(&ANSWER = 1997-02-05)

%DateAdd(:1, :2) (:1 and :2 are bind variables in SQLExec PeopleCode)
```

%DateDiff

Syntax

```
%DateDiff(date_from, date_to)
```

Description

%DateDiff returns an integer representing the difference between two dates in number of days.

Examples

```
%DateDiff(%DateIn('1997-01-01'), %DateIn('1966-06-30'))
```

```
%DateDiff( date1_column, date2_column)
```

```
%DateDiff ( %DateAdd(date1_column, 30), date2_column)
```

The following usage is **illegal** (always use %DateIn for inputting date literals).

```
%DateDiff('1997-01-01', '1996-06-30') (should use %DateIn for inputting date literals)
```

%DateIn

%DateIn(*dt*), where *dt* is either a Date value or a date literal in YYYY-MM-DD format, expands into platform-specific SQL syntax for the date. %DateIn should be used whenever a date literal or Date bind variable is used in a comparison in the WHERE clause of a SELECT or UPDATE statement, or when a Date value is passed in an INSERT statement.

Restrictions using COBOL

You can only use string literals when using this function in COBOL. You can *not* use it with bind parameters in COBOL. For example, the following will work in COBOL:

```
UPDATE PS_PERSONAL_DATA SET LASTUPDT = %DATEIN('2002-12-11')
```

Whereas the following SQL will fail:

```
UPDATE PS_PERSONAL_DATA SET LASTUPDT = %DATEIN(:1)
```

%DateOut

%DateOut(*dt*), where *dt* is a date column, expands to a platform-specific SQL substring representing *dt* in the SELECT clause of an SQL query.

%DateTimeDiff

Syntax

```
%DateTimeDiff(datetime_from, datetime_to)
```

Description

`%DateTimeDiff` returns a time value, representing the difference between two datetimes in minutes.

Example

The following example returns the difference in hours between the current datetime and the requested date time:

```
%DateTimeDiff(%CurrentDateIn, RQSTDTTM) < " | RECORD.FIELDNAME * 60;
```

The following example returns the difference in minutes:

```
%DateTimeDiff(%CurrentDateIn, RQSTDTTM) < " | RECORD.FIELDNAME;
```

%DateTimeIn

`%DateTimeIn(dti)` expands to platform-specific SQL for a DateTime value in the WHERE clause of a SQL SELECT or UPDATE statement, or when a datetime value is passed in an INSERT statement. The parameter *dti* is either a Datetime bind variable or a String literal in the form:

```
YYYY-MM-DD-hh.mm.ss.ssssss
```

Restrictions using COBOL

You can only use string literals when using this function in COBOL. You can *not* use it with bind parameters in COBOL. For example, the following will work in COBOL:

```
UPDATE PS_PERSONAL_DATA SET LASTUPDTTM = %DATETIMEIN('2002-12-11
11:59:00:000000')
```

Whereas the following SQL will fail:

```
UPDATE PS_PERSONAL_DATA SET LASTUPDTTM = %DATETIMEIN(:1)
```

%DateTimeOut

`%DateTimeOut(datetime_col)`, where *datetime_col* is a Datetime column, expands to a platform-specific SQL substring representing *datetime_col* in the SELECT clause of an SQL query.

%DecDiv

Syntax

```
%DecDiv(a,b)
```

Description

%DecDiv returns an integer representing the value of a divided by b, where a and b are numeric expressions.

It is recommended to use this function on the INSERT .. SELECT only.

If the result needs to be picked up by a bind variable, it is recommended to pick it up using character type or PIC X(50).

Parameters

a, b	Numeric expressions.
--------	----------------------

Example

```
%DecDiv(1000.0, :1)
```

where :1 is a bind variable in SQLExec PeopleCode.

%DecMult

Syntax

$$\% \text{DecMult}(a, b)$$

Description

%DecMult returns an integer representing a multiplied by b , where a and b are numeric expressions.

It is recommended to use this function on the INSERT .. SELECT only.

If the result needs to be picked up by a bind variable, it is recommended to pick it up using character type or PIC X(50).

Parameters

a, b	Numeric expressions.
--------	----------------------

Example

```
%DecMult(12.3, 34.67)
```

```
%DecMult (c1 + c2, c3)
```

where c1, c2 and c3 are fields of NUMBER datatype.

%DTTM

Syntax

```
%DTTM(date, time)
```

Description

%DTTM combines the database date in the *date* value with the database time in the *time* value and returns a database timestamp value.



This meta-SQL is not implemented for COBOL.

Example

```
INSERT INTO TABLE1 (TIMESTAMP) SELECT %DTTM(DATE,TIME) FROM TABLE2
```

%EffDtCheck

Syntax

```
%EffDtCheck(recordname [correlation_id1], correlation_id2, as_of_date)
```

Description

The **%EffDtCheck** function expands into an effective date sub-query suitable for a WHERE clause. The value for *as_of_date* will automatically be wrapped in %Datein() unless *as_of_date* is already wrapped in %Datein() or refers to other database columns.



This meta-SQL is not implemented for COBOL.

%EffDtCheck only works with effective dates. It does *not* take effective sequence numbers (EFFSEQ) into account. It also doesn't do any effective status (EFF_STATUS) checking.

Parameters

recordname

Specify the record name to be used as the record in the effective date checking. This can be a bind variable, a record object, or a record name in the form *recname*. You *can't* specify a RECORD.*recname*, a record name in quotes, or a table name.



If you specify a bind variable, it should refer to a record object, *not* a string variable.

<i>correlation_id1</i>	Specify the letter used inside the effective dating subselect. This is an optional parameter. If it isn't specified, <i>recordname</i> is used.
<i>correlation_id2</i>	Specify the letter already assigned to the main record in the FROM clause of the SQL statement.
<i>as_of_date</i>	Specify the date to be used in the effective date. This can be a bind variable, a variable or a hard-coded date. The value for <i>as_of_date</i> will automatically be wrapped in <code>%Datein()</code> .

Example

The following is a generic code sample:

```
SELECT. . .

FROM. . .

WHERE %EffDtCheck(recordname, correlation_id, as_of_date)
```

resolves into the following:

```
SELECT . . .

FROM. . .

WHERE correlation_id.EFFDT = (SELECT MAX(EFFDT) FROM recordname

WHERE recordname.KEYFIELD1 = correlation_id.KEYFIELD1

AND recordname.KEYFIELD2 = correlation_id.KEYFIELD2

AND. . .

AND recordname.EFFDT <= %DATEIN(as_of_date))
```

In the following example, `&Date` has the value of 01/02/1998. And the example `&Rec` object has an EFFDT key field. The following code sample:

```
SQLExec("SELECT FNUM FROM PS_REC A WHERE %EffDtCheck(:1, A, :2)", &Rec, &Date);
```

resolves into the following:

```
"Select FNUM from PS_REC A where EFFDT = (select MAX(EFFDT)

from PS_REC

where PS_REC.FNUM = A.FNUM
```

```
and PS_REC.EFFDT <= %DateIn('1998-01-02') )"
```

The following example uses correlation ids. The following code sample:

```
SELECT A.DEPTID

FROM %Table(DEPT_TBL) A

WHERE

%EffDtCheck(DEPT_TBL B, A, %CurrentDateIn)

AND A.EFF_STATUS = 'A'
```

resolves into the following:

```
SELECT A.DEPTID

FROM %Table(DEPT_TBL) A

WHERE

A.EFFDT =

(SELECT MAX(B.EFFDT)

FROM DEPT_TBL B

WHERE

A.SETID = B.SETID

AND A.DEPTID = B.DEPTID

AND B.EFFDT <=%CurrentDateIn)

AND A.EFF_STATUS = 'A'
```

%FirstRows

Syntax

```
%FirstRows (n)
```

Description

The **%FirstRows** meta-SQL is replaced by the database specific SQL syntax to optimize retrieval of *n* rows. Depending on the database, it will optimize:

- the query path
- the number of rows returned
- the number of rows returned per fetch buffer

Considerations using %FirstRows

- This function is *not* implemented for COBOL or Dynamic View SQL.
- This function is *not* implemented for Sybase. It will do nothing on this platform.
- This function *must not be used* if there's a chance that the application requires more than *n* rows fetched. The results of fetching more than *n* rows varies by platform. Some return the extra rows, but performance may be sub-optimal. Others return "ROW NOT FOUND".
- This function must only be between the SELECT that begins the SQL statement and the SELECT LIST. It must *not* be used in subqueries, views, INSERT/SELECTS, and so on. The SELECT list must *not* be *.
- This function must not be used with DISTINCT, because SELECT TOP 1 DISTINCT will fail on SqlServer.
- This function is implicitly embedded in all SELECT SqlExecs for all platforms except ORACLE.

Parameters

n Specify the number of rows to be returned.

Example

The following is checking for the existence of a row:

```
&SQL = CreateSQL("select %firstrows(1) 'x' from PS_EXAMPLE where COL1 = :1",
&temp);
```

The following populates a 10 element array:

```
&SQL = CreateSQL("select %firstrows(10) COL2, COL3 from PS_EXAMPLE_VW where COL1
= :1", &temp);
```

%InsertSelect

Syntax

```
%InsertSelect([DISTINCT, ]insert_recname, select_recname [ correlation_id] [,
select_recname_n [ correlation_id_n]] [, override_field = value] . . .)
```

Description

The **%InsertSelect** function will generate an INSERT statement with a SELECT for you. It does *not* generate the FROM statement. You must specify all the select records before you specify any over ride fields.

The INSERT column list is composed of all the fields in the specified *insert_recname*, with the exception of LONG_CHAR or IMAGE fields.



Because of the way longs (long character and image fields) are handled in the various database platforms for INSERT statements, all longs in the *insert_recname* will be skipped in the generated INSERT statement. This implies that these fields should be defined in such a manner as to allow NULLs.

The corresponding value in the SELECT list will be generated based on the following precedence:

1. If the INSERT fieldname appears as an *override_field*, the corresponding *value* will be used in the SELECT list.
2. If the INSERT fieldname matches a fieldname in one of the *select_recname* specified, the corresponding SELECT field will be used in the SELECT list.
3. The search order of the *select_recname* records is the order that they are specified in the %InsertSelect.
4. If the INSERT fieldname has a constant default value defined in Application Designer, that value will be used in the SELECT list.
5. A default value appropriate for the data type of the INSERT field will be used (blank for character, zero for numerics, NULL for date/time/datetime values.)

The optional *override_field* is used to specify values for a particular field. For each field you specify, the matching logic described above will not be performed. Instead, the value you specify after the equal sign will be used for that field in the actual SELECT list. Use this technique when you want to let PeopleTools or Application Engine handle most of the fields in the record, but need to specify some of them explicitly. Also, you can use *override_field* to specify aggregate functions like sum(), max(), etc.



This meta-SQL is not implemented for COBOL.

Parameters

DISTINCT

Optionally specify if the select statement being generated will contain the word DISTINCT.

insert_recname

Specifies the name of record being inserted into. You must specify a record name, *not* a RECORD.*recname*, a record name in quotes, a bind variable or a table name.



If the record for *insert_recname* is a temporary table, %InsertSelect will automatically substitute the corresponding table instance (PS_TARGET*nn* instead of PS_TARGET).

<i>select_recname</i>	Specifies the name of record being selected from. You can specify more than one record to be selected from. You must specify a record name, not a RECORD. <i>recname</i> , a record name in quotes, or a table name.
<i>correlation_id</i>	Identifies the correlation ID to be used for the <i>select_recname</i> records and fields.
<i>override_field</i>	Specify the name of a field on <i>insert_recname</i> that you want to supply a value for (instead of using the value supplied from the <i>select_recname</i> .)
<i>Value</i>	Specify the value that should be used for the <i>override_field</i> instead of the value from <i>select_recname</i> .

Example

The following code:

```
%InsertSelect(AE_SECTION_TBL, AE_STEP_TBL S, AE_SECTION_TYPE = ' ')
FROM PS_AE_STEP_TBL S, PS_AS_STMT_TBL T
WHERE. . .
```

will resolve into the following:

```
INSERT INTO PS_AE_SECTION_TBL (AE_APPLID, AE_SECTION,..., AE_SECTION_TYPE)
SELECT S.AE_APPL_ID, S.AE_SECTION, ... ' '
FROM PS_AE_STEP_TBL S, PS_AS_STMT_TBL T
WHERE. . .
```

Another example would be the case where you have a temporary table, PS_MY_TEMP, which is based on a join between two of your other tables, PS_MY_TABLE1 and PS_MY_TABLE2.

```
%InsertSelect(MY_TEMP, MY_TABLE1, MY_TABLE2 T2)
FROM PS_MY_TABLE1 T1, PS_MY_TABLE2 T2
WHERE %Join(COMMON_KEYS, MY_TABLE1 T1, MY_TABLE2 T2) . . .
```

This code will resolve into:

```
INSERT INTO PS_MY_TEMP (FIELD1, FIELD2 . . .)
SELECT T2.FIELD1, T2.FIELD2, . . .
FROM PS_MY_TABLE1 T1, PS_MYTABLE2 T2
WHERE T1.FIELD1 = T2.FIELD1
```

```
AND T1.FIELD2 = T2.FIELD2 . . .
```

The following example assigns a SQL statement to a variable:

```
Function insert_draft_type(&RECNAME)

&SQL = "%InsertSelect(" | &RECNAME | " ," | &RECNAME | ", SETID = :2, DRAFT_TYPE
= :3) FROM %Table(:1) WHERE " | FIELD.SETID | " = :4 and " | FIELD.DRAFT_TYPE |
" =:5";

SQLExec(&SQL, @("RECORD." | &RECNAME), SETID, DRAFT_TYPE,
DRAFT_TYPE_TABLE.SETID, DRAFT_TYPE_TBL.DRAFT_TYPE);

End-Function;
```

The following example will create a distinct select.

```
%InsertSelect(DISTINCT, MY_TABLE, TABLE1, TABLE2 T2)

FROM PS_TABLE1 T1, PS_TABLE2 T2

WHERE %Join(COMMON_KEYS, TABLE1 T1, TABLE2 T2) . . .
```

This code will resolve into:

```
INSERT INTO PS_MYTABLE (FIELD1, FIELD2 . . .)

SELECT DISTINCT T2.FIELD1, T2.FIELD2, . . .

FROM PS_TABLE1 T1, PS_TABLE2 T2

WHERE T1.FIELD1 = T2.FIELD1

AND T1.FIELD2 = T2.FIELD2 . . .
```

%InsertValues

Syntax

```
%InsertValues(recname)
```

Description

The **%InsertValues** function produces a comma separated list of the record's nonnull field values. Input processing is applied to the fields, that is:

- if the field is a date, a time, or a datetime, its value will automatically be wrapped in %DateIn(), %TimeIn(), or %DateTimeIn(), respectively
- if the field is a string, its value will automatically be wrapped in quotes
- if the field has a null value, it will not be included in the list



This meta-SQL can only be used in PeopleCode programs, not in Application Engine SQL actions. Also, this meta-SQL is not implemented for COBOL.

Parameters

recname

Specify the name of the record to be used for inserting. This can be a bind variable, a record object, or a record name in the form *recname*. You *can't* specify a `RECORD.recname`, a record name in quotes, or a table name.

Example

The following code:

```
SQLExec("Insert into TABLE (%List(NonNull_Fields, :1)) values
(%InsertValues(:1))", &Rec);
```

will be expanded into:

```
"Insert into TABLE (FNUM, FCHAR, FDATE) values (27, 'Y', %datein('1989-11-27'))"
```

%Join

Syntax

```
%Join({COMMON_KEYS | COMMON_FIELDS}, join_recname
[ correlation_id1], to_recname [ correlation_id2]
[, override_field_list])
```

where *override_field_list* is an arbitrary-length list of fields to be substituted in the resulting text string, in the form:

```
field1 [, field2] . . .
```

Description

Use the **%Join** function to dynamically build a WHERE clause joining one table to another. At runtime, the entire function will be replaced with a character string.



This meta-SQL is not implemented for COBOL.

Parameters

COMMON_KEYS	Specifies that all common primary key fields will be used in constructing a WHERE clause. You can select either COMMON_KEYS or COMMON_FIELDS.
COMMON_FIELDS	Specifies that all common fields will be used in constructing a WHERE clause. You can select either COMMON_KEYS or COMMON_FIELDS.
<i>join_recname</i>	Specifies the name of the record to be joined. This can be a bind variable, a record object, or a record name in the form <i>recname</i> . You <i>can't</i> specify a RECORD. <i>recname</i> , a record name in quotes, or a table name.
<i>correlation_id1</i>	Identifies the correlation ID to be used to relate the record specified by <i>join_recname</i> and its fields.
<i>to_recname</i>	Specifies the name of the record to be joined to. This can be a bind variable, a record object, or a record name in the form <i>recname</i> . You <i>can't</i> specify a RECORD. <i>recname</i> , a record name in quotes, or a table name.
<i>correlation_id2</i>	Identifies the correlation ID to be used to relate the record specified by <i>to_recname</i> and its fields.
<i>override_field_list</i>	Specifies a list of any fields that you do not want used in the join. For example, if fields A, B, and C were common to two records, and you didn't want to join on C, list C as an <i>override_field</i> .

Examples

The following code:

```
%Join(COMMON_KEYS, PSAESECTDEFN ABC, PSAESTEPDEFN XYZ)
```

will result in the following being generated:

```
ABC.AE_APPLID = XYZ.AE_APPLID
AND ABC.AE_SECTION = XYZ.AE_SECTION
AND ABC.DBTYPE = XYZ.DBTYPE
AND ABC.EFFDT = XYZ.EFFDT
```

The following code:

```
%Join(COMMON_FIELDS, PSAEAPPLDEFN ABC, PSAESECTDEFN XYZ)
```

will result in the following being generated:

```
ABC.AE_APPLID = XYZ.AE_APPLID
```

```
AND ABC.DESCR = XYZ.DESCR
```

However, you don't want to join using the DESCR. So, use the *override_field*, as shown in the following code:

```
%Join(COMMON_FIELDS, PSAEAPPLDEFN ABC, PSAESECTDEFN XYZ, DESCR)
```

to get:

```
ABC.AE_APPLID = XYZ.AE_APPLID
```

You can also specify a value for a field. Suppose you want to join two tables, but not on the field C3. In addition, you would like to specify a value for C3. Your code could look like the following:

```
%Join(COMMON_FIELDS, MY_TABLE1 A, MY_TABLE2 B, C3) AND C3 = 'XX'
```

%KeyEqual

Syntax

```
%KeyEqual(recname [ correlation_id] )
```

Description

The **%KeyEqual** function expands into a conditional phrase suitable for use in a WHERE clause.

The conditional phrase consists of a conjunction (AND) of *[correlation_id].keyfieldname = 'keyfieldvalue'* phrases for each key field of the given record.

No autoupdate processing is done, but other input processing is applied to the values. That is:

- if the field is a date, a time, or a datetime, its value will automatically be wrapped in %DateIn(), %TimeIn(), or %DateTimeIn(), respectively.
- if a value is a string, its value will automatically be wrapped in quotes.
- if a value is NULL, the "*=value*" part is replaced with "IS NULL".



This meta-SQL can only be used in PeopleCode programs, not in Application Engine PeopleCode actions. Also, this meta-SQL is not implemented for COBOL.

Parameters

recname

Specify the name of the record to be used for inserting. This can be a bind variable, a record object, or a record name in the form *recname*. You *can't* specify a RECORD.*recname*, a record name in quotes, or a table name.

correlation_id Identifies the single letter correlation ID to be used to relate the record specified by *recname* and its fields.

Example

The record &REC has three keys: FNUM, FDATE, FSMART. Therefore, the following code:

```
Local record &REC;

&REC = CreateRecord(RECORD.MYRECORD);

&REC.FNUM.Value = 27;

&REC.FDATE.Value = %Date;

SQLExec("Delete from MYRECORD A where %KeyEqual(:1, A)", &REC);
```

expands to:

```
"Delete from TABLE A

where A.FNUM = 27

AND A.FDATE = %Date('1989-11-27')

AND A.FSMART IS NULL"
```

%KeyEqualNoEffDt

Syntax

```
%KeyEqualNoEffDt(recname [ correlation_id] )
```

Description

The **%KeyEqualNoEffDt** function expands into a conditional phrase suitable for use in a WHERE clause.

The conditional phrase consists of a conjunction (AND) of *[correlation_id].keyfieldname = 'keyfieldvalue'* phrases for all key fields of the given record, *except* that it omits any key field named EFFDT.

No autoupdate processing is done, but other input processing is applied to the values. That is:

- if the field is a date, a time, or a datetime, its value will automatically be wrapped in %DateIn(), %TimeIn(), or %DateTimeIn(), respectively
- if a value is a string, its value will automatically be wrapped in quotes
- if a value is NULL, the "*=value*" part is replaced with "IS NULL"



This meta-SQL can only be used in PeopleCode programs, not in Application Engine PeopleCode actions. Also, this meta-SQL is not implemented for COBOL.

Parameters

<i>recname</i>	Specify the name of the record to be used for inserting. This can be a bind variable, a record object, or a record name in the form <i>recname</i> . You <i>can't</i> specify a <code>RECORD.recname</code> , a record name in quotes, or a table name.
<i>correlation_id</i>	Identifies the single letter correlation ID to be used to relate the record specified by <i>recname</i> and its fields.

Example

The EMPL_CHECKLIST record has three keys: EMPLID, CHECK_SEQ, and EFFDT. Therefore, the following code:

```
&REC = CreateRecord(EMPL_CHECKLIST);

SQLExec("Delete from TABLE A where %KeyEqualNoEffdt(:1, A)", &REC)
```

expands to:

```
"Delete from TABLE A

where A.EMPLID = 8001

AND A.CHECK_SEQ = 00001"
```

%Like

Syntax

```
%Like("Literal")
```

Description

The **%Like** function expands to look for literal values. This meta-SQL should be used when looking for like values. A % is appended to *literal*.



This meta-SQL is not implemented for COBOL.

If you're using a bind marker (such as ":1") for the literal argument in a SQLExec, you must wrap the SQL string with ExpandSqlBinds. ExpandSqlBinds replaces bind markers with the actual input values.



For more information, see Considerations using Bind Markers.

%Like generates the following:

```
like 'literal%
```

If the literal value contains '_' or '%', %Like generates the following:

```
like 'literal%' escape '\'
```

Using %Like and Eliminating Blanks

Some platforms require that you use RTRIM to get the correct value. The following characters are wildcards even when preceded with the backslash '\' escape character.

- %
- _

Therefore, on some platforms, the literal must end with a '%' wildcard that isn't preceded by '\'.

```
literal = 'ABC%' --> no need for RTRIM on any platform
```

```
literal = 'ABC\%' --> need RTRIM on MSS and DB2.
```

Using %Like and Trailing Blanks

It is important to note that not all executions of LIKE perform the same. When dealing with trailing blanks, some platforms behave as if there is an implicit % at the end of the comparison string, while most don't, as shown in the following example:

In this example, if the selected column contains the string "ABCD " (three trailing blanks), the following statement may or may not return the any rows:

```
select * from t1 Where c like 'ABCD'
```

Therefore, it is always important to explicitly code the % the end of matching strings for those columns where you want to include trailing blanks.

DBMS	<i>Implicit %?</i>
PeopleSoft Standard Usage	Always use %
DB2/400	No
DB2/MVS	No

DB2/Unix	No
Informix	No
Microsoft SQL Server	Yes
Oracle	No
SQLBase	No
Sybase	Yes

LIKE Wild Cards

SQL specifies two wild cards that can be used when specifying pattern matching strings for use with the LIKE predicate. The underscore is used as a substitution for a single character within a string, the percent sign is used to represent any number of character spaces within a string.

DBMS	Single Character	Multiple Characters
PeopleSoft Standard Usage	—	%
DB2/400	—	%
DB2/MVS	—	%
DB2/Unix	—	%
Informix	—	%
Microsoft SQL Server	—	%
Oracle	—	%
SQLBase	—	%
Sybase	—	%

Parameters

literal Specify the value you want to search for.

%LikeExact

Syntax

```
%LikeExact (fieldname, "Literal")
```

Description

The **%LikeExact** meta-SQL expands to look for literal values. This should be used when exact matches are necessary, taking into account wildcards in the literal values.



This meta-SQL is not implemented for COBOL.

%LikeExact generates one of the following:

- If the literal contains no wildcards:

```
fieldname = 'literal'
```

- If the literal ends with the '%' wildcard:

```
fieldname like 'literal' [escape '\']
```

Some platforms require that you use RTRIM to get the correct value. The following characters are wildcards even when preceded with the backslash '\' escape character.

- %
- _

Therefore, on some platforms, the literal must end with a '%' wildcard that isn't preceded by '\'.

```
literal = 'ABC%' --> no need for RTRIM on any platform
```

```
literal = 'ABC\%' --> need RTRIM on MSS and DB2.
```

Considerations using Bind Markers

If you're using a bind marker (such as ":1") for the literal argument in a SQLExec, you must wrap the SQL string with ExpandSqlBinds. ExpandSqlBinds replaces bind markers with the actual input values.

The following forms work:

- AE SQL (with or without Reuse enabled)

```
UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE %LIKEEXACT(AE_APPL_ID,
%Bind(AE_APPL_ID, STATIC))
```

The STATIC modifier is only required if Reuse is enabled, but it can't hurt to use it (if Reuse is not enabled, static is the default anyway).

- PeopleCode

```
AE_TESTAPPL_AET.AE_APPL_ID = "AB\_C";
```

```
SQLExec("UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE %LIKEEXACT(AE_APPL_ID,
:AE_TESTAPPL_AET.AE_APPL_ID)");
```

or

```
SQLExec(ExpandSqlBinds("UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE
%LikeExact(AE_APPL_ID, :1)", "AB\_C"));
```

This form does *not* work:

```
SQLExec("UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE %LIKEEXACT(AE_APPL_ID,
:1)", "AB\_C");
```

Parameters

<i>fieldname</i>	Specify a field to be used in the first part of the LIKE comparison.
<i>literal</i>	Specify the value you want to search for.

Example

The following code

```
UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE %LIKEEXACT(AE_APPL_ID, 'ABC')
```

resolves into the following:

```
UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE AE_APPL_ID = 'ABC'
```

The following code

```
UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE %LIKEEXACT(AE_APPL_ID, 'AB%C')
```

resolves into the following:

```
UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE RTRIM(AE_APPL_ID) LIKE 'AB%C'
```

The following code

```
UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE LIKEEXACT(AE_APPL_ID, 'AB%C%')
```

resolves into the following:

```
UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE AE_APPL_ID LIKE 'AB%C%'
```

The following code

```
UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE %LIKEEXACT(AE_APPL_ID, 'AB%C%
')
```

resolves into the following:

```
UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE AE_APPL_ID LIKE 'AB%C% '
```

The following example shows using ExpandSqlBinds:

```
SQLExec(ExpandSqlBinds("SELECT COUNT(*) FROM PS_ITEM WHERE
%LIKEEXACT(BUSINESS_UNIT, :1)", "M04"), %COUNT);
```

%List

Syntax

```
%List({FIELD_LIST | FIELD_LIST_NOLONGS | KEY_FIELDS | ORDER_BY }, recordname [
correlation_id])
```

Description

The **%List** function expands into a list of field names, delimited by commas. Which fields are included in the expanded list depends on the parameters passed to the function.



This meta-SQL is not implemented for COBOL, Dynamic View SQL or PeopleCode.

Parameters

FIELD_LIST	Use all field names in the given record. You can select only one option from FIELD_LIST, ORDER_BY, FIELD_LIST_NOLONGS or KEY_FIELDS..
KEY_FIELDS	Use all key fields in the given record. You can select only one option from FIELD_LIST, FIELD_LIST_NOLONGS, KEYFIELDS or ORDER_BY.
ORDER_BY	Use all the key fields of <i>recordname</i> , adding DESC for descending key columns. This parameter is often used when the list being generated is for an ORDER BY clause. You can select only one option from FIELD_LIST, KEY_FIELDS, ORDER_BY, or FIELD_LIST_NOLONGS.
FIELD_LIST_NOLONGS	Use all field names in the given record, <i>except</i> any "long" columns (that is, long text or image fields.) You can select only one option from FIELD_LIST, ORDER_BY, KEY_FIELDS, or FIELD_LIST_NOLONGS.
<i>recordname</i>	Identifies either a record or a subrecord that the field names will be drawn from. This can be a bind variable, a record object, or a record name in the form <i>recname</i> . You <i>can't</i> specify a RECORD. <i>recname</i> , a record name in quotes, or a table name.
<i>correlation_id</i>	Identifies the single letter correlation ID to be used to relate the record specified by <i>recordname</i> and its fields.

Considerations Using %List

When using %List in an insert/select or insert/values or %Select statement, you *must* have matching pairs of %List (or %ListBind) in the target and source field lists, using the same list type argument and record name to ensure consistency.

Example

The following is a *good* example of using %List. Note that both the INSERT and SELECT statements use the same %List.

```
INSERT INTO PS_PO_DISTRIB_STG ( %Sql(POCOMMONDISTSTGFDDLSTU)

, %List(FIELD_LIST, CF16_AN_SBR)

, MERCHANDISE_AMT

, MERCH_AMT_BSE

, QTY_DEMAND

, QTY_PO

, QTY_PO_STD

, QTY_REQ)

SELECT %Sql(POCOMMONDISTSTGFDDLSTU)

, %List(FIELD_LIST, CF16_AN_SBR)

, MERCHANDISE_AMT

, MERCH_AMT_BSE

, QTY_DEMAND

, QTY_PO

, QTY_PO_STD

, QTY_REQ

FROM PS_PO_DIST_STG_WRK WRK

WHERE WRK.PROCESS_INSTANCE = %Bind(PROCESS_INSTANCE)
```

The following example shows both a *poor* example of how to use %List, as well as how the example might be rewritten to use %List correctly.

In the following SQL Object, INSERT and SELECT field lists both use %List, but the SELECT field list is only partly dynamic. The rest is hard-coded.

Poor code example:

```
INSERT INTO PS_EN_TRN_CMP_TMP (%List(FIELD_LIST, EN_TRN_CMP_TMP))
```

```

SELECT B.EIP_CTL_ID

, %List(SELECT_LIST, EN_BOM_COMPS A)

, E.COPY_DIRECTION

, E.BUSINESS_UNIT_TO

, E.BOM_TRANSFER_STAT

, 'N'

, B.MASS_MAINT_CODE

, 0

FROM PS_EN_BOM_COMPS A

, PS_EN_ASSY_TRN_TMP B

, PS_EN_TRNS_TMP E

WHERE ...

```

Rewritten *Good* code example:

```

INSERT INTO PS_EN_TRN_CMP_TMP (EIP_CTL_ID,

, %List(FIELD_LIST, EN_BOM_COMPS)

, COPY_DIRECTION

, BUSINESS_UNIT_TO

, BOM_TRANSFER_STAT

, EN_MMC_UPDATE_FLG

, MASS_MAINT_CODE

, EN_MMC_SEQ_FLG01

, ...

, EN_MMC_SEQ_FLG20)

SELECT B.EIP_CTL_ID

, %List(FIELD_LIST, EN_BOM_COMPS A)

, E.COPY_DIRECTION

, E.BUSINESS_UNIT_TO

, E.BOM_TRANSFER_STAT

, 'N'

```

```

, B.MASS_MAINT_CODE

, 0

, ...

, 0

FROM PS_EN_BOM_COMPS A

, PS_EN_ASSY_TRN_TMP B

, PS_EN_TRNS_TMP E

WHERE ...

```

The following example shows both a *poor* example of how to use %List, as well as how the example might be rewritten to use %List correctly.

In the following SQL Object, only the field list of on the INSERT portion is dynamically generated, but the SELECT is statically coded. If the table STL_NET_TBL is reordered, the INSERT will be incorrect.

Example of *poor* code:

```

INSERT INTO PS_STL_NET_TBL (%List(FIELD_LIST, STL_NET_TBL ) )

SELECT :1

, :2

, :3

, :4

, :5

, :6

, :7

, :8

FROM PS_INSTALLATION

```

Rewritten *Good* code example

```

INSERT INTO PS_STL_NET_TBL (%List(FIELD_LIST, STL_NET_TBL))

VALUES (%List(BIND_LIST, STL_NET_TBL MY_AET))

```

%ListBind

Syntax

```
%ListBind({FIELD_LIST | FIELD_LIST_NOLONGS | KEY_FIELDS}, recordname [
State_record_alias])
```

Description

The **%ListBind** function expands a field list as bind references for use in an INSERT/VALUE statement.



This meta-SQL is not implemented for COBOL, Dynamic View SQL or PeopleCode.

Parameters

FIELD_LIST	Use all field names in the given record. You can select only one option from FIELD_LIST, FIELD_LIST_NOLONGS or KEY_FIELDS.
FIELD_LIST_NOLONGS	Use all field names in the given record, <i>except</i> any "long" columns (i.e., long text or image fields.) You can select only one option from FIELD_LIST, FIELD_LIST_NOLONGS or KEY_FIELDS.
KEY_FIELDS	Use all key field names in the given record. You can select only one option from FIELD_LIST, FIELD_LIST_NOLONGS or KEY_FIELDS.
<i>recordname</i>	Identifies either a record or a subrecord that the field names will be drawn from. This can be a bind variable, a record object, or a record name in the form <i>recname</i> . You <i>can't</i> specify a RECORD. <i>recname</i> , a record name in quotes, or a table name.
<i>State_record_alias</i>	Specify the Application Engine State Record buffer that contains the values (this could be different than the record used to derive the field list.) If missing, the default State Record is assumed.

Considerations Using %ListBind

When using %List in an insert/select or insert/values or %Select statement, you *must* have matching pairs of %List or %ListBind in the target and source field lists, using the same list type argument and record name to ensure consistency.

Example

```
INSERT INTO PS_TARGET (FIELD1, FIELD2, %List(FIELD_LIST, CF_SUBREC), FIELDN)
VALUES (%Bind(MY_AET.FIELD1), %Bind(MY_AET.FIELD2), %ListBind(FIELD_LIST,
CF_SUBREC MY_AET), %Bind(MY_AET.FIELDN))
```

%ListEqual

Syntax

```
%ListEqual({ALL | KEY }, Recordname [alias], RecordBuffer [, Separator])
```

Description

The **%ListEqual** function maps each field, possibly to an alias, with a %Bind value, with a separator added before each equality. Each field is mapped as follows:

```
<alias>.X = %Bind(<recbuffer>.X)
```

This function can be used in the SET clause of an update or in a WHERE clause.



This meta-SQL is not implemented for COBOL, Dynamic View SQL or PeopleCode.

Parameters

ALL | KEY

Specify the type of fields you want, whether you want all the fields, or just the key fields.

recordname

Identifies either a record or a subrecord that the field names will be drawn from. This can be a bind variable, a record object, or a record name in the form *recname*. You *can't* specify a RECORD.*recname*, a record name in quotes, or a table name.

alias

Specify an optional alias to prefix each field name.

RecordBuffer

Specify the record buffer for the binds (this could be different than the record used to derive the field list.)

Separator

Specify a separator to be used between each equality. Valid values are AND or OR. The default value of a comma will be used if no separator is specified.

Example

```
UPDATE PS_TEMP

SET   %ListEqual(ALL,  CF_SUBREC, MY_AET)
```

```
WHERE %ListEqual(KEYS, TEMP, MY_AET, AND)
```

%OldKeyEqual

Syntax

```
%OldKeyEqual(recname [correlation_id])
```

Description

The **%OldKeyEqual** function is similar to the **%KeyEqual** function, except that it uses the "original" values of the record fields, rather than the current values. Since the rules by which values are original and which are current are not very clear, especially for stand-alone record objects, the use of this meta-SQL function is discouraged. You should use separate records to hold the desired previous values. This will make your code much clearer and more maintainable.



This meta-SQL can only be used in PeopleCode programs, not in Application Engine PeopleCode actions. Also, this meta-SQL is not implemented for COBOL.



For more information, see %KeyEqual.

%OPRCLAUSE

The %OPRCLAUSE metastring is used in the viewtext of dynamic views. In PeopleTools 6 the %OPRCLAUSE metastring expanded in the following manner:

```
SELECT EMPLID, ABSENCE_TYPE, oprid
FROM PS_ABSENCE_HIST
WHERE %OPRCLAUSE
```

```
SELECT EMPLID, ABSENCE_TYPE, OPRID FROM PS_ABSENCE_HIST WHERE ( OPRCLASS
='HRADMIN') AND (EMPLID='8001' AND ABSENCE_TYPE='CNF') ORDER BY EMPLID,
ABSENCE_TYPE
```

In PeopleTools 7, to support the new concept of a specific "Row Level Security Class", this metastring now fills in the WHERE clause with the value from PSOPRDEFN.ROWSECCLASS.

%OPRCLAUSE must be either all upper case or all lowercase. Mixed case isn't allowed.

%Round

Syntax

```
%Round(expression, factor)
```

Description

%Round rounds an expression to a specified scale before or after the decimal point. If *factor* is a literal, it can be rounded to a negative number.

Parameters

<i>expression</i>	An arbitrary numeric expression involving numeric constants and database columns.
<i>factor</i>	An integer or bind variable in SQLExec PeopleCode. The range of a factor is from -31 to +31 for literals. Non-literals can only be positive.

Examples

```
%Round(10.337, 2) = 10.34
```

```
%Round(13.67, 0) = 14
```

```
SQLExec("SELECT %Round(field_c1, :1) from RECORD_T", field_c2, &Result);
```

where field_c1 and field_c2 are two fields in the record.

The following cases are illegal, and may cause incorrect results or runtime SQL errors:

```
%Round(10.337, 2 + 1) (factor can not be an expression)
```

```
%Round(field_c1, field_c2) (factor can not be database columns)
```

%SQL

Syntax

```
%SQL(SQLid [, paramlist])
```

where *paramlist* is a list of arguments that are used for dynamic substitutions at runtime, in the form:

```
arg1 [, arg2] . . .
```

Description

Use the **%SQL** function for common SQL fragments that you have already defined and want to re-use, substituting additional values dynamically. *SQLid* is the name of a SQL definition created using either Application Designer or the **StoreSQL** function.



This meta-SQL is not implemented for COBOL.

Note. A SQL definition is *not* the same as the SQL Object that is instantiated from the SQL Class at runtime. A SQL definition is created either using Application Designer at design time, or using the StoreSQL function. See SQL Definition for more information. A SQL Object is instantiated at runtime from the SQL Class, and has methods and properties associated with it like any other object. See SQL Class for more information.

When a SQL definition specified has more than one version, the database type always takes precedence. That is:

- If one or more versions of a SQL definition are found for the database type of the current database connection, and if any of the versions have an effective date less than or equal to the value returned for %AsOfDate, the most recent version is used.
- If no versions are found for the current database type, or if all of the versions have effective dates greater than the value returned for %AsOfDate, the system looks for an effective version of the SQL definition under the database type "generic". If no version is found, an error will occur.

Application Engine Considerations

Application Engine programs use the current date to compare with the effective date, not the date returned by %AsOfDate.

Special SQL Characters

The following meta-SQL constructs can be used as part of the %SQL function to represent special characters as SQL parameters.

%Comma	Represents a single comma.
%LeftParen	Allows you to pass a left parenthesis, "(", to a %P() variable, without closing the SQL object.
%RightParen	Allows you to pass a right parenthesis, ")", to a %P() variable, without closing the SQL object.
%Space	Represents a space.

Parameters

SQLid Specify the name of an existing SQL definition.

paramlist

An optional list of arguments that are used for dynamic substitutions at runtime. The first argument replaces all occurrences of %P(1) in the referenced SQL definition, the second argument replaces %P(2), and so forth. You can specify up to 99 arguments in paramlist.

Example

In the following example, the SQL definition MY_SQL was created in Application Designer to be the following:

```
%P(1).EFFDT = (SELECT MAX(EFFDT) FROM ...)
```

Therefore the following code (the **bold** portions will be dynamically generated):

```
UPDATE PS_TEMP
SET ...
WHERE ...
AND %SQL(MY_SQL, PS_TEMP)
```

resolves to the following:

```
UPDATE PS_TEMP
SET ...
WHERE ...
AND PS_TEMP.EFFDT = (SELECT MAX(EFFDT) FROM ...)
```

Related Topics

SQL Class

%Substring

%Substring(*source_str*, *start*, *length*) expands to a substring of *source_str*, where *start* specifies the substring's beginning position (the first character of *source_str* is position 1), and *length* is the length of the substring.

%SUBREC

%SUBREC(*subrec_name*, *corel_name*) is used only in Dynamic View SQL where it expands to the columns of a subrecord: you can't use this statement in SQLExec or any other SQL statement. *subrec_name* is the name of the subrecord; *corel_name* is the correlation name.

%SUBREC must be either all upper case or all lowercase. Mixed case isn't allowed.

For example, suppose you have a record definition AAA_VW that is a dynamic view with fields CHR, SUB, and NUM. The field SUB is a subrecord with fields CHR_SUB, NUM_SUB, and IMG_SUB. The viewtext for AAA_VW could be:

```
"select a.chr, %subrec(sub,a), a.num from ps_aaa a"
```

The Create View SQL generated by this viewtext would be:

```
"CREATE VIEW SYSADM.PS_AAA_VW (CHR, CHR_SUB, NUM_SUB, IMG_SUB, NUM) AS SELECT  
A.CHR, A.CHR_SUB, A.NUM_SUB, A.IMG_SUB, A.NUM FROM PS_AAA A"
```

%Table

Syntax

```
%Table(recname [, instance])
```

Description

The **%Table** function returns the SQL table name for the record specified with *recname*.

For example, %Table(ABSENCE_HIST) will return PS_ABSENCE_HIST.



This meta-SQL is not implemented for COBOL.

If the record is a temporary table and the current process has a temp table instance number assigned, %Table resolves to that instance of the temp table (that is, PS_ABSENCE_HIST nn , where nn is the instance number).

Or, you can override this value with the *instance* parameter. For example, suppose you know you want the third instance of a temp table, you could specify it with %Table(&MYREC, 3). You can use the SetTempTableInstance function to set the instance of a temp table that will be used with %Table.

This function can be used to specify temporary tables for running parallel Application Engine processes.



For more information, see %Table description in Application Engine.

Parameters

<i>recname</i>	Identifies the record that the table name will be drawn from. This can be a bind variable, a record object, or a record name in the form <i>recname</i> . You <i>can't</i> specify a RECORD. <i>recname</i> , a record name in quotes, or a table name.
<i>instance</i>	Specify the instance of the temp table to be used.

Example

The following function deletes records based on two other fields:

```
Function delete_draft_type(&RECNAME)

&SQL = "Delete from %Table(:1) where " | FIELD.SETID | " = :2 and " |
FIELD.DRAFT_TYPE | " = :3";

SQLExec(&SQL, @("RECORD." | &RECNAME), SETID, DRAFT_TYPE);

End-Function;
```

Related Topics

SetTempTableInstance function

%TextIn

Syntax

```
%TextIn(BindVariable)
```

Description

%TextIn function, when used with a bind variable, allows the insertion and updating of a text string into a LONG CHARACTER field (column).



This Meta-SQL is not implemented for COBOL.



This Meta-SQL is mandatory for any LONG CHARACTER field insertion or update to be compatible on all database platforms. If you don't use this Meta-SQL wrapper, this type of operation fails on Sybase and Informix.

Parameters

BindVariable A bind variable

Example

In the following example, :1 is a bind variable in PeopleCode.

```
&String1 = "This is a test."
```

```
SqlExec("INSERT INTO PS_TABLE1 (STMTID, SQLSTMT) VALUES (1, %TextIn(:1))",
&String1)
```

%TimeAdd

Syntax

```
%TimeAdd(datetime, add-minutes)
```

Description

This function generates the SQL that will add *add-minutes* (which can be a positive or negative integer literal or expression, provided that the expression resolves to a data type that can be used in datetime arithmetic for the given RDBMS) to the provided *datetime* (which can be a datetime literal or expression).



On some platforms, you can use a *time-value* in place of *datetime*. However, this can give a SQL error on other platforms (for example, Informix) if the result of the %TimeAdd would result in a new date (for example, 11:59PM + 2 minutes).



This meta-SQL is not implemented for COBOL.

Parameters

<i>time</i>	Specify a time or datetime value that you want to add more time to.
<i>addminutes</i>	Specify the number of minutes you want to add to <i>time</i> . This must be a numeric value or an expression that resolves to a numeric value.

Example

```
SELECT %TimeAdd(%CurrentTimeIn, 60) FROM PS_INSTALLATION
```

%TimeIn

`%TimeIn(tm)` expands to platform-specific SQL for a Time value in the WHERE clause of a SQL SELECT or UPDATE statement, or when a time value is passed in an INSERT statement. The parameter *tm* is either a Time bind variable or a String literal in the form:

```
hh.mm.ss.ssssss
```

Restrictions using COBOL

You can only use string literals when using this function in COBOL. You can *not* use it with bind parameters in COBOL. For example, the following will work in COBOL:

```
UPDATE PS_PERSONAL_DATA SET LASTUPTM = %TIMEIN('11:59:00:000000')
```

Whereas the following SQL will fail:

```
UPDATE PS_PERSONAL_DATA SET LASTUPTM = %TIMEIN(:1)
```

%TimeOut

`%TimeOut(time_col)`, where *time_col* is a time column, expands to a platform-specific SQL substring representing *time_col* in the SELECT clause of an SQL query.

%TrimSubstr

`%TrimSubstr(source_str, start, length)` is similar to `%Substring` except that trailing blanks are removed from the substring.

%Truncate

Syntax

```
%Truncate(expression, factor)
```

Description

`%Truncate` truncates an expression to a specified scale before or after the decimal point.

Parameters

<i>Expression</i>	Is an arbitrary expression involving numeric constants and database columns.
<i>Factor</i>	An integer or bind variable in SQLExec PeopleCode. The range of a factor is -30 to +31, negative number truncates to left of the decimal point.

Examples

```
%Truncate(10.337, 2) = 10.33
```

```
%Truncate(13.37, 0) = 13
```

```
%Truncate(19.337, -1) = 10
```

```
SQLExec("SELECT %Truncate(field_c1, :1) from RECORD_T", field_c2, &Result);
```

where *field_c1* and *field_c2* are two fields in the record.

Illegal case

```
%Truncate(10.337, 2 + 1) (factor can not be an expression)
```

You may get wrong results or runtime SQL errors when using the illegal case.

%TruncateTable

Syntax

```
%TruncateTable(table_name)
```

Description

%TruncateTable deletes all the rows in a table.



You must use a table name, not a record name, with this statement.

On ORACLE, Microsoft SQL Server, and SYBASE, the use of %TruncateTable causes an implicit commit. The rows deleted by this command, and any other pending database updates, will all be committed. If you want to postpone the commit until subsequent database updates have been successfully completed, use the SQL statement 'DELETE FROM *table_name*' instead of %TruncateTable(*table_name*). The advantage of using %TruncateTable is that its execution is faster than 'DELETE FROM *table_name*'. %TruncateTable is often used for removing rows from a work table or a temporary table.

If you're calling %TruncateTable from an Application Engine program step, you should commit after the step that immediately precedes the step containing the %TruncateTable. Also, %TruncationTable should never be used on a step that is executed multiple times within a loop. In general, it's best to use this construct early in your Application Engine program as an initialization task.

In addition, you want to avoid using this meta-SQL when your Application Engine program is started from **CallAppEngine**.

Example

```
%TruncateTable(PS_TEMP_TABLE)
```

%UpdatePairs

Syntax

```
%UpdatePairs(recname [correlation_id])
```

Description

The **%UpdatePairs** function produces a comma separated list of *fieldname = 'fieldvalue'* phrases for each changed field of the given record. Input processing is applied to the values, that is:

- if the field is a date, a time, or a datetime, its value will automatically be wrapped in %DateIn(), %TimeIn(), or %DateTimeIn(), respectively
- if the field is a string, its value will automatically be wrapped in quotes
- if the field has a null value, NULL will be the given value



This meta-SQL can only be used in PeopleCode programs, not in Application Engine PeopleCode actions. Also, this meta-SQL is not implemented for COBOL.

Parameters

<i>recname</i>	Specify the name of the record to be used for updating. This can be a bind variable, a record object, or a record name in the form <i>recname</i> . You <i>can't</i> specify a RECORD. <i>recname</i> , a record name in quotes, or a table name.
<i>correlation_id</i>	Identifies the single letter correlation ID to be used to relate the record specified by <i>recname</i> and its fields.

Example

The record &REC has one key: FNUM. The FCHAR field has changed. Therefore, the following code:

```
Local record &REC;

&REC = CreateRecord(RECORD.MYRECORD);

&REC.FNUM.Value = 27;

&REC.FCHAR.Value = 'Y';
```

```
SQLExec("Update TABLE set %UpdatePairs(:1) where %KeyEqual(:1)", &REC)
```

expands to:

```
"Update TABLE set FCHAR = 'Y' where FNUM = 27"
```

The following example updates all the fields on a base record (&REC) that are *not* also fields on the related language record (&REC_RELATED_LANG). It creates a "holding" record (&REC_TEMP), copies the specific fields you want to update from the base record to the holding record, then uses the holding record for the update.

```
&UPDATE = CreateSQL("Update %Table(:1) set %UpdatePairs(:1) Where
%KeyEqual(:2)");

&REC_TEMP = CreateRecord(@"RECORD." | &REC.Name);

&FIELD_LIST_ARRAY = CreateArray();

For &I = 1 to &REC_RELATED_LANG.FieldCount

    &FIELD_LIST_ARRAY.Push(&REC_RELATED_LANG.GetField(&I).Name);

End-For;

For &I = 1 to &REC.FieldCount

    If &FIELD_LIST_ARRAY.Find(&REC.GetField(&I).Name) = 0 then

        &REC_TEMP.GetField(&I).Value = &REC.GetField(&I).Value;

    End-If;

End-For;

&UPDATE.Execute(&REC_TEMP, &REC);
```

%Upper

Syntax

```
%Upper(charstring)
```

Description

The **%Upper** function converts the string *charstring* to uppercase. You can use wildcards with *charstring*, like %.



This meta-SQL is not implemented for COBOL.

Parameters

charstring The string you want converted to uppercase.

Example

```
SELECT EMPLID, NAME FROM PS_EMPLOYEES WHERE %UPPER(NAME) LIKE %UPPER(sch%)
```

Meta-SQL Shortcuts

The following shortcuts are provided for when you want use the entire list of key fields for a record.



The meta-SQL shortcuts can only be used in PeopleCode programs, not in Application Engine PeopleCode actions. Also, none of the meta-SQL shortcuts are implemented for COBOL.

%Delete(:num)

This is a shorthand for:

```
Delete from %Table(:num) where %KeyEqual(:num)
```

%Insert(:num)

This is a shorthand for:

```
Insert into %Table(:num) (%List(Nonnull_Fields :num)) values
(%InsertValues(:num))
```

%SelectAll(:num [correlation_id])

%SelectAll is shorthand for selecting all fields in the specified record, wrapping date/time fields with %DateOut, %TimeOut, and so on.

The pseudo-code looks like this:

```
Select (All Fields, :num correlation_id) from %Table(:num) prefix
```

This shortcut is only appropriate if the statement is being used in PeopleCode or Application Engine to read data into memory. Dynamic Views should retain the internal database formats for DateTime fields.

%SelectDistinct(:num [*prefix*])

%SelectDistinct is shorthand for selecting all fields in the specified record, wrapping date/time fields with %DateOut, %TimeOut, and so on.

The pseudo-code looks like this:

```
Select DISTINCT (All Fields, :num correlation_id) from %Table(:num) prefix
```

This shortcut is only appropriate if the statement is being used in PeopleCode or Application Engine to read data into memory. Dynamic Views should retain the internal database formats for DateTime fields.

%SelectByKey(:num [*correlation_id*])

This is a shorthand for:

```
Select %List(Select_List, :num correlation_id) from %Table(:num) correlation_id
where %KeyEqual(:num, correlation_id)
```

%SelectByKeyEffDt(:num1, :num2)

This is a shorthand for:

```
Select %List(Select_List, :num1) from %Table(:num1) A where
%KeyEqualNoEffDt(:num1 A) and %EffDtCheck(:num1 B, A, :num2)
```

%Update(:num [, :num2])

This is a shorthand for:

```
Update %Table(:num) set %UpdatePairs(:num) where %KeyEqual(:num2)
```

If *:num2* is omitted, it defaults to *:num*.

CHAPTER 11

System Variables

PeopleTools provides a number of system variables that provide access to system information. System variables are prefixed with the ‘%’ character, rather than the ‘&’ character. You can use these system variables wherever you can use a constant, passing them as parameters to functions or assigning their values to fields or to temporary variables.

%AsOfDate

Returns the as-of-date of the environment that the PeopleCode is running in. In most cases, this will be the current date, but for Application Engine environments, it will be the processing date of the Application Engine program.

%AuthenticationToken

This system variable returns a single signon authentication token for the user after SwitchUser is executed. For example, you can use this system variable to write a single signon cookie to the http response after a new user is authenticated.



Note. This system variable only returns a valid value after SwitchUser executes successfully.

%BPName

%BPName is relevant when the user has accessed a page from a worklist entry. It returns a string containing the name of the Business Process for the worklist entry. It returns an empty string if the user didn’t access the current page group from a worklist.

%ClientDate

%ClientDate returns the current date on the client, adjusted for the user’s time zone. This is potentially one day different than the server date.

Restrictions on Use in Three-Tier Mode

%ClientDate is a client-only constant, which limits its use in three-tier mode. In three-tier mode **%ClientDate** can be used only in processing groups set to run on the client. If **%ClientDate** is called in a processing group running on the application server, a runtime error occurs.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

%ClientTimeZone

%ClientTimeZone returns the current time zone on the client as a three-character string.

Restrictions on Use in Three-Tier Mode

%ClientTimeZone is a client-only constant, which limits its use in three-tier mode. In three-tier mode **%ClientTimeZone** can be used only in processing groups set to run on the client. If **%ClientTimeZone** is called in a processing group running on the application server, a runtime error occurs.



For more information, see PeopleCode and PeopleSoft Internet Architecture.

%Component

%Component returns an uppercase character string containing the name of the current component, as set in the component definition.

%CompIntfcName

%CompIntfcName returns the name of the Component Interface, if the currently executing PeopleCode program is being run from a Component Interface. If the currently executing PeopleCode program is *not* being run from a Component Interface, this variable will return NULL (if the program is running from PeopleCode) or "Nothing" (if running from Visual Basic.)

%Currency

This system variable returns the preferred currency for the current user.

%Date

%Date returns a Date value equal to the current server date.

%DateTime

%DateTime returns the current server date and time as a Datetime value.

%DbName

%DbName returns the name of the current database as a String value.

%DbType

%DbType returns a string representing the type of the current database; for example, "MICROSFT" or "SYBASE".

%EmailAddress

This system variable returns the email address of the current user.

%EmployeeId

%EmployeeId returns an uppercase character string containing the Employee ID of the user currently logged on. This is typically used to restrict access to an employee's own records.

%ExternalAuthInfo

This system variable returns external connect information. Programmers can customize the authentication process by passing in binary data. This data is encoded with base64 encoding and passed to signon PeopleCode as a string using this system variable.



This system variable can only be used in Signon PeopleCode.



Note. This system variable isn't applicable with the PeopleSoft Internet Architecture.

%Import

%Import returns True if an import is being performed by Import Manager and False if not.

%IsMultiLanguageEnabled

This system variable returns True if the current user is multi-language enabled.



For more information, see [Working With Language-Sensitive Application Data](#).

%Language

%Language returns a string value representing the current setting from the Language menu if there is one, otherwise it returns the user's language setting (from the security administrator or user definition). It does not use the Language setting from the configuration tool, display page. This value can be changed with the SetLanguage function.

%Language_Base

%Language_Base returns the base language for the current database, as set with the PeopleTools Options page.



For more information, see [PeopleTools Options](#).

%Market

The %Market system variable returns a three-character String value for the Market property of the current page group. This is useful if you want to add market-specific PeopleCode functionality to a component. For example:

```
if %Component = COMPONENT.PERSONAL_DATA then

    /* do some stuff that applies to all localized version */

    :

    :

    /* do some stuff that differs by market */

    evaluate %Market
```

```

when = "USA"

/* do usa stuff */

break;

when = "GER"

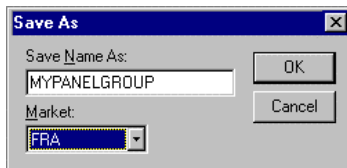
/* do german stuff */

end-evaluate;

end-if;

```

The Market property of a component specifies a component's target market. This property is set when a component is initially saved or cloned.



Setting the Component Market Property

Components that are used on a global basis have a market setting of "GBL". Variations of components targeted at a specific market can have a local Market setting, for example "FRA". This enables developers to avoid cloning, renaming, and coding distinct PeopleCode in market-specific components. Instead, they can create a single component with market-specific PeopleCode, then clone the component, applying different Market property settings.

Because the %Market string is a three-character string like Country Code, Country Codes can be used as market settings where appropriate.



For more information, see [Creating Component Definitions](#).

%MaxInterlinkSize

%MaxInterlinkSize returns the current size limit of Interlink objects, as set on the PeopleTools Options page.



For more information, see [PeopleTools Options](#).

%MaxMessageSize

%MaxMessageSize returns the current size limit of application messages, as set on the PeopleTools Options page.



For more information, see PeopleTools Options.

%Menu

%Menu returns an uppercase string containing the current menu name. It can be used to restrict edits or processing to a specific menu.

%MessageAgent

%MessageAgent returns a String representing the current Message Definition name, if the component was invoked by the Message Agent. If the current page was not invoked by the Message Agent, %MessageAgent returns a blank string.

%Mode

%Mode returns a String value consisting of an uppercase character specifying the action a user selected when starting the current component. The following values can be returned. You can check either for the string value ("A", "U", etc.) or for the constant:

Value	Constant	Description
A	%Action_Add	Add
U	%Action_UpdateDisplay	Update/Display
L	%Action_UpdateDisplayAll	Update/Display All
C	%Action_Correction	Correction
E	%Action_DataEntry	Data Entry
P	%Action_Prompt	Prompt

%NavigatorHomePermissionList

This system variable returns the navigator homepage permission list for the current user.

%OperatorClass

This system variable returns a string representing the primary or base class of the current operator.



This system variable is supported for compatibility with previous releases of PeopleTools. Future applications should use %PasswordExpired instead.

%OperatorId

%OperatorId returns an uppercase character string containing the operator currently logged on. This is typically used to restrict access to records or fields to specific operators.



This system variable is supported for compatibility with previous releases of PeopleTools. Future applications should use %UserId instead.

%OperatorRowLevelSecurityClass

This system variable returns a string representing the row-level security class of the current operator. The row-level security class is now distinct from the operator's primary class.



This system variable is supported for compatibility with previous releases of PeopleTools. Future applications should use %RowSecurityPermissionList instead.

%Page

%Page returns an uppercase character string containing the current page name. It is typically used to restrict processing to a specific page, which is often necessary, because PeopleCode programs are associated with record definitions that can be shared by multiple pages.

%Panel

%Panel returns an uppercase character string containing the current panel name.



This system variable is supported for compatibility with previous releases of PeopleTools. Future applications should use %OperatorClass instead.

%PanelGroup

%PanelGroup returns an uppercase character string containing the name of the current component, as set in the component definition.



This system variable is supported for compatibility with previous releases of PeopleTools. Future applications should use %Component instead.

%PasswordExpired

This system variable returns a Boolean indicating if the current user's password has expired. This system variable should be used after using SwitchUser, to verify if the password of the user that the user has just switched to is expired or not.

%PermissionLists

This system variable returns an array object containing entries for all the permission lists to which the current user belongs.

%PrimaryPermissionList

This system variable returns a string representing the primary permission list of the current user.

%ProcessProfilePermissionList

This system variable returns the process profile Permission List for the current user.

%PSAuthResult

This system variable returns the result (True or False) of PeopleSoft ID and password authentication for the user signing on.

%Request

%Request returns a reference to the request object. This reference can be used like an object, that is, you can use this as part of a dot notation string. For example:

```
&LOGOUT = %Request.LogoutURL;
```

This constant is only applicable in an Internet Script.



For more information, see Internet Script Classes.

%Response

%Response returns a reference to the response object. This reference can be used like an object, that is, you can use this as part of a dot notation string. For example:

```
&CookieArray = %Response.CookieNames();
```

This constant is only applicable in an Internet Script.



For more information, see Internet Script Classes.

%ResultDocument

This system variable returns a string containing an HTML document displayed to a user. This system variable is used with SwitchUser to pass any messages from the signon process (or Signon PeopleCode) to the user.



This system variable can only be used in Signon PeopleCode.

%Roles

This system variable returns an array object containing entries for all the roles to which the current user belongs.

%RowSecurityPermissionList

This system variable returns a string representing the row-level PermissionList of the current user. The row-level security PermissionList is distinct from the user's primary PermissionList.

%ServerTimeZone

%ServerTimeZone returns the current time zone on the server as a three-character string.

%Session

%Session returns a reference to the current, existing session. If you use %Session successfully, you don't have to use the GetSession function and Connect method. If you don't have a current session, %Session returns NULL.

```
Local ApiObject &MySession

&MySession = %Session;

If Not (&MySession) Then

    /*Not connected, connect manually */

    &MySession = GetSession();

    &MySession.Connect(1, "EXISTING", "", "", 0);

End-If;
```

%SignonUserId

%SignonUserId returns the value the user typed in at the signon page.



This system variable can only be used in Signon PeopleCode.

%SignOnUserPswd

%SignOnUserPswd returns the value the user typed in at the signon page. This value is encrypted. This ensures end-user passwords can't be "captured" by a Signon PeopleCode program.



This system variable can only be used in Signon PeopleCode.

%SQLRows

%SQLRows returns the number of rows affected by the most recent UPDATE, DELETE, or INSERT executed through **SQLExec**.

%SQLRows can also be used after SELECT. It will return 0 if no rows are returned, a non-zero value if one or more rows are returned. In this case, the non-zero value does not indicate the total number of rows returned.

%Time

%Time retrieves the current server time.

%UserDescription

This system variable returns the description (if any) listed for the current user.

%UserId

%UserId returns an uppercase character string containing the user currently logged on. This is typically used to restrict access to records or fields to specific users.

%WLInstanceID

%WLInstanceID returns a string containing the name of the Worklist Instance ID for the current worklist entry. It returns a blank string if the current page was not accessed via a worklist.

%WLName

%WLName returns a string containing the name of the Worklist for the current worklist entry. It returns a blank string if the current page was not accessed via a worklist.

Cheat Sheets

Class List Cheat Sheet

The following tables contain all of the classes, their functions, properties and methods introduced for PeopleTools.

A method marked with an asterisk (*) is the default method for that class.

Properties that are read-only are marked with RO (read-only). If a property isn't marked RO, it's read-write.

Throughout this section, we use typographical conventions to distinguish between different elements of the PeopleCode language, such as bold to indicate function names, italics for arguments, and so on.



For more information, see [Typographical Conventions and Visual Cues](#).

Summary of PeopleCode Classes

This table lists the new classes, their functions, methods and properties, and what each object returns.

Class Name	Functions, Methods and Properties	Returns
AESession	Functions	
	GetAESession(<i>applid</i> , <i>ae_section</i> [, <i>effdt</i>])	AESession object
AESession	Methods	
	AddStep(<i>ae_step_name</i> [, <i>NewStepName</i>])	
	Close()	
	Open(<i>ae_applid</i> , <i>ae_section</i> , [<i>effdt</i>])	AESession object
	Save()	
	SetSQL(<i>action_type_string</i> , <i>string</i>)	

Class Name	Functions, Methods and Properties	Returns
	SetTemplate(<i>ae_applid</i> , <i>ae_section</i>)	
AESection	Properties	
	IsOpen	Boolean, RO
Array	Functions	
	CreateArray(<i>paramlist</i>)	Array object
	CreateArrayRept(<i>val</i> , <i>count</i>)	Array object
	Split(<i>string</i> , <i>separator</i>)	Array object
Array	Methods	
	Clone()	Array object
	Find(<i>value</i>)	Index of an element
	Get(<i>index</i>)	Element of array
	Join([<i>separator</i> [, <i>arraystart</i> , <i>arrayend</i>]])	String
	Next(& <i>index</i>)	Boolean
	Pop()	Value of last array element
	Push(<i>paramlist</i>)	
	Replace(<i>start</i> , <i>length</i> , <i>paramlist</i>)	
	Reverse()	
	Set(<i>index</i>)	
	Shift()	Value of first array element
	Sort([<i>order</i>])	
	Subarray(<i>start</i> , <i>length</i>)	Array object
	Substitute(<i>old_val</i> , <i>new_val</i>)	
	Unshift(<i>paramlist</i>)	
Array	Properties	
	Dimension	Number, RO
	Len	Number
Business Interlink	Function	
	GetBIDoc([<i>XMLString</i>])	BIDoc object
	GetInterlink(INTERLINK. <i>name</i>)	Business Interlink object

Class Name	Functions, Methods and Properties	Returns
Business Interlink	Methods	
	AddInputRow(<i>inputname</i> , <i>value</i>)	Boolean
	BulkExecute(RECORD. <i>inputrec</i> [, RECORD. <i>outputrec</i>] [, <i>user_process_inst</i> <i>user_operid</i>])	Number
	Clear()	
	Execute()	Number
	FetchIntoRecord(RECORD. <i>recname</i> , [, <i>user_process_inst</i> <i>user_operid</i>])	Number
	FetchIntoRowset(& <i>Rowset</i>)	Number
	FetchNextRow(<i>outputname</i> , <i>value</i>)	Output row object
	GetFieldCount()	Number
	GetFieldType(<i>index</i>)	Number
	GetFieldValue(<i>index</i>)	String
	InputRowset(& <i>Rowset</i>)	Number
	MoveFirst()	Boolean
	MoveNext()	Boolean
Business Interlink	Property	
	StopAtError	Boolean
Business Interlink	Configuration Parameters	
	URL	String
	Configuration parameters can be accessed like RW properties. Values depend on type of Interlink plug-in.	
Business Interlink BIDoc	Methods	
	AddAttribute(<i>attributename</i> , <i>attributevalue</i>)	Number
	AddComment(<i>comment</i>)	Number
	AddProcessingInstruction(<i>instruction</i>)	Number
	AddText(<i>text</i>)	Number
	CreateElement(<i>elementname</i>)	BIDoc object

Class Name	Functions, Methods and Properties	Returns
	GenXMLString()	String
	GetAttributeName(<i>attributenum</i>)	String
	GetAttributeValue(<i>{attributenum attributename}</i>)	String
	GetNode(<i>{nodenum nodename}</i>)	BIDoc object
Business Interlink BIDoc	Properties	
	AttributeCount	Number, RO
	ChildNodeCount	Number, RO
	NodeName	String, RO
	NodeType	Number, RO
	NodeValue	String, RO
Component Interface	See Summary of Component Interface API Properties and Methods	
Field	Function	
	GetField(<i>[recname.fieldname]</i>)	Field object
Field	Methods	
	GetLongLabel(<i>LabelID</i>)	String
	GetRelated(<i>recname.fieldname</i>)	Field object
	GetShortLabel(<i>LabelID</i>)	String
	SetCursorPos(PAGE . <i>pagename</i> %Page)	
	SetDefault()	
Field	Properties	
	DisplayFormat	String
	DisplayOnly	Boolean
	EditError	Boolean
	Enabled	Boolean
	FormattedValue	
	IsAltKey	Boolean, RO
	IsAuditFieldAdd	Boolean, RO
	IsAuditFieldChg	Boolean, RO
	IsAuditFieldDel	Boolean, RO
	IsAutoUpdate	Boolean, RO

<i>Class Name</i>	<i>Functions, Methods and Properties</i>	<i>Returns</i>
	IsChanged	Boolean, RO
	IsDateRangeEdit	Boolean, RO
	IsDescKey	Boolean, RO
	IsDupKey	Boolean, RO
	IsEditTable	Boolean, RO
	IsEditXlat	Boolean, RO
	IsFromSearchField	Boolean, RO
	IsInBuf	Boolean, RO
	IsKey	Boolean, RO
	IsListItem	Boolean, RO
	IsRequired	Boolean, RO
	IsSearchItem	Boolean, RO
	IsSystem	Boolean, RO
	IsThroughSearchField	Boolean, RO
	IsUseDefaultLabel	Boolean, RO
	IsYesNo	Boolean, RO
	Label	String
	LongTranslateValue	Any
	MessageNumber	Number, RO
	MessageSetNumber	Number, RO
	Name	String, RO
	OriginalValue	Depends on field
	ParentRecord	Record Object, RO
	SeachDefault	Boolean
	SearchEdit	Boolean
	ShortTranslateValue	Any
	ShowRequiredFieldCue	Boolean
	StoredFormat	String, RO
	Style	String
	Type	String, RO
	Value	Depends on field
	Visible	Boolean

Class Name	Functions, Methods and Properties	Returns
File	Functions	
	FileExists(<i>filename</i> [, <i>pathtype</i>])	Boolean
	FindFiles(<i>filespec_pattern</i> [, <i>pathtype</i>])	Array object
	GetFile(<i>filename</i> , <i>mode</i> [, <i>charset</i>] [, <i>pathtype</i>])	File object
File	Methods	
	Close()	
	CreateRowset()	Rowset object
	GetPosition()	Number
	Open(<i>filename</i> , <i>mode</i> [, <i>charset</i>] [, <i>pathtype</i>])	
	ReadLine(<i>string</i>)	Boolean
	ReadRowset()	Rowset object
	SetFileId(<i>fileid</i> , <i>position</i>)	
	SetFileLayout(FILELAYOUT. <i>filelayoutname</i>)	Boolean
	SetPosition(<i>position</i>)	
	WriteLine(<i>string</i>)	
	WriteRaw(<i>RawBinary</i>)	
	WriteRecord(<i>record</i>)	Boolean
	WriteRowset(<i>rowset</i>)	Boolean
	WriteString(<i>string</i>)	
File	Properties	
	CurrentRecord	String, RO
	IgnoreInvalidId	Boolean
	IsError	Boolean, RO
	IsNewFileId	Boolean, RO
	IsOpen	Boolean, RO
	Name	String, RO
	RecTerminator	String
Grid	Function	
	GetGrid(PAGE. <i>pagename</i> , <i>gridname</i> [, <i>occursnumber</i>])	Grid object

Class Name	Functions, Methods and Properties	Returns
Grid	Methods	
	*GetColumn(<i>columnname</i>)	Grid column object
	Properties	
	<i>columnname</i>	Grid column object
	Label	String
GridColumn	Properties	
	Enabled	Boolean
	Label	String
	Name	String, RO
	Visible	Boolean
Internet Script	See Summary of Methods and Properties for Internet Script Classes	
Java	Functions	
	CreateJavaArray(<i>ElementClassName</i> [], <i>NumberOfElements</i>)	Java object
	CreateJavaObject(<i>ClassName</i> [<i>ConstructorParams</i>])	Java Object
	GetJavaClass(<i>ClassName</i>)	Java Object
Message	Functions	
	AddSystemPauseTimes(<i>StartDay</i> , <i>StartTime</i> , <i>EndDay</i> , <i>EndTime</i>)	Boolean
	CreateMessage(MESSAGE. <i>messagename</i>)	Message object
	DeleteSystemPauseTimes(<i>StartDay</i> , <i>StartTime</i> , <i>EndDay</i> , <i>EndTime</i>)	Boolean
	GetMessage()	Message object
	GetMessageInstance(<i>pub_id</i> , <i>pub_nodename</i> , <i>channelname</i>)	Message object
	GetPubContractInstance(<i>pub_id</i> , <i>pub_nodename</i> , <i>channelname</i> , <i>sub_nodename</i>)	Message object
	GetSubContractInstance(<i>pub_id</i> , <i>pub_nodename</i> , <i>channelname</i> , <i>messagename</i> , <i>sub_name</i>)	Message object
	PingNode(<i>MsgNodeName</i>)	Array of number
	ReturnToServer({TRUE FALSE &NODE_ARRAY})	

Class Name	Functions, Methods and Properties	Returns
	SetChannelStatus(<i>ChannelName</i> , <i>Status</i>)	Boolean
Message	Methods	
	Cancel()	
	Clone()	Message object
	CopyRowset(<i>source_rowset</i> , [, <i>record_list</i>])	
	CopyRowsetDelta(<i>source_rowset</i> , [, <i>record_list</i>])	
	ExecuteEdits([<i>editlevels</i>])	
	GenXMLString()	String
	GenFormattedXMLString()	String
	GetFormattedRawXML([<i>version</i>])	String
	GetMessageVersion(<i>version</i>)	Boolean
	GetRawXML([<i>version</i>])	String
	GetRowset()	Rowset object
	LoadXMLString(<i>XMLString</i>)	
	Publish()	Boolean
	Resubmit()	
	SetEditTable(% <i>PromptTable</i> , RECORD. <i>recname</i>)	
	SetStatus(<i>status</i>)	
	Update()	
Message	Properties	
	ChannelName	String, RO
	DoNotPubToNodeName	RO
	IsActive	Boolean, RO
	IsDelta	Boolean, RO
	IsEditError	Boolean, RO
	IsLocal	Boolean, RO
	Name	String, RO
	PubID	String, RO
	PubNodeName	String, RO
	Size	Number, RO

Class Name	Functions, Methods and Properties	Returns
	SubName	String, RO
	SubscriptionProcessID	String, RO
Page	Function	
	GetPage(PAGE.pagename)	Page object
Page	Properties	
	DisplayOnly	Boolean, RO
	Name	String, RO
	Visible	Boolean
PortalRegistry	See Summary of Classes, Methods and Properties for PortalRegistry Classes.	
ProcessRequest	Function	
	CreateProcessRequest()	Process request object
ProcessRequest	Method	
	Schedule()	
ProcessRequest	Properties	
	EmailAttachLog	String
	EmailSubject	String
	EmailText	String
	JobName	String
	LanguageCd	String
	OutDest	String
	OutDestFormat	String
	OutDestType	String
	ProcessInstance	Number, RO
	ProcessName	String
	ProcessType	String
	RunControlID	String
	RunDateTime	DateTime
	RunLocation	String
	RunRecurrence	String
	Status	Number, RO
Record	Functions	

Class Name	Functions, Methods and Properties	Returns
	CreateRecord(RECORD. <i>recname</i>)	Record object
	GetRecord([RECORD. <i>recname</i>])	Record object
Record	Methods	
	CompareFields(<i>RecordObject</i>)	Boolean
	CopyChangedFieldsTo(<i>RecordObj</i>)	
	CopyFieldsTo(<i>RecordObject</i>)	
	Delete()	Boolean
	ExecuteEdits([<i>EditLevel</i>])	
	*GetField(<i>n</i> /FIELD. <i>fieldname</i>)	Field Object
	Insert()	Boolean
	SelectByKey()	Boolean
	SetDefault()	
	SetEditTable(% <i>PromptField</i> , RECORD. <i>recname</i>)	
	Update([<i>KeyRecord</i>])	Boolean
Record	Properties	
	FieldCount	Number, RO
	<i>fieldname</i>	Field Object, RO
	IsChanged	Boolean, RO
	IsDeleted	Boolean, RO
	IsEditError	Boolean, RO
	Name	String, RO
	ParentRow	Row Object, RO
	RelLangRecName	String, RO
Row	Function	
	GetRow()	Row object
Row	Methods	
	CopyTo(<i>rowobject</i>)	Row Object
	GetNextEffRow()	Row Object
	GetPriorEffRow()	Row Object
	*GetRecord(<i>n</i> /RECORD. <i>recname</i>)	Record Object
	GetRowSet(<i>n</i> /SCROLL. <i>scrollname</i>)	Rowset Object

Class Name	Functions, Methods and Properties	Returns
	<i>scrollname(n)</i>	Row Object
Row	Properties	
	ChildCount	Number, RO
	DeleteEnabled	Boolean
	IsChanged	Boolean, RO
	IsDeleted	Boolean, RO
	IsEditError	Boolean, RO
	IsNew	Boolean, RO
	ParentRowSet	Rowset object, RO
	<i>recname</i>	Record object, RO
	RecordCount	Number, RO
	RowNumber	Number, RO
	Selected	Boolean
	Style	String
	Visible	Boolean
Rowset	Functions	
	CreateRowset({RECORD. <i>recname</i> &Rowset} [, {FIELD. <i>fieldname</i> , RECORD. <i>recname</i> &Rowset}] . . .)	Rowset object
	GetLevel0()	Rowset object
	GetRowset([SCROLL. <i>scrollname</i>])	Rowset object
Rowset	Methods	
	CopyTo(&DestRowset [, RECORD. <i>SourceRecname</i> , RECORD. <i>DestRecname</i>] . . .)	
	DeleteRow(<i>n</i>)	Boolean
	Fill([<i>wherestring</i> [, <i>bindvalue</i>] . . .])	Number
	FillAppend([<i>wherestring</i> [, <i>bindvalue</i>] . . .])	Number
	Flush()	
	FlushRow(<i>n</i>)	
	GetCurrEffRow()	Row Object
	*GetRow(<i>n</i>)	Row Object
	HideAllRows()	

Class Name	Functions, Methods and Properties	Returns
	InsertRow(<i>n</i>)	Boolean
	Refresh()	
	Select([<i>parmlist</i>], RECORD. <i>selrecord</i> [, <i>wherestr</i> , <i>bindvars</i>])	Number
	SelectNew([<i>parmlist</i>], RECORD. <i>selrecord</i> [, <i>wherestr</i> , <i>bindvars</i>])	Number
	SetDefault(<i>recname.fieldname</i>)	
	ShowAllRows()	
	Sort([<i>paramlist</i> ,] <i>sort_fields</i>)	
Rowset	Properties	
	ActiveRowCount	Number, RO
	DBRecordName	String, RO
	DeleteEnabled	Boolean
	EffDt	Date, RO
	EffSeq	Number, RO
	InsertEnabled	Boolean
	IsEditError	Boolean, RO
	Level	Number, RO
	Name	String, RO
	ParentRow	Row Object, RO
	ParentRowSet	Rowset Object, RO
	RowCount	Number, RO
	TopRowNumber	Number, RO
Session	Many objects are instantiated from a session object. For more information, see Summary of Session Class Properties and Methods	
Session	Function	
	GetSession()	Session object
Session	Methods	
	Connect(<i>version</i> , {EXISTING <i>ConnectID:Port</i> }, <i>UserID</i> , <i>Password</i> , <i>EXTAUTH</i>)	Boolean
	Disconnect()	Boolean

Class Name	Functions, Methods and Properties	Returns
	FindCompIntfc(<i>partial_name</i>)	Component Interface collection
	FindPortalRegistries(<i>partial_name</i>)	PortalRegistry collection
	FindQueryDBRecords()	QueryDBRecord collection
	FindQueries()	Query collection
	FindQueriesByDateRange(<i>StartDateString</i> , <i>EndDateString</i>)	Query collection
	FindTree(<i>SetId</i> , <i>UserKey</i> , <i>TreeName</i> , <i>EffDt</i> , <i>BranchName</i>)	Tree collection
	FindTreeStructure(<i>StructureID</i>)	Tree structure collection
	GetCompIntfc([CompIntfc.] <i>name</i>)	Component Interface object
	GetPortalRegistry()	PortalRegistry object
	GetQuery()	Query object
	GetSearchQuery()	SearchQuery object
	GetTree(<i>SetId</i> , <i>UserKey</i> , <i>TreeName</i> , <i>EffDt</i> , <i>BranchName</i>)	Tree object
	GetTreeStructure(<i>StructureID</i>)	Tree structure
Session	Properties	
	ErrorPending	Boolean, RO
	PSMessages	PSMessage collection object
	RegionalSettings	Regional settings object
	Repository	Repository object
	SuspendFormating	Boolean
	TraceSettings	Trace settings object
	WarningPending	Boolean, RO
SQL	Functions	
	CreateSQL(<i>sqlstring</i> [, <i>paramlist</i>])	
	DeleteSQL([SQL.] <i>sqlname</i> [, <i>dbtype</i> [, <i>effdt</i>]]))	Boolean
	FetchSQL([SQL.] <i>sqlname</i> [, <i>dbtype</i> [,	String

Class Name	Functions, Methods and Properties	Returns
	<i>effdt</i>]])	
	GetSQL(SQL. <i>sqlname</i> [, <i>paramlist</i>])	SQL object
	StoreSQL(<i>sqlstring</i> , [SQL.] <i>sqlname</i> [, <i>dbtype</i> [, <i>effdt</i>]])	
SQL	Methods	
	Close()	Boolean
	Execute(<i>paramlist</i>)	Boolean
	Fetch(<i>paramlist</i>)	Boolean
	Open(<i>sql</i> [, <i>paramlist</i>])	
SQL	Properties	
	BulkMode	Boolean
	IsOpen	Boolean, RO
	RowsAffected	Number, RO
	Status	Number, RO
	TraceName	String
	Value	String, RO
Tree	See Summary of Methods for Tree Classes API and Summary of Properties for Tree Classes API	

Summary of Session Class Properties and Methods

The session object controls access to the PeopleSoft APIs, as well as the environment in which they run. This table contains the properties and methods for controlling the environment. The API objects (business components, trees, and so on) are listed, but have their own separate tables.

Objects	Methods and Properties	Returns
Component Interfaces and CI Collection	See Summary of Component Interface API Properties and Methods	
PortalRegistry classes	See Summary of Classes, Methods and Properties for PortalRegistry Classes	
Query classes	See Summary of Classes, Methods and Properties for Query Classes	
PSMessage Collection	Methods	

	DeleteAll()	Boolean
	DeleteItem(<i>number</i>)	Boolean
	First()	PSMessage object
	Item(<i>number</i>)	PSMessage object
	Next()	PSMessage object
PSMessage Collection	Property	
	Count	Number, RO
PSMessage	Properties	
	Code	String, RO
	ExplainText	String, RO
	MessageNumber	Number, RO
	MessageSetNumber	Number, RO
	Source	String, RO
	Text	String, RO
	Type	Number, RO
Regional Settings	Properties	
	ClientTimeZone	String
	CurrencyFormat	Number
	CurrencySymbol	String
	DateFormat	Number
	DateSeparator	String
	DecimalSymbol	String
	DigitGroupingSymbol	String
	LanguageCD	String
	UseLocalTime	Boolean
	1159Separator	String
	2359Separator	String
Repository	See Summary of Repository Methods and Summary of Repository Properties	
TraceSettings	Properties	
	API	Boolean
	COBOLStmtTimings	Boolean

	ConDisRollbackCommit	Boolean
	DBSpecificCalls	Boolean
	ManagerInfo	Boolean
	NetworkServices	Boolean
	NonSSBs	Boolean
	OutputUNICODE	Boolean
	PCExtFcnCalls	Boolean
	PCFcnReturnValues	Boolean
	PCFetchedValues	Boolean
	PCIntFcnCalls	Boolean
	PCListProgram	Boolean
	PCParameterValues	Boolean
	PCProgramStatements	Boolean
	PCStack	Boolean
	PCStartOfPrograms	Boolean
	PCTraceProgram	Boolean
	PCVariableAssignments	Boolean
	RowFetch	Boolean
	SQLStatement	Boolean
	SQLStatementVariables	Boolean
	SSBs	Boolean
	SybBindInfo	Boolean
	SybFetchInfo	Boolean
	TraceFile	String
Tree, Tree Collection, Tree Structure, Tree Structure Collection	See Summary of Methods for Tree Classes API and Summary of Properties for Tree Classes API	

Summary of Component Interface API Properties and Methods

Component Interface collections and Component Interface objects are instantiated from a session object. CompIntfPropInfo and Data Collections are instantiated from a Component Interface object.

<i>Class Name</i>	<i>Functions, Methods and Properties</i>	<i>Returns</i>
Component Interface	Methods	

Class Name	Functions, Methods and Properties	Returns
Collection		
	First()	Component Interface structure (no data)
	Item(<i>number</i>)	Component Interface structure (no data)
	Next()	Component Interface structure (no data)
Component Interface Collection	Properties	
	Count	Number, RO
Component Interface	Methods	
	Cancel()	Boolean
	CopyRowset(&rowset [,InitialRow] [, record_list])	
	CopyRowsetDelta(&rowsetI, [,InitialRow] [, record_list])	
	CopySetupRowset(&rowset [,InitialRow] [, record_list])	
	CopySetupRowsetDelta(&rowsetI, [,InitialRow] [, record_list])	
	Create()	Component Interface with data
	Find()	Component Interface collection
	Get()	Component Interface with data
	Save()	Boolean
Component Interface	Properties	
	CreateKeyInfoCollection	Component InterfacePropertyInfo collection object, RO
	FindKeyInfoCollection	Component InterfacePropertyInfo collection object, RO
	GetHistoryItems	Boolean
	GetKeyInfoCollection	Component InterfacePropertyInfo collection object, RO

Class Name	Functions, Methods and Properties	Returns
	InteractiveMode	Boolean
	PropertyInfoCollection	Component InterfacePropertyInfo collection object, RO
	StopOnFirstError	Boolean
	Every user-defined property for a business component definition can be used like a property on the instantiated object .	Depends on property
CompIntfPropInfo Collection	Methods	
	First()	Component InterfacePropertyInfo object
	Item(<i>number</i>)	Component InterfacePropertyInfo object
	Next()	Component InterfacePropertyInfo object
CompIntfPropInfo Collection	Properties	
	Count	Number, RO
CompIntfPropInfo	Properties	
	Format	String, RO
	IsCollection	Boolean, RO
	Key	Boolean, RO
	LabelLong	String, RO
	LabelShort	String, RO
	Name	String, RO
	Prompt	Boolean, RO
	PropertyInfoCollection	Component InterfacePropertyInfo collection object
	Required	Boolean, RO
	Type	String, RO
	Xlat	Boolean, RO

Class Name	Functions, Methods and Properties	Returns
	YesNo	Boolean, RO
Data Collection	Methods	
	CurrentItem()	data collection item
	DeleteItem(<i>number</i>)	
	GetEffectiveItem(<i>DateString</i> , <i>SeqNo</i>)	data collection item
	GetEffectiveItemNum(<i>DateString</i> , <i>SeqNo</i>)	Number
	InsertItem(<i>number</i>)	data collection item
	Item(<i>number</i>)	data collection item
	ItemByKey(<i>key_values</i>)	data collection item
Data Collection	Properties	
	Count	Number, RO
	CurrentItemNumber	Number, RO

Summary of Search API Methods and Properties

Search API objects are instantiated from the GetSearchQuery property on a session object, or the GetSearchQuery method on a PortalRegistry object.

Class Name	Methods and Properties	Returns
SearchQuery	Method	
	Execute(<i>Start</i> , <i>Size</i>)	SearchResult collection
	Properties	
	HitCount	Number, RO
	IndexName	String
	Language	String
	ProcessedCount	Number, RO
	QueryText	String
SearchResult Collection	Methods	
	First()	SearchResult
	Item(<i>number</i>)	SearchResult
	Next()	SearchResult
SearchResult Collection	Properties	

Class Name	Methods and Properties	Returns
	Count	Number, RO
SearchResult	Properties	
	Key	String, RO
	Score	Number, RO
	SearchFields	SearchFields collection, RO
SearchField Collection	Methods	
	First()	SearchField
	ItemByName(<i>Name</i>)	SearchField
	Next()	SearchField
SearchField Collection	Properties	
	Count	Number, RO

Summary of Repository Methods

Repository objects are instantiated from the Repository property on a session object. This table contains a list of all the Repository objects plus their methods. Methods that can be used by a class are marked with an "X".

Method	Bindings collection	Namespaces collection	Classinfo collection	MethodInfo collection	Propertyinfo collection
CreateObject (<i>classname</i>)		X			
Generate()	X				
Item(<i>number</i>)	X	X	X	X	X
ItemByName (<i>name</i>)		X	X	X	X

Summary of Repository Properties

Repository collection objects are instantiated from the Repository property on a session object. This table contains a list of all the Repository objects plus their properties. All properties are read-only.

Property	Rep coll/ object	Binding coll/ object	Namespace coll/ object	Classinfo coll/ object	MethodInfo coll/ object	PropertyInfo coll/ object
Arguments					RO	

<i>Property</i>	<i>Rep coll/ object</i>	<i>Binding coll/ object</i>	<i>Namespace coll/ object</i>	<i>Classinfo coll/ object</i>	<i>MethodInfo coll/ object</i>	<i>PropertyInfo coll/ object</i>
Bindings	RO					
Classes			RO			
Count		RO	RO	RO	RO	RO
Documentati on				RO	RO	RO
Generate		RO				
Methods				RO		
Name		RO	RO	RO	RO	RO
Namespaces	RO					
Properties				RO		
Type					RO	RO
Usage						RO

Summary of Methods and Properties for Internet Script Classes

<i>Class Name</i>	<i>Methods and Properties</i>	<i>Returns</i>
Request	Methods	
	GetContentBody()	String
	GetCookieNames()	Array of string
	GetCookieValue(<i>name</i>)	String
	GetHeader(<i>name</i>)	String
	GetHeaderNames()	Array of string
	GetHelpURL	String
	GetParameter(<i>name</i>)	String
	GetParameterNames()	Array of string
	GetParameterValues()	Array of string
Request	Properties	
	BrowserPlatform	String, RO
	BrowserType	String, RO
	BrowserVersion	String, RO
	ExpireMeta	String, RO
	FullURI	String, RO

Class Name	Methods and Properties	Returns
	HTTPMethod7	String, RO
	LogoutURL	String, RO
	PathInfo	String, RO
	Protocol	String, RO
	QueryString	String, RO
	RequestURI	String, RO
	Scheme	String, RO
	ServerName	String, RO
	ServerPort	String, RO
	Timeout	Integer, RO
Response	Methods	
	Clear()	
	CreateCookie(<i>name</i>)	Cookie object
	GetCookie(<i>name</i>)	Cookie object
	GetCookieNames()	Array of string
	GetHeaderNames()	Array of string
	GetImageURL(<i>ImageName</i>)	String
	GetJavaScriptURL(<i>HTML.Name</i>)	String
	GetStyleSheetURL(<i>STYLESHEET.name</i>)	String
	RedirectURL(<i>name</i>)	
	SetContentType(<i>Type</i>)	
	SetHeader(<i>name, value</i>)	
	Write(<i>HTML</i>)	
	WriteLine(<i>HTML</i>)	
Cookie	Properties	
	Domain	String
	MaxAge	Number
	Name	String, RO
	Path	String
	Secure	Boolean
	Value	String

Summary of Classes, Methods and Properties for PortalRegistry Classes

Class Name	Methods and Properties	Returns
PortalRegistry	Methods	
	BuildSearchIndex(<i>Language</i>)	Boolean
	Close()	Boolean
	Create(<i>RegistryName</i>)	Boolean
	Delete(<i>RegistryName</i>)	Boolean
	FindCRefByName(<i>Name</i>)	ContentReference object
	FindCRefByURL(<i>URL</i>)	ContentReference object
	FindFolderByName(<i>Name</i>)	Folder object
	GetQualifiedURL(<i>ContentProvider</i> , <i>URL</i>)	URL string
	GetSearchQuery()	SearchQuery object
	Open(<i>RegistryName</i>)	Boolean
	Save()	Boolean
PortalRegistry	Properties	
	ContentProviders	ContentProvider collection, RO
	DefaultTemplate	String
	Description	String
	Name	String, RO
	RootFolder	Folder object, RO
	TemplateObject	ContentReference object, RO
PortalRegistry Collection	Methods	
	First()	PortalRegistry object
	Item(<i>number</i>)	PortalRegistry object
	Next()	PortalRegistry object
PortalRegistry Collection	Property	
	Count	Number, RO
ContentProvider	Properties	
	DefaultTemplate	String

Class Name	Methods and Properties	Returns
	Description	String
	Name	String, RO
	TemplateObject	ContentReference object, RO
	URI	String
ContentProvider Collection	Methods	
	DeleteItem(<i>ContentProviderName</i>)	Boolean
	First()	ContentProvider object
	InsertItem(<i>ContentProviderName</i>)	ContentProvider object
	ItemByName(<i>ContentProviderName</i>)	ContentProvider object
	ItemByURI(<i>URI</i>)	ContentProvider object
	Next()	ContentProvider object
ContentProvider Collection	Property	
	Count	Number, RO
Folder	Method	
	Save()	Boolean
Folder	Properties	
	Attributes	Attribute collection object, RO
	Author	String, RO
	AuthorAccess	Boolean
	Authorized	Boolean
	CascadedPermissions	PermissionValue collection, RO
	ContentRefs	ContentReference Collection, RO
	CreationDate	String, RO
	Description	String
	Folders	Folder collection, RO

Class Name	Methods and Properties	Returns
	Label	String
	Name	String, RO
	ParentName	String, RO
	Path	String, RO
	Permissions	PermissionValue collection, RO
	Product	String
	SequenceNumber	Number
	PublicAccess	Boolean
	ValidFrom	String
	ValidTo	String
Folder Collection	Methods	
	DeleteItem(<i>FolderName</i>)	Boolean
	First()	Folder object
	InsertItem(<i>FolderName</i>)	Folder object
	ItemByName(<i>FolderName</i>)	Folder object
	Next()	Folder object
Folder Collection	Properties	
	Count	Number, RO
ContentReference	Method	
	Save()	Boolean
ContentReference	Properties	
	Attributes	Attribute collection object, RO
	Author	String, RO
	AuthorAccess	Boolean
	Authorized	Boolean
	CascadedPermissions	PermissionValue collection, RO
	ContentProvider	String
	CreationDate	String, RO
	Data	String
	Description	String

Class Name	Methods and Properties	Returns
	Label	String
	Name	String, RO
	ParentName	String, RO
	Path	String, RO
	Permissions	PermissionValue collection, RO
	Product	String
	PublicAccess	Boolean
	QualifiedURL	String
	SequenceNumber	Number
	StorageType	String
	Template	String
	TemplateObject	ContentReference object, RO
	TemplateType	String
	URL	String
	URLType	String
	UsageType	String
	ValidFrom	String
	ValidTo	String
ContentReference Collection	Methods	
	DeleteItem(<i>ContentReferenceName</i>)	Boolean
	First()	ContentReference object
	InsertItem(<i>ContentReferenceName</i> , <i>ContentProvider</i> , <i>RelativeURL</i>)	ContentReference object
	ItemByName(<i>ContentReferenceName</i>)	ContentReference object
	Next()	ContentReference object
ContentReference Collection	Property	
	Count	Number, RO
AttributeValue	Properties	

Class Name	Methods and Properties	Returns
	Label	String
	Name	String, RO
	Translatable	Boolean
	Value	String
Attribute Collection	Methods	
	DeleteItem(<i>AttributeValueName</i>)	Boolean
	First()	AttributeValue object
	InsertItem(<i>AttributeValueName</i>)	AttributeValue object
	ItemByName(<i>AttributeValueName</i>)	AttributeValue object
	Next()	AttributeValue object
Attribute Collection	Property	
	Count	Number, RO
PermissionValue	Properties	
	Cascade	Boolean
	Name	String
PermissionValue Collection	Methods	
	DeleteItem(<i>PermissionValueName</i>)	Boolean
	First()	PermissionValue object
	InsertItem(<i>PermissionValueName</i>)	PermissionValue object
	ItemByName(<i>PermissionValueName</i>)	PermissionValue object
	Next()	PermissionValue object
PermissionValue Collection	Property	
	Count	Number, RO

Summary of Classes, Methods and Properties for Query Classes

Class Name	Methods and Properties	Returns
Query Collection	Methods	

Class Name	Methods and Properties	Returns
	First()	Query object
	Item(<i>Number</i>)	Query object
	ItemByName(<i>Name</i>)	Query object
	Next()	Query object
Query Collection	Property	
	Count	Number, RO
QueryObject	Methods	
	AddAllFields(<i>QueryRecord</i>)	
	AddCriteria()	QueryCriteria object
	AddExpression()	QueryExpression object
	AddQueryOutputField(<i>QueryRecord</i> , <i>index</i>)	QueryField object
	AddQuerySelectedField(<i>QueryRecord</i> , <i>index</i>)	QueryField object
	AddQueryRecord(<i>QueryRecordName</i>)	QueryRecord object
	Close()	
	Create(<i>QueryName</i> , <i>Pubic</i> , <i>Distinct</i> , <i>Type</i> , <i>Description</i> , <i>LongDescription</i>)	Query object
	Delete()	Number
	DeleteCriteria(<i>index</i>)	Number
	DeleteExpression(<i>Index</i>)	Number
	DeleteField(<i>Index</i>)	Number
	Open(<i>QueryName</i> , <i>Public</i> , <i>Update</i>)	Number
	QueryOutputFields()	QueryField collection
	QueryRecords()	QueryRecord collection
	QuerySelectedFields()	QueryField collection
	Rename(<i>NewQueryName</i>)	Number
	Save()	Number
Query Object	Properties	
	Criteria	QueryCriteria collection, RO
	Description	String

Class Name	Methods and Properties	Returns
	Distinct	Boolean
	Expressions	QueryExpression collection, RO
	LongDescription	String
	Metadata	Metadata collection, RO
	Name	String, RO
	PermissionList	PermissionList collection, RO
	Public	Boolean
	SQL	String, RO
	Type	Number
QueryRecord Collection	Methods	
	First()	QueryRecord object
	Item(<i>Number</i>)	QueryRecord object
	ItemByName(<i>Name</i>)	QueryRecord object
	Next()	QueryRecord object
QueryRecord Collection	Property	
	Count	Number, RO
QueryRecord	Properties	
	Alias	String
	Name	String, RO
	QueryFields	QueryFields collection, RO
	RecordHierarchy	QueryRecordHierarchy collection, RO
QueryField Collection	Methods	
	First()	QueryField object
	Item(<i>Number</i>)	QueryField object
	ItemByName(<i>Name</i>)	QueryField object
	Next()	QueryField object
QueryField	Property	

Class Name	Methods and Properties	Returns
Collection		
	Count	Number, RO
QueryField	Properties	
	Aggregate	Number
	ColumnNumber	Number
	Format	Number
	HeadingText	String
	HeadingType	String
	HeadingUniqueFieldName	String
	Name	String, RO
	OrderByDirection	Number
	QueryDBRecordField	QueryDBRecordField, RO
	QueryRecord	QueryRecord object, RO
	QueryRelatedRecords	QueryRecordHierarchy object, RO
	TranslateEffDtLogic	Number
	TranslateOption	Number
	Type	Number
QueryCriteria Collection	Methods	
	First()	QueryCriteria object
	Item(<i>Number</i>)	QueryCriteria object
	Next()	QueryCriteria object
QueryCriteria Collection	Property	
	Count	Number, RO
QueryCriteria	Methods	
	AddExpr1Expression()	QueryExpression object
	AddExpr1Field(<i>QueryRecord</i> , <i>Index</i>)	QueryField object
	AddExpr2Expression()	QueryExpression object

Class Name	Methods and Properties	Returns
	AddExpr2Field(<i>QueryRecord</i> , <i>index</i>)	QueryField object
	AddExpr2Subquery()	Query object
QueryCriteria	Properties	
	Expr1Expression	QueryExpression object, RO
	Expr1Field	QueryField object, RO
	Expr1Type	Number
	Expr2Constant	String
	Expr2Expression	QueryExpression object, RO
	Expr2Field	QueryField object, RO
	Expr2Subquery	Query object, RO
	Expr2Type	Number
	Logical	Number
	Operator	Number
QueryExpression Collection	Methods	
	First()	QueryExpression object
	Item(<i>Number</i>)	QueryExpression object
	Next()	QueryExpression object
QueryExpression Collection	Property	
	Count	Number, RO
QueryExpression	Properties	
	Aggregate	Number
	Decimal	Number
	Length	Number
	Text	String
	Type	Number
QueryExpression Collection	Methods	

Class Name	Methods and Properties	Returns
	First()	QueryExpression object
	Item(<i>Number</i>)	QueryExpression object
	Next()	QueryExpression object
QueryExpression Collection	Property	
	Count	Number, RO
QueryRecordHierarchy Collection	Methods	
	First()	QueryRecordHierarchy object
	Item(<i>Number</i>)	QueryRecordHierarchy object
	ItemByName(<i>Name</i>)	QueryRecordHierarchy object
	Next()	QueryRecordHierarchy object
QueryRecordHierarchy Collection	Property	
	Count	Number, RO
QueryRecordHierarchy	Properties	
	Description	String, RO
	Level	Number, RO
	Name	String, RO
	ParentFlag	Number, RO
Metadata Collection	Methods	
	First()	Metadata object
	Item(<i>Number</i>)	Metadata object
	ItemByName(<i>Name</i>)	Metadata object
	Next()	Metadata object
Metadata Collection	Property	

Class Name	Methods and Properties	Returns
	Count	Number, RO
Metadata	Properties	
	Name	String, RO
	Value	String, RO
PermissionList Collection	Methods	
	First()	PermissionList object
	Item(<i>Number</i>)	PermissionList object
	ItemByName(<i>Name</i>)	PermissionList object
	Next()	PermissionList object
PermissionList Collection	Property	
	Count	Number, RO
PermissionList	Property	
	Name	
QueryDBRecord Collection	Methods	
	First()	QueryDBRecord object
	Item(<i>Number</i>)	QueryDBRecord object
	ItemByName(<i>Name</i>)	QueryDBRecord object
	Next()	QueryDBRecord object
QueryDBRecord Collection	Property	
	Count	Number, RO
QueryDBRecord	Properties	
	Description	String, RO
	Name	String, RO
	QueryDBRecordFields	QueryDBRecordFields collection
QueryDBRecordField Collection	Methods	

Class Name	Methods and Properties	Returns
	First()	QueryDBRecordField object
	Item(<i>Number</i>)	QueryDBRecordField object
	ItemByName(<i>Name</i>)	QueryDBRecordField object
	Next()	QueryDBRecordField object
QueryDBRecordField Collection	Property	
	Count	Number, RO
QueryDBRecordField	Properties	
	Decimal	Number, Ro
	Format	Number, Ro
	Length	Number, RO
	LongName	String, RO
	Name	String, RO
	ShortName	String, RO
	Type	Number, RO

Summary of Methods for Tree Classes API

Tree and tree structures are instantiated from a session object. This table contains a list of all the tree classes API plus their methods. Methods that can be used by a class are marked with an "X".

Methods	Branch Coll	Leaf	Level Coll/ Level	Node	Tree/Tree Coll	Tree Structure / Tree Structure Coll
Add			X			
Audit					X	
AuditByName					X	
Branch				X		
Close					X	X
Copy					X	X
Create			X		X	X

Methods	Branch Coll	Leaf	Level Coll/ Level	Node	Tree/Tree Coll	Tree Structure / Tree Structure Coll
Cut		X		X		
Delete		X	X	X	X	X
DeleteByName				X		
DeleteByRange		X				
Exists					X	
Expand				X		
FindLeaf					X	
FindNode					X	
FindRoot					X	
InsertChildLeaf				X		
InsertChildNode				X		
InsertChildRecord				X		
InsertDynChildLeaf				X		
InsertDynSib		X				
InsertRoot					X	
InsertSib		X		X		
InsertSibRecord				X		
Item	X		X		X	X
MoveAsChild		X		X		
MoveAsChildByName		X		X		
MoveAsSib		X		X		
MoveAsSibByName				X		
MoveAsSibByRange		X				
Open					X	X
PasteChild				X		
PasteSib		X		X		
Remove			X			

Methods	Branch Coll	Leaf	Level Coll/ Level	Node	Tree/Tree Coll	Tree Structure / Tree Structure Coll
Rename				X	X	X
Save			X		X	X
SaveAs					X	
SaveAsDraft					X	
SaveDraft					X	
SwitchLevel				X		
Unbranch				X		

Summary of Properties for Tree Classes API

Tree and tree structures are instantiated from a session object. This table contains a list of all the tree classes API plus their properties. Properties that are read-only are marked with RO (read-only). Properties that are valid for a class are marked with an "X". Also, see the footnotes following the table.

Properties	Branch Coll	Leaf	Level	Node	Tree	Tree Structure
AllChildCount				RO		
AllChildNodeCount				RO		
AllValues					X	
AllValuesAudit			X			
AuditDetails					X	
Branches					RO	
BranchLevel					RO	
BranchName					RO	
Category					X	
CheckLeafUserData					X	
ChildLeafCount				RO		
ChildNodeCount				RO		
Count	X		X ²		X ²	X ²
Description			X	RO	X	X
DetailComponent						X
DetailField						X

Properties	Branch Coll	Leaf	Level	Node	Tree	Tree Structure
DetailMenu						X
DetailMenuBar						X
DetailMenuItem						X
DetailPage						X
DetailRecord						X
DuplicateLeaves					X	
Dynamic		X				
EffDt					X	
First	X		X ²		X ²	X ²
FirstChildLeaf				RO		
FirstChildNode				RO		
HasChildLeaves				RO		
HasChildNodes				RO		
HasChildren				RO		
HasDetailRanges					RO	
HasNextSib		RO		RO		
HasPrevSib		RO		RO		
IndirectionMethod						X
IsBranched				RO	RO	
IsChanged		RO		RO	RO	
IsDeleted		RO		RO		
IsInserted		RO		RO		
IsOpen					RO	
IsQueryTree					RO	
IsRoot				RO		
IsValid					RO	
KeyBranchName ¹					RO	
KeyEffDt ¹					RO	
KeyName ¹					RO	RO
KeySetId ¹					RO	
KeyUserKeyValue ¹					RO	

Properties	Branch Coll	Leaf	Level	Node	Tree	Tree Structure
Last	X		X ²		X ²	X ²
LastChildLeaf				RO		
LastChildNode				RO		
LeafCount					RO	
LevelComponent						X
LevelCount					RO	
LevelMenu						X
LevelMenuBar						X
LevelMenuItem						X
LevelNumber				X		
LevelPage						X
LevelRecord						X
Levels					RO	
LevelUse					X	
Name			X	X	X	X
Next	X		X ²		X ²	X ²
NextSib		RO		RO		
NodeComponent						X
NodeCount					RO	
NodeField						X
NodeMenu						X
NodeMenuBar						X
NodeMenuItem						X
NodePage						X
NodeRecord						X
NodeUserKeyField						X
Number			X			
Parent		RO		RO		
ParentLevel					RO	
ParentName					RO	
PerformanceMethod					X	

Properties	Branch Coll	Leaf	Level	Node	Tree	Tree Structure
PerformanceSelector					X	
PerformanceSelectorOption					X	
PrevSib		RO		RO		
RangeFrom		X				
RangeTo		X				
SetId					X	
SilentMode					X	
State				RO		
Status					X	
Structure					RO	
StructureName					X	
SummarySetId						X
SummaryLevelNumber						X
SummaryTreeName						X
SummaryUserKeyValue						X
TreeBranchName		RO	RO	RO		
TreeEffDt		RO	RO	RO		
TreeName		RO	RO	RO		
TreeSetId		RO	RO	RO		
TreeUserKeyValue		RO	RO	RO		
TotalNodeCount					X	
Type				RO		X
UserKeyValue					X	

¹ These properties can be used with a closed tree (or tree structure.)

² These properties are used with the tree collection, the tree structure collection, the level collection and the branch collection objects, not the tree, tree structure or level objects.

Chart, TreeView, and ImageList Cheat Sheets

The following tables list the Chart, Treeview, and ImageList ActiveX control properties and methods, and all the objects instantiated from these objects, plus their properties and methods.

A property marked with an asterisk (*) is the default property for that object.

Properties that are read-only are marked with RO (read-only). If a property isn't marked RO, it's read-write.

Chart Control Objects, Properties and Methods

<i>Object Name</i>	<i>Properties and Methods</i>	<i>Returns</i>
Chart	Properties	
	ActiveSeriesCount	Number, RO
	AllowDithering	Boolean
	AllowDynamicRotation	Boolean
	AllowSelections	Boolean
	AllowSeriesSelection	Boolean
	AutoIncrement	Boolean
	Backdrop	Backdrop Object, RO
	BorderStyle	Number
	Chart3d	Boolean, RO
	ChartType	Number
	Column	Number
	ColumnCount	Number
	ColumnLabel	String
	ColumnLabelCount	Number
	ColumnLabelIndex	Number
	Data	Number
	DataGrid	DataGrid object, RO
	DoSetCursor	Boolean
	DrawMode	Number
	Enabled	Boolean
	Footnote	Footnote Object, RO
	FootnoteText	String

	Legend	Legend Object, RO
	MousePointer	Number
	OLEDragMode	Number
	OLEDropMode	Number
	Plot	Plot object, RO
	RandomFill	Boolean
	Repaint	Boolean
	Row	Number
	RowCount	Number
	RowLabel	String
	RowLabelCount	Number
	RowLabelIndex	Number
	SeriesColumn	Number
	SeriesType	Number
	ShowLegend	Boolean
	Stacking	Boolean
	TextLengthType	Number
	Title	Title object
	TitleText	String
Chart	Methods	
	EditCopy()	
	EditPaste()	
	Layout()	
	OLEDrag()	
	Refresh()	
	SelectPart(<i>part, index1, index2, index3, index4</i>)	
	ToDefaults()	
Axis	Properties	
	AxisGrid	AxisGrid Object, RO
	AxisScale	AxisScale Object, RO
	AxisTitle	AxisTitle object, RO
	CategoryScale	CategoryScale object, RO

	Intersection	Intersection object, RO
	LabelLevelCount	Number, RO
	Labels	Labels object, RO
	Pen	Pen object, RO
	Tick	Tick object, RO
	ValueScale	ValueScale object, RO
AxisGrid	Properties	
	MinorPen	Pen Object, RO
	MajorPen	Pen object, RO
AxisScale	Properties	
	Hide	Boolean
	LogBase	Number
	PercentBasis	Number
	Type	Number
AxisTitle	Properties	
	Backdrop	Backdrop object, RO
	Font	Font object
	Text*	String
	TextLayout	TextLayout object, RO
	TextLength	Number, RO
	Visible	Boolean
	VtFont	VtFont object, RO
BackDrop	Properties	
	Fill	Fill Object, RO
	Frame	Frame object, RO
	Shadow	Shadow object, RO
Brush	Properties	
	FillColor	VtColor object, RO
	Index	Number
	PatternColor	VtColor object, RO
	Style	Number
CategoryScale	Properties	
	Auto	Boolean

	DivisionsPerLabel	Number
	DivisionsPerTick	Number
	LabelTick	Boolean
Coor	Properties	
	X	Number
	Y	Number
Coor	Method	
	Set(<i>X</i> , <i>Y</i>)	
DataGrid	Properties	
	ColumnCount	Number
	ColumnLabelCount	Number
	RowCount	Number
	RowLabelCount	Number
DataGrid	Special Property: can only be used with ObjectSetProperty built-in function.	
	RowLabel(<i>row</i> , <i>labelIndex</i>)	String
DataGrid	Methods	
	ColumnLabel(<i>Column</i> , <i>LabelIndex</i>)	
	CompositeColumnLabel(<i>Column</i>)	
	CompositeRowLabel(<i>Row</i>)	
	DeleteColumnLabels(<i>LabelIndex</i> , <i>Count</i>)	
	DeleteColumns(<i>Column</i> , <i>Count</i>)	
	DeleteRowLabels(<i>LabelIndex</i> , <i>Count</i>)	
	DeleteRows(<i>Row</i> , <i>Count</i>)	
	GetData(<i>Row</i> , <i>Column</i> , & <i>DataPoint</i> , <i>nullFlag</i>)	Variable
	InitializeLabels()	
	MoveData(<i>Top</i> , <i>Left</i> , <i>Bottom</i> , <i>Right</i> , <i>OverOffset</i> ,	

	<i>DownOffset)</i>	
	RandomDataFill()	
	RandomFillColumns(<i>Column</i> <i>, Count)</i>	
	RandomFillRows(<i>Row</i> , <i>Count)</i>	
	SetData(<i>Row</i> , <i>Column</i> , <i>DataPoint</i> , <i>nullFlag</i>)	
	SetSize(<i>RowLabelCount</i> , <i>ColumnLabelCount</i> , <i>DataRowCount</i> , <i>DataColumnCount</i>)	
DataPoint	Properties	
	Brush	Brush object, RO
	DataPointLabel	DataPointLabel object, RO
	EdgePen	Pen object, RO
	Marker	Marker object, RO
	Offset	Number
DataPoint	Methods	
	ResetCustom()	
	Select()	DataPoint
DataPointLabel	Properties	
	Backdrop	Backdrop object
	Component	Number, RO
	Custom	Boolean
	Font	Font object
	LineStyle	Number
	LocationType	Number
	Offset	Number, RO
	PercentFormat	String
	Text*	String
	TextLayout	TextLayout object, RO
	TextLength	Number
	ValueFormat	Number
	VtFont	VtFont, RO

DataPointLabel	Methods	
	ResetCustomLabel()	
	Select()	DataPointLabel
DataPoints	Properties	
	Item(<i>Index</i>)	DataPoint object, RO
	Count()	Number, RO
Fill	Properties	
	Brush	Brush object, RO
	Style	Number
Font	Properties	
	Bold	Boolean
	Charset	Number
	Italic	Boolean
	Name	String
	Size	Number
	Strikethrough	Boolean
	Underline	Boolean
	Weight	Number
Footnote	Properties	
	Backdrop	Backdrop object, RO
	Font	Font object
	Location	Location object, RO
	Text*	String
	TextLayout	TextLayout object, RO
	TextLength	Number
	VtFont	VtFont object, RO
Footnote	Method	
	Select()	Footnote object
Frame	Properties	
	FrameColor	VtColor object, RO
	SpaceColor	VtColor object, RO
	Style	Number
	Width	Number

Intersection	Properties	
	Auto	Boolean
	AxisID	Number, RO
	Index	Number, RO
	LabelsInsidePlot	Boolean
	Point	Number
Label	Properties	
	Auto	Boolean
	Backdrop	Backdrop object, RO
	Font	Font object
	Format	Number
	FormatLength	String, RO
	Standing	Boolean
	TextLayout	TextLayout object, RO
	VtFont	VtFont object, RO
Labels (collection)	Properties	
	Count	Number, RO
	Item(<i>Item</i>)	Label object, RO
LCoor	Properties	
	X	Number
	Y	Number
LCoor	Method	
	Set(<i>X</i> , <i>Y</i>)	
Legend	Properties	
	Backdrop	Backdrop object, RO
	Font	Font object
	Location	Location object, RO
	TextLayout	TextLayout object, RO
	VtFont	VtFont object, RO
Legend	Method	
	Select()	Legend object
Light	Properties	
	AmbientIntensity	Number

	EdgeIntensity	Number
	EdgeVisible	Boolean
	LightSources	LightSources object, RO
Lightsource	Properties	
	Intensity	Number
	X	Number
	Y	Number
	Z	Number
Lightsource	Method	
	Set(<i>X, Y, Z, Intensity</i>)	
Lightsources (collection)	Properties	
	Count	Number, RO
	Item(<i>Index</i>)	LightSource object, RO
Lightsources (collection)	Methods	
	Add(<i>X, Y, Z, Intensity</i>);	
	Remove(<i>Index</i>)	
Location	Properties	
	LocationType	Number
	Rect	Rect object, RO
	Visible	Boolean
Marker	Properties	
	FillColor	VtColor object, RO
	Pen	Pen object, RO
	Size	Number
	Style	Number, RO
	Visible	Boolean
Pen	Properties	
	Cap	Number
	Join	Number
	Limit	Number
	Style	Number
	VtColor	VtColor object, RO
	Width	Number

Plot	Properties	
	AngleUnit	Number
	AutoLayout	Boolean
	Backdrop	Backdrop object, RO
	BarGap	Number
	Clockwise	Boolean
	DataSeriesInRow	Boolean
	DefaultPercentBasis	Number, RO
	DepthToHeightRatio	Number
	Light	Light object, RO
	LocationRect	Rect object, RO
	PlotBase	Plotbase object, RO
	Projection	Number
	SeriesCollection	SeriesCollection object, RO
	Sort	Number
	StartingAngle	Number
	SubPlotLabelPosition	Number
	UniformAxis	Boolean
	View3d	View3D object, RO
	Wall	Wall object, RO
	Weighting	Weighting object, RO
	WidthToHeightRatio	Number
	XGap	Number
	ZGap	Number
Plot	Property (Additional)	
	Axis(<i>axisID</i> , [<i>&Index</i>]);	Axis object
Plotbase	Properties	
	BaseHeight	Number
	Brush	Brush object, RO
	Pen	Pen object, RO
Rect	Properties	
	Min	Coor object, RO
	Max	Coor object, RO

Series	Properteis	
	DataPoints	Datapoints Collection, RO
	GuideLinePen	Pen object, RO
	LegendText	String
	Pen	Pen object, RO
	Position	SeriesPostiion object, RO
	SecondaryAxis	Boolean
	SeriesMarker	SeriesMarker object, RO
	SeriesType	SeriesType object
	ShowLine	Boolean
	StatLine	StatLine object, RO
Series	Special Properties: can only be used with ObjectSetProperty built-in function.	
	ShowGuideLine([<i>axisID</i>], [, <i>Index</i>]) = [<i>boolean</i>]	Boolean
	TypeByChartType(<i>chtype</i>) = [<i>SeriesType</i>]	Number
Series	Method	
	Select()	Series object, RO
SeriesCollection	Property	
	Item(<i>Item</i>)	Series Object, RO
SeriesCollection	Method	
	Count	Number
SeriesMarker	Properties	
	Auto	Boolean
	Show	Boolean
SeriesPosition	Properties	
	Excluded	Boolean
	Hidden	Boolean
	Order	Number
	StackOrder	Number
Shadow	Properties	

	Brush	Brush object, RO
	Offset	Number, RO
	Style	Number
StatLine	Properties	
	Flag	Number
	VtColor	VtColor Object, RO
	Width	Number
StatLine	Special Property: can only be used with ObjectSetProperty built-in function.	
	Style(<i>type</i>) = [<i>style</i>]	Number
TextLayout	Properties	
	HorzAlignment	Number
	Orientation	Number
	VertAlignment	Number
	WordWrap	Boolean
Tick	Properties	
	Length	Number
	Style	Number
Title	Properties	
	Backdrop	Backdrop object, RO
	Font	Font object
	Location	Location object, RO
	Text*	String
	TextLayout	TextLayout object, RO
	TextLength	Number
	VtFont	VtFont object, RO
Title	Method	
	Select()	Title object
ValueScale	Properties	
	Auto	Boolean
	MajorDivision	Number
	Maximum	Number

	Minimum	Number
	MinorDivision	Number
View3D	Properties	
	Elevation	Number
	Rotation	Number
View3d	Method	
	<i>Set(Rotation, Elevation)</i>	
VtColor	Properties	
	Automatic	Boolean
	Blue	Number
	Green	Number
	Red	Number
VtColor	Method	
	<i>Set(Red, Green, Blue)</i>	
VtFont	Properties	
	Effect	Number
	Name	String
	Size	Number
	Style	Number
	VtColor	VtColor object, RO
Wall	Properties	
	Brush	Brush object, RO
	Pen	Pen object, RO
	Width	Number
Weighting	Properties	
	Basis	Number
	Style	Number
Weighting	Method	
	<i>Set(Basis, Style)</i>	

TreeView Objects, Properties and Methods

Object Name	Properties and Methods	Returns
TreeView	Properties	
	Appearance	Number
	BorderStyle	Number
	Checkboxes	Boolean
	DropHighlight	Node object, RO
	Enabled	Boolean
	Font	Font object, RO
	FullRowSelect	Boolean
	HideSelection	Boolean
	HotTracking	Boolean
	ImageList;	ImageList object
	Indentation	Number
	LabelEdit	Number
	LineStyle	Number
	MousePointer	Number
	Nodes	Nodes object, RO
	OLEDragMode	Number
	OLEDropMode	Number
	PathSeparator	String
	Scroll	Boolean
	SelectedItem	Node object, RO
	SingleSel	Boolean
	Sorted	Boolean
	Style	Number
TreeView	Methods	
	GetVisibleCount()	Number
	HitTest(<i>X</i> , <i>Y</i>)	Node Object
	StartLabelEdit()	
Font	Properties	
	Bold	Boolean

	Charset	Number
	Italic	Boolean
	Name	String
	Size	Number
	Strikethrough	Boolean
	Underline	Boolean
	Weight	Number
Node	Properties	
	Bold	Boolean
	Checked	Boolean
	Child	Node object, RO
	Children	Node object, RO
	Expanded	Boolean
	ExpandedImage	Number
	FirstSibling	Node object, RO
	FullPath	String
	Image	Number
	Index	Number
	Key	String
	LastSibling	Node object, RO
	Next	Node object, RO
	Parent	Node object, RO
	Previous	Node object, RO
	Root	Node object, RO
	Selected	Boolean
	SelectedImage	Number
	Sorted	Boolean
	Tag	String
	Text*	String
	Visible	Boolean
Node	Methods	
	CreateDragImage()	Picture Object
	EnsureVisible()	Boolean

Nodes (collection)	Properties	
	Count	Number
Nodes (collection)	Methods	
	Add([<i>relative</i> ,] [<i>relationship</i> ,] [key,] [text,] [<i>image</i> ,] [<i>selectedimage</i>])	Node object
	Clear()	
	Item(<i>Index</i>)	Node object
	Remove (<i>Index</i>)	
Picture	Properties	
	Height	Number
	Type	Number
	Width	Number

ImageList Objects, Properties and Methods

Object Name	Properties and Methods	Returns
ImageList	Properties	
	ImageHeight	Number, RO
	ImageWidth	Number, RO
	ListImages	Images object
	MaskColor	Number
	UseMaskColor	Boolean
ImageList	Method	
	Overlay(<i>index1</i> , <i>index2</i>)	
Images (collection)	Properties	
	Count	Number
Images (collection)	Methods	
	Add([<i>index</i> ,] [key,] <i>Picture</i>)	Image object
	Clear()	
	Item(<i>Index</i>)	Image object
	Remove (<i>Index</i>)	
Image	Properties	
	Index	Number

	Key	String
	Picture	Picture object, RO
	Tag	String
Image	Methods	
	ExtractIcon()	Picture object
	Index(<i>Index</i>)	Image object

Data Buffer Access Classes Cheat Sheet

RO = Read Only. * indicates default method for class

Rowset Class	Returns	Row Class	Returns
Functions		Function	
CreateRowset({RECORD. <i>recname</i> &Rowset} [, {FIELD. <i>fieldname</i> , RECORD. <i>recname</i> &Rowset}] . . .)	Rowset Object	GetRow()	Row Object
GetLevel0()	Rowset Object	Methods	
GetRowSet([SCROLL. <i>scrollname</i>])	Rowset Object	CopyTo(<i>rowobject</i>)	Row Object
Methods		GetNextEffRow()	Row Object
CopyTo(&DestRowset, [, RECORD. <i>SourceRec</i> , RECORD. <i>DestRec</i>] . . .)		GetPriorEffRow()	Row Object
DeleteRow(<i>n</i>)	Boolean	*GetRecord(<i>n</i> / RECORD. <i>recname</i>)	Record Object
Fill([<i>wherestr</i> , [, <i>bindvalue</i>] . . .])	Number	GetRowSet(<i>n</i> / SCROLL. <i>scrollname</i>)	Rowset Object
FillAppend([<i>wherestr</i> , [, <i>bindvalue</i>]...])	Number	<i>Scrollname</i> (<i>n</i>)	Row Object
Flush()		Properties	
FlushRow(<i>n</i>)		ChildCount	Number, RO
GetCurrEffRow()	Row Object	DeleteEnabled	Boolean
*GetRow(<i>n</i>)	Row Object	IsChanged	Boolean, RO
HideAllRows()		IsDeleted	Boolean,

Rowset Class	Returns	Row Class	Returns
			RO
InsertRow(<i>n</i>)	Boolean	IsEditError	Boolean, RO
Refresh()		IsNew	Boolean, RO
Select([<i>parmlist</i>], RECORD.selrecord [<i>, wherestr, bindvars</i>])	Number	ParentRowSet	Rowset object, RO
SelectNew([<i>parmlist</i>], RECORD.selrecord [<i>, wherestr, bindvars</i>])	Number	<i>Recname</i>	Record object, RO
SetDefault(<i>recname.fieldname</i>)		RecordCount	Number, RO
ShowAllRows()		RowNumber	Number, RO
Sort([<i>paramlist</i> ,] <i>sort_fields</i>)		Selected	Boolean
Properties		Style	String
ActiveRowCount	Number, RO	Visible	Boolean
DBRecordName	String, RO		
DeleteEnabled	Boolean		
EffDt	Date, RO		
EffSeq	Number, RO		
InsertEnabled	Boolean		
IsEditError	Boolean, RO		
Level	Number, RO		
Name	String, RO		
ParentRow	Row Object, RO		
ParentRowSet	Rowset Object, RO		
RowCount	Number, RO		
TopRowNumber	Number, RO		

Record Class	Returns	Field Class	Returns
Functions		Function	
CreateRecord(RECORD. <i>recname</i>)	Record Object	GetField([<i>recname.fieldname</i>])	Field Object
GetRecord([RECORD. <i>recname</i>])	Record Object	Methods	
Methods		GetLongLabel(<i>LabelID</i>)	String
CompareFields(<i>RecordObject</i>)	Boolean	GetRelated(<i>recname.fieldname</i>)	Field Object
CopyChangedFieldsTo(<i>RecordObj</i>)		GetShortLabel(<i>LabelID</i>)	String
CopyFieldsTo(<i>RecordObject</i>)		SetCursorPos(PAGE. <i>pagename</i> %Page)	
Delete()	Boolean	SetDefault()	
ExecuteEdits(<i>EditLevel</i>)		Properties	
*GetField(n/FIELD. <i>fieldname</i>)	Field Object	DisplayFormat	String
Insert()	Boolean	DisplayOnly	Boolean
SelectByKey()	Boolean	EditError	Boolean
SetDefault()		Enabled	Boolean
SetEditTable(% <i>EdiFldNml</i> , RECORD. <i>rec</i>)		FormattedValue	
Update([<i>KeyRecord</i>])	Boolean	IsAltKey	Boolean, RO
Properties		IsAuditFieldAdd	Boolean, RO
FieldCount	Number, RO	IsAuditFieldChg	Boolean, RO
<i>fieldname</i>	Field Object, RO	IsAuditFieldDel	Boolean, RO
IsChanged	Boolean, RO	IsAutoUpdate	Boolean, RO
IsDeleted	Boolean, RO	IsChanged	Boolean, RO
IsEditError	Boolean, RO	IsDateRangeEdit	Boolean, RO

Record Class	Returns	Field Class	Returns
Name	String, RO	IsDescKey	Boolean, RO
ParentRow	Row Object, RO	IsDupKey	Boolean, RO
RelLangRecName	String, RO	IsEditTable	Boolean, RO
		IsEditXlat	Boolean, RO
		IsFromSearchField	Boolean, RO
		IsInBuf	Boolean, RO
		IsKey	Boolean, RO
		IsListItem	Boolean, RO
		IsRequired	Boolean, RO
		IsSearchItem	Boolean, RO
		IsSystem	Boolean, RO
		IsThroughSearchField	Boolean, RO
		IsUseDefaultLabel	Boolean, RO
		IsYesNo	Boolean, RO
		Label	String
		LongTranslateValue	Any
		MessageNumber	Number
		MessageSetNumber	Number
		Name	String, RO
		OriginalValue	Depends on field
		ParentRecord	Record Object, RO

Record Class	Returns	Field Class	Returns
		SeachDefault	Boolean
		SearchEdit	Boolean
		ShortTranslateValue	Any
		ShowRequiredFieldCue	Boolean
		StoredFormat	String, RO
		Style	String
		Type	String, RO
		Value	Depends on field
		Visible	Boolean

Old ScrollSelect code filling level 1 scroll:

```
ScrollFlush(RECORD.PMN_SRVRLIST);

ScrollSelect(1, RECORD.PMN_SRVRLIST, RECORD.PMN_SRVRLIST);
```

New Select code:

```
&RSServerList = GetLevel0() (1).GetRowset(SCROLL.PMN_SRVRLIST);

&RSServerList.Flush();

&RSServerList.Select(RECORD.PMN_SRVRLIST);
```

The following selects into a row on a third level scroll, using the current context.

```
RowScrollSelect(3, Record.WORKLISTAPI_VW, CurrentRowNumber(1),
Record.PSWORKLIST, CurrentRowNumber(2), Record.WLKEYINFO_VW,
Record.WLKEYINFO_VW, "where RECNAME = :1 ", WORKLISTAPI_VW.WLRECNAME);
```

The following code also selects into a row, using the current context.

```
&PSWorkList = GetRowset();

&WLKeyInfo = &PSWorkList.GetRowset(SCROLL.WLKEYINFO_VW);

&WLKeyInfo.Select(RECORD.WLKEYINFO_VW, "where RECNAME = :1 ",
WORKLISTAPI_VW.WLRECNAME);
```

Because the previous code is executing on the second level of the scroll and selecting into the third level, it could be reduced to the following:

```
&WLKeyInfo = GetRowset(SCROLL.WLKEYINFO_VW);

&WLKeyInfo.Select(RECORD.WLKEYINFO_VW, "where RECNAME = :1 ",
WORKLISTAPI_VW.WLRECNAME);
```

Mapping of Functions to Methods and Properties

The first column in following table lists built-in PeopleCode functions that existed prior to PeopleTools 8.0. The second column lists the new method or property that should be used instead of the function. The existing functions still work in PeopleTools 8.0, however, they are being retained for backward compatibility only. New applications should be created using the new classes, methods and properties.

Existing function name	Use instead
ActiveRowCount	ActiveRowCount Rowset property
ClearSearchDefault	SearchDefault Field property
ClearSearchEdit	SearchEdit Field property
CompareLikeFields	CompareFields Record method
CopyFields	CopyFieldsTo or CopyChangedFields Record method
CopyRow	CopyTo Row method
CurrEffDt	Effdt Rowset property
CurrEffRowNum	RowNumber Row property, in combination with the GetCurrEffRow Rowset method
CurrEffSeq	EffSeq Rowset property
CurrentRowNumber	RowNumber Row property
DeleteRecord	Delete Record method
DeleteRow	DeleteRow Rowset method
FetchValue	Value Field property
FieldChanged	IsChanged Field property
GetRelField	GetRelated Field method
GetStoredFormat	StoredFormat Field property
Gray	Enabled Field property
Hide	Visible Field property
HideRow	Visible Row property
HideScroll	HideAllRows Rowset method
InsertRow	InsertRow Rowset method
IsHidden	Visible Row property
NextEffDt	GetNextEffRow().REC.FIELD.Value
NextRelEffDt	GetNextEffRow().REC.FIELD.GetRelated(rec.field).value
PriorEffDt	GetPriorEffRow().REC.Field.Value
PriorRelEffDt	GetPriorEffRow().REC.FIELD.GetRelated(rec.field).value

Existing function name	Use instead
RecordChanged	IsChanged Record property
RecordDeleted	IsDeleted Record property
RecordNew	IsNew Record property
RemoteCall for Application Engine	CallAppEngine function
RowFlush	FlushRow Rowset method
RowScrollSelect	Select Rowset method
RowScrollSelectNew	SelectNew Rowset method
ScheduleProcess	CreateProcessRequest Function
ScrollFlush	Flush Rowset method
ScrollSelect	Select Rowset method
ScrollSelectNew	SelectNew Rowset method
SetDefault	SetDefault Field property
SetDefaultAll	SetDefault Rowset method
SetDefaultNext	GetNextEffRow().REC.FIELD.SetDefault()
SetDefaultNextRel	GetNextEffRow().REC.Field.GetRelated(REC.FIELD).SetDefault()
SetDefaultPrior	GetPriorEffRow().REC.FIELD.SetDefault()
SetDefaultPriorRel	GetPriorEffRow().REC.Field.GetRelated(REC.FIELD).SetDefault()
SetDisplayFormat	DisplayFormat Field property
SetLabel	Label Field property
SetSearchDefault	SearchDefault Field property
SetSearchEdit	SearchEdit Field property
SetTracePC	If using API (Session object), use TraceSetting properties
SetTraceSQL	If using API (Session object), use TraceSetting properties
SortScroll	Sort Rowset method
TotalRowCount	RowCount Rowset property
Ungray	Enabled Field property
UnHide	Visible Field property
UnHideRow	Visible Row property
UnhideScroll	ShowAllRows Rowset method
UpdateValue	Value Field property

Mapping of Old Names to New Names

In PeopleTools 8.1, the names of some of the definitions in Application Designer changed. The names of related built-in functions have changed accordingly.

The first table lists the old terms and new terms.

In the second table, the left column lists the old names of the functions, system variables or reserved words. The right column lists the new names. The existing functions, system variables and reserved words in this table still work in PeopleTools 8.0, however, they are being retained for backward compatibility only. New applications should be created using the new functions, system variables and reserved words.

Old Term	New Term
Operator	User
Panel	Page
Panel Group	Component
Business Component	Component Interface

Deprecated Function, System Variable or Reserved Word	New Function, System Variable or Reserved Word
DoModalPanelGroup built-in function	DoModalComponent built-in function
IsModalPanelGroup built-in function	IsModalComponent built-in function
IsOperatorInClass built-in function	IsUserInPermissionList built-in function
PanelGroupChanged built-in function	ComponentChanged built-in function
SetNextPanel built-in function	SetNextPage built-in function
TransferPanel built-in function	TransferPage built-in function
%OperatorClass System Variable	%PrimaryPermissionList System Variable
%OperatorID System Variable	%UserID System Variable
%OperatorRowLevelSecurityClass System Variable	%RowSecurityPermissionList System Variable
%Panel System Variable	%Page System Variable
%PanelGroup System Variable	%Component System Variable
PANEL reserved word	PAGE reserved word
PANELGROUP reserved word	COMPONENT reserved word
PanelGroup variable declaration	Component variable declaration

Business Components are now named Component Interfaces. For Component Interfaces, the old reserved word, methods and system variables *are no longer valid*. You *must* use the new reserved word, methods or system variables.

No Longer Valid	Use Instead
COMPONENT reserved word	COMPINTFC reserved word
GetComponent method	GetCompIntfc method
FindComponent method	FindCompIntfc method
%Component system variable	%CompIntfc system variable

PeopleCode Syntax Cheat Sheet

The following is a description of the PeopleCode syntax, as derived from the PeopleCode parser. This description uses these "metacharacters":

	colons are used to separate the defined symbol from its definition.
[]	brackets are used to indicate optional portions.
...	ellipses are used to indicate that the preceding part (often optional) repeats an arbitrary number of times.
	vertical bars are used to separate alternatives. They indicate that one of the alternatives should be present. The alternatives only extend to the immediately enclosing parentheses or brackets (if any).
()	parentheses are used to group portions of definitions to limit the scope of alternatives () or optionality (...).
'symbols'	quoted characters in blue are keywords or punctuation in the language. They should appear literally, without the quotes.
Program	DeclList TopStList
TopStList	TopStmt [';' TopStmt]...
TopStmt	Stmt
StList	Stmt [';' Stmt]...
Stmt	[LValue [Assign] If Evaluate While Repeat For Accept Break Return Exit Error Warning]
Assign	'=' Expression
If	'if' LogicalExpression 'then' StList ['else' StList] 'end-if'

Evaluate	'evaluate' Expression [WhenExpr... StList]... ['when-other' StList] 'end-evaluate'
WhenExpr	'when' [RelOp] Expression
While	'while' LogicalExpression StList 'end-while'
Repeat	'repeat' StList 'until' LogicalExpression
For	'for' Variable '=' Expression 'to' Expression ['step' Expression] StList 'end-for'
Accept	'accept'
Return	'return' [Expression]
Break	'break'
Exit	'exit' [Expression]
Warning	'warning' Expression
Error	'error' Expression
Logical Expression	LogicalTerm ['or' LogicalTerm]...
Logical Term	Relation ['and' Relation]...
Relation	'not' Relation Expression [Relop Expression]
Relop	'not' Relop '<' '<=' '=' '>=' '>' '!=' '<>'
Expression	Term [('+' '-' ' ') Term]...
Term	Power [('*' '/') Power]...
Power	Primary ['**' Primary]...
Primary	LValue '%' BuiltinVar Number String 'true' 'false' 'null' '(' LogicalExpression ')' '-' Primary
LValue	(Variable Id Call '@' Primary) [LValueObject LValueSubscript]...
LValueObject	('.' Id [Call] Call)
LValueSubscript	'[' Expression [',' Expression]... ']'
Variable	Id ['.' (Id String)] '^' '&' Id
Call	'(' [Expression [',' Expression]...] ')'

DeclList	[('declare' Declare 'local' VarDeclare 'pagegroup' VarDeclare 'global' VarDeclare 'function' Function) ';']...
Declare	'function' Id ('library' DeclareLibrary 'peoplecode' DeclarePC)
DeclareLibrary	String ['alias' String] ['(' ExternalParameters ')'] ['returns' ExternalType ['as' PeopleCodeType]]
ExternalParameters	[ExternalParameter [',' ExternalParameter]...]
ExternalParameter	ExternalType ['value' 'ref'] ['as' PeopleCodeType]
DeclarePC	Variable EventType
VarDeclare	PeopleCodeType ['&' Id [',' '&' Id]...]
Function	Id ['(' InternalParameters ')'] ['returns' PeopleCodeType] ['noexport'] ['doc' String]TopStList 'end-function'
InternalParameters	[InternalParameter [',' InternalParameter]...]
InternalParameter	'&' Id ['as' PeopleCodeType]
PeopleCodeType	['array' 'of']... ('number' 'string' 'date' 'any' 'boolean' 'time' 'datetime' 'object' 'array' ObjectClass)
ExternalType	'boolean' 'integer' 'long' 'uinteger' 'ulong' 'string' 'lstring' 'ustring' 'float' 'double'

Index

%

- %Abs meta-SQL function 10-5
 - %AsOfDate variable 11-1
 - %AuthenticationToken variable 11-1
 - %BPName variable 11-1
 - %ClientDate variable 11-1
 - %ClientTimeZone variable 11-2
 - %CompIntfcName variable 11-2
 - %Component variable 11-2
 - %Concat meta-SQL function 10-6
 - %Currency variable 11-2
 - %CurrentDateIn meta-SQL function 10-6
 - %CurrentDateOut meta-SQL function 10-6
 - %CurrentDateTimeIn meta-SQL function 10-6
 - %CurrentDateTimeOut meta-SQL function 10-7
 - %CurrentTimeIn meta-SQL function 10-7
 - %CurrentTimeOut meta-SQL function 10-7
 - %Date variable 11-3
 - %DateAdd meta-SQL function 10-7
 - %DateDiff meta-SQL function 10-7, 10-8
 - %DateIn meta-SQL function 10-8
 - %DateOut meta-SQL function 10-8
 - %DateTime variable 11-3
 - %DateTimeIn meta-SQL function 10-9
 - %DateTimeOut meta-SQL function 10-9
 - %DbName variable 11-3
 - %DbType variable 11-3
 - %DecDiv meta-SQL function 10-9
 - %DecMult meta-SQL function 10-10
 - %DTTM meta-SQL function 10-11
 - %EffDtCheck meta-SQL function 10-11
 - %EmailAddress variable 11-3
 - %EmployeeId variable 11-3
 - %ExternalAuthInfo variable 11-3
 - %Import variable 11-4
 - %InsertSelect meta-SQL function 10-14
 - %InsertValues meta-SQL function 10-17
 - %IsMultiLanguageEnabled variable 11-4
 - %Join meta-SQL function 10-18
 - %KeyEqual meta-SQL function 10-20
 - %KeyEqualNoEffDt meta-SQL function 10-21
 - %Language variable 11-4
 - %Language_Base variable 11-4
 - %Like meta-SQL function 10-22
 - %LikeExact meta-SQL function 10-24
 - %List meta-SQL function 10-27
 - %ListBind meta-SQL function 10-31
 - %ListEqual meta-SQL function 10-32
 - %Market variable 11-4
 - %MaxInterlinkSize variable 11-5
 - %MaxMessageSize variable 11-6
 - %Menu variable 11-6
 - %MessageAgent variable 11-6
 - %Mode variable 11-6
 - %NavigatorHomePermissionList variable 11-6
 - %OldKeyEqual meta-SQL function 10-33
 - %OPRCLAUSE meta-SQL function 10-33
 - %Page variable 11-7
 - %PasswordExpired variable 11-8
 - %PermissionLists variable 11-8
 - %PrimaryPermissionList variable 11-8
 - %ProcessProfilePermissionList variable 11-8
 - %PSAuthResult variable 11-8
 - %Request variable 11-8
 - %Response variable 11-9
 - %ResultDocument variable 11-9
 - %Roles variable 11-9
 - %Round meta-SQL function 10-34
 - %RowSecurityPermissionList variable 11-9
 - %ServerTimeZone variable 11-9
 - %Session variable 11-10
 - %SignonUserId variable 11-10
 - %SignonUserPswd variable 11-10
 - %SQL meta-SQL function 10-34
 - %SQLRows variable 11-10
 - %SUBREC meta-SQL function 10-36
 - %Substring meta-SQL function 10-36
 - %Table meta-SQL function 10-37
 - %TextIn meta-SQL function 10-38
 - %Time variable 11-11
 - %TimeAdd meta-SQL function 10-39
 - %TimeIn meta-SQL function 10-40
 - %TimeOut meta-SQL function 10-40
 - %TrimSubstr meta-SQL function 10-40
 - %Truncate meta-SQL function 10-40
 - %Truncatetable meta-SQL function 10-41
 - %UpdatePairs meta-SQL function 10-42
 - %Upper meta-SQL function 10-43
 - %UserDescription variable 11-11
 - %UserId variable 11-11
 - %WLInstanceID variable 11-11
 - %WLName variable 11-11
-
- ## A
- Abs function 1-16
 - accessing
 - AESection 4-4
 - Component Interface structure 4-183
 - PSMessages collection 7-30

- AccruableDays function 1-17
- AccruableFactor function 1-18
- Acos function 1-19
- ActiveRowCount function 1-20
- ActiveX Controls 8-1
 - chart control 8-1
 - ImageList 9-29
 - TreeView 9-1
- ActiveX functions
 - GetControl 2-47
 - GetControlOccurrence 2-49
- AddAttachment function 1-21
- AddKeyListItem function 1-27
- AddSystemPauseTimes function 1-27
- AddToDate function 1-30
- AddToDateTime function 1-31
- AddToTime function 1-32
- AESection class
 - accessing 4-4
 - declaring 4-7
 - example 4-5
 - methods 4-7
 - property 4-12
 - scope of 4-7
- All function 1-33
- AllOrNone function 1-34
- AllowEmplidChg function 1-35
- Amortize function 1-36
- API functions
 - CreateProcessRequest 1-84
 - GetSession 2-83
 - RemoteCall 3-27
 - ScheduleProcess 3-44
 - SendMail 3-55
- Application Engine
 - CallAppEngine function 1-41
 - CommitWork function 1-53
 - file class example 5-46
 - GetAESection function 2-39
 - SetTempTableInstance function 3-79
 - SQL object considerations 7-51
 - using with Business Interlink 4-47
- Application Messaging
 - %MaxMessageSize variable 11-6
 - AddSystemPauseTimes function 1-27
 - CreateMessage function 1-82
 - DeleteSystemPauseTimes function 1-121
 - GetMessage function 2-68
 - GetMessageInstance function 2-69
 - GetPubContractInstance function 2-75
 - GetSubMessage function 2-88
 - message class 5-121
 - PingNode function 3-11
 - ReturnToServer function 3-33
 - SetChannelStatus function 3-60
- Array class
 - CreateArray function 1-77
 - CreateArrayRept function 1-78
 - creating empty arrays 4-17
 - declaring 4-21
 - flattening and promotion 4-21
 - methods 4-22
 - multi-dimensional arrays 4-18
 - populating an array 4-15
 - properties 4-39
 - removing items 4-16
 - scope of 4-22
 - Split function 3-88
- Array functions
 - CreateArray 1-77
 - CreateArrayRept 1-78
 - Split 3-88
- Asin function 1-37
- Atan function 1-37
- attachment functions
 - AddAttachment 1-21
 - DeleteAttachment 1-112
 - GetAttachment 2-40
 - PutAttachment 3-16
 - ViewAttachment 3-132
- AttributeValue class
 - properties 6-64
- AttributeValue collection
 - methods 6-65
 - property 6-68
- Axis object properties 8-45
- AxisGrid object properties 8-46
- AxisGrid properties
 - MajorPen 8-47
 - MinorPen 8-46
- AxisScale object properties 8-47
- AxisScale properties
 - Hide 8-47
 - LogBase 8-47
 - PercentBasis 8-47
 - Type 8-48
- AxisTitle object properties 8-49
- AxisTitle properties
 - Backdrop 8-49
 - Font 8-49
 - Text 8-49
 - TextLayout 8-50
 - TextLength 8-50
 - Visible 8-50
 - VtFont 8-50

B

- Backdrop object properties 8-50
- Backdrop properties
 - Fill 8-50
 - Frame 8-51
 - Shadow 8-51
- BIDocs methods 4-97
- BIDocs properties 4-109

- BlackScholesCall function 1-38
- BlackScholesPut function 1-39
- BootstrapYTM function 1-39
- branch collection
 - method 7-80
 - properties 7-81
- Break 1-40
- Brush object properties 8-51
 - FillColor 8-51
 - Index 8-51
 - PatternColor 8-52
 - Style 8-52
- Business Interlink class
 - %MaxInterlinkSize variable 11-5
 - BIDocs methods 4-97
 - BIDocs properties 4-109
 - configuration parameters 4-115
 - declaring 4-48
 - GetBiDoc 2-43
 - GetInterlink function 2-62
 - methods 4-49
 - property 4-114
 - scope of 4-48
 - state of object 4-47
 - using 4-40
 - using with Application Engine 4-47

C

- CallAppEngine function 1-41
- Canceling functions
 - DoCancel 1-125
 - Exit 2-17
 - WinEscape 3-140
- CategoryScale properties 8-52
- CD-ROM
 - ordering lxvii
- Char function 1-44
- chart control 8-1
 - appearance 8-7
 - axis object 8-45
 - AxisGrid object 8-46
 - AxisScale object 8-47
 - AxisTitle object 8-49
 - BackDrop object 8-50
 - Brush object 8-51
 - CategoryScale object 8-52
 - constants 8-12
 - Coor object 8-53
 - DataGrid object 8-54
 - DataPoint object 8-67
 - DataPointLabel object 8-69
 - DataPoints object 8-73
 - declaring 8-30
 - Fill object 8-73
 - font object 8-74
 - fonts 8-8

- Footnote object 8-76
- Frame object 8-78
- Intersection object 8-79
- Label object 8-80
- Labels collection 8-82
- LCoor object 8-82
- Legend object 8-83
- Light object 8-84
- lighting 8-10
- Lightsources object 8-85
- Lightsources collection 8-87
- Location object 8-88
- Marker object 8-89
- Pen object 8-91
- PeopleCode events 8-20
- Plot object 8-93
- PlotBase object 8-100
- properties 8-31
- quick reference 9-42
- Rect object 8-101
- rotating 8-9
- scope of 8-30
- Series object 8-101
- SeriesCollection collection 8-105
- SeriesMarker object 8-105
- SeriesPosition object 8-106
- setting data points 8-5
- Shadow object 8-107
- special properties 8-4
- Statline object 8-108
- TextLayout object 8-110
- Tick object 8-112
- Title object 8-113
- using data grid 8-6
- ValueScale object 8-114
- View3D object 8-117
- VtColor object 8-118
- VtFont object 8-120
- Wall object 8-122
- Weighting object 8-123
- chart control methods
 - EditCopy 8-42
 - EditPaste 8-42
 - Layout 8-42
 - OLEDrag 8-43
 - Refresh 8-43
 - SelectPart 8-43
 - ToDefaults 8-44
- chart control properties
 - ActiveSeriesCount 8-31
 - AllowDithering 8-31
 - AllowDynamicRotation 8-31
 - AllowSelections 8-31
 - AllowSeriesSelection 8-32
 - AutoIncrement 8-32
 - Backdrop 8-32
 - BorderStyle 8-32
 - Chart3d 8-33

- ChartType 8-33
- Column 8-33
- ColumnCount 8-34
- ColumnLabel 8-34
- ColumnLabelCount 8-34
- ColumnLabelIndex 8-34
- Data 8-35
- DataGrid 8-35
- DoSetCursor 8-35
- DrawMode 8-36
- Enabled 8-36
- Footnote 8-36
- FootnoteText 8-37
- Legend 8-37
- MousePointer 8-37
- OLEDragMode 8-37
- OLEDropMode 8-38
- Plot 8-38
- RandomFill 8-38
- Repaint 8-39
- Row 8-39
- RowCount 8-40
- RowLabel 8-40
- RowLabelCount 8-40
- RowLabelIndex 8-40
- SeriesColumn 8-40
- SeriesType 8-40
- ShowLegend 8-41
- Stacking 8-41
- TextLengthType 8-41
- Title 8-41
- TitleText 8-41
- CharType function 1-44
- ChDir function 1-47
- ChDrive function 1-48
- cheat sheets 12-1
- CheckMenuItem function 1-49
- Clean function 1-50
- ClearKeyList function 1-50
- ClearSearchDefault function 1-51
- ClearSearchEdit function 1-52
- Code function 1-52
- Codeb function 1-53
- CommitWork function 1-53
- CompareLikeFields function 1-56
- CompIntfPropInfoCollection collection methods 4-186
- CompIntfPropInfoCollection collection properties 4-187
- CompIntfPropInfoCollection object properties 4-187
- component buffer functions
 - ActiveRowCount 1-20
 - AddKeyListItem 1-27
 - CompareLikeFields 1-56
 - ComponentChanged 1-58
 - CopyFields 1-74
 - CopyRow 1-75
 - CurrentLevelNumber 1-95
 - CurrentRowNumber 1-96
 - DeleteRecord 1-117
 - DeleteRow 1-117
 - DiscardRow 1-124
 - ExpandBindVar 2-18
 - FetchValue 2-23
 - FieldChanged 2-24
 - GetNextNumber 2-70
 - GetNextNumberWithGaps 2-72
 - GetRelField 2-77
 - GetSetID 2-83
 - InsertRow 2-111
 - PriorValue 3-13
 - RecordChanged 3-19
 - RecordDeleted 3-21
 - RecordNew 3-24
 - RowFlush 3-38
 - RowScrollSelect 3-39
 - RowScrollSelectNew 3-42
 - ScrollFlush 3-48
 - ScrollSelect 3-49
 - ScrollSelectNew 3-53
 - SetControlValue 3-61
 - SetCursorPos 3-64
 - SetDefault 3-66
 - SetDefaultAll 3-67
 - SetDefaultNext 3-68
 - SetDefaultNextRel 3-68
 - SetDefaultPrior 3-69
 - SetDefaultPriorRel 3-69
 - SetNextPage 3-74
 - SetTempTableInstance 3-79
 - SortScroll 3-87
 - StopFetching 3-96
 - TotalRowCount 3-109
 - TreeDetailInNode 3-116
 - UpdateSysVersion 3-129
 - UpdateValue 3-129
- component buffer meta-SQL functions
 - %TruncateTable 10-41
- component buffer system variables
 - %Component 11-2
 - %Menu 11-6
 - %Mode 11-6
- Component Interface class
 - %CompIntfcName variable 11-2
 - accessing structure 4-183
 - CompIntfPropInfoCollection collection methods 4-186
 - CompIntfPropInfoCollection collection properties 4-187
 - CompIntfPropInfoCollection object properties 4-187
 - Component Interface collection methods 4-152
 - Component Interface collection property 4-154
 - data collection methods 4-173
 - data collection standard properties 4-181

- data collections 4-135
- data item standard property 4-182
- declaring 4-123
- effective dated data 4-148
- GetMethodNames function 2-68
- implementing 4-124
- keys 4-119
- life cycle 4-118
- quick reference 12-16
- reusing existing code 4-149
- scope of 4-124
- session object methods 4-150
- setting keys 4-119
- standard methods 4-154
- standard properties 4-122, 4-171
- timeouts 4-135
- traversing 4-135
- using data collection 4-135
- Component statement 1-57
- ComponentChanged function 1-58
- ContainsCharType function 1-59
- ContainsOnlyCharType function 1-61
- content provider class
 - properties 6-37
- content provider collection
 - methods 6-39
 - properties 6-42
- content reference class
 - methods 6-51
 - properties 6-52
- content reference collection
 - methods 6-60
 - property 6-63
- Conversion functions
 - Char 1-44
 - Code 1-52
 - Codeb 1-53
 - ConvertChar 1-63
 - String 3-98
 - Value 3-131
- ConvertChar function 1-63
- ConvertCurrency function 1-68
- ConvertDatetimeToBase function 1-70
- ConvertRate function 1-71
- ConvertTimeToBase function 1-73
- Cookie object properties 5-107
- Coor object method 8-54
- Coor object properties 8-53
- CopyFields function 1-74
- CopyRow function 1-75
- Cos function 1-76
- Cot function 1-76
- CreateArray function 1-77
- CreateArrayRept function 1-78
- CreateJavaArray function 1-80
- CreateJavaObject function 1-80
- CreateMessage function 1-82
- CreateObject function 1-83

- CreateProcessRequest function 1-84
- CreateRecord function 1-85
- CreateRowset function 1-87
- CreateSQL function 1-90
- creating web libraries 5-87
- CubicSpline function 1-91
- CurrEffDt function 1-93
- CurrEffRowNum function 1-94
- CurrEffSeq function 1-95
- Currency and financial functions
 - %Currency 11-2
 - Amortize 1-36
 - ConvertCurrency 1-68
 - RoundCurrency 3-37
 - SinglePaymentPV 3-86
 - UniformSeriesPV 3-128
- Current date and time functions
 - %CurrentDateIn 10-6
 - %CurrentDateOut 10-6
 - %CurrentDateTimeIn 10-6
 - %CurrentDateTimeOut 10-7
 - %CurrentTimeIn 10-7
 - %CurrentTimeOut 10-7
- CurrentLevelNumber function 1-95
- CurrentRowNumber function 1-96
- Custom display format functions
 - GetStoredFormat 2-87
 - SetDisplayFormat 3-70

D

- data buffer access
 - quick reference 12-55
- Data Buffer Access Functions
 - CreateRecord 1-85
 - CreateRowset 1-87
 - GetField 2-52
 - GetLevel0 2-66
 - GetRecord 2-75
 - GetRow 2-78
 - GetRowset 2-79
- data collection methods 4-173
- data collection standard properties 4-181
- datagrid
 - using 8-6
- DataGrid object methods 8-55
- DataGrid object properties 8-54
- DataGrid object special properties 8-66
- DataPoint object methods 8-68
- DataPoint object properties 8-67
- DataPointLabel object methods 8-72
- DataPointLabel object properties 8-69
- DataPoints object properties 8-73
- Date and time functions
 - %DateAdd 10-7
 - %DateDiff 10-7, 10-8
 - %DateIn 10-8

- %DateOut 10-8
- %DateTimeIn 10-9
- %DateTimeOut 10-9
- %DTTM 10-11
- %TimeAdd meta-SQL function 10-39
- %TimeIn 10-40
- %TimeOut 10-40
- AddToDate 1-30
- AddToDateTime 1-31
- AddToTime 1-32
- considerations 10-1
- ConvertDatetimeToBase 1-70
- ConvertTimeToBase 1-73
- Date 1-97
- Date3 1-97
- DatePart 1-98
- DateTime6 1-98
- DateTimeToLocalizedString 1-99
- DateTimeToTimeZone 1-101
- DateTimeValue 1-103
- DateValue 1-104
- Day 1-105
- Days 1-105
- Days360 1-106
- Days365 1-106
- FormatDateTime 2-34
- GetCalendarDate 2-45
- Hour 2-108
- IsDaylightSavings 2-113
- Minute 2-134
- Month 2-135
- Second 3-55
- Time 3-104
- Time3 3-104
- TimePart 3-105
- TimeToTimeZone 3-106
- TimeValue 3-107
- TimeZoneOffset 3-108
- Weekday 3-139
- Year 3-147
- Date function 1-97
- Date3 function 1-97
- DatePart function 1-98
- DateTime6 function 1-98
- DateTimeToLocalizedString function 1-99
- DateTimeToTimeZone function 1-101
- DateTimeValue function 1-103
- DateValue function 1-104
- Day function 1-105
- Days function 1-105
- Days360 function 1-106
- Days365 function 1-106
- DBCSTrim function 1-107
- debugging
 - Java 5-119
- Declare Function 1-107
- Decrypt function 1-111
- Default functions

- SetDefault 3-66
- SetDefaultAll 3-67
- SetDefaultNext 3-68
- SetDefaultNextRel 3-68
- SetDefaultPrior 3-69
- SetDefaultPriorRel 3-69
- Degrees function 1-112
- DeleteAttachment function 1-112
- DeleteImage function 1-116
- DeleteRecord function 1-117
- DeleteRow function 1-117
- DeleteSQL function 1-119
- DeleteSystemPauseTimes function 1-121
- DisableMenuItem function 1-123
- DiscardRow function 1-124
- DoCancel function 1-125
- DoModal function 1-126
- DoModalComponent function 1-128
- DOS functions
 - ChDir 1-47
 - ChDrive 1-48
 - ExpandEnvVar 2-19
 - GetCwd 2-51
 - GetEnv 2-52
- DoSave function 1-132
- DoSaveNow function 1-133
- Double-byte character functions
 - CharType 1-44
 - ContainsCharType 1-59
 - ContainsOnlyCharType 1-61
 - ConvertChar 1-63
 - DBCSTrim 1-107
- Dynamic tree control functions
 - GetSelectedTreeNode 2-80
 - GetTreeNodeParent 2-91
 - GetTreeNodeRecordName 2-92
 - GetTreeNodeValue 2-93
 - RefreshTree 3-26

E

- Effective date functions
 - %EffDtCheck 10-11
 - CurrEffDt 1-93
 - CurrEffRowNum 1-94
 - CurrEffSeq 1-95
 - NextEffDt 3-1
 - NextRelEffDt 3-1
 - PriorEffDt 3-11
 - PriorRelEffDt 3-12
- EnableMenuItem function 2-1
- EncodeURL function 2-2
- EncodeURLForQueryString function 2-3
- Encrypt function 2-4
- EndMessage function 2-4
- EndModal function 2-6
- Error 2-7

- EscapeHTML function 2-9
- EscapeJavascriptString function 2-10
- EscapeWML function 2-10
- Evaluate 2-11
- Exact function 2-12
- Exec function 2-12
- Executable files
 - Exec function 2-12
 - WinExec function 3-141
- ExecuteRolePeopleCode function 2-14
- ExecuteRoleQuery function 2-15
- ExecuteRoleWorkflowQuery function 2-16
- Existence functions
 - All 1-33
 - AllOrNone 1-34
 - None 3-2
 - OnlyOne 3-8
 - OnlyOneOrNone 3-9
- Exit 2-17
- exiting loops 1-40
- Exp function 2-18
- ExpandBindVar function 2-18
- ExpandEnvVar function 2-19
- ExpandSqlBinds function 2-20

F

- Fact function 2-21
- FetchSQL function 2-22
- FetchValue function 2-23
- field class
 - declaring 5-1
 - methods 5-2
 - properties 5-7
 - scope of 5-2
- FieldChanged function 2-24
- file class
 - access interruptions 5-25
 - Application Engine example 5-46
 - declaring 5-24
 - error processing 5-48
 - file layout 5-28
 - file rowset considerations 5-40
 - FileExists function 2-26
 - FindFiles function 2-30
 - GetFile function 2-53
 - methods 5-50
 - multiple file layouts 5-40
 - plain text files 5-26
 - properties 5-72
 - reading multiple file layouts 5-40
 - ReadRecord example 5-32
 - ReadRowset example 5-39
 - scope of 5-24
 - security considerations 5-25
 - WriteRecord example 5-29
 - WriteRowset example 5-34

- writing multiple file layouts 5-44
- FileExists function 2-26
- Fill object properties 8-74
- Financial functions
 - AccruableDays 1-17
 - AccruableFactor 1-18
 - BlackScholesCall 1-38
 - BlackScholesPut 1-39
 - BootstrapYTM 1-39
 - ConvertRate 1-71
 - CubicSpline 1-91
 - HermiteCubic 2-101
 - HistVolatility 2-107
 - LinearInterp 2-123
- Find function 2-28
- Findb function 2-29
- FindFiles function 2-30
- flattening arrays 4-21
- FlushBulkInserts function 2-32
- folder class
 - method 6-43
 - properties 6-43
- folder collection
 - methods 6-47
 - property 6-51
- Font object properties
 - Chart control 8-75
 - TreeView control 9-18
- Footnote object method 8-77
- Footnote object properties 8-76
- For 2-34
- FormatDateTime function 2-34
- Frame object properties 8-78
- Function statement 2-36
- functions
 - declaring 1-107
 - defining 2-36
 - return values 3-32

G

- GenerateTree function 2-38
- GetAESction function 2-39
- GetAttachment function 2-40
- GetBiDoc function 2-43
- GetCalendarDate function 2-45
- GetControl function 2-47
- GetControlOccurrence function 2-49
- GetCwd function 2-51
- GetEnv function 2-52
- GetField function 2-52
- GetFile function 2-53
- GetGrid function 2-57
- GetHTMLText function 2-60
- GetImageExtents function 2-61
- GetInterlink function 2-62
- GetJavaClass function 2-65

- GetLevel0 function 2-66
- GetMessage function 2-68
- GetMessageInstance function 2-69
- GetNextNumber function 2-70
- GetNextNumberWithGaps function 2-72
- GetPage function 2-73
- GetPubContractInstance function 2-75
- GetRecord function 2-75
- GetRelField function 2-77
- GetRow function 2-78
- GetRowset function 2-79
- GetSelectedTreeNode function 2-80
- GetSession function 2-83
- GetSetID function 2-83
- GetSQL function 2-85
- GetStoredFormat function 2-87
- GetSubMessage function 2-88
- GetTreeNodeParent function 2-91
- GetTreeNodeRecordName function 2-92
- GetTreeNodeValue function 2-93
- GetURL function 2-95
- GetWLFieldValue function 2-96
- Global statement 2-97
- Gray function 2-98
- grid class
 - changing a grid name 2-58
 - declaring 5-77
 - GetGrid function 2-57
 - method 5-78
 - scope of 5-77
 - specifying grid column name 5-78
 - TreeView example 9-41
 - using in PeopleCode 5-76
- gridcolumn class
 - properties 5-82

H

- Hash function 2-100
- HermiteCubic function 2-101
- Hide function 2-101
- HideMenuItem function 2-103
- HideRow function 2-104
- HideScroll function 2-106
- HistVolatility function 2-107
- Hour function 2-108
- HTML
 - GetHTMLText function 2-60

I

- If 2-108
- image functions
 - DeleteImage 1-116
 - GetImageExtents 2-61
 - InsertImage 2-109

- image object methods 9-35
- image object properties 9-34
- ImageList control
 - associating with TreeView 9-3
 - declaring 9-30
 - image object 9-34
 - images collection 9-32
 - method 9-31
 - properties 9-30
 - quick reference 9-56
 - scope of 9-30
 - TreeView example 9-39
- images collection methods 9-32
- images collection properties 9-32
- implementing
 - Component Interface objects 4-124
 - tree classes 7-70
- Import manager
 - %Import system variable 11-4
- InsertImage function 2-109
- InsertRow function 2-111
- Int function 2-113
- internet functions
 - %EmailAddress variable 11-3
 - %Request variable 11-8
 - %Response variable 11-9
 - %UserDescription variable 11-11
 - EncodeURL 2-2
 - EncodeURLForQueryString 2-3
 - EscapeHTML 2-9
 - EscapeJavascriptString 2-10
 - EscapeWML 2-10
 - GenerateTree 2-38
 - GetHTMLText 2-60
 - GetMethodNames 2-68
 - GetURL 2-95
 - Unencode 3-121
 - ViewURL 3-135
- Internet Script
 - creating 5-87
- Internet Script classes
 - accessing 5-92
 - Cookie class 5-95
 - Cookie object properties 5-107
 - creating web libraries 5-87
 - overview 5-85
 - quick reference 12-21
 - Request class 5-94
 - Request object methods 5-95
 - Request object properties 5-98
 - Response class 5-95
 - Response object methods 5-102
 - scope of 5-93
 - security 5-89
 - URI 5-86
 - URL 5-86
 - when to use 5-91
- Intersection object properties 8-79

IsDaylightSavings function 2-113
 IsHidden function 2-115
 IsMenuItemAuthorized function 2-116
 IsModal function 2-117
 IsModalComponent function 2-118
 IsSearchDialog function 2-120
 IsUserInPermissionList function 2-121
 IsUserInRole function 2-121

J

Java class
 considerations using PeopleCode 5-118
 CreateJavaArray function 1-80
 CreateJavaObject function 1-80
 declaring 5-120
 error handling 5-119
 from Java to PeopleCode 5-114
 from PeopleCode to Java 5-111
 GetJavaClass function 2-65
 mapping PeopleCode to Java data types 5-117
 naming classes and packages 5-109
 scope of 5-120
 state management concerns 5-111
 supported versions of Java 5-109
 using the Java debuggin environment 5-119
 using your own classes 5-110

L

Label object properties 8-80
 Labels collection properties 8-82
 Language constructs
 Break 1-40
 Component 1-57
 Declare function 1-107
 Evaluate 2-11
 Exit 2-17
 For 2-34
 Function 2-36
 Global 2-97
 If 2-108
 Local 2-124
 Repeat 3-30
 Return 3-32
 While 3-139
 Language preferences
 %IsMultiLanguageEnabled variable 11-4
 %Language variable 11-4
 %Language_Base variable 11-4
 SetLanguage function 3-72
 LCoor object
 method 8-83
 properties 8-82
 leaf object
 methods 7-81

 properties 7-88
 Left function 2-122
 Legend object method 8-84
 Legend object properties 8-83
 Len function 2-122
 Lenb function 2-123
 level collection
 methods 7-91
 properties 7-93
 level object
 methods 7-93
 properties 7-94
 Light object properties 8-84
 Lightsource object method 8-86
 Lightsource object properties 8-85
 Lightsources collection methods 8-87
 Lightsources collection properties 8-87
 LinearInterp function 2-123
 Ln function 2-124
 Local statement 2-124
 Location object properties 8-88
 Log10 function 2-125
 loops
 exiting 1-40
 Lower function 2-126
 LTrim function 2-126

M

mapping
 functions to methods and properties 12-60
 old names to new names 12-62
 Marker object properties 8-89
 MarkWLItemWorked function 2-127
 Math functions
 %Abs meta-SQL function 10-5
 Acos 1-19
 Asin 1-37
 Atan 1-37
 Cos 1-76
 Cot 1-76
 Degrees 1-112
 Radians 3-18
 Sin 3-86
 Tan 3-103
 Menu functions
 CheckMenuItem 1-49
 DisableMenuItem 1-123
 EnableMenuItem 2-1
 HideMenuItem 2-103
 UnCheckMenuItem 3-120
 UnGray 3-122
 message class
 considerations with character fields 5-127
 considerations with partial records 5-126
 considerations with rowset 5-124
 CreateMessage function 1-82

- declaring 5-122
- error handling 5-127
- GetMessage function 2-68
- GetMessageInstance function 2-69
- GetPubContractInstance function 2-75
- GetSubMessage function 2-88
- methods 5-129
- populating 5-122
- properties 5-149
- ReturnToServer function 3-33
- scope of 5-122
- Message display functions
 - EndMessage 2-4
 - Error 2-7
 - MessageBox 2-127
 - MsgGet 2-135
 - MsgGetExplainText 2-136
 - MsgGetText 2-138
 - Prompt 3-14
 - Warning 3-137
 - WinMessage 3-143
- MessageBox function 2-127
- Meta-SQL functions 10-1
 - %Abs 10-5
 - %Concat 10-6
 - %CurrentDateIn 10-6
 - %CurrentDateOut 10-6
 - %CurrentDateTimeIn 10-6
 - %CurrentDateTimOut 10-7
 - %CurrentTimeIn 10-7
 - %CurrentTimeOut 10-7
 - %DateAdd 10-7
 - %DateDiff 10-7, 10-8
 - %DateIn 10-8
 - %DateOut 10-8
 - %DateTimeIn 10-9
 - %DateTimOut 10-9
 - %DecDiv 10-9
 - %DecMult 10-10
 - %DTTM 10-11
 - %FirstRows 10-13
 - %OPRCLAUSE 10-33
 - %Round 10-34
 - %SUBREC 10-36
 - %Substring 10-36
 - %TimeIn 10-40
 - %TimeOut 10-40
 - %TrimSubstr 10-40
 - %Truncate 10-40
 - %TruncateTable 10-41
- date considerations 10-1
- placement considerations 10-2
- shortcuts 10-44
- Minute function 2-134
- Mod function 2-134
- Modal component functions
 - DoModalComponent 1-128
 - IsModalComponent 2-118

- Month function 2-135
- MsgGet function 2-135
- MsgGetExplainText function 2-136
- MsgGetText function 2-138

N

- NextEffDt function 3-1
- NextRelEffDt function 3-1
- node object (tree)
 - methods 7-96
 - properties 7-108
- node object methods
 - TreeView control 9-25
- node object properties
 - TreeView control 9-21
- nodes collection methods
 - TreeView control 9-26
- nodes collection property
 - TreeView control 9-26
- None function 3-2
- Numeric functions
 - %DecDiv 10-9
 - %DecMult 10-10
 - %Round 10-34
 - %Truncate 10-40
 - Abs 1-16
 - Acos 1-19
 - Asin 1-37
 - Cos 1-76
 - Cot 1-76
 - Degress 1-112
 - Exp 2-18
 - Fact 2-21
 - Int 2-113
 - Ln 2-124
 - Log10 2-125
 - Mod 2-134
 - Product 3-13
 - Radians 3-18
 - Rand 3-19
 - Round 3-37
 - Sign 3-85
 - Sin 3-86
 - Sqrt 3-95
 - Tan 3-103
 - Truncate 3-119

O

- ObjectDoMethod function 3-3
- ObjectGetProperty function 3-4
- ObjectSetProperty function 3-6
- OLE functions
 - CreateObject 1-83
 - ObjectDoMethod 3-3

- ObjectGetProperty 3-4
- ObjectSetProperty 3-6
- OnlyOne function 3-8
- OnlyOneOrNone function 3-9
- Operator Security functions
 - %OPRCLAUSE meta-SQL 10-33
- Operator Security system variables
 - %EmployeeId 11-3

P

page appearance functions

- GetPage 2-73
- Gray 2-98
- Hide 2-101
- HideRow 2-104
- HideScroll 2-106
- IsHidden 2-115
- SetLabel 3-71
- Unhide 3-124
- UnhideRow 3-125
- UnhideScroll 3-127

page class

- declaring 6-1
- properties 6-2
- scope of 6-1

Pen object properties 8-91

PeopleBooks

- CD-ROM, ordering lxvii
- printed, ordering lxvii

PeopleCode

- from Java 5-114
- Java class considerations 5-118
- mapping to Java data types 5-117
- to Java 5-111

PeopleCode classes

- AESession 4-2
- array 4-13
- business interlink 4-40
- Component Interface 4-116
- field 5-1
- file 5-24
- grid 5-75
- gridcolumn 5-82
- Internet Script 5-85
- Java 5-109
- message 5-121
- page 6-1
- PortalRegistry 6-3
- ProcessRequest 6-99
- query 6-120
- quick reference 12-1
- record 6-197
- row 6-221
- rowset 6-235
- search 7-1
- session 7-19

- SQL 7-45

- tree 7-63

PeopleCode events

- chart control 8-20

PeopleCode syntax

- quick reference 12-63

PermissionValue class

- properties 6-69

PermissionValue collection

- methods 6-70
- property 6-73

picture object properties

- TreeView control 9-29

PingNode function 3-11

Plot object properties 8-93

Plotbase object properties 8-100

portal registry classes

- AttributeValue collection methods 6-65
- AttributeValue collection property 6-68
- AttributeValue properties 6-64
- content provider collection methods 6-39
- content provider collection properties 6-42
- content provider properties 6-37
- content reference collection methods 6-60
- content reference collection property 6-63
- content reference methods 6-51
- content reference properties 6-52
- content references 6-14
- declaring 6-22
- deleting content considerations 6-21
- error handling 6-10
- examples 6-73
- folder collection methods 6-47
- folder collection property 6-51
- folder method 6-43
- folder properties 6-43
- hierarchy 6-13
- life-cycle 6-12
- methods 6-25
- naming conventions 6-21
- overview 6-3
- PermissionValue collection methods 6-70
- PermissionValue collection property 6-73
- PermissionValue properties 6-69
- portal registry collection methods 6-35
- portal registry collection property 6-37
- properties 6-34
- quick reference 12-23
- saving content considerations 6-22
- scope of 6-23
- search class methods 7-7
- security considerations 6-6
- session object methods 6-23
- using 6-6
- using Cascading Permissions 6-7
- using PermissionValue 6-7
- ValidFrom and ValidTo 6-9

PriorEffDt function 3-11

- PriorRelEffDt function 3-12
- PriorValue function 3-13
- ProcessRequest class
 - considerations 6-100
 - declaring 6-102
 - example 6-100
 - method 6-103
 - properties 6-104
 - scope of 6-102
- Product function 3-13
- promotion in arrays 4-21
- Prompt function 3-14
- Proper function 3-16
- PSMessage object
 - properties 7-33
- PSMessages collection
 - accessing 7-30
 - methods 7-31
 - property 7-33
- PutAttachment function 3-16

Q

- query class
 - methods 6-141
 - properties 6-152
- query classes 6-120
 - collections in 6-125
 - declaring 6-134
 - error handling 6-133
 - hierarchy 6-127
 - life-cycle 6-126
 - metadata class properties 6-185
 - metadata collection methods 6-183
 - metadata collection property 6-185
 - overview 6-121
 - PermissionList class property 6-189
 - PermissionList collection methods 6-186
 - query class methods 6-141
 - query class properties 6-152
 - query collection 6-138
 - query collection methods 6-138
 - query collection property 6-140
 - query fields 6-125
- QueryCriteria
 - collection property 6-166
- QueryCriteria class methods 6-166
- QueryCriteria class properties 6-169
- QueryCriteria collection methods 6-164
- QueryDBRecord class methods 6-192
- QueryDBRecord class properties 6-193
- QueryDBRecord collection methods 6-189
- QueryDBRecord collection property 6-191
- QueryDBRecordField class properties 6-196
- QueryDBRecordField collection methods 6-193
- QueryDBRecordField collection property 6-195

- QueryExpression class properties 6-178
- QueryExpression collection methods 6-175
- QueryExpression collection property 6-177
- QueryField class properties 6-160
- QueryField collection methods 6-158
- QueryField collection property 6-160
- QueryOutputFields 6-125
- QueryRecord
 - collection property 6-156
- QueryRecord class properties 6-157
- QueryRecord collection methods 6-154
- QueryRecordHierarchy class properties 6-182
- QueryRecordHierarchy collection methods 6-179
- QueryRecordHierarchy collection property 6-181
- QuerySelectedFields 6-125
- quick reference 12-27
- scope of 6-134
- session object methods 6-135
- using metadata 6-130
- working with criteria and expressions 6-128
- query collection
 - methods 6-138
 - property 6-140
- QueryCriteria
 - class methods 6-166
 - class properties 6-169
 - collection methods 6-164
- QueryCriteria collection property 6-166
- QueryDBRecord
 - class methods 6-192
 - class properties 6-193
 - collection methods 6-189
 - collection property 6-191
- QueryDBRecordField
 - class properties 6-196
 - collection methods 6-193
 - collection property 6-195
- QueryExpression
 - class properties 6-178
 - collection methods 6-175
 - collection property 6-177
- QueryField
 - class properties 6-160
 - collection methods 6-158
 - collection property 6-160
- QueryRecord
 - class properties 6-157
 - collection methods 6-154
- QueryRecord collection property 6-156
- QueryRecordHierarchy
 - class properties 6-182
 - collection methods 6-179
 - collection property 6-181

R

- Radians function 3-18
- Rand function 3-19
- record class
 - declaring 6-199
 - methods 6-199
 - properties 6-218
 - scope of 6-199
 - SQL methods 6-198
- RecordChanged function 3-19
- RecordDeleted function 3-21
- RecordNew function 3-24
- Rect object properties 8-101
- RefreshTree function 3-26
- regional settings object
 - about 7-22
 - properties 7-37
- RemoteCall function 3-27
- Repeat 3-30
- Replace function 3-31
- repository API
 - quick reference 12-20
- Rept function 3-32
- Request object methods 5-95
- Request object properties 5-98
- Response object methods 5-102
- Return 3-32
- ReturnToServer function 3-33
- RevalidatePassword function 3-35
- Right function 3-36
- Round function 3-37
- RoundCurrency function 3-37
- row class
 - declaring 6-221
 - methods 6-222
 - properties 6-228
 - scope of 6-222
- RowFlush function 3-38
- RowScrollSelect function 3-39
- RowScrollSelectNew function 3-42
- rowset class
 - declaring 6-236
 - methods 6-237
 - properties 6-259
 - scope of 6-236
- RTrim function 3-44
- Running executable files
 - Exec function 2-12
 - WinExec function 3-141

S

- Saving functions
 - DoSave 1-132
 - DoSaveNow 1-133

- ScheduleProcess function 3-44
- ScrollFlush function 3-48
- ScrollSelect function 3-49
- ScrollSelect functions
 - RowScrollSelect 3-39
 - RowScrollSelectNew 3-42
 - ScrollFlush 3-48
 - ScrollSelect 3-49
 - ScrollSelectNew 3-53
 - SortScroll 3-87
- ScrollSelectNew function 3-53
- search classes 7-1
 - declaring 7-6
 - error handling 7-4
 - examples 7-15
 - hierarchy 7-3
 - overview 7-1
 - PortalRegistry object methods 7-7
 - quick reference 12-19
 - scope of 7-6
 - SearchField collection methods 7-13
 - SearchField collection property 7-14
 - SearchField object properties 7-14
 - SearchQuery object methods 7-9
 - SearchQuery object properties 7-9
 - SearchResult collection methods 7-11
 - SearchResult collection property 7-12
 - SearchResult object properties 7-12
 - session class method 7-6
- Search functions
 - AddKeyListItem 1-27
 - ClearKeyList 1-50
 - ClearSearchDefault 1-51
 - ClearSearchEdit 1-52
 - IsSearchDialog 2-120
 - SetSearchDefault 3-76
 - SetSearchDialogBehavior 3-77
 - SetSearchEdit 3-78
- search query *See* search classes
- SearchField collection
 - methods 7-13
 - property 7-14
- SearchField object
 - properties 7-14
- SearchQuery object
 - method 7-9
 - properties 7-9
- SearchResult collection
 - methods 7-11
 - property 7-12
 - using 7-2
- SearchResult object
 - properties 7-12
 - understanding 7-2
- Second function 3-55
- Secondary page functions
 - DoModal 1-126
 - EndModal 2-6

- IsModal 2-117
- security
 - %AuthenticationToken variable 11-1
 - %ExternalAuthInfo variable 11-3
 - %NavigatorHomePermissionList variable 11-6
 - %PasswordExpired variable 11-8
 - %PermissionLists variable 11-8
 - %PrimaryPermissionList variable 11-8
 - %ProcessProfilePermissionList variable 11-8
 - %PSAuthResult variable 11-8
 - %ResultDocument variable 11-9
 - %Roles variable 11-9
 - %RowSecurityPermissionList variable 11-9
 - %SignonUserId variable 11-10
 - %SignonUserPswd variable 11-10
 - %UserId variable 11-11
 - AllowEmplidChg function 1-35
 - Decrypt function 1-111
 - Encrypt function 2-4
 - ExecuteRolePeopleCode function 2-14
 - ExecuteRoleQuery function 2-15
 - ExecuteRoleWorkflowQuery function 2-16
 - Hash function 2-100
 - Internet Scripts 5-89
 - IsMenuItemAuthorized function 2-116
 - IsUserInPermissionList function 2-121
 - IsUserInRole function 2-121
 - portal registry classes 6-6
 - RevalidatePassword function 3-35
 - SetAuthenticationResult function 3-59
 - SetPasswordExpired function 3-75
 - SwitchUser function 3-101
- SendMail function 3-55
- Series object
 - method 8-105
 - properties 8-102
 - special properties 8-103
- SeriesCollection collection
 - method 8-105
 - properties 8-105
- SeriesMarker object properties 8-106
- SeriesPosition object properties 8-106
- session class
 - access to PeopleSoft system 7-20
 - Component Interface methods 4-150
 - declaring 7-24
 - error handling 7-20
 - methods 7-25
 - portal registry methods 6-23
 - properties 7-28
 - PSMessages collection 7-30
 - PSMessages collection methods 7-31
 - PSMessages collection property 7-33
 - PSMessages object properties 7-33
 - query classes methods 6-135
 - quick reference 12-14
 - regional settings overview 7-22
 - regional settings properties 7-37
 - scope of 7-25
 - search method 7-6
 - security 7-20
 - trace settings overview 7-22
 - trace settings properties 7-40
 - tree classes methods 7-75
- SetAuthenticationResult function 3-59
- SetChannelStatus function 3-60
- SetControlValue function 3-61
- SetCursorPos function 3-64
- SetDefault function 3-66
- SetDefaultAll function 3-67
- SetDefaultNext function 3-68
- SetDefaultNextRel function 3-68
- SetDefaultPrior function 3-69
- SetDefaultPriorRel function 3-69
- SetLabel function 3-71
- SetLanguage function 3-72
- SetNextPage function 3-74
- SetPasswordExpired function 3-75
- SetReEdit function 3-75
- SetSearchDefault function 3-76
- SetSearchDialogBehavior function 3-77
- SetSearchEdit function 3-78
- SetTempTableInstance function 3-79
- SetTracePC function 3-80
- SetTraceSQL function 3-83
- Shadow object properties 8-108
- Sign function 3-85
- Sin function 3-86
- SinglePaymentPV function 3-86
- SortScroll function 3-87
- special properties
 - chart control 8-4
- Split function 3-88
- SQL
 - %FirstRows meta-SQL function 10-13
 - %InsertSelect meta-SQL function 10-14
 - %InsertValues meta-SQL function 10-17
 - %Join meta-SQL function 10-18
 - %KeyEqual meta-SQL function 10-20
 - %KeyEqualNoEffDt meta-SQL function 10-21
 - %Like meta-SQL function 10-22
 - %LikeExact meta-SQL function 10-24
 - %List meta-SQL function 10-27
 - %ListBind meta-SQL function 10-31
 - %ListEqual meta-SQL function 10-32
 - %OldKeyEqual meta-SQL function 10-33
 - %SQL meta-SQL function 10-34
 - %SQLRows variable 11-10
 - %Table meta-SQL function 10-37
 - %UpdatePairs meta-SQL function 10-42
 - binding 7-46
 - CreateSQL function 1-90
 - DeleteSQL function 1-119
 - executing 7-46
 - ExpandSqlBinds function 2-20
 - FetchSQL function 2-22

- FlushBulkInserts function 2-32
- GetSQL function 2-85
- Meta functions 10-1
- reusing a cursor 7-49
- SetTempTableInstance function 3-79
- SQLExec function 3-89
- StoreSQL function 3-97
- SQL class
 - application engine considerations 7-51
 - binding and executing statements 7-46
 - CreateSQL function 1-90
 - declaring 7-51
 - DeleteSQL function 1-119
 - fetching 7-48
 - FetchSQL function 2-22
 - GetSQL function 2-85
 - global declaration considerations 7-51
 - methods 7-52
 - properties 7-58
 - record methods 7-46
 - reusing a cursor 7-49
 - scope of 7-51
 - SELECT styles 7-48
 - SQL definition 7-46
 - SQLExec function 3-89
 - StoreSQL function 3-97
- SQL date and time functions
 - %DateIn 10-8
 - %DateOut 10-8
 - %DateTimeIn 10-9
 - %DateTimeOut 10-9
 - %DTTM 10-11
 - %TimeIn 10-40
 - %TimeOut 10-40
- SQL definition
 - creating 7-46
- SQL shortcuts
 - %Delete 10-44
 - %Insert 10-44
 - %SelectAll 10-44
 - %SelectByKey 10-45
 - %SelectByKeyEffDt 10-45
 - %SelectDistinct 10-45
 - %Update 10-45
- SQLExec function 3-89
- Sqrt function 3-95
- StatLine object properties 8-109
- StatLine object special properties 8-109
- StopFetching function 3-96
- StoreSQL function 3-97
- String function 3-98
- String functions
 - %Concat 10-6
 - %Substring 10-36
 - %TextIn 10-38
 - %TrimSubstr 10-40
 - %Upper 10-43
 - Clean 1-50
 - Code 1-52
 - Codeb 1-53
 - DBCSTrim 1-107
 - Exact 2-12
 - ExpandBindVar 2-18
 - Find 2-28
 - Findb 2-29
 - Left 2-122
 - Len 2-122
 - Lenb 2-123
 - Lower 2-126
 - LTrim 2-126
 - Proper 3-16
 - Replace 3-31
 - Rept 3-32
 - Right 3-36
 - RTrim 3-44
 - String 3-98
 - Substitute 3-99
 - SubString 3-100
 - Substringb 3-100
 - Upper 3-131
 - Value 3-131
- Subrecords
 - %SUBREC meta-SQL function 10-36
- Substitute function 3-99
- SubString function 3-100
- Substringb function 3-100
- SwitchUser function 3-101
- System variables 11-1
 - %AsOfDate 11-1
 - %AuthenticationToken 11-1
 - %BPName 11-1
 - %ClientDate 11-1
 - %ClientTimeZone 11-2
 - %CompIntfcName 11-2
 - %Component 11-2
 - %Currency 11-2
 - %Date 11-3
 - %DateTime 11-3
 - %DbName 11-3
 - %DbType 11-3
 - %EmailAddress 11-3
 - %EmployeeId 11-3
 - %ExternalAuthInfo 11-3
 - %Import 11-4
 - %IsMultiLanguageEnabled 11-4
 - %Language 11-4
 - %Language_Base 11-4
 - %Market 11-4
 - %MaxInterlinkSize 11-5
 - %MaxMessageSize 11-6
 - %Menu 11-6
 - %MessageAgent 11-6
 - %Mode 11-6
 - %NavigatorHomePermissionList 11-6
 - %OperatorClass 11-7
 - %OperatorId 11-7

- %OperatorRowLevelSecurityClass 11-7
- %Page 11-7
- %Panel 11-7
- %PanelGroup 11-8
- %PasswordExpired 11-8
- %PermissionLists 11-8
- %PrimaryPermissionList 11-8
- %ProcessProfilePermissionList 11-8
- %PSAuthResult 11-8
- %Request 11-8
- %Response 11-9
- %ResultDocument 11-9
- %Roles 11-9
- %RowSecurityPermissionList 11-9
- %ServerTimeZone 11-9
- %Session 11-10
- %SignonUserId 11-10
- %SignonUserPswd 11-10
- %SQLRows 11-10
- %Time 11-11
- %UserDescription 11-11
- %UserId 11-11
- %WLInstanceID 11-11
- %WLName 11-11

T

- Tan function 3-103
- TextLayout object properties 8-110
- Tick object properties 8-112
- Time and date functions
 - %DateAdd 10-7
 - %DateDiff 10-7, 10-8
 - %DateIn 10-8
 - %DateOut 10-8
 - %DateTimeIn 10-9
 - %DateTimeOut 10-9
 - %DTTM 10-11
 - %TimeAdd meta-SQL function 10-39
 - %TimeIn 10-40
 - %TimeOut 10-40
 - AddToDate 1-30
 - AddToDateTime 1-31
 - AddToTime 1-32
 - ConvertDatetimeToBase 1-70
 - ConvertTimeToBase 1-73
 - Date 1-97
 - Date3 1-97
 - DatePart 1-98
 - DateTime6 1-98
 - DateTimeToLocalizedString 1-99
 - DateTimeToTimeZone 1-101
 - DateTimeValue 1-103
 - DateValue 1-104
 - Day 1-105
 - Days 1-105
 - Days360 1-106

- Days365 1-106
- FormatDateTime 2-34
- GetCalendarDate 2-45
- Hour 2-108
- IsDaylightSavings 2-113
- Minute 2-134
- Month 2-135
- Second 3-55
- Time 3-104
- Time3 3-104
- TimePart 3-105
- TimeToTimeZone 3-106
- TimeValue 3-107
- TimeZoneOffset 3-108
- Weekday 3-139
- Year 3-147
- Time function 3-104
- Time3 function 3-104
- TimePart function 3-105
- TimeToTimeZone function 3-106
- TimeValue function 3-107
- TimeZoneOffset function 3-108
- Title object method 8-114
- Title object properties 8-113
- TotalRowCount function 3-109
- Trace functions
 - SetTracePC 3-80
 - SetTraceSQL 3-83
- trace settings object
 - about 7-22
 - properties 7-40
- Transfer function 3-110
- Transfer functions
 - AddKeyListItem 1-27
 - ClearKeyList 1-50
 - DoModalComponent 1-128
 - IsModalComponent 2-118
 - SetNextPage 3-74
 - Transfer 3-110
 - TransferPage 3-115
- TransferPage function 3-115
- tree classes
 - branch collection methods 7-80
 - branch collection properties 7-81
 - branch collections 7-79
 - collections in 7-66
 - declaring 7-69
 - error handling 7-67
 - implementing 7-70
 - insertion verification 7-69
 - leaf 7-81
 - leaf methods 7-81
 - leaf properties 7-88
 - level 7-93
 - level collection 7-91
 - level collection methods 7-91
 - level methods 7-93
 - level properties 7-94

- node methods 7-96
- node object 7-96
- node properties 7-108
- quick reference 7-163, 12-34
- relationship between 7-65
- scope of 7-70
- session object methods 7-75
- tree 7-114
- tree collection 7-145
- tree collection method 7-146
- tree collection properties 7-147
- tree methods 7-115
- tree properties 7-134
- tree structure 7-148
- tree structure collection 7-161
- tree structure collection method 7-162
- tree structure collection properties 7-162
- tree structure methods 7-148
- tree structure properties 7-151
- TreeView control example 9-35
- tree collection
 - method 7-146
 - properties 7-147
- Tree control functions
 - GetSelectedTreeNode 2-80
 - GetTreeNodeParent 2-91
 - GetTreeNodeRecordName 2-92
 - GetTreeNodeValue 2-93
 - RefreshTree 3-26
- tree structure collection
 - method 7-162
 - properties 7-162
- tree structure object
 - methods 7-148
 - properties 7-151
- TreeDetailInNode function 3-116
- TreeView control
 - associating with ImageList 9-3
 - declaring 9-9
 - events 9-4
 - examples 9-35
 - font object 9-18
 - grid example 9-41
 - ImageList example 9-39
 - methods 9-17
 - node object 9-19
 - nodes collection 9-25
 - picture object 9-28
 - properties 9-10
 - quick reference 9-54
 - scope of 9-9
 - Tree API example 9-35
- TriggerBusinessEvent function 3-118
- Truncate function 3-119

U

- UnCheckMenuItem function 3-120
- Unencode function 3-121
- UnGray function 3-122
- UnHide function 3-124
- UnhideRow function 3-125
- UnhideScroll function 3-127
- UniformSeriesPV function 3-128
- UpDateSysVersion function 3-129
- UpdateValue function 3-129
- Upper function 3-131
- URL vs. URI 5-86
- User Security functions
 - AllowEmplidChg 1-35
 - Decrypt 1-111
 - EncodeURL 2-4
 - ExecuteRolePeopleCode 2-14
 - ExecuteRoleQuery 2-15
 - ExecuteRoleWorkflowQuery 2-16
 - Hash 2-100
 - IsMenuItemAuthorized 2-116
 - IsUserInPermissionList 2-121
 - IsUserInRole 2-121
 - RevalidatePassword 3-35
 - SetAuthenticationResult 3-59
 - SetPasswordExpired 3-75
 - SwitchUser 3-101
- using
 - content references 6-14

V

- validation functions
 - Error 2-7
 - RevalidatePassword 3-35
 - SetCursorPos 3-64
 - SetReEdit 3-75
 - Warning 3-137
- Value function 3-131
- ValueScale object properties 8-114
- variables
 - declaring 1-57, 2-97, 2-124
- View3D object method 8-118
- View3D object properties 8-117
- ViewAttachment function 3-132
- ViewURL function 3-135
- VtColor object method 8-120
- VtColor object properties 8-119
- VtFont object properties 8-121

W

- Wall object properties 8-122
- Warning 3-137

- web libraries
 - creating 5-87
 - GetMethodNames function 2-68
 - security 5-89
- Weekday function 3-139
- Weighting object properties 8-123, 8-124
- While 3-139
- WinEscape function 3-140
- WinExec function 3-141
- WinMessage function 3-143
- Workflow functions
 - GetWLFieldValue 2-96

- MarkWLItemWorked 2-127
- TriggerBusinessEvent 3-118
- Workflow system variables
 - %BPName 11-1
- Worklist system variables
 - %WLInstanceID 11-11
 - %WLName 11-11

Y

- Year function 3-147