



PeopleTools 8.12 PeopleCode Developer's Guide

SKU MTPDr8SP1B 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	xv
Before You Begin.....	xvi
Related Documentation	xvi
Documentation on the Internet.....	xvii
Documentation on CD-ROM	xvii
Hardcopy Documentation	xvii
Typographical Conventions and Visual Cues.....	xviii
Comments and Suggestions.....	xx

Chapter 1

Introducing What's New

Objects and Classes	1-1
List of Classes	1-1
Dot Notation and Objects	1-3
Instantiating Objects.....	1-4
Code Enhancements Using Dot Notation.....	1-4
Existing Code	1-4
Re-Written Code	1-5
Data Buffer Access	1-6
Instead of SQLExec.	1-7
Using the Record Class	1-7
SQL Definitions and the SQL Class	1-10
Record Class vs. SQL Object.....	1-11
Using SQL Definitions with SQLExec	1-11
Using Record Objects With SQLExec	1-11
Meta-SQL.....	1-12
Standalone Rowset	1-13
Performance Enhancement for SQLExec and ScrollSelect	1-14
Using Literal Parameters.....	1-16
Grids	1-16
Existing Code	1-16
Re-Written Code	1-17

Accessing PeopleCode Programs	1-17
Case Sensitivity in PeopleCode Programs.....	1-19
Application Reviewer	1-20
Runtime Checking	1-21
Additional Features.....	1-22
New Component Variable Type.....	1-22
New Data Types.....	1-22
Automatic Backup of PeopleCode	1-22
Enhanced Find In Function	1-22
Refreshing your Page	1-23
Using Constants Instead of Numeric Values.....	1-23
Enhanced Transfer Functionality	1-24
Attachment PeopleCode	1-24
Mapping of Functions to Methods and Properties.....	1-24
Mapping of Old Names to New Names.....	1-26

Chapter 2

Understanding PeopleCode and Events

Accessing PeopleCode in Application Designer (Overview).....	2-2
Record Field PeopleCode	2-4
Accessing Record Field PeopleCode From a Record Definition	2-4
Accessing Record Field PeopleCode From a Page Definition.....	2-6
Fields and Record Fields	2-7
Record Field Event Set.....	2-7
Component Record Field PeopleCode.....	2-8
Component Record Field Event Set	2-9
Component Record PeopleCode	2-9
Component Record Event Set	2-10
Component PeopleCode	2-11
Component Event Set.....	2-11
Page PeopleCode	2-12
Page Activate Event	2-13
Page Field Control PeopleCode.....	2-13
Page Field Event Set	2-14
Menu Item PeopleCode	2-14
Menu Item ItemSelected Event	2-16
Application Message PeopleCode	2-17
Accessing Message PeopleCode	2-17
Accessing Message Channel PeopleCode.....	2-18
Application Message Event Sets	2-19

How PeopleCode Programs are Stored and Saved	2-20
Automatic Backup of PeopleCode	2-20
Copying PeopleCode with a Parent Definition	2-21
Upgrading PeopleCode Programs.....	2-21

Chapter 3

Using the PeopleCode Editor

Navigating Between PeopleCode Programs	3-1
Understanding the PeopleCode Editor Window	3-1
Using the Drop-down Definition List	3-2
Selecting a Record Field	3-3
Selecting a Component Definition	3-3
Selecting a Page or ActiveX Control	3-4
Selecting a Menu Item.....	3-4
Selecting a Message or Message Subscription.....	3-5
Selecting a Message Channel.....	3-6
Using the Drop-down Event List	3-6
Using the PeopleCode Editor.....	3-7
Editing Functions	3-8
Find and Replace.....	3-8
Validating Syntax.....	3-9
Auto Formatting	3-10
Drag-and-Drop Editing	3-10
Accessing PeopleCode External Functions.....	3-10
Accessing Definitions and Associated PeopleCode.....	3-11
Context-Sensitive Help	3-12
Choosing a Font for the PeopleCode Editor.....	3-12
Generating PeopleCode using Drag-and-Drop	3-13
Generating Definition References	3-13
Generating PeopleCode for a Business Interlink	3-14
Generating PeopleCode for a Component Interface.....	3-15

Chapter 4

Introducing the SQL Editor

Understanding the SQL Editor Window.....	4-1
Accessing the SQL Editor	4-2
SQL Definitions	4-2
Dynamic View or SQL View Records	4-4
Application Engine Programs	4-5
Using the SQL Editor	4-6

Chapter 5

PeopleCode Language

Data Types	5-1
Conventional Data Types	5-1
Object-Based Data Types	5-2
Data Buffer Access Types	5-2
Page Display Types	5-2
Internet Script Types	5-2
Miscellaneous Object-Based Types	5-3
API Object Type	5-3
Comments and Statements	5-4
Comments	5-4
Statements	5-4
Separators	5-5
Assignment Statements	5-5
Language Constructs	5-6
Functions as Subroutines	5-6
Control Statements	5-6
Branching Statements	5-6
For Loops	5-8
Conditional Loops	5-9
Functions	5-10
Defining Functions	5-10
Declaring Functions	5-11
Calling Functions	5-11
Function Return Values	5-12
Expressions	5-12
Constants	5-13
Numeric	5-13
String Constants	5-13
Boolean Constants	5-13
Functions as Expressions	5-14
Variables	5-14
User-Defined Variable Declaration and Scope	5-14
Using User-Defined Variables	5-16
Passing Variables to Functions	5-17
System Variables	5-17
Metastings	5-17
Record Field References	5-18
Record Field Reference Syntax	5-18

Legal Record Field Names.....	5-19
Definition Name References	5-19
Legal and Illegal Definition Names	5-20
Reserved Word Summary Table	5-20
Operators.....	5-21
Math Operators.....	5-21
Operations on Dates and Times	5-22
String Concatenation.....	5-22
@ Operator	5-22
Comparison Operators	5-24
Boolean Operators.....	5-24

Chapter 6

Understanding Objects and Classes in PeopleCode

What is a Class?	6-1
What is an Object?	6-1
Instantiating Objects.....	6-2
Working with Objects	6-2
Object Properties	6-3
Object Methods	6-3
Object Assignment	6-5
Passing Objects	6-6

Chapter 7

Using Methods and Built-in Functions

Restrictions on Method and Function Use.....	7-1
Think-Time Functions.....	7-1
WinMessage and MessageBox	7-2
Program Execution with Fields not in the Data Buffer	7-4
Errors and Warnings	7-4
DoSave	7-4
Record Object Database Methods.....	7-5
SQL Object Methods and Functions	7-5
Component Interface Restricted Functions	7-6
CallAppEngine.....	7-6
ReturnToServer.....	7-7
GetPage	7-7
GetGrid.....	7-7
GetControl.....	7-7
Publish Method	7-8

Implementing Modal Transfers	7-8
Considerations Before Implementing a Modal Transfer	7-9
Using the ImageReference Field.....	7-11
Using PeopleCode with PeopleSoft Internet Architecture.....	7-12
Internet Scripts	7-12
Unsupported Functions	7-12
Using the Field Object Style Property.....	7-12
HTML Area.....	7-14
Using HTML Definitions and the GetHTMLText Function.....	7-15
Using HTML Definitions and the GetJavaScriptURL Method.....	7-17
Increasing the Internet/Tuxedo Timeout	7-18
Inserting using PeopleCode	7-18
Using the GenerateTree Function.....	7-19
Building your HTML Tree Page	7-20
The HTML Tree Rowset Records	7-21
HTML Tree End-User Actions (Events).....	7-24
Customizing the PeopleCode for the HTML Tree	7-25
PostBuild PeopleCode Example	7-27
FieldChange PeopleCode Example.....	7-31
Using the Attachment Functions.....	7-37
Using the Select and SelectNew Methods	7-39
What Select Does	7-40
Select Syntax.....	7-40
Specifying Child Rowsets	7-41
Specifying the Select Record	7-42
The WHERE Clause	7-42
Using Select like RowScrollSelect.....	7-42
Using Standalone Rowsets	7-43
The Fill Method.....	7-44
The CopyTo Method.....	7-44
Adding Child Rowset.....	7-45
Using Standalone Rowsets to Write a File.....	7-46
Using Standalone Rowsets to Read a File.....	7-49
Errors and Warnings	7-51
Syntax of Errors and Warnings	7-51
Errors, Warnings, and Edits	7-51
Errors and Warnings in FieldEdit	7-52
Errors and Warnings in SaveEdit.....	7-52
Errors and Warnings in RowSelect.....	7-52
Errors and Warnings in RowDelete	7-53

Errors and Warnings in Other Events	7-53
Using RemoteCall.....	7-53
RemoteCall Components.....	7-55
PeopleCode API.....	7-55
Remote Program API	7-56
PeopleSoft RemoteCall Service	7-57
RemoteCall and Process Scheduler.....	7-57
Modifying a Process Scheduler Program to Use RemoteCall.....	7-57
Programming Guidelines.....	7-58

Chapter 8

Referencing Data in the Component Buffer

Component Buffer Structure and Contents.....	8-1
Comparing Rowsets to Scrolls	8-3
What Record Fields Are in the Component Buffer?	8-3
Contextual References	8-4
Understanding Current Context	8-4
Contextual Reference Processing Order	8-5
Contextual Row References	8-6
Contextual Buffer Field References	8-7
Contextual Buffer Field Reference Ambiguity	8-7
Ambiguous Contextual References to Buffer Fields on Level Zero	8-8
Resolving Ambiguous References with Objects.....	8-8
References Using Scroll Path Syntax and Dot Notation.....	8-9
Scroll Path Syntax in PeopleTools 7.5	8-9
Scroll Path Syntax with RECORD.recordname	8-10
Scroll Path Syntax with SCROLL.scrollname	8-11
Scroll Level, Row, and Buffer Field References.....	8-12
Referring to Scroll Levels	8-13
Referring to Rows	8-15
Referring to Buffer Fields	8-16
Using CurrentRowNumber	8-17
Looping through Scroll Levels.....	8-18
Scroll Path Syntax prior to PeopleTools 7.5	8-18

Chapter 9

Data Buffer Access

Access Classes	9-1
Data Buffer Model and Data Access Objects.....	9-1
Data Buffer Classes Examples.....	9-3

Object Creation Examples.....	9-5
Accessing Level 0	9-5
Rowset Object	9-6
Row Object	9-9
Record Object	9-10
Field Object.....	9-12
Traversing the Data Buffer Hierarchy Example.....	9-13
Rowset.....	9-13
Rowsets contain rows.....	9-14
Rows can contain child rowsets	9-14
Rowsets contain rows.....	9-14
Rows can contain child rowsets, rowsets contain rows	9-14
Rows contain records	9-15
Records contain fields	9-15
Using shortcuts.....	9-16
Traversing a Rowset Example.....	9-17
Using a Hidden Work Scroll Example	9-19
Current Context	9-22
Creating Records or Rowsets and Current Context	9-23
Accessing Secondary Component Buffer Data.....	9-23
Instantiating Rowsets using non-Component Buffer data	9-23

Chapter 10

PeopleCode and the Component Processor

Events Outside the Component Processor Flow	10-1
How PeopleCode Programs Are Triggered	10-2
Accessing PeopleCode Programs.....	10-3
Execution Order of Events and PeopleCode	10-4
Events after User Changes Field	10-5
Events after User Saves.....	10-5
Component Processor Behavior.....	10-7
From Page Start to Page Display.....	10-7
End-User Actions in the Component	10-8
Processing Sequences	10-9
Default Processing.....	10-10
Field-Level Default Processing.....	10-10
Default Processing on Component Level.....	10-11
Search Processing in Update Modes	10-12
Search Processing in Add Modes.....	10-14
Component Build Processing in Update Modes.....	10-16

Row Select Processing	10-18
Component Build Processing in Add Modes	10-19
Field Modification	10-20
Row Insert Processing	10-23
Row Delete Processing	10-24
PushButtons	10-26
Prompts	10-26
Pop-up Menu Display	10-26
ItemSelected Processing	10-27
PSLostFocus Processing	10-27
Save Processing	10-28
Exit Component	10-30
PeopleSoft Internet Architecture Processing Considerations	10-30
Deferred Processing Mode	10-30
PeopleCode Events	10-33
Activate Event	10-33
FieldChange Event	10-34
FieldDefault Event	10-34
FieldEdit Event	10-35
FieldFormula Event	10-35
ItemSelected Event	10-36
PostBuild Event	10-36
PreBuild Event	10-36
PrePopup Event	10-37
PSControlInit Event	10-37
PSLostFocus Event	10-38
RowDelete Event	10-38
Considerations when Deleting all Rows from a Scroll	10-39
RowInit Event	10-39
Exception to RowInit Firing	10-40
RowInsert Event	10-40
RowSelect Event	10-41
SaveEdit Event	10-42
SavePostChange Event	10-43
SavePreChange Event	10-43
SearchInit Event	10-44
SearchSave Event	10-44
Workflow Event	10-45
PeopleCode Execution in Multiple Scroll Pages	10-45
Scroll Level One	10-46

Scroll Level Two	10-46
Scroll Level Three	10-46

Chapter 11

PeopleCode and PeopleSoft Internet Architecture

Using PeopleCode in the PeopleSoft Internet Architecture.....	11-1
Avoiding Features Not Supported by PeopleSoft Internet Architecture	11-2
Using PeopleCode to Populate Search Dialog Key Fields	11-2
Client-Only PeopleCode	11-3
Functions That Are Always Client-Only.....	11-3
Functions That Are Client-Only under Specific Conditions	11-3
ScheduleProcess	11-4
Functions That Behave Differently in Three-Tier Mode	11-4
Calling Executables on the Application Server	11-4
Calling Dynamic Link Library Functions on the Application Server.....	11-5
Sample Cross-Platform External Test Function.....	11-5
Updating the Installation and PSOPTIONS Tables	11-7

Chapter 12

Debugging Your Application

Accessing the PeopleCode Debugger	12-1
PeopleCode Debugger Features.....	12-2
Visible Current Line of Execution	12-2
Visible Breakpoints.....	12-2
Hover Inspect	12-3
Single Debugger.....	12-4
Variables Panes	12-4
Field Values	12-6
General Debugging Tips	12-7
DoModal Considerations	12-8
PeopleCode Debugger Options.....	12-8
Additional Features	12-11
Setting Up the Debugging Environment.....	12-11
Debugging Subscription PeopleCode	12-12
Compiling all PeopleCode Programs.....	12-12
Setting PeopleCode Debugger Log Options	12-13
Interpreting the PeopleCode Debugger Log File	12-15
Log File Contents	12-16
Sample Trace File	12-20
About Operations and Operands	12-21

Stacks	12-21
START	12-21
STOP	12-22
BRANCH	12-22
PUSH	12-22
FETCH	12-23
BUILTIN	12-23
CALL	12-24
START EXT	12-24
RETURN	12-25
STORE	12-25
ERROR	12-25
STATEMENT	12-25
Printed Data Values	12-25
Other Items in the Log File	12-26
Find In	12-27
Cross Reference Reports	12-30

Chapter 13

Using Three-Tier and Windows Client

Implementing Dynamic Tree Controls	13-1
Recursive Single-Table Dynamic Trees	13-6
How It Works	13-9
Controlling the Root Node of the Dynamic Tree	13-10
Implementing ActiveX Controls	13-11
Manipulating Events for ActiveX Controls	13-12
PSControlInit	13-13
PSLostFocus	13-13
Control Specific Events	13-14
Manipulating ActiveX Control Properties and Methods	13-15
Data Types for Declaring ActiveX Controls	13-16
Initializing an ActiveX Control with Data	13-16
Using ScrollSelect Functions	13-17
What ScrollSelect Does	13-18
ScrollSelect Syntax	13-19
Specifying the Target Scroll Area	13-19
The SQL String	13-20
Specifying the Select Record	13-21
Turbo ScrollSelect	13-21
Other ScrollSelect Functions	13-22

New Data Behavior	13-22
Specific Row Behavior	13-22
Using OLE Functions	13-23
Data Types	13-24
Sharing a Single Object Instance	13-24
OLE versus WinExec	13-25
Processing Groups	13-25
Component Build	13-26
FieldChange PeopleCode	13-26
Component Save	13-27
SaveEdit PeopleCode	13-28
Other Processing Groups.....	13-28
Note on External Function Location	13-29
Default Processing Locations	13-29
Controlling Process Location.....	13-30

Index

ABOUT THIS PEOPLEBOOK

This PeopleBook covers the concepts of PeopleCode, the proprietary scripting language used in the development of PeopleSoft applications. Its chapters describe 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.

The accompanying book, PeopleCode Reference, is a complete reference of the PeopleCode language. Its chapters describe the syntax and fundamental elements of the PeopleCode language.



For more information see PeopleCode Reference.

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 on how to use PeopleCode as you build and augment applications with PeopleTools. For information specific to your application, please refer to your PeopleSoft application documentation.

Introducing What's New is for developers who are familiar with PeopleCode. It provides a brief overview of new functionality for this release.

Understanding PeopleCode and Events describes how PeopleCode programs and PeopleCode events are integrated into Application Designer's framework. It also give information on how to add PeopleCode programs to applications in Application Designer.

Using the PeopleCode Editor provides information on how to use the PeopleCode Editor.

Introducing the SQL Editor provides information on how to use the SQL Editor.

PeopleCode Language covers the syntax and fundamental elements of the PeopleCode language.

Understanding Objects and Classes in PeopleCode describes at a high level how to use objects and classes in your PeopleCode programs.

Using Methods and Built-in Functions includes a number of issues related to the use of PeopleCode methods and built-in functions. It discusses common restrictions on the use of functions and methods in certain PeopleCode events. It examines groups of functions that are related by common syntactic complexities (such as functions that access data in multiple-scroll panels). And it views specific functions in the context of a specific development task (such as implementing a dynamic tree control or a remote call).

Referencing Data in the Component Buffer discusses the logical and syntactic problems involved in referencing scrolls, rows of data, and buffer fields.

Data Buffer Access furthers the discussion of how to access data in the panel buffers using the data buffer access classes (rowset, row, record, field.)

PeopleCode and the Component Processor discusses the flow of execution of the Component Processor at runtime and its interaction with PeopleCode programs. It describes when PeopleCode events are generated during the Component Processor's flow of execution, and how PeopleCode events trigger PeopleCode programs.

PeopleCode and PeopleSoft Internet Architecture discuss the PeopleCode considerations writing PeopleCode for PeopleSoft Internet Architecture applications.

Debugging Your Application discusses tools and techniques for debugging PeopleCode in applications.

Using Three-Tier and Windows Client covers a number of issues related to the use of PeopleCode methods and built-in functions when used in a three-tier or windows client architecture. PeopleSoft applications are written to work in the PeopleSoft Internet Architecture. However, you may have legacy applications or an environment that requires using this older architecture.

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:

<code>monospace font</code>	Indicates a PeopleCode program or other example
Bold	<p>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.</p> <p>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).</p> <p>We also use boldface when we refer to navigational paths, menu names, or process actions (such as Save and Run).</p>
<i>Italics</i>	<p>In PeopleCode syntax, italic items are placeholders for arguments that your program must supply.</p> <p>Throughout the rest of this PeopleBook italics indicates a PeopleSoft or other book-length publication. We also use italics for <i>emphasis</i> and to indicate specific field values. When we cite a field value under the page on which it appears, we use this style: <i>field value</i>.</p> <p>We also use italics when we refer to words as words or letters as letters, as in the following: Enter the number <i>0</i>, not the letter <i>O</i>.</p>
...	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 <i>italics</i> 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

Introducing What's New

What's new for PeopleTools PeopleCode? Plenty! There's dot notation, new objects and classes, more places to put your PeopleCode, and an enhanced debugger, to name a few.

This section provides an overview of the new features. This section is written for developers who have a background in PeopleCode. If you're new to PeopleCode, you probably shouldn't start by reading this section.

Objects and Classes

PeopleSoft is introducing object classes that can be manipulated by PeopleCode. Manipulating these objects with PeopleCode is an easy and consistent way to manipulate data in the buffer. These object classes enable you to write code that's more readable, more easily maintained, and more reusable. Coding with objects is simply easier.

List of Classes

The following list defines most of the new PeopleTools classes. This high-level list includes the primary objects that can be instantiated from PeopleCode. It doesn't include all the sub-classes, that is, the additional objects that can be instantiated from each class. Each class is described in detail in PeopleCode Classes.

<i>Class Name</i>	<i>Description</i>
AESession Class	Use this class to modify, with PeopleCode, the steps and SQL associated with a given Application Engine section.
Array Class	Use this class to manipulate 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.
Business Interlink Class	Use this class to contact an external system and access its data in a synchronous manner.
Component Interface Classes	Use this class to gain real-time synchronous access to the PeopleSoft business rules and data associated from outside the PeopleSoft online system. A Component Interface is one of the APIs that work with the Session object.
Field Class	Use this class, based on a field definition, to gain access to all the properties of a field, such as its visibility, whether

Class Name	Description
	it's enabled, its value, and so on.
File Class	Use this class for reading from and writing to external files. These files can either be unformatted files or files with a pre-set structure defined with a File Layout Definition.
Grid Class, GridColumn Class	Use these classes to access grids and grid columns on a page.
Internet Script Classes	Use this class in PeopleSoft Internet Architecture to dynamically generate a web page.
Message Class	Use this class to create and access an application message from PeopleCode. Application messages are self-describing messages that contain application data.
Page Class	Use this class to manipulate a page in a component. Usually you want to hide or display a page in a component, either based on field values or the user's security level.
PortalRegistry Classes	Use this class to add and register content on your portal. The PortalRegistry is one of the APIs that work with the Session object.
ProcessRequest Class	Use this class to schedule a process or job. This class succeeds the ScheduleProcess PeopleCode function.
Query Classes	Use these classes to access or modify existing PeopleSoft queries, or to build new queries.
Record Class	Use this class to access a single record within a row. You can also use this class to create standalone records. This class is based on a record definition. A record object consists of one to n fields.
Row Class	Use this class to access a row of data. The Row class is a single row of data that consists of one to n records of data. You can also think of a row as a single row in a component scroll.
Rowset Class	This class is a data structure that describes hierarchical data. It's composed of a collection of rows. A component scroll is a rowset. You can also have a level 0 rowset. Use rowsets with component buffer data, as well as with application messages and File Layouts.
Search Classes	Use these classes to access a search index and query its contents.
Session Class	This class is the root of the entire family of PeopleSoft APIs that provide external access into the PeopleSoft system. It controls access to the PeopleSoft system,

Class Name	Description
	controls the environment, and enables you to do error handling for all APIs from a central location.
SQL Class	This class provides access to SQL definitions in your PeopleCode program at runtime. You can create SQL definitions in Application Designer or in your PeopleCode program. They can be entire SQL programs, or just fragments of SQL statements that you want to re-use.
Tree Classes	These classes provide access to all the functionality of Tree Manager in your PeopleCode program. The Tree classes are one of the APIs that work with the Session class.

Dot Notation and Objects

For the experienced application developer who writes a lot of PeopleCode, the new PeopleCode object syntax might be the most valuable PeopleTools enhancement. This new object syntax includes a new set of standard classes and support for Visual Basic-style dot notation to access properties and methods of those classes.

It's a simpler, more intuitive, more standard approach to writing code. With this release, PeopleCode can be used extensively outside of the standard components (such as Application Engine programs, Application Messaging subscriptions, and Component Interfaces). The new object syntax de-couples PeopleCode from components and enables the application developer to write vanilla PeopleCode that executes under the various runtime environments.

Prior to this release, there were only built-in functions, like `FetchValue`, `ScrollSelect`, and so on. In this release, there are now **methods**. The primary difference between a built-in function and a method is:

- A method can only be executed from an object by using dot notation. You must create an instance of the object before you can use the method.
- A built-in function, in your code, is on a line by itself, and doesn't (generally) have any dependencies. You don't have to create an instance of an object before you can use a built-in unless it takes an object as one of its parameters.

For example, both the `GetRowset` built-in function and the `GetRowset` method return a reference to a rowset object.

The `GetRowset` built-in could be used almost anywhere in your code. It returns a reference to the rowset in the current context:

```
&MYROWSET = GetRowset();
```

The method can only be used as part of a row object to return a reference to a child rowset, that is, a rowset contained by a row. You must first get a row object before you can get the child rowset of that row. In order to get the row we use the `GetRow` built-in function. This returns a reference to the current row:

```
&MYROW = GetRow();

&MYROWSET = &MYROW.GetRowset(SCROLL.scrollname);
```

If you're not going to access the row again, you could combine the two lines of code (above) into the following:

```
&MYROWSET = GetRow().GetRowset(SCROLL.scrollname);
```

These two lines are combined using *dot notation*, that is, the reference to the object (what's returned with the GetRow built-in function) followed by a period, then a method or property name.



For more information about objects and dot notation, see Understanding Objects and Classes in PeopleCode.

Sometimes the built-in function and the method with the same name have the same syntax, sometimes they don't. You need to check the documentation to be certain.

Instantiating Objects

A class is the blueprint for something, like a bicycle, a car, or a data structure. An object is the actual thing that's built using that class (or blueprint.) From the blueprint for a bicycle, you can build a specific mountain bike with 23 gears and tight suspension. From the blueprint of a data structure class, you build a specific instance of that class. *Instantiation* is the term for building that copy, or an instance, of a class.

Code Enhancements Using Dot Notation

In the following example, instead of making a series of FetchValue calls, the re-written code takes advantage of creating and defining one record object plus the Value property for fields. The code is now more readable and easier to maintain if the structure of the page changes. If you have many lines of code calling the same structure (such as a record, a page, a field, and so on), you may see better performance if you write your code to call the structure once.

Existing Code

```
For &ROW_2 = 1 To &TOT_ROWS

    /* Fetch values from DEMAND_PO_VW that we need for processing */

    &STAGED_DATE = FetchValue(RECORD.DEMAND_CANCP_VW, &ROW_1,
    DEMAND_PO_VW.STAGED_DATE, &ROW_2);

    &INV_LOT_ID = FetchValue(RECORD.DEMAND_CANCP_VW, &ROW_1,
    DEMAND_PO_VW.INV_LOT_ID, &ROW_2);
```



```

    &CONTAINER_ID = FetchValue(RECORD.DEMAND_CANCP_VW, &ROW_1,
    DEMAND_PO_VW.CONTAINER_ID, &ROW_2);

    &SERIAL_ID = FetchValue(RECORD.DEMAND_CANCP_VW, &ROW_1,
    DEMAND_PO_VW.SERIAL_ID, &ROW_2);

    &STORAGE_AREA = FetchValue(RECORD.DEMAND_CANCP_VW, &ROW_1,
    DEMAND_PO_VW.STORAGE_AREA, &ROW_2);

    &STOR_LEVEL_1 = FetchValue(RECORD.DEMAND_CANCP_VW, &ROW_1,
    DEMAND_PO_VW.STOR_LEVEL_1, &ROW_2);

    &STOR_LEVEL_2 = FetchValue(RECORD.DEMAND_CANCP_VW, &ROW_1,
    DEMAND_PO_VW.STOR_LEVEL_2, &ROW_2);

    &STOR_LEVEL_3 = FetchValue(RECORD.DEMAND_CANCP_VW, &ROW_1,
    DEMAND_PO_VW.STOR_LEVEL_3, &ROW_2);

    &STOR_LEVEL_4 = FetchValue(RECORD.DEMAND_CANCP_VW, &ROW_1,
    DEMAND_PO_VW.STOR_LEVEL_4, &ROW_2);

    &UOM = FetchValue(RECORD.DEMAND_CANCP_VW, &ROW_1,
    DEMAND_PO_VW.UNIT_OF_MEASURE, &ROW_2);

    &RECEIVER_ID = FetchValue(RECORD.DEMAND_CANCP_VW, &ROW_1,
    DEMAND_PO_VW.RECEIVER_ID, &ROW_2);

    &RECV_LN_NBR = FetchValue(RECORD.DEMAND_CANCP_VW, &ROW_1,
    DEMAND_PO_VW.RECV_LN_NBR, &ROW_2);

End-For;

```

Re-Written Code

```

Local row &ROW_1, &ROW_2;

Local record &RECORD;

.
.
.

For &ROW_2 = 1 To &TOT_ROWS

    /* Fetch values from DEMAND_PO_VW that we need for processing */

    &RECORD =
    GetLevel0() (1).DEMAND_CANCP_VW(&ROW_1).GetRowSet(Scroll.DEMAND_PO_VW).(&ROW_2).D
    EMAND_PO_VW;

```

```

&STAGED_DATE = &RECORD.STAGED_DATE.Value;

&INV_LOT_ID = &RECORD.INV_LOT_ID.Value;

&CONTAINER_ID = &RECORD.CONTAINER_ID.Value;

&SERIAL_ID = &RECORD.SERIAL_ID.Value;

&STORAGE_AREA = &RECORD.STORAGE_AREA.Value;

&STOR_LEVEL_1 = &RECORD.STOR_LEVEL_1.Value;

&STOR_LEVEL_2 = &RECORD.STOR_LEVEL_2.Value;

&STOR_LEVEL_3 = &RECORD.STOR_LEVEL_3.Value;

&STOR_LEVEL_4 = &RECORD.STOR_LEVEL_4.Value;

&UOM = &RECORD.UNIT_OF_MEASURE.Value;

&RECEIVER_ID = &RECORD.RECEIVER_ID.Value;

&RCV_LN_NBR = &RECORD.RCV_LN_NBR.Value;

End-For;

```

Another advantage of the re-written code is that it can be called from practically any record in the component because you get the level 0 rowset first. If this code is to be called from the level one scroll (DEMAND_CANCP_VW), you might shorten the declaration for &RECORD to the following:

```

&RECORD =
GetRowset ( ) . (&ROW_1) . GetRowSet (SCROLL.DEMAND_PO_VW) . (&ROW_2) . DEMAND_PO_VW;

```

Data Buffer Access

There have been many different ways and built-in functions for accessing page buffer data. A consistent data model is now available, based on the data buffer access classes Rowset, Row, Record and Field.



The data buffer access support does not replace the existing functions, such as ActiveRecordCount, ScrollSelect, and so on. These functions are still valid and are being kept for backwards compatibility.

These four classes are built on the data model of a PeopleTools component, in which scrollbars or grids are used to describe a hierarchical, multiple-occurrence data structure. You can access these classes using dot notation.

The four data buffer classes relate to each other in a hierarchical manner. The main thing to remember when learning these relationships is:

- A *record* contains one or more *fields*.
 - Records contain the fields that make up that record.
- A *row* contains one or more *records* and zero or more child *rowsets*
 - A row contains the records that make up that row. It may also contain child rowsets.
- A *rowset* contains one or more *rows*
 - A rowset is a data structure that describes hierarchical data. For component buffers, think of a rowset as a scroll on a page that contains all of that scroll's data. A level 0 rowset contains all the data for the entire component.
 - You can use rowsets with application messages, file layouts, Business Interlinks, and other definitions besides components.
 - A level 0 rowset from a component buffer only contains one row, that is, the keys that the user specifies to initiate that component. A level 0 rowset from data that isn't a component, such as a message or a file layout, might contain more than one level 0 row.



For more information about these classes, see Data Buffer Access.

To view a rowset, start the PeopleCode debugger in Application Designer, then select **Break at start**. Start a PeopleSoft application. When the first PeopleCode program is reached, the debugger displays the first line of code. Select **Debug, View Component Buffers**. In the view window that appears you can look through the entire component buffer. The top level is a rowset.



For more information see Debugging Your Application

Instead of SQLExec. . .

You now have options other than SQLExec for using SQL.

Using the Record Class

With the Record class you can build and execute a SQL statement by using the following methods:

- Delete
- Insert
- SelectByKey
- Update

In the following example, the existing code selects all fields into one record then copies that information to another record. The existing code used SQLExec. The rewritten code uses a record object method SelectByKey.

Existing Code

```
&MYKEY = "001";

SQLExec("select %dateout(msdate1), %dateout(msdate2), %timeout(mstime1),
%timeout(mstime2), %timeout(mstime3), %datetimeout(msdtm1),
%datetimeout(msdtm2), %datetimeout(msdtm3) from ps_xmstbl1 where mskey1 = :1",
&MYKEY, &MYDATE1, &MYDATE2, &MYTIME1, &MYTIME2, &MYTIME3, &MYDTM1, &MYDTM2,
&MYDTM3);

SQLExec("delete from ps_xms_out1 where mskey1 = :1", &MYKEY);

SQLExec("insert into ps_xms_out1
(mskey1,msdateout1,msdateout2,mstimeout1,mstimeout2,mstimeout3,msdtmout1,msdtm
out2,msdtmout3)values(:1,%datein(:2),%datein(:3),%timein(:4),%timein(:5),%timei
n(:6),%datetimein(:7),%datetimein(:8),%datetimein(:9))", &MYKEY, &MYDATE1,
&MYDATE2, &MYTIME1, &MYTIME2, &MYTIME3, &MYDTM1, &MYDTM2, &MYDTM3);
```

Re-Written Code

SelectByKey works by using the keys you've already assigned values for. It returns successfully if you assign enough key values to return a unique record. In this example, the record has a single key, so only that key value is set before executing SelectByKey. If your record has several keys, you must set enough of those key values to return a unique record.

```
Local record &REC, REC2;

&REC = CreateRecord(RECORD.XMSTBL1);

&REC.MSKEY1 = "001";

&REC.SelectByKey();

&REC2 = CreateRecord(RECORD.XMS_OUT1);

&REC.CopyFieldsTo(&REC2);

&REC2.Delete();

&REC2.Insert();
```



For more information see Data Buffer Access, ProcessRequest Class and SelectByKey.

In the following example, again, the record object can be used instead of SQLExec.

Existing Code

```

If None(&EXISTS) Then

    SQLExec("insert into ps_rt_rate_tbl (rt_rate_index, term, from_cur, to_cur,
rt_type, effdt, rate_mult, rate_div) values (:1, :2, :3, :4, :5, %DateIn(:6),
:7, :8)", RT_RATE_INDEX, TERM, TO_CUR, FROM_CUR, RT_TYPE, EFFDT, RATE_DIV,
RATE_MULT);

    SQLExec("select 'x' from ps_rt_rate_def_tbl where rt_rate_index = :1 and
term = :2 and from_cur = :3 and to_cur = :4", RT_RATE_INDEX, TERM, TO_CUR,
FROM_CUR, &DEFEXISTS);

    If None(&DEFEXISTS) Then

        SQLExec("insert into ps_rt_rate_def_tbl (rt_rate_index, term, from_cur,
to_cur, max_variance, error_type, int_basis) values (:1, :2, :3, :4, :5, :6,
:7)", RT_RATE_INDEX, TERM, TO_CUR, FROM_CUR, RT_RATE_DEF_TBL.MAX_VARIANCE,
RT_RATE_DEF_TBL.ERROR_TYPE, RT_RATE_DEF_TBL.INT_BASIS);

        End-If;

    Else

        SQLExec("update ps_rt_rate_tbl set rate_mult = :7, rate_div = :8 where
rt_rate_index = :1 and term = :2 and from_cur = :3 and to_cur = :4 and rt_type =
:5 and effdt = %DateIn(:6)", RT_RATE_INDEX, TERM, TO_CUR, FROM_CUR, RT_TYPE,
EFFDT, RATE_DIV, RATE_MULT);

        End-If;

```

Re-Written Code

```

Local record &RT_RATE_TBL, &RT_RATE_DEF_TBL;

.

.

.

If None(&EXISTS) Then

    &RT_RATE_TBL = CreateRecord(RT_RATE_TBL);

    &RT_RATE_DEF_TBL = CreateRecord(RT_RATE_DEF_TBL);

    &RT_RATE_TBL.Insert();

    &RT_RATE_DEF_TBL.SelectByKey();

```

```

If None(&DEFEXISTS) Then

    &RT_RATE_DEF_TBL.Insert();

End-If;

Else

    &RT_RATE_TBL.Update();

End-If;

```

SQL Definitions and the SQL Class

You can create SQL definitions in Application Designer. These can be complete SQL statements or just fragments that you wish to re-use. You can share code between an online and a batch process that use the same SQL statements. If a table changes, your SQL must be changed in only one place. And you can dynamically create or change SQL statements online before you start a batch process.

SQLExec only allows one row at a time to be fetched. With the SQL object, you can process more than one row of data.

You can use the SQL class to create temporary SQL statements for manipulating data. The following example creates a temporary SQL statement, then writes all the rows of data from a record to a file:

```

Local Record &LN;

Local File &MYFILE;

Local SQL &SQL2;

&MYFILE = GetFile("record.txt", "A");

If &MYFILE.IsOpen Then

    If &MYFILE.SetFileLayout(FILELAYOUT.ABS_HIST) Then

        &LN = CreateRecord(RECORD.ABSENCE_HIST);

        &SQL2 = CreateSQL("%Selectall(:1)", &LN);

        While &SQL2.Fetch(&LN)

            &MYFILE.WriteRecord(&LN);

        End-While;

    End-If;

End-If;

```

```

        End-If;

    End-If;

    &MYFILE.Close();

```

Record Class vs. SQL Object

If you're doing many iterations of the same operation (like a million UPDATES) use the SQL object with the BulkMode property set to True.

The SQL object maintains a state (that is, a cursor). If your database can take advantage of BulkMode, instead of a million operations, the commands are committed altogether and only once. This can improve performance dramatically.

Using SQL Definitions with SQLExec

There are times when SQLExec is the appropriate function to use. If you only need a single row, use SQLExec, which can only SELECT a single row of data. If your SQL statement retrieves more than one row of data, **SQLExec** outputs only the first row to its output variables and discards subsequent rows. This can improve single-operation performance.

However, you don't have to hardcode your SQL statement. SQLExec is enhanced to accept the SQL definitions, and the new meta-SQL, so you can reuse your SQL statements.

For example, consider the following statement:

```

SQLExec("Update %Table(:1) set %UpdatePairs(:1) where %KeyEqual(:2)", &REC,
&FIELD);

```

If you have created a SQL definition with this SQL statement and named it MYUPDATE, you can use it with SQLExec as follows:

```

SQLExec(SQL.MYUPDATE, &REC, &FIELD);

```

Using Record Objects With SQLExec

The syntax for SQLExec now accepts Record objects as arguments. Instead of listing a group of fields, you can use all the fields within a record.

In the following example, a SQLExec statement selects 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 */

```

Meta-SQL

Of the many meta-SQL statements for this release, some are expandable, that is, you enter the code in the PeopleCode editor as a short meta-SQL construct, but at runtime it expands into a full SQL statement. For example, with the %List construct, you can list all of the fields for a record. The re-written code is easier to maintain because you don't have to change it if a field was added, deleted, or renamed. In addition, input processing is applied to the values:

- If the field is a date, a time, or a datetime, its value is automatically wrapped in the appropriate %DateIn(), %DateOut(), %TimeIn(), %TimeOut(), and so on.
- If a value is a string, its value is automatically wrapped in quotation marks.
- If a value is NULL, the "*=value*" part is replaced with "IS NULL".

The following example uses the %EffDtCheck meta-SQL statement, which expands into an effective date sub-query suitable for a WHERE clause. &DATE has the value of 01/02/1998. The &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 SQL definition FTP_TEMPLATE_SELECT has the following code. The %List and %EFFDTCHECK meta-SQL statements make the code easier to maintain. If there are any changes to the underlying record structure, you don't have to change this SQL definition:


```

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'

```

Standalone Rowset

In addition to being able to create records on the fly, you can create a standalone rowset. You can use this type of object to get rid of hidden work scrolls on a page.

Standalone rowsets are **not** tied to the database. This means if you make changes to data in a standalone rowset, it will not be automatically saved to the database. In addition, a standalone rowset isn't tied to the component processor. When you fill it with data, no PeopleCode programs or events run (like RowInit, FieldDefault, and so on.)

Use the CreateRowset function to create a standalone rowset. The parameters for this function determine if the structure of the rowset you're creating is based on an already instantiated rowset, on a record definition, or both.

For example, to create a rowset based on an existing rowset, pass in the name of the existing rowset.

```

&Level0 = GetLevel0();

&MyRowset = CreateRowset(&Level0);

```

You can create rowsets based on record definitions. The following code creates a rowset structure composed of four records in an hierarchical structure:

```

QA_INVEST_HDR
    QA_INVEST_LN
        QA_INVEST_TRANS
            QA_INVEST_DTL

```



You must start at the **bottom** of the hierarchy and add the upper levels.

```

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);

```



For more information see the CreateRowset function.

Performance Enhancement for SQLExec and ScrollSelect

Users familiar with PeopleCode know that runtime parameter markers in SQL strings were replaced with the associated literal values. For databases that offer SQL statement caching, a match was never found in the cache so the SQL was re-parsed and re-assigned a query path.

For example:

```

ScrollSelect(1, RECORD.ORD_HDR_HOLD_VW, RECORD.ORD_HDR_HOLD_VW, "where
business_unit = :1 and order_no = :2", &BU, &ORDER_NO, True);

```

Old SQL sent to Database (important part in bold):

```

SELECT BUSINESS_UNIT, ORDER_NO, . . ., FROM PS_ORD_HDR_HOLD_VW where
business_unit = 'M04a' and order_no = 'AAA-50001'

```

New SQL sent to Database (change in bold):

```

SELECT BUSINESS_UNIT, ORDER_NO, . . ., FROM PS_ORD_HDR_HOLD_VW where
business_unit = :1 and order_no = :2

```

Now, in order to handle skipped parameter markers, each parameter marker is assigned a unique number. This doesn't change the value associated with the parameter markers, but it might cause some confusion when it first appears in a Tools SQL trace.

The following code

```

SELECT SETID, SHIP_TO_CUST_ID FROM PS_CUST_SHPOPT_VW

where setid = :1 and ship_to_cust_id = :2 and effdt =

(select max(effdt) from ps_cust_shpopt_vw s2 where s2.setid = :1 and
s2.ship_to_cust_id = :2 and s2.eff_status = 'A' and s2.effdt <=
TO_DATE(:3, 'YYYY-MM-DD'))

```

will appear in the trace as follows:

```
SELECT SETID, SHIP_TO_CUST_ID FROM PS_CUST_SHPOPT_VW

where setid = :1 and ship_to_cust_id = :2 and effdt =

(select max(effdt) from ps_cust_shpopt_vw s2 where s2.setid = :3 and
s2.ship_to_cust_id = :4 and s2.eff_status = 'A' and s2.effdt <=
TO_DATE(:5,'YYYY-MM-DD'))

1-2519      09.34.19      0.000 Cur#1 RC=0 Dur=0.000 Bind-1 type=2 length=3
value=MFG

1-2520      09.34.19      0.000 Cur#1 RC=0 Dur=0.000 Bind-2 type=2 length=5
value=50001

1-2521      09.34.19      0.000 Cur#1 RC=0 Dur=0.000 Bind-3 type=2 length=3
value=MFG

1-2522      09.34.19      0.000 Cur#1 RC=0 Dur=0.000 Bind-4 type=2 length=5
value=50001

1-2523      09.34.19      0.000 Cur#1 RC=0 Dur=0.000 Bind-5 type=2 length=10
value=1999-08-04
```



Some SQL statements can't contain parameter markers because of database compatibility.

One case is if you concatenate parameter markers to literals, as follows:

```
ScrollSelect(1, RECORD.TARGET, RECORD.SELECT, "where p.setid = 'M':1 and t.setid
= 'M':2 . . ." &var1, &var2, &var3, &var4, &var5, True);
```

Another case is if you use a second bind variable with a %Truncate statement.

The following code is valid without any changes:

```
%Truncate(:1, 5)
```

The following code is no longer valid:

```
%Truncate(:1, :2);
```

To handle these exceptions, use the ExpandSqlBinds function. This function expands the bind variable reference, and can be used within a SQLExec statement or on its own.

You can then change the above example to:

```
ScrollSelect(1, RECORD.TARGET, RECORD.SELECT, ExpandSqlBinds("where p.setid =
'M':1 and t.setid = 'M':2 . . ." &var1, &var2, &var3, &var4, &var5), True);
```

You should only use ExpandSQLBinds for those parts of the string into which you want to put literal values. For example, don't use ExpandSQLBinds with meta-SQL. The following code shows how to use ExpandSQLBinds with %Table:

```
SQLExec(ExpandSQLBind("Insert.... Select A.Field, :1, :2 from ", "01", "02") |
"%table(:1)", Record.MASTER_ITEM_TBL);
```

Using Literal Parameters

If your code uses inline (literal) parameters (as opposed to standard parameter markers), they remain as literals. In the following example, there are three standard parameter markers and one literal (**in bold**):

```
ScrollSelect(2, Record.BI_LINE, Record.BI_LINE_DST_AR, Record.BI_LINE_DST_AR,
"where business_unit = :1 and invoice = :2 and
line_seq_num>=:derived_work_bi.line_seq_num and line_seq_num <=:3",
BI_HDR.BUSINESS_UNIT, BI_HDR.INVOICE, &LAST_BI_LINE);
```

The standard parameter markers are sent to the database as markers, while the literal remains as a literal. PeopleSoft recommends that you create no new code with literals.



For significant performance improvement, convert your literals to parameter markers.

Grids

You can now access both a grid and a grid column using PeopleCode (using the Grid and GridColumn objects). You can have multiple grids on a page, and the grid no longer has to be the last field on a page.

To hide a column in a grid, you no longer must loop through every row in the grid and hide that field.

Now you can use the GridColumn property **Visible**.

The **Visible** property will also hide grid columns that are displayed as tabs in the PeopleSoft Internet Architecture.

Existing Code

```
If COMPLETE_FLAG = "Y" Then

    For &I = 1 to &ACTIVE_ROW2

        Hide(SCROLL.TASK_RESOURCE, &CURRENT_ROW1, SCROLL.TASK_EFFORT, &I,
TASK_EFFORT.EFFORT_DT);

    End-For;
```

```
End-if;
```

Re-Written Code

```
Local Grid &GRID;

Local GridColumn &COLUMN;

If COMPLETE_FLAG = "Y" Then

    &GRID = GetGrid(PAGE.RESOURCE, "GRID1");

    &COLUMN = &GRID.GetColumn("COL5");

    &COLUMN.Visible = False;

End-If;
```



For more information on how to use the new grid objects, see File Class.

There is no visible property on a grid, but you can still hide an entire grid. Most of the rowset methods and properties work on a grid. To hide an entire grid, get its rowset, then use the HideAllRows() rowset method.

Accessing PeopleCode Programs

Now, PeopleCode programs can be associated with items other than record fields, menus and pushbuttons.

The following table locates the different types of PeopleCode programs in Application Designer.

<i>PeopleCode Programs</i>	<i>In Application Designer</i>
Record field	Record definitions and page definitions
Component record field, component record, and component	Component definitions
Page	Page definitions
Page field (ActiveX control)	Page definitions
Menu component	Menu definitions
Application message	Message definitions and message channels

Component Interfaces	Component Interface definitions
Application Engine	Application Engine program (definition)

You can still access PeopleCode from the project window or from the record definition by clicking the name of the PeopleCode program or by using **Edit, View PeopleCode**. To access record field PeopleCode from a page definition select a field, then select **View Record PeopleCode** from either the context window or from the **Edit** menu.

To access PeopleCode from a page definition select an element on the page, then select **View PeopleCode** from either the context window or the **Edit** menu.

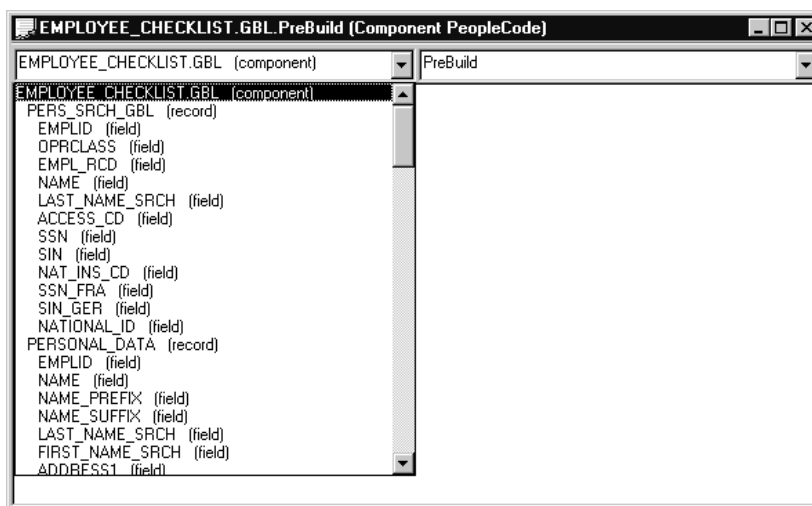
To access page field (ActiveX control) PeopleCode select an ActiveX control on the page, then select **View PeopleCode** from either the context window or the **Edit** menu.



For more information about accessing PeopleCode from different components, see [Accessing PeopleCode in Application Designer \(Overview\)](#).

You can access component PeopleCode only from the **Structure** tab of a component definition. Select any element from that page, then select **View PeopleCode** from either the context window or the **Edit** menu.

All the items with which you can associate PeopleCode are listed in the left-hand drop-down menu in a component's PeopleCode editor window.



A Component's PeopleCode Editor

A component is listed in bold if it has PeopleCode associated with it.

When you select a item of a component, the possible events associated with item are listed in the right-hand drop-down window.

Every PeopleCode program is associated with a PeopleCode event, and is often referred to by that name, such as RowInit PeopleCode, or FieldChange PeopleCode. These programs are accessible from, and associated with, different items. The following table lists events and types of PeopleCode programs.



These events are part of the normal component processor flow. For more information about events outside of the component processor (such as Component Interfaces, messages, and so on) see the documentation specific to that technology.



The SearchInit and SearchSave events (under Component Record) are only available for the search record associated with a component.

<i>Record Field</i>	<i>Component Record Field</i>	<i>Component Record</i>	<i>Component</i>	<i>Page</i>	<i>Menu</i>
FieldChange	FieldChange	RowDelete	PostBuild	Activate	ItemSelected
FieldDefault	FieldDefault	RowInit	PreBuild		
FieldEdit	FieldEdit	RowInsert	SavePostChg		
FieldFormula	PrePopup	RowSelect	SavePreChg		
PrePopup		SaveEdit	Workflow		
RowDelete		SavePostChg			
RowInit		SavePreChg			
RowInsert		SearchInit			
RowSelect		SearchSave			
SaveEdit					
SavePostChg					
SavePreChg					
SearchInit					
SearchSave					
Workflow					

Case Sensitivity in PeopleCode Programs

PeopleCode formats programs differently now. Variables are no longer automatically translated into uppercase, but keep the case in which they're typed. If a variable is declared at the start of the program, all occurrences are converted to the same case in which it is declared. Some PeopleCode language statements are also no longer converted into uppercase, but instead are converted to initial caps, for example, data types and keywords.

The following example has two parts: the first part is the code as it's typed in. The user variables are in bold. The second part is how the code is displayed to the user after saving.

```
local record &Rec;

global number &num;


function HandleRec(&InRec as record)

&inrec = &rec;

&num = 0;

end-function;


&rec = createrecord(record.xyz);

handlerec(&rec);
```

After saving, the above code would be formatted as follows:

```
Local Record &Rec;

Global Number &num;


Function HandleRec(&InRec As Record)

    &Inrec = &Rec;

    &num = 0;

End-Function;


&Rec = CreateRecord(Record.XYZ);

HandleRec(&Rec);
```

Application Reviewer

Access to the Application Reviewer is simplified and its function is greatly expanded. It is now called the **PeopleCode Debugger**.

The PeopleCode Debugger is integrated into Application Designer. The interface to the debugger has a visual indicator of breakpoints, an arrow indicating the current line and the ability to step through code. You can inspect variables in the different variable panes. In addition, you can inspect the value of a variable by positioning your cursor over it and reading the pop-up help bubble. The debugger provides variable inspection windows for Globals, Locals, Function Parameters, and the new Component scoped variables. It also allows the new PeopleCode Objects to be expanded so you can inspect their component parts.

You can also debug your application by compiling all the existing PeopleCode to see if it's valid.



For more information see [Debugging Your Application](#)

Runtime Checking

Changes to PeopleCode runtime checking include edits that cause certain programs that worked in PeopleTools 7.5 to abort when run in this release.

- Errors caused by missing pushbuttons

A PeopleCode program that refers to a pushbutton that was removed from a page now causes a runtime error.

- Errors caused by referencing missing page fields

Gray or **Hide** built-in functions that reference page fields that no longer exist now cause a runtime error.

- Syntax checking of declared variables

If you declare your variables at the start of your program, the PeopleCode editor runs some fundamental syntax checking when you save your program. For example, if you declare &DATE as type Field and then try to assign it as a rowset, you receive a design time error when you try to save your PeopleCode.

- Checking of SQLExec Bind Variables

A SQLExec with more bind variables than are actually required now causes a runtime error. For example, the following line of code is now invalid because there are 11 bind variables and only 10 are required.

```
SQLExec("Insert Into PS_PF_TEMP_REC_TBL Select :1, PF_RECNAME, %datetimein(:3),
:4, :5, :6, :7, %datein(:8), :9, :10 From PS_PF_META_REC_TBL", RECSUITE_ID,
&PF_RECNAME, &NULL_DATETIME, &NULL_CHAR, &NULL_CHAR, &NULL_NUM, &NULL_NUM,
&NULL_DATE, &NULL_CHAR, &NO, &RETURN);
```

Additional Features

Below are described some of the new features with this release. Again, for more in-depth information about each new feature, read the appropriate section.

New Component Variable Type

Variables declared as Component type exist for the life of the component. Declare variables of the Component type just as you declare Local or Global variables: at the top of your program and in every program in which you use them.



For more information, see Data Types.

New Data Types

For most PeopleCode classes in this release, a corresponding data type enables you to instantiate objects from that class. For example, you can declare your rowsets as type Rowset, your grids as type Grid, and so on. These are object-based data types.



For more information, see Object-Based Data Types.

Automatic Backup of PeopleCode

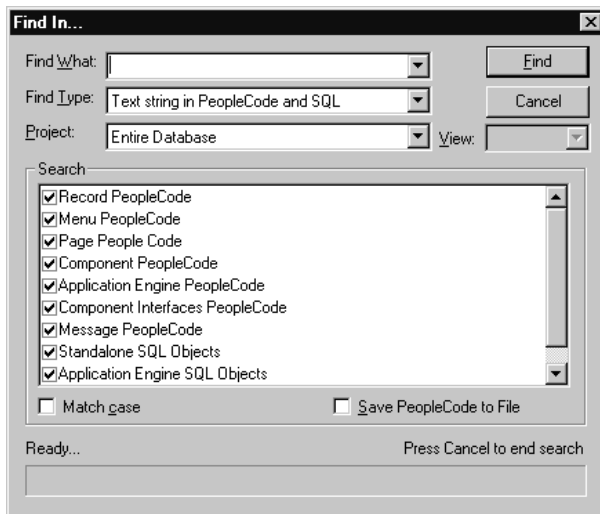
Has your system ever crashed while you were writing a PeopleCode program? Or worse yet, while you were fixing a PeopleCode program? Now, you no longer need to fear such failures. The PeopleCode you're currently working on is automatically saved to a file on your system in between times you save it yourself.



For more information see Automatic Backup of PeopleCode.

Enhanced Find In Function

The **Find In PeopleCode** function is now **Find In**, because it searches for both PeopleCode and SQL strings. You can search a project or specify individual records to search. You can also specify whether to search for PeopleCode and SQL, just SQL, or just PeopleCode.



Find In. . . Dialog



For more information see Find In . . .

Refreshing your Page

What do you want to do after your PeopleCode calls a RemoteCall, Component Interface, Business Interlink or Application Engine program? You want to refresh the values displayed on the page. A new PeopleCode method does just that.

The **Refresh** rowset method reloads the rowset (scroll) using the current page keys and redraws the page. `GetLevel0().Refresh()` refreshes the entire page. You can also limit the refresh to a particular scroll.



For more information see Refresh.

Using Constants Instead of Numeric Values

Some functions now accept or return constants. Use a constant instead of a numeric value to make your code easier to read.

For example, the *char_code* parameter of **CharType** now accepts constants.

```
&ISKANJI = CharType(&STRTOTEST, 5);
```

Here's the same code, using a constant:

```
&ISKANJI = CharType(&STRTOTEST, %CharType_Kanji);
```

Similarly, when you're looking for a return value, it's much easier when using constants than numeric return values:

```
&ONLYHIRAGANA = ContainsOnlyCharType(&STRTOTEST, %CharType_Hiragana,
%CharType_JapanesePunctuation);

If &ONLYHIRAGANA = %CharType_Matched Then

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

Else

    If &ONLYHIRAGANA = %CharType_NotMatched Then

        WinMessage("Mixed characters");

    Else

        WinMessage("UNKNOWN");

    End-If

End-If
```

Enhanced Transfer Functionality

In addition to specifying a list of `RECORD.fieldnames` for the **Transfer** built-in function, you can now specify a record object. Any field of the record object that's also a field of the search record for the destination component is added to the list. This enhancement is particularly useful when dealing with Workflows.

Attachment PeopleCode

Several PeopleCode functions have been added to enable you to transfer, view or delete files using ftp.

Mapping of Functions to Methods and Properties

The first column in the following table lists built-in PeopleCode functions that existed prior to this release. The second column lists the new method or property that should be used instead of the function. The existing functions still work in this release, but, they are being kept only for backwards compatibility. New applications should be created using the new classes, methods, and properties.

<i>Existing function name</i>	<i>Use instead</i>
ActiveRowCount	ActiveRowCount Rowset property

Existing function name	Use instead
ClearSearchDefault	SearchDefault Field property
ClearSearchEdit	SearchEdit Field property
CompareLikeFields	CompareFields Record method
CopyFields	CopyFieldsTo or CopyChangedFieldsTo 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
RecordChanged	IsChanged Record property
RecordDeleted	IsDeleted Record property
RecordNew	IsNew Record property
RemoteCall for Application Engine	CallAppEngine function

Existing function name	Use instead
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	SearchDefault 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 Trace Setting Class Properties
SetTraceSQL	If using API (Session object), use Trace Setting Class 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 first column in the following lists the old names of the functions, system variables or reserved words. The second 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 just being kept for backwards compatibility. New applications should be created using the new functions, system variables and reserved words.

<i>Old Term</i>	<i>New Term</i>
Operator	User
Panel	Page
Panel Group	Component
Business Component	Component Interface

<i>Deprecated Function, System Variable or Reserved Word</i>	<i>New Function, System Variable or Reserved Word</i>
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.

<i>No Longer Valid</i>	<i>Use Instead</i>
COMPONENT reserved word	COMPINTFC reserved word

<i>No Longer Valid</i>	<i>Use Instead</i>
GetComponent method	GetCompIntfc method
FindComponent method	FindCompIntfc method
%Component system variable	%CompIntfc system variable

CHAPTER 2

Understanding PeopleCode and Events



With PeopleTools 8, PeopleCode introduces object classes into its programming model. To reduce confusion, this section refers to Application Designer objects (such as records, pages, and so on) as definitions, to distinguish them from PeopleCode data objects. You may encounter references to the older terms "object" or "object definition" in Application Designer and other documentation.

Every PeopleCode program is associated with an Application Designer definition and with an event. Events are predefined points either in the Component Processor flow or in the program flow (for application messages.) As each point is encountered, the event fires on each definition, triggering any PeopleCode program associated with that definition and that event. Each class of definitions in Application Designer can have an **event set**—a group of events appropriate to that definition. A definition can have zero or one PeopleCode programs for each event in its event set. In PeopleTools 7.5, only record fields and menu items had event sets. In PeopleTools 8, several more types of definitions have event sets as well:

Definition Type	Availability
Record field	Prior to PeopleTools 8
Component record field	New in PeopleTools 8
Component record	New in PeopleTools 8
Component	New in PeopleTools 8
Page field	New in PeopleTools 8
ActiveX control	New in PeopleTools 8
Pop-up Menu item	Prior to PeopleTools 8
Message	New in PeopleTools 8
Message Channel	New in PeopleTools 8



Note on Application Engine PeopleCode. An Application Engine program can have a PeopleCode program as an action. Though the right-hand drop-down menu on the PeopleCode editor window shows the text "OnExecute," it really isn't an event. Any PeopleCode contained in an Application Engine action is executed only when the action is executed. For more information see PeopleCode Actions in the Application Engine book.

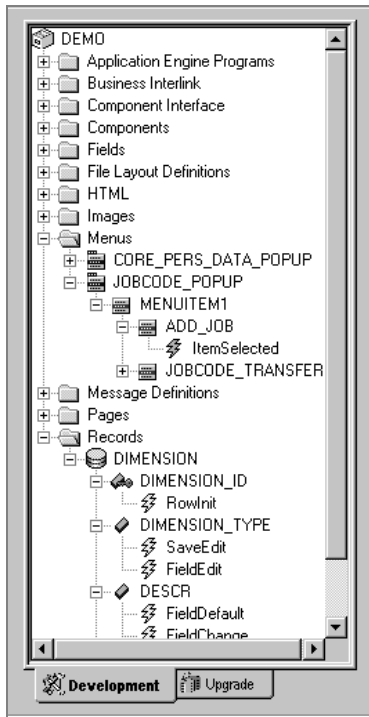
Note on Component Interface PeopleCode. A Component Interface can have user-defined methods associated with it. These methods aren't part of any processor flow. They're called as needed by the program executing the Component Interface. For more information see Component Interface Classes.

Note on "PeopleCode Types." The term "PeopleCode type" is still frequently used, but fits poorly into the PeopleTools event-driven metaphor. The term **PeopleCode event** should now be used instead. However, it's often convenient to qualify a class of PeopleCode programs triggered by a specific event with the event name; for example, PeopleCode programs associated with the RowInit events are collectively called RowInit PeopleCode.

Accessing PeopleCode in Application Designer (Overview)

You can access PeopleCode associated with the various Application Designer definitions in several ways.

With record fields and pop-up menu items, the Project View displays PeopleCode programs within the project hierarchy using a lightning bolt symbol. The programs are children of the fields and pop-up menu items with which they're associated, and are named according to their associated events, such as ItemSelected, RowInit, or SaveEdit, as shown in the following illustration. Double-click a record field or pop-up menu item program in the Project View to launch the PeopleCode Editor and load that program for editing.



PeopleCode Programs in the Project View Hierarchy



For more information about record field and menu item PeopleCode programs and other ways to access them, see Record Field PeopleCode and Component PeopleCode.

You can associate PeopleCode with many other types of definitions, such as

- components
- pages
- Component Interfaces

Such PeopleCode programs don't display in the Project View. Instead, right-click the definition's name and select View PeopleCode from the pop-up menu. You can also access them from their associated definitions, as described for each definition in this section.

PeopleCode can also be associated with the following:

- component records, that is, with specific records included in components
- component record fields, that is, with specific record fields included in components
- ActiveX controls

Since component record fields, component records and ActiveX controls don't appear in the Project View at all, you must access their associated programs through their parent definitions, as described for each definition in this section.

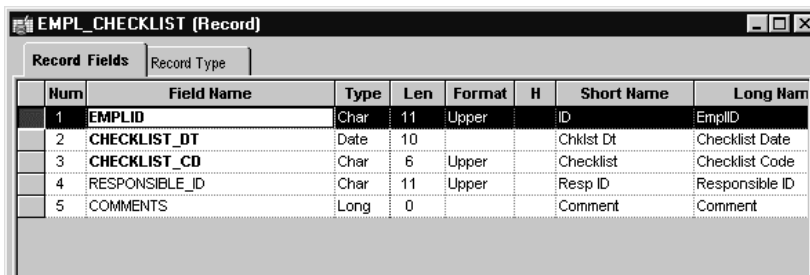
Record Field PeopleCode

A record is a table-level definition, such as an SQL table, a View, or a Derived/Work Record. Record fields are child definitions of records, and Record Field PeopleCode programs are child definitions of record fields. A record field can have zero or one PeopleCode programs for each event in the record field event set.

In addition to accessing programs through the Project View, you can also access them from the record definition, and from any page definition that includes that record.

Accessing Record Field PeopleCode From a Record Definition

Record definition fields that have PeopleCode associated with them appear bold in all record views.

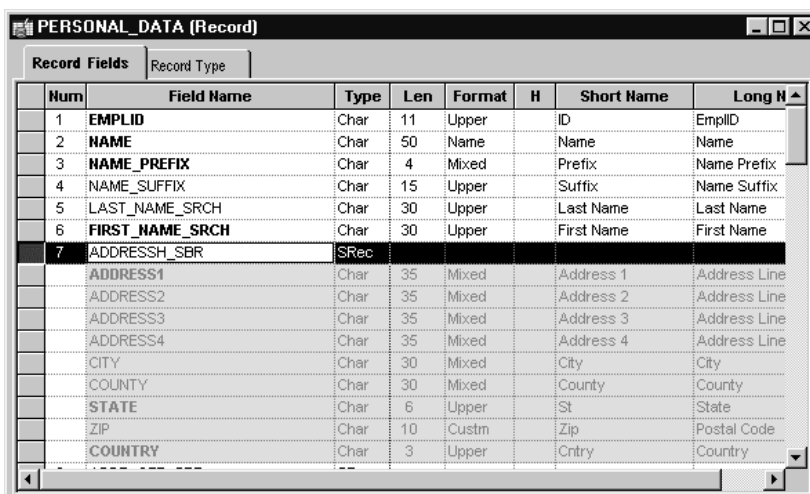


The screenshot shows the 'EMPL_CHECKLIST (Record)' window with the 'Record Fields' tab selected. The table below represents the data shown in the window:

Num	Field Name	Type	Len	Format	H	Short Name	Long Nam
1	EMPLID	Char	11	Upper		ID	EmplID
2	CHECKLIST_DT	Date	10			Chklist Dt	Checklist Date
3	CHECKLIST_CD	Char	6	Upper		Checklist	Checklist Code
4	RESPONSIBLE_ID	Char	11	Upper		Resp ID	Responsible ID
5	COMMENTS	Long	0			Comment	Comment

Record Definition with Three Fields with PeopleCode

In the above example, the first three fields (in bold) have PeopleCode associated with them. In addition, if you expand the subrecords in a record definition, any fields in the subrecord that have PeopleCode associated with them will display in bold.



The screenshot shows the 'PERSONAL_DATA (Record)' window with the 'Record Fields' tab selected. The table below represents the data shown in the window:

Num	Field Name	Type	Len	Format	H	Short Name	Long N
1	EMPLID	Char	11	Upper		ID	EmplID
2	NAME	Char	50	Name		Name	Name
3	NAME_PREFIX	Char	4	Mixed		Prefix	Name Prefix
4	NAME_SUFFIX	Char	15	Upper		Suffix	Name Suffix
5	LAST_NAME_SRCH	Char	30	Upper		Last Name	Last Name
6	FIRST_NAME_SRCH	Char	30	Upper		First Name	First Name
7	ADDRESSH_SBR	SRec					
	ADDRESS1	Char	35	Mixed		Address 1	Address Line
	ADDRESS2	Char	35	Mixed		Address 2	Address Line
	ADDRESS3	Char	35	Mixed		Address 3	Address Line
	ADDRESS4	Char	35	Mixed		Address 4	Address Line
	CITY	Char	30	Mixed		City	City
	COUNTY	Char	30	Mixed		County	County
	STATE	Char	6	Upper		St	State
	ZIP	Char	10	Custm		Zip	Postal Code
	COUNTRY	Char	3	Upper		Cntry	Country

Record Definition with Subrecords with PeopleCode

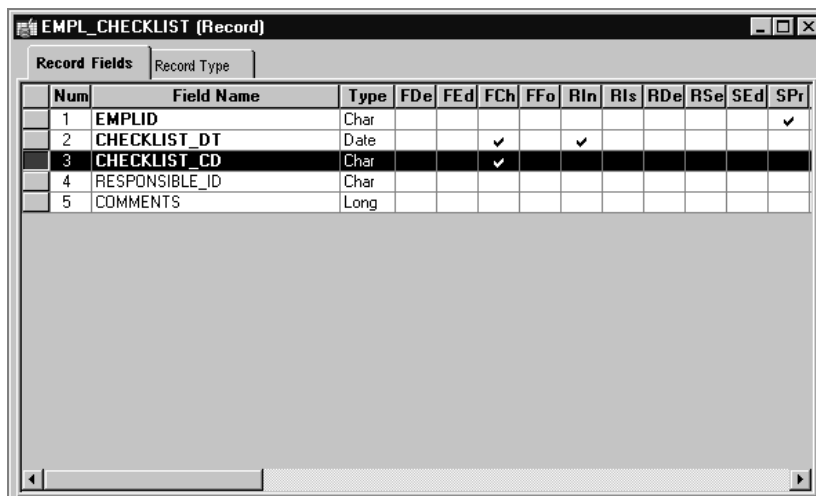
To access record field PeopleCode from an open record definition:

1. Click the  PeopleCode Display button on the toolbar.

A grid appears with a column for each event in the record field event set. Each cell represents a field/event combination. The column names are abbreviations of the record field event names, for example, **FCh** for the FieldChange event and **RIn** for the RowInit event. A checkmark appears in the appropriate cell for each field/event combination that has an associated PeopleCode program.



For more information about individual events, see Record Field Event Set.



Record Fields		Record Type												
Num	Field Name	Type	FDe	FEd	FCh	FFo	RIn	RIIs	RDe	RSe	SEd	SPr		
1	EMPLID	Char												
2	CHECKLIST_DT	Date			✓		✓							
3	CHECKLIST_CD	Char			✓									
4	RESPONSIBLE_ID	Char												
5	COMMENTS	Long												

Accessing Record Field PeopleCode from a Record Definition

2. Access the PeopleCode.

You can access the PeopleCode for a given cell by:

- Double-click the cell.
- Right-click the cell, and select View PeopleCode from the pop-up menu.
- Select **View, PeopleCode** from the main menu.

The PeopleCode Editor appears. If the field/event combination has an associated program, it displays in the editor.



For more information, see Using the PeopleCode Editor.

Accessing Record Field PeopleCode From a Page Definition

You can associate a PeopleCode program with any page control that you can associate with a record field.

To access record field PeopleCode from a page definition, right-click a page control and select **View Record PeopleCode** from the pop-up menu. The PeopleCode Editor appears, displaying the first event in the event set associated with that control's underlying record field.

Pushbutton controls are a special case. You can associate a PeopleCode program with a pushbutton only if its destination is defined as **PeopleCode Command**. When the end-user clicks a pushbutton defined like this, the FieldEdit and FieldChange events are triggered, so your PeopleCode must be associated with one of those two events. FieldChange is normally used.

To define a command pushbutton:

1. In the page definition, double-click the pushbutton to access its properties.

Page Field Properties Dialog Box

2. Select **PeopleCode Command** as the pushbutton Destination.
3. Select the record and field with which your pushbutton, and PeopleCode, are associated.

It's best to associate the pushbutton with a Derived/Work record field, which separates its PeopleCode from the PeopleCode associated with any of the page's other underlying record fields. You can then store generic PeopleCode with this field so you can reuse it with pushbuttons on other pages.

4. Click **OK** to return to the page.

Right-click the pushbutton and select View PeopleCode from the pop-up menu to access the PeopleCode Editor.



For more information, see Using the PeopleCode Editor.

Fields and Record Fields

It's important to distinguish clearly between *fields* and *record fields*:

- Fields are stand-alone definitions, on the same level as records, whose attributes are shared across all records that use the field.
- Record fields are owned by the record definitions that include them.



Fields and record fields are also distinct from *component record fields*. These are the record fields used in a component, as they appear in the component's structure view. Within that context, they have their own independent event sets and PeopleCode programs. For more information, see Component Record Field PeopleCode.

Properties of a field, such as data type and size, affect all records that include the field; therefore any change to a field property affects all records that include the field. Properties of a record field, such as PeopleCode programs and key settings, are not shared among records; a change to a record field property affects only the record that owns the record field.



PeopleCode programs are owned by record fields and component record fields, not fields.

Record Field Event Set

The events in the Record Field event set will be familiar to users of previous releases of PeopleTools. These have traditionally been called "PeopleCode types" (now an obsolete term):

- FieldChange Event
- FieldDefault Event
- FieldEdit Event
- FieldFormula Event
- RowInit Event

- RowSelect Event
- RowDelete Event
- PrePopup Event
- SaveEdit Event
- SavePreChange Event
- Workflow Event
- SavePostChange Event
- SearchInit Event
- SearchSave Event



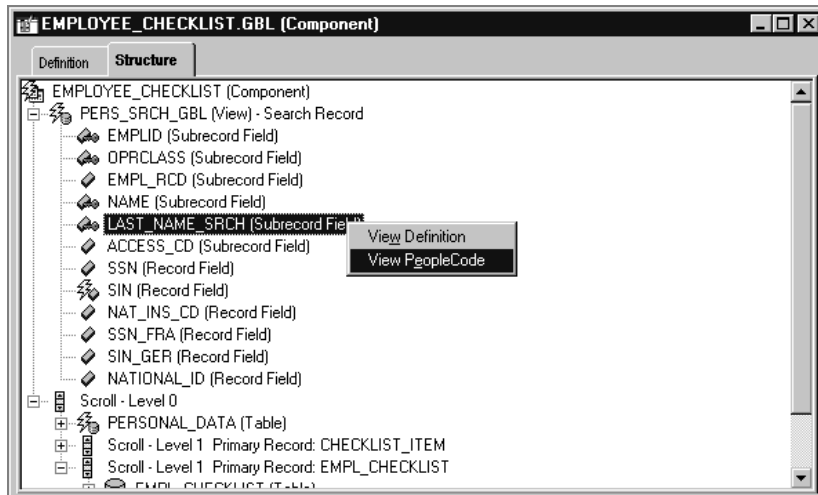
For more information, see PeopleCode and the Component Processor.

Component Record Field PeopleCode

Component record field PeopleCode is different from *record field* PeopleCode.

Component record field PeopleCode is associated with a record field, but only with respect to a component and one of its events. Use this type of association to tailor your programs to a particular component. This PeopleCode is only accessible through a component's structure display, not from any record definition.

To access PeopleCode associated with a component record field, open the component's structure view, select a field, right-click the field name, and select **View PeopleCode** from the pop-up menu. A lightening bolt displays next to the field name if PeopleCode is associated with the field *at the component level*. If PeopleCode is associated with the field at the record level, a lightening bolt doesn't display.



Accessing Component Record Field PeopleCode from the Component Structure

The PeopleCode Editor appears. If that field has associated PeopleCode, the first program in the component record field event set displays in the editor.



For more information, see Using the PeopleCode Editor.

Through a component's structure display, you can access the definition of a **record field** included in a page of the component. That record field has its own event set and associated PeopleCode.



For more information see Record Field PeopleCode.

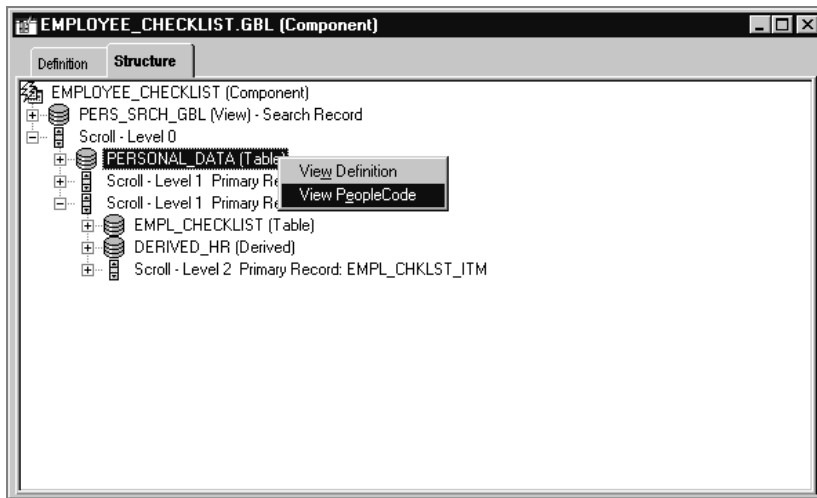
Component Record Field Event Set

- FieldChange Event
- FieldDefault Event
- FieldEdit Event
- PrePopup Event

Component Record PeopleCode

Component record PeopleCode is associated with a record definition, but only with respect to a component and one of its events. Use this type of association to tailor your programs to a particular component. This PeopleCode is directly accessible through a component's structure display, not from the record definition.

To access PeopleCode associated with a component record, open the component's structure view, select a record, right-click the record name, and select **View PeopleCode** from the pop-up menu.



Accessing Component Record PeopleCode from the Component Structure

The PeopleCode Editor appears. If that record has associated PeopleCode, the first program in the component record event set displays in the editor.



For more information, see Using the PeopleCode Editor.

Component Record Event Set

Search records and non-search records in components have different associated event sets. The following events are associated with component *search* records:

- SearchInit Event
- SearchSave Event

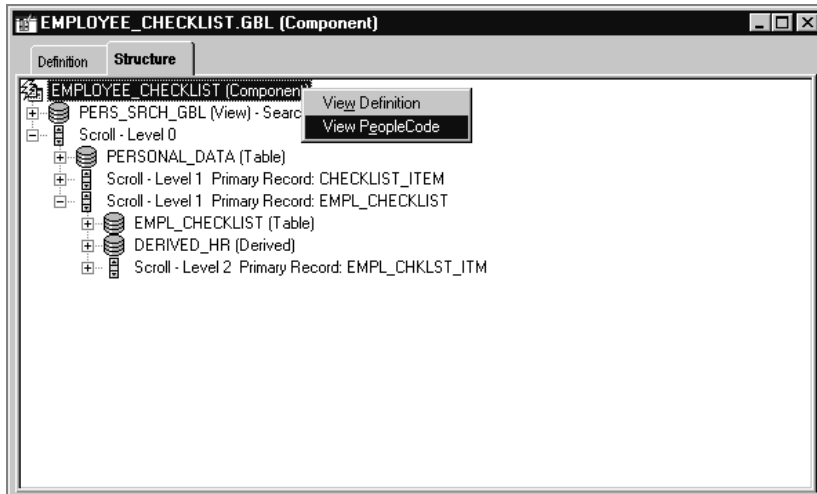
The following events are associated with component *non-search* records:

- RowDelete Event
- RowInit Event
- RowInsert Event
- RowSelect Event
- SaveEdit Event
- SavePostChange Event

- SavePreChange Event

Component PeopleCode

Component PeopleCode is associated with a component definition and an event. To access PeopleCode associated with a component, open its structure view, select the component name, right-click the name, and select **View PeopleCode** from the pop-up menu.



Accessing Component PeopleCode from the Component Structure

The PeopleCode Editor appears. If that component has associated PeopleCode, the first program in the component event set displays in the editor.



For more information, see Using the PeopleCode Editor.

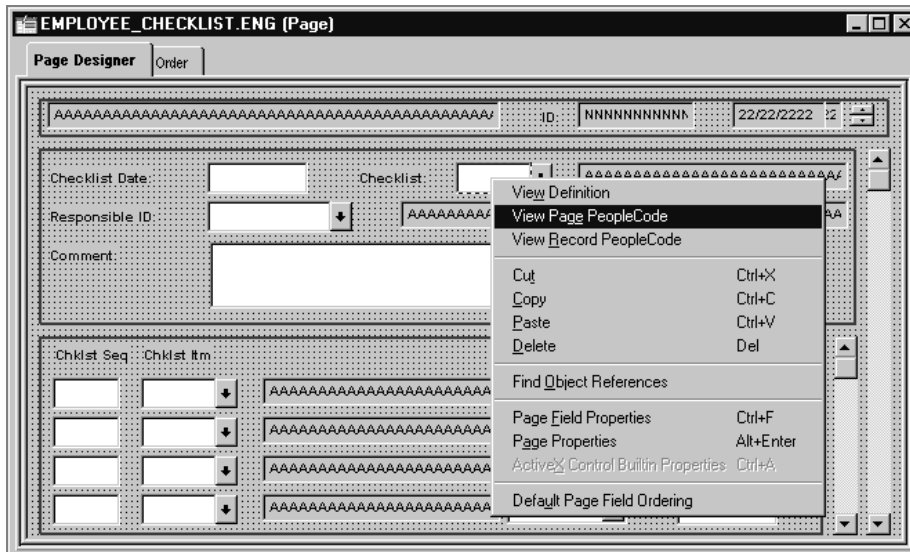
Component Event Set

- PostBuild Event
- PreBuild Event
- SavePostChange Event
- SavePreChange Event
- Workflow Event

Page PeopleCode

Page PeopleCode is associated with a page definition. Currently, pages have only the *Activate* event. This event is only valid for pages that are defined as Standard or Secondary. This event is *not* supported for subpages.

To access PeopleCode associated with a page, right-click on any part of the page's definition, except an ActiveX control, and select **View Page PeopleCode** from the pop-up menu.



Accessing Page PeopleCode from the Page Definition

The PeopleCode Editor appears. If that page has associated PeopleCode, it displays in the editor.



The term "page PeopleCode" refers to PeopleCode programs owned by pages. It's important not to confuse page PeopleCode with PeopleCode properties related to the appearance of pages, such as the Visible page class property.



For more information, see Using the PeopleCode Editor.



If you select a non-ActiveX page control, you can select View Record PeopleCode from the pop-up menu, which displays PeopleCode associated with that record field. For more information, see Record Field PeopleCode.

Page Activate Event

The page event set consists of a single event, the *Activate* event, which fires every time the page is activated.



For more information, see the Activate Event.

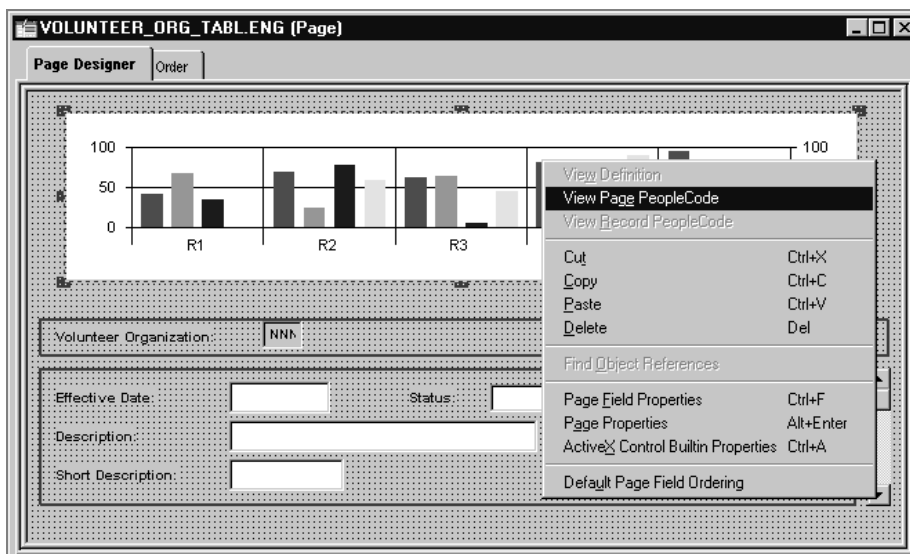
Page Field Control PeopleCode

Page Field PeopleCode is associated with page fields. The only control that has PeopleCode associated with it is an ActiveX control.



ActiveX controls are not supported in PeopleSoft Internet Architecture.

To access PeopleCode associated with an ActiveX control, select the ActiveX control in the page definition, right-click it and select **View Page PeopleCode** from the pop-up menu.



Accessing Page Field PeopleCode from a Page Definition



For more information, see Using the PeopleCode Editor.

Page Field Event Set

The only control that has PeopleCode associated with it is an ActiveX control. Every ActiveX control has its own unique event set. The following events are common to all ActiveX control event sets; only these events are part of the Component Processor flow:

- PSControlInit Event
- PSLostFocus Event



For more information, see [Implementing ActiveX Controls and ActiveX Controls in PeopleTools](#).

Menu Item PeopleCode

PeopleTools menus come in two types, pop-up and standard, both of which are stand-alone definitions in the project hierarchy.

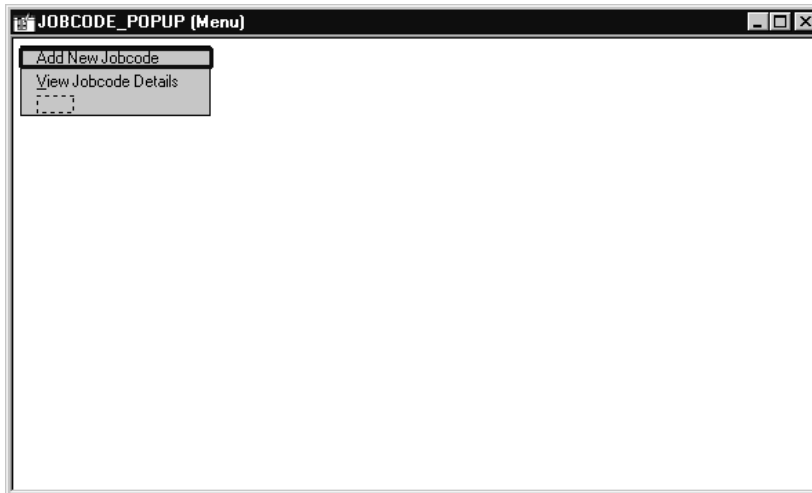


Important! For the PeopleSoft Internet Architecture, you can only associate PeopleCode with menu items in a pop-up menu.

Menu item PeopleCode programs are associated with pop-up menu items, which are child definitions of pop-up menus.

To define a PeopleCode pop-up menu item:

1. In the open pop-up menu definition, double-click the menu item to access its properties. If you're creating a new menu item, double-click the empty rectangle at the bottom of the pop-up menu.



Pop-up Menu Definition in Application Designer

2. The Menu Item Properties dialog box appears. If this is a new menu item, enter a name and a label for the item.



PeopleCode Menu Item Properties Dialog Box



For more information about menus, see [Creating Menu Definitions](#).

3. Select **PeopleCode** from the Type group.



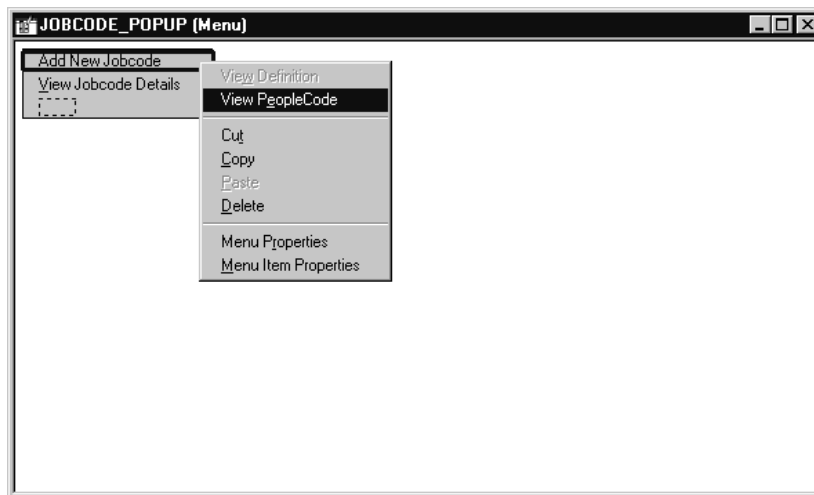
For more information about different types of menu items, see [Defining Menu Items](#).

4. Click **OK** to close the Menu Item Properties dialog box.

To access pop-up menu item PeopleCode:

1. Open the pop-up menu definition.
2. Access the PeopleCode.

Right-click the menu item. If it's defined as a PeopleCode menu item, **View PeopleCode** is enabled on the pop-up menu.



Accessing Pop-up Menu Item PeopleCode from a Menu Definition

3. Select View PeopleCode.

The PeopleCode Editor appears with that menu item's associated program, if any, displayed.



The term "menu item PeopleCode" refers to PeopleCode programs owned by menu items. It's important not to confuse menu item PeopleCode with PeopleCode functions related to the appearance of menu items, such as CheckMenuItem.



For more information, see Using the PeopleCode Editor.

Menu Item ItemSelected Event

The menu item event set consists of a single event, the ItemSelected Event. This event fires whenever an user chooses a menu item from a pop-up menu; so naturally this is where you put PeopleCode programs that are executed from menu items.



For more information, see the ItemSelected Event

Application Message PeopleCode

Application messages have the following types of Application Designer definitions with associated PeopleCode:

- the message
- the message channel

These definitions have separate definitions and different event sets.

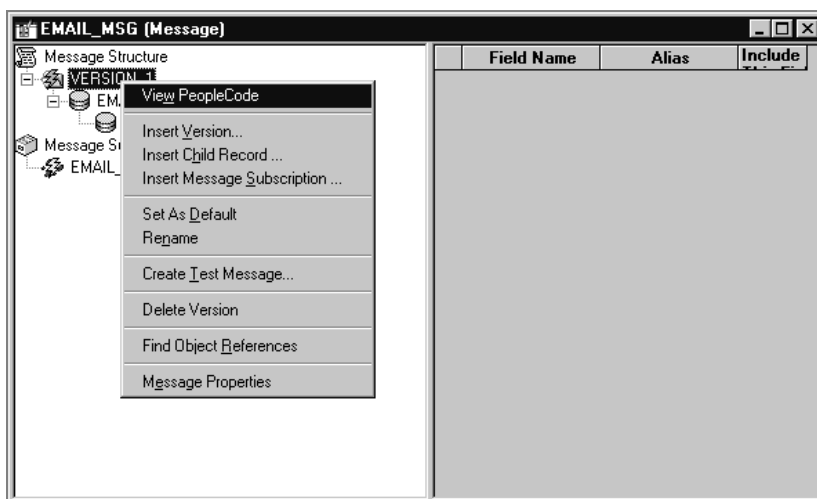


For more information about application message definitions, see PeopleSoft Application Messaging.

Accessing Message PeopleCode

In a message definition, you can associate a PeopleCode program with the message itself and with each message subscription included in the definition.

To access PeopleCode associated with a message, open the message definition, right-click anywhere in the Message Structure display, and select **View PeopleCode** from the pop-up menu.



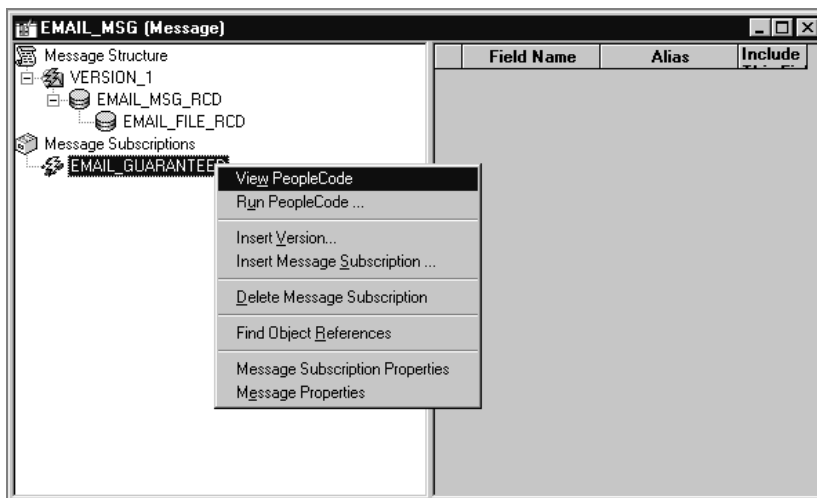
Accessing Message PeopleCode from the Message Definition

The PeopleCode Editor appears. If that message definition has associated programs, the first one in the application message event set displays in the editor.



For more information, see Using the PeopleCode Editor.

Each message subscription entry in a message definition represents a PeopleCode program. To access that program, open the message definition, select the subscription, right-click the subscription name, and select View PeopleCode from the pop-up menu.



Accessing Message Subscription PeopleCode from the Message Definition

The PeopleCode Editor appears, with the message subscription's associated program displayed in the editor.

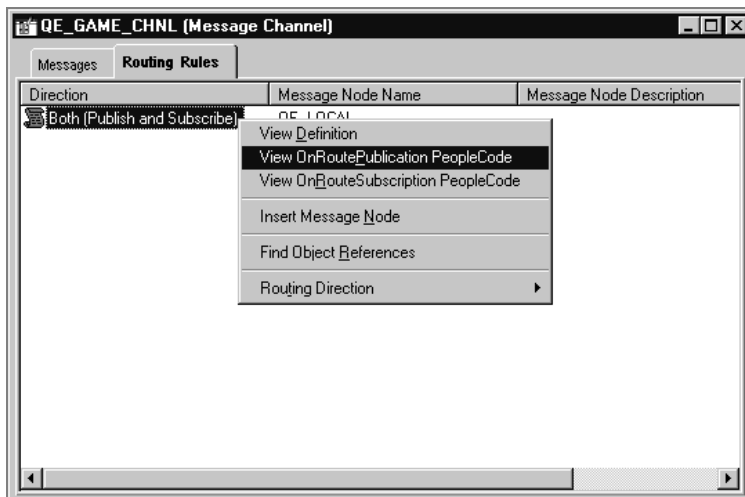


For more information, see Using the PeopleCode Editor.

Accessing Message Channel PeopleCode

In a message channel definition, PeopleCode is associated with the message channel through its message nodes. To access PeopleCode associated with a message channel, open the **Routing Rules** view, select a message node, right-click on the node's name, and select one of the following from the pop-up menu:

- View OnRoutePublication PeopleCode
- View OnRouteSubscription PeopleCode



Accessing Message PeopleCode from the Message Channel Definition

The PeopleCode Editor appears, with the selected PeopleCode program, if any, displayed in the editor.



For more information, see Using the PeopleCode Editor.

Application Message Event Sets

These events are not considered part of the Component Processor flow, so they're documented separately from the majority of PeopleCode events. The following events are associated with messages:

- OnPublishTransform Event
- OnSubscribeTransform Event

The following event is associated with message subscriptions:

- Subscription Event

The following events are associated with message channels:

- OnRoutePublication Event
- OnRouteSubscription Event



For more information, see PeopleSoft Application Messaging.

How PeopleCode Programs are Stored and Saved

When you save an Application Designer definition, PeopleTools saves all PeopleCode programs belonging to that definition that have been added or modified. Component record field PeopleCode and component record PeopleCode belong to the component. If new PeopleCode programs have been added, PeopleTools creates an association between the owning definition and new rows in the PeopleCode table. If a PeopleCode program is deleted, the association between the owning definition and the program is also deleted.

Saving PeopleCode programs together with their owning definitions helps to guarantee the integrity of the application. In the event of system failure, it's now extremely unlikely that your application could end up storing PeopleCode programs that are not associated with any definition; or worse, a definition associated with a PeopleCode program that was never saved.



When you save any Application Designer definition, all PeopleCode programs that you've added or changed since the last save will be checked, formatted, and saved at the same time. If you want to format and check the syntax of a single PeopleCode program, use the Validate Syntax command instead of the Save command.

Automatic Backup of PeopleCode

A PeopleCode program is automatically saved to a file while you're working on it. This checkpoint occurs at the following times:

- Every 10 keystrokes.
- On a save command, just prior to the save being executed (in case the save doesn't actually execute because the code is invalid).
- When another PeopleCode program is selected to be edited (if you have two PeopleCode editor windows open at the same time, and you move from one to the other).

The file is saved to your temp directory (as specified in your environment), in a file with the following name:

```
PPCMMDDYY_HHMMSS.txt
```

where MMDDYY represents the month, date and year, respectively, of the checkpoint, and HHMMSS represents the hour, minute and second, respectively.

The top of the checkpoint file contains the following information:

```
[PeopleCode Checkpoint File]
```

```
[RECORD.recordname.FIELD.fieldname.METHOD.eventname]
```

If your PeopleCode program is saved successfully, any checkpoint files associated with that program are automatically deleted.

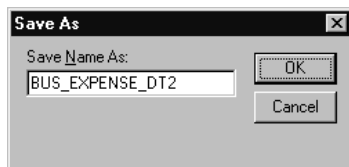
Copying PeopleCode with a Parent Definition

When you create a copy of an Application Designer definition that contains PeopleCode, Application Designer lets you choose whether to copy all PeopleCode programs along with the definition. Each copy of the definition gets a separate copy of the PeopleCode programs.

To copy a definition with its PeopleCode:

1. Open the definition you want to copy.
2. Choose File, Save As.

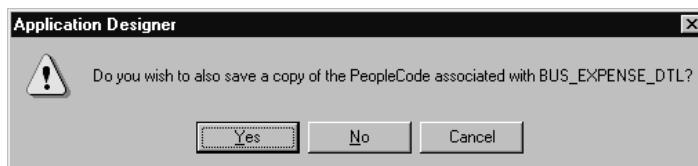
The Save As dialog appears. Type a name for the new definition in the dialog box.



Save As Dialog Box

3. Click **OK**, then click **Yes** to copy the PeopleCode.

Click Yes in this dialog box to copy all PeopleCode associated with the definition.



Saving Associated PeopleCode Dialog Box

Upgrading PeopleCode Programs

You can upgrade PeopleCode programs independently of the definitions with which they're associated.



For more information, see the Upgrade Documentation.

CHAPTER 3

Using the PeopleCode Editor

This section covers the practical aspects of adding PeopleCode to PeopleSoft applications through the Application Designer interface. It discusses various techniques for using the PeopleCode Editor and its features.

Navigating Between PeopleCode Programs

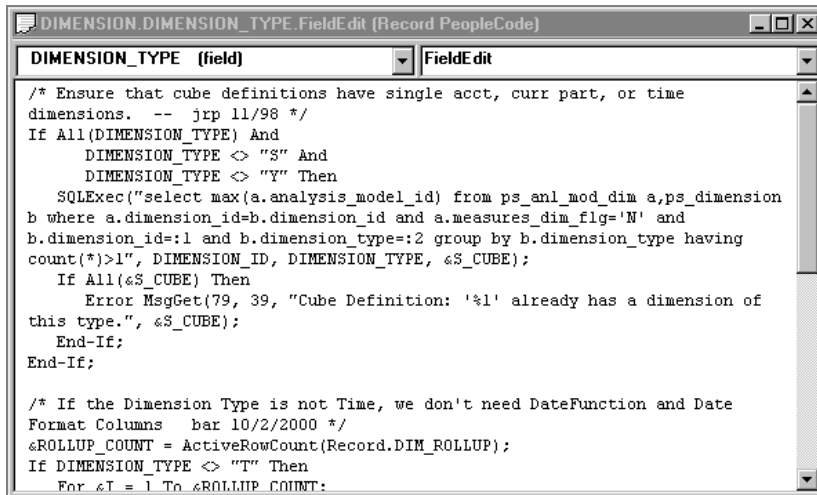
The PeopleCode Editor navigational features have been significantly enhanced, making it a more powerful and convenient tool. Once you access a PeopleCode program associated with an Application Designer definition, you can access programs associated with other related definitions, without having to close the Editor window.



For more information about accessing PeopleCode within Application Designer, see [Understanding PeopleCode and Events](#).

Understanding the PeopleCode Editor Window

Application Designer supplies an independent Editor window for each record, definition, menu, message, and so on, for which you invoke the Editor—these are the *parent definitions*. The Editor window's title bar displays the name and type of the parent *definition*.



PeopleCode Editor Window with Record Field PeopleCode

The Editor window contains the main *edit pane*, the drop-down *definition list* at the top left, and the drop-down *event list* at the top right. The drop-down lists enable you to navigate directly to the PeopleCode associated with related *child definitions*—for example, fields within a record—and their event sets.



When you make a selection from either drop-down list, your selected entry has a yellow background, indicating that you must click in the edit pane before you can start typing.



For more information, see [Using the Drop-down Definition List](#) and [Using the Drop-down Event List](#).

You can open as many Editor windows as you want, and resize them within the Application Designer. Each line of code wraps automatically, based on the window's current width. A vertical scroll bar appears if the program has more lines than the Editor can display in the edit pane.



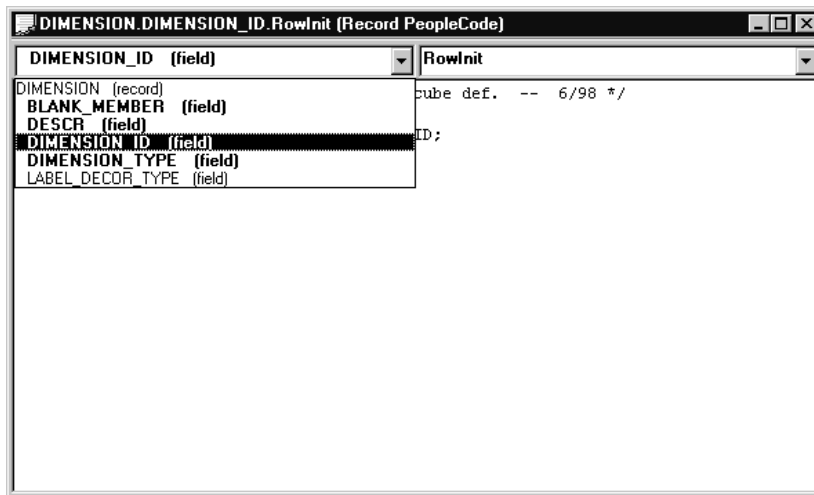
You can't open two Editor windows for a single parent definition, or for any two of its child definitions.

Using the Drop-down Definition List

The PeopleCode Editor's drop-down definition list enables you to navigate between PeopleCode programs that are associated with a given parent definition and its children. The list displays the complete hierarchy of child definitions to which you can navigate. The structure of the definition list depends on the type of parent definition. Each type of list is illustrated and described in the following sections.

Selecting a Record Field

The record PeopleCode drop-down definition list displays all the record fields included in a record. Each record field displayed in bold has a PeopleCode program associated with at least one of the events in its event set.

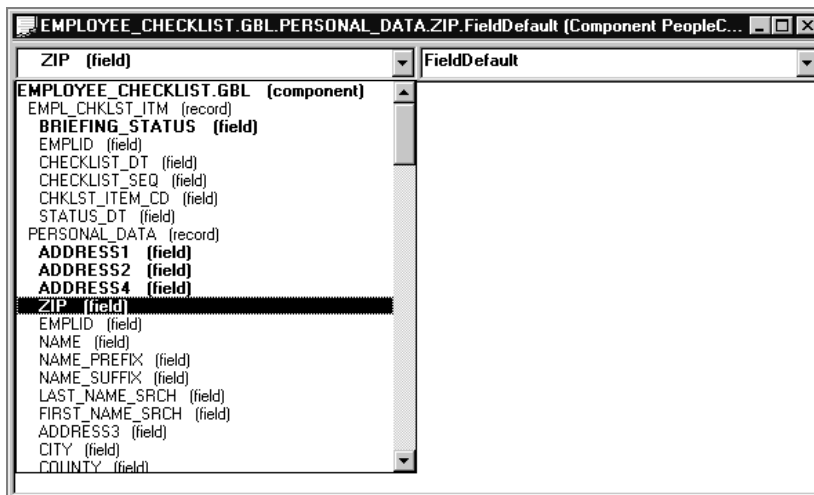


Selecting a Record Field from the Record Definition List

The record name appears at the top of the list, but you can't associate PeopleCode with just a record; you can only associate it with a record field. The record name is displayed to visually clarify the location of the record fields.

Selecting a Component Definition

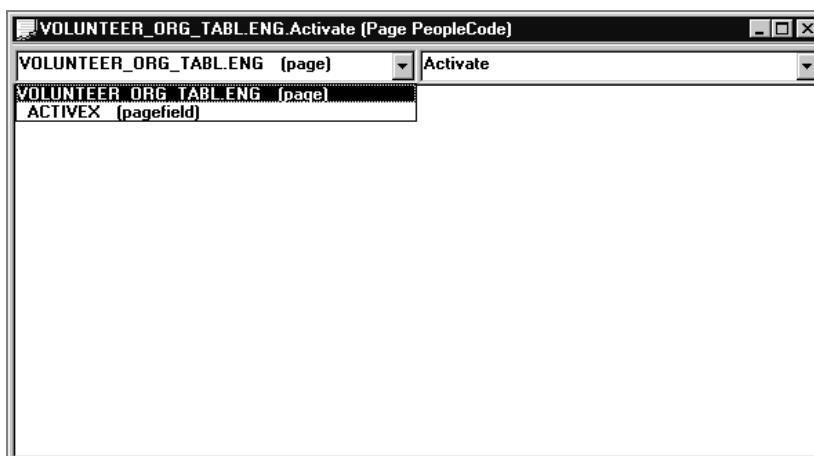
The component PeopleCode drop-down definition list displays all the component record fields and component records included in a component. Each definition displayed in bold has a PeopleCode program associated with at least one of the events in its event set.



Selecting a Component Record Field from the Component Definition List

Selecting a Page or ActiveX Control

The page PeopleCode drop-down definition list displays a page and all the ActiveX controls included on the page. Each definition displayed in bold has a PeopleCode program associated with at least one of the events in its event set.



Selecting a Page from the Page Definition List

Selecting a Menu Item

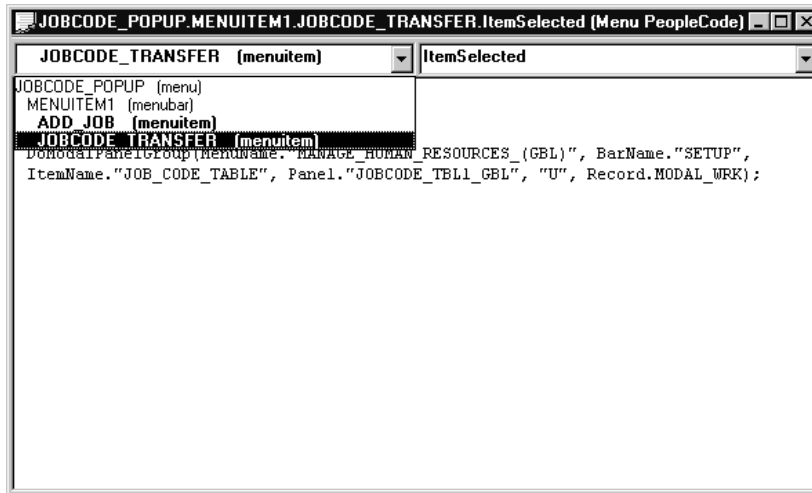
The menu PeopleCode drop-down definition list displays the menu, its menu bars and menu items. Each menu item displayed in bold has an associated PeopleCode program.



Important! For the PeopleSoft Internet Architecture, you can only associate PeopleCode with menu items in a pop-up menu.



For more information about defining PeopleCode menu items, see [Understanding PeopleCode and Events](#).

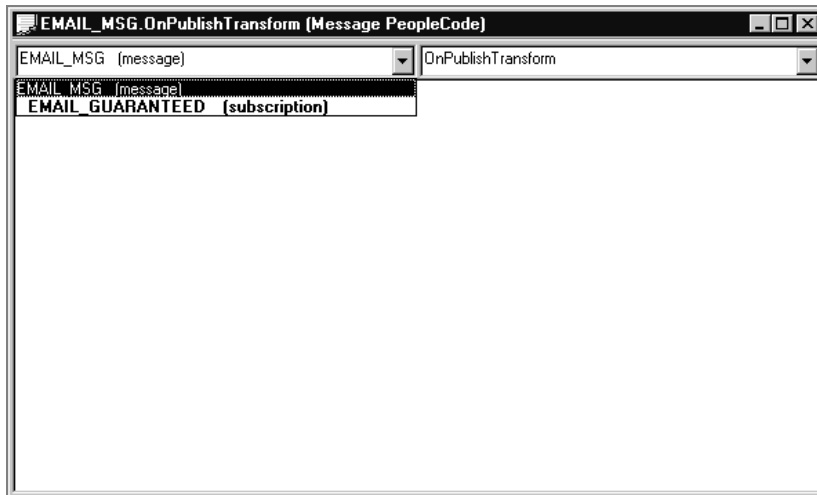


Selecting a PeopleCode Menu Item from the Menu Definition List

The menu and menu bar names appear on the list, but you can't associate PeopleCode with just a menu or a menu bar; you can only associate it with a menu item. The menu and menu bar names are displayed to visually clarify the positions of the PeopleCode menu items in the pop-up menu hierarchy.

Selecting a Message or Message Subscription

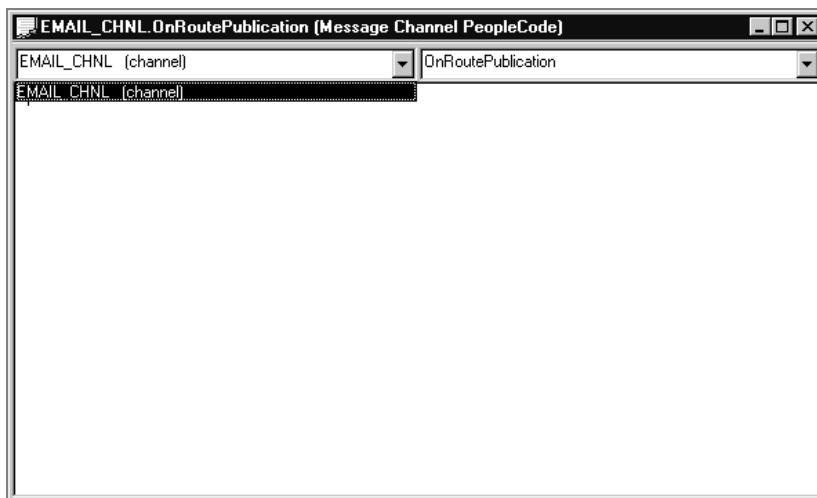
The message PeopleCode drop-down definition list displays a message, and all the message subscriptions included in the message definition. Each definition displayed in bold has a PeopleCode program associated with at least one of the events in its event set. The message subscriptions *are* PeopleCode programs, so they will always be bold.



Selecting a Message from the Message Definition List

Selecting a Message Channel

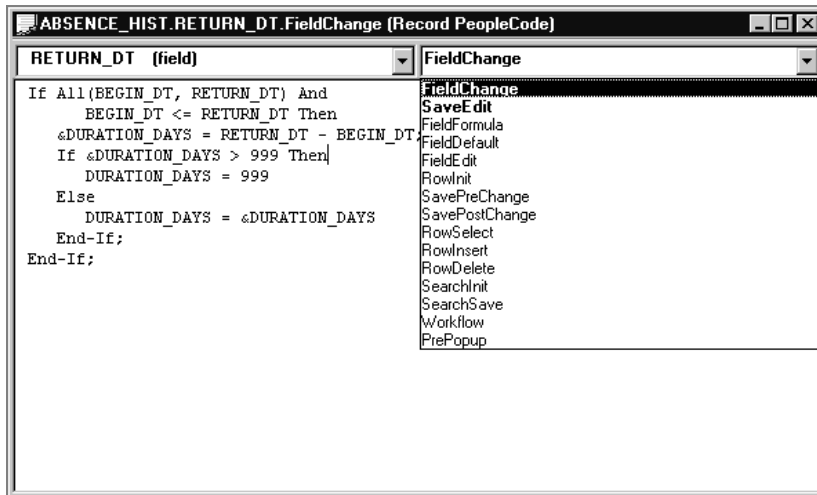
The drop-down definition list for message channel PeopleCode contains only one entry—the message channel itself, which is always the selected definition. If it's displayed in bold, it has a PeopleCode program associated with at least one of the events in its event set.



Viewing the Single-Entry Message Channel Definition List

Using the Drop-down Event List

The PeopleCode Editor's drop-down event list enables you to select an event from the event set of the currently selected definition. You can use this event list to navigate between PeopleCode programs that are associated with that definition. For every definition/event combination with associated PeopleCode, the event name is displayed in bold, and it appears at the top of the event list.



Selecting an Event from the Record Event Set



For more information about events and event sets, see [Understanding PeopleCode and Events](#) and [PeopleCode and the Component Processor](#).

Using the PeopleCode Editor

The PeopleCode Editor works much like any other text editor, but has capabilities specifically geared toward the PeopleTools environment.








- Its editing functions are integrated with the menus and toolbar of Application Designer and are also accessible from a pop-up window.
- It checks, formats, and saves all programs associated with Application Designer definitions simultaneously when any definition is saved (see [How PeopleCode Programs are Stored and Saved](#)).
- It includes a Validate Syntax command for checking and formatting a single PeopleCode program without saving.
- It supports standard Windows drag-and-drop editing.
- You can open separate instances of the Editor simultaneously, and you can drag and drop text between programs.
- You can open the definition with which the current set of PeopleCode programs is associated from within the PeopleCode Editor.
- You can open a field, record, page, file layout, message definition or other definitions from a PeopleCode reference to the field, record, page, file layout or message, and so on.
- You can access the PeopleCode programs associated with a field, record, page, file layout message definition or other definitions from a PeopleCode reference to the field, record, page,

file layout or message, and so on.

- You can open a PeopleCode Editor window containing an external function definition from a function declaration or function call.
- You can press F1 with the cursor in a PeopleCode built-in function, method, meta-SQL, and so on, to open the PeopleSoft help for that item.

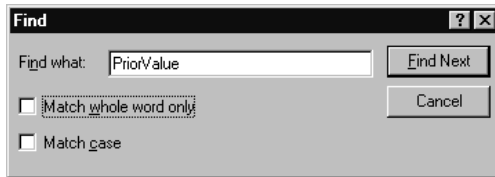
Editing Functions

The PeopleCode Editor supports the standard editing functions **Save**, **Cancel**, **Cut**, **Copy**, **Paste**, **Find**, **Replace**, and **Undo**; from the PeopleCode Editor pop-up menu. **Cut**, **Copy**, and **Paste** use standard Windows keyboard shortcuts. You can also cut, copy, and paste within the same PeopleCode program or across multiple programs.

Command	Key	Button
Save	Ctrl+S	
Cancel	Esc	
Cut	Ctrl+X or Shift+Del	
Copy	Ctrl+C or Ctrl+Ins	
Paste	Ctrl+V or Shift+Ins	
Find	Ctrl+F	
Replace	Ctrl+H	
Undo	Ctrl+Z or Alt+Bksp	
Validate		

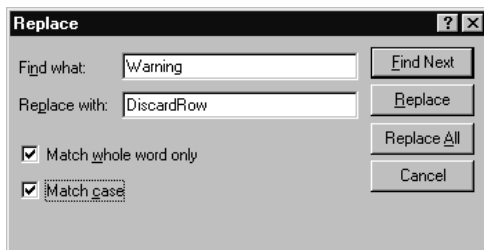
Find and Replace

When you use the Find and Replace functions, any text string that is highlighted appears when either the **Find** or **Replace** dialog boxes are called. For example, if you select the function PriorValue, it appears in the **Find** dialog box when it's called.



Find Dialog Box


You can step through finding and replacing text strings one string at a time, or click on **Replace All** to replace globally. The **Undo** function is available to undo the last replace or replace all.



Replace Dialog box

Validating Syntax

To check the syntax of the current PeopleCode program and format it if it is syntactically correct, do one of the following:

- Click the **Validate Syntax** button  on the Application Designer toolbar.
- Select **Tools, Validate Syntax** from the Application Designer menu.
- Right-click in the PeopleCode Editor window, then select **Validate Syntax** from the pop-up menu.

Validate is a utility with several functions, one of which is to check for PeopleCode that won't run in the PeopleSoft Internet Architecture. You can check either a single component or an entire project.



For more information see [Validating Projects](#).



This feature is convenient if you have written multiple PeopleCode programs and you want to check the syntax of one without saving. If you save the current record or menu, all of the PeopleCode programs associated with the record or menu are checked prior to saving.

Auto Formatting

You do not need to format your PeopleCode statements; you only need to use the correct syntax. In fact, when you save or validate, the system formats the code according to the rules in the PeopleCode tables—no matter how you entered it originally. It automatically converts field names to upper case and indents statements for you. This makes your PeopleCode look consistent with other programs in the system.

PeopleCode is case-insensitive, except for quoted literals. PeopleCode does not format anything surrounded by quotation marks. String comparisons, however, are case-sensitive. When you compare the contents of a field or a variable to a string literal, make sure the literal is in the right case.

Drag-and-Drop Editing

In addition to the standard keyboard shortcuts and toolbar buttons for editing, you can copy or move text within a window or between two PeopleCode Editor windows, using the mouse and the CTRL key on your keyboard.



You can't open two Editor windows for a single parent definition, or for any two of its child definitions.

To move text between instances of the PeopleCode Editor:

1. Select the text you want to move.
2. Click in the selection and drag the mouse to the other PeopleCode Editor window.
3. When the cursor appears at the place where you wish to insert the text, release the mouse button.

To copy text between instances of the PeopleCode Editor:

1. Select the text you want to move.
2. Holding down the **Ctrl** key, click in the selection and drag the mouse to the other PeopleCode Editor window.
3. When the cursor appears at the place where you wish to insert the text, release the mouse button.

Accessing PeopleCode External Functions

An external PeopleCode function is a function written in PeopleCode (as opposed to a built-in function or external DLL function) and defined in a program outside the one from which it is

called. External PeopleCode functions can be defined in any Record PeopleCode program, but conventionally they are stored in the FieldFormula event in records beginning with FUNCLIB_.

The PeopleCode Editor gives you immediate access to external PeopleCode function definitions. Right-click the function name in the program where the function is called, then choose **View Function *FunctionName*** from the pop-up menu. This opens a new PeopleCode Editor window containing the external function definition.



Internet Scripts are contained in records similar to FUNCLIB_ records. However, they're named WEBLIB_XXX.

Accessing Definitions and Associated PeopleCode

You can open a field, record, page, message, and other definitions from the PeopleCode Editor. Or you can open a new PeopleCode Editor window containing the programs associated with a field, record, page, and so on.

To open a definition from the PeopleCode Editor:

1. Right-click on a PeopleCode definition reference
2. Choose **View Definition** from the pop-up menu.

For example, you could open definitions by clicking in any of the following references:

```
RECORD.BUS_EXPENSE_PER
BUS_EXPENSE_PER.EXPENSE_PERIOD_DT
PAGE.BUSINESS_EXPENSES
```

If you access a record definition from a record field reference (that is, *recordname.fieldname*) the specified record field is selected when the record definition opens.

To open a new PeopleCode editor window:

1. Right-click on a reference to the definition.
2. Choose **View PeopleCode** from the pop-up menu.

For example, you can access record PeopleCode from the following record and record field references:

```
RECORD.BUS_EXPENSE_PER
BUS_EXPENSE_PER.EXPENSE_PERIOD_DT
```

Context-Sensitive Help

The PeopleCode Editor has context-sensitive online reference help for all PeopleCode built-in functions, methods, properties, system variables, and meta-SQL. To access online help, place the cursor in the name of what you want to look up, then press F1. If there is a corresponding entry in the online reference system it will be displayed; otherwise a "No Help Available" error message appears.

If more than one entry is applicable, a popup window showing all applicable entries displays. You can choose the correct entry from there.

In order to use the F1 functionality, you must have specified where the documentation exists on your system, on the PeopleTools Options page, in ***F1 Help URL***.



For more information, see [PeopleTools Options](#).

The following is an example of ***F1 Help URL***:

```
http://Pandora/doc/flsearch.htm?ContextID=%CONTEXT_ID%&LangCD=%LANG_CD%
```

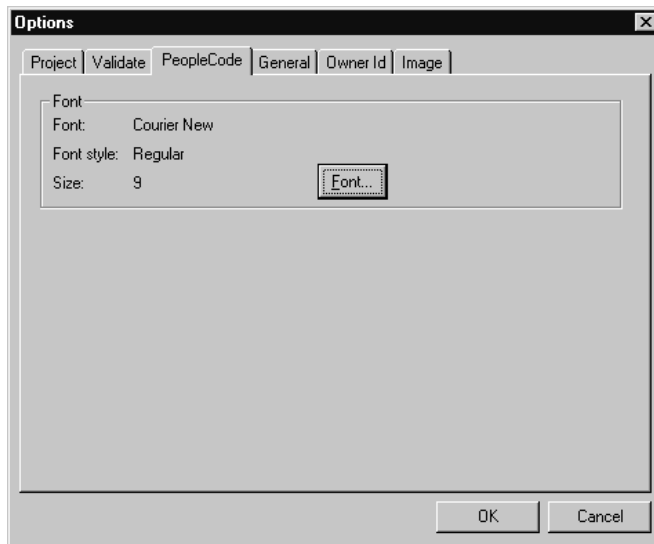
After you set the F1 help location, you must quit out of all PeopleTools sessions and start again before the F1 functionality is active.

Choosing a Font for the PeopleCode Editor

The default font for the PeopleCode Editor is 9-point Courier New. You can change the font in the PeopleCode Editor. This may be necessary for some languages that don't display correctly using Courier New.

To change the PeopleCode editor font:

1. Select **Tools, Options**, then the PeopleCode tab.



Tools Options Dialog Box

Use this dialog to change the font in the PeopleCode editor.



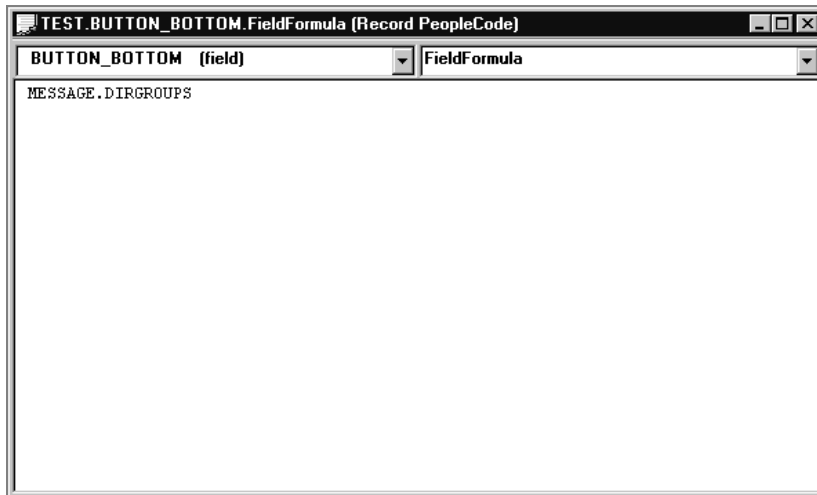
Note on displaying foreign characters. When you select a font for the PeopleCode Editor, the font selection dialog presents you with choices based on a character set appropriate for your international version of Windows. If you experience trouble embedding foreign characters (such as Thai characters) in PeopleCode, you might need to change the font setting. If you are trying to display Thai characters in Windows 95, you might also need to change your keyboard input settings for the characters to display correctly. You can change your keyboard input settings from the Input Locales tab on the Windows Regional Settings control page, or on the Keyboard control page.

Generating PeopleCode using Drag-and-Drop

You can generate references to definitions using drag-and-drop. You can also generate PeopleCode templates for accessing Business Interlinks and Component Interfaces.

Generating Definition References

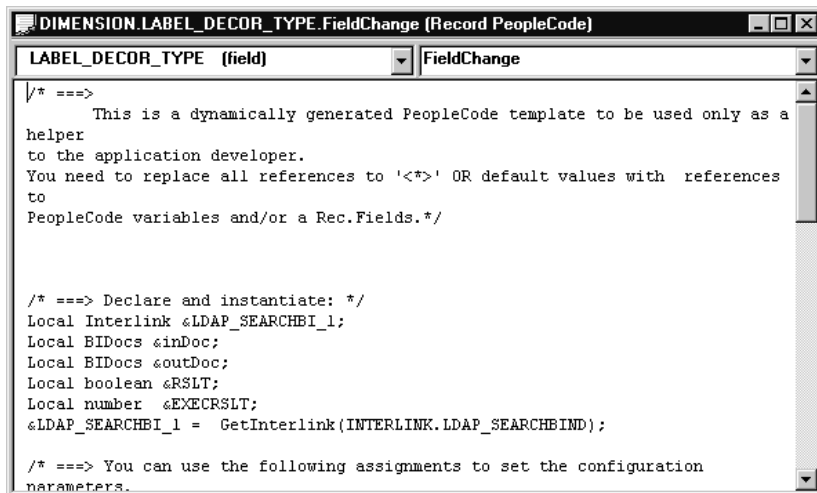
You can drag-and-drop definitions, such as menus, records, record fields, pages, and so on, from a project into an open PeopleCode editor window. Doing this generates a reference to the definition. For example, suppose your project contain a message definition named DIRGROUPS. If you select the DIRGROUPS message definition from the project, drag it into an open PeopleCode window, the following is generated:



Example PeopleCode editor window

Generating PeopleCode for a Business Interlink

After you create your Business Interlink definition, you must use PeopleCode to instantiate an Interlink object and execute the 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.



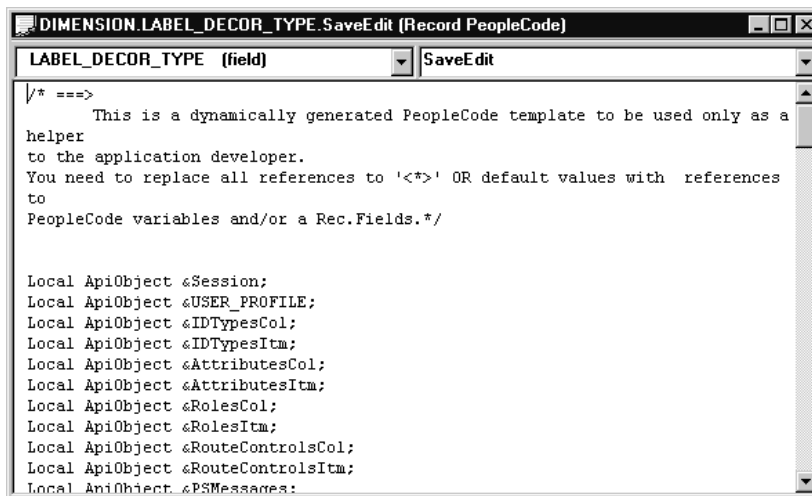
Example of Business Interlink code template



For more information, see the [PeopleSoft Business Interlink Application Developer Guide](#).

Generating PeopleCode for 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.



Example of Component Interface code template

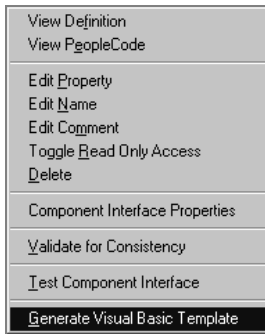


For more information, see [Component Interface](#).

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.

To generate a Visual Basic Template:

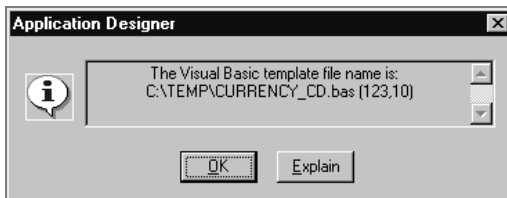
1. Open a Component Interface in Application Designer.
2. Right-Click anywhere in the open Component Interface.
3. Select Generate Visual Basic Template from the pop-up menu.



Component Interface pop-up menu

You must save the Component Interface before generating the template.

When the template is successfully generated, a message displays with the full path and name of the file containing the template.



Message with path and file name

4. Open the generated file and modify the source code to meet the needs of your application.

Here's an example template.

```
' ==>

'This is a dynamically generated Visual Basic 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

'Visual Basic variables.
```

```
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 = <*>

strOperatorID = <*>

strPassword = <*>


***** 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 = <*>


***** Execute Get Or Create *****

oCURRENCY_CD.Get
```

```

'oCURRENCY_CD.Create

'***** BEGIN:  Set Component Interface Properties *****

'***** BEGIN:  Set Component Interface Properties *****

'Set CURRENCY_CD_TBL Collection Field Properties -- Parent: PS_ROOT
Collection

Set oCURRENCY_CD_TBL = oCURRENCY_CD.CURRENCY_CD_TBL

'For <*> = 1 to oCURRENCY_CD_TBL.Count

Set oCURRENCY_CD_TBLItem = oCURRENCY_CD_TBL.Item(<*>)

oCURRENCY_CD_TBLItem.CURRENCY_CD = <*>

oCURRENCY_CD_TBLItem.EFFDT = <*>

oCURRENCY_CD_TBLItem.EFF_STATUS = <*>

oCURRENCY_CD_TBLItem.DESCR = <*>

oCURRENCY_CD_TBLItem.DESCRSHORT = <*>

oCURRENCY_CD_TBLItem.COUNTRY = <*>

oCURRENCY_CD_TBLItem.CUR_SYMBOL = <*>

oCURRENCY_CD_TBLItem.DECIMAL_POSITIONS = <*>

oCURRENCY_CD_TBLItem.SCALE_POSITIONS = <*>

'Next <*>

'***** END:  Set Component Interface Properties *****

'***** END:  Set Component Interface Properties *****

'***** 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
```


CHAPTER 4

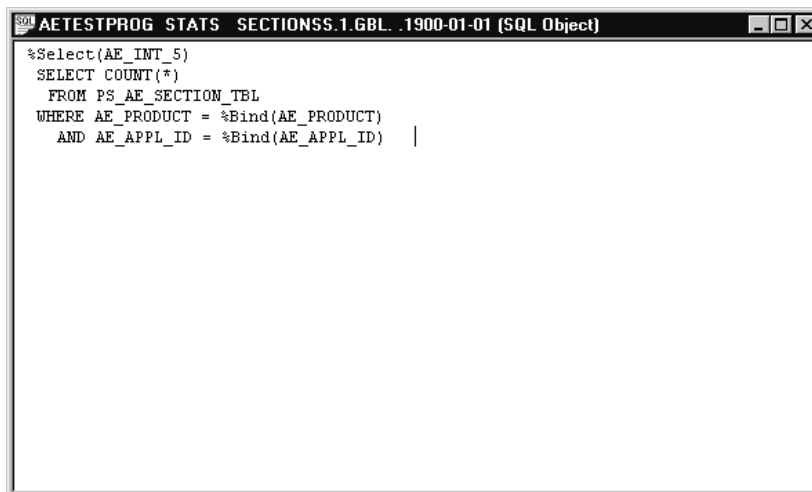
Introducing the SQL Editor

Use the SQL editor to create SQL for SQL definitions, record views, and Application Engine programs.

The SQL editor and the PeopleCode editor interfaces are similar. You can add, delete and change text: you can use the find and replace function; and you can validate your SQL. When you save your SQL definition, the code is automatically formatted (indented, and so on), just as it is for a PeopleCode program.

Understanding the SQL Editor Window

The Editor window's title bar displays either the name of the SQL definition, or the name of the component that contains the SQL. For example, if the SQL statement is part of an Application Engine program, the names of the program and the section are listed in the title bar.



SQL Editor Window with Application Engine Program SQL

The Editor window consists of the main *edit pane*.

For SQL definitions and SQL used with records, there is also a drop-down *database list* at the top left. For SQL definitions, you can also use a drop-down *effective date list* at the top right.



When you make a selection from either drop-down list, your selected entry has a yellow background, indicating that you must click in the edit pane before you can start typing.

Accessing the SQL Editor

Use the SQL editor to create and edit SQL for the following components:

- SQL definitions
- Dynamic view or SQL view records
- Application Engine programs

You access the SQL editor differently for each type of component.

SQL Definitions

A SQL definition contains SQL statements, which can be entire SQL programs or just fragments that you want to re-use. You can access a SQL definition using Application Designer. You can add SQL definitions to a project. In addition, they're upgradable.



For more information see Using Application Designer.

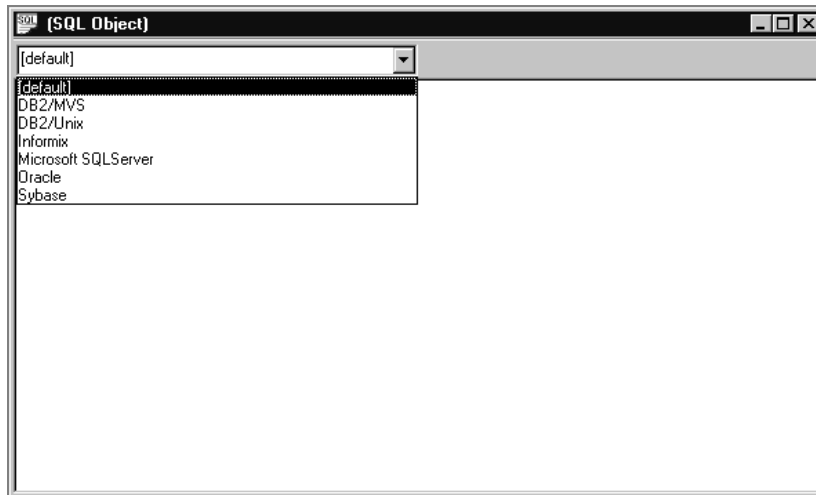
You can create, change or delete SQL definitions using Application Designer. You can also do the same things programmatically with the SQL class in PeopleCode.



For more information see SQL Class.

To create a SQL definition:

1. From Application Designer, select **File, New, SQL**.
2. Specify the database type you want associated with your SQL definition.



SQL Definition Database Types

You can associate more than one database type with a single SQL definition. In your PeopleCode, you can specify the appropriate database type for your program. However, at least one of your SQL statements must be of type **Default**.

3. Specify an effective date (optional)

SQL definitions can be effective dated. If you'd like to specify an effective date with your SQL definition, go to the **Advanced** tab of the SQL definition object property. Access the object properties by doing one of the following:

- Go to File, Object Properties.
- Select the SQL definition, right-click, then select **Object Properties**.
- Press Alt + Enter.
- Click the Advanced tab, then click **Show Effective Date**. When you click **OK**, the SQL definition shows a date in the right-hand drop-down menu.



SQL definition with effective date

4. Add you SQL code

You don't need to format your code. The SQL editor formats it when you save your SQL definition.

Dynamic View or SQL View Records

When you create a SQL View or Dynamic View record definition, you must enter a SQL View Select Statement to indicate what field values you want to join from which tables. You do this in the SQL editor.

To access the SQL editor with records:

1. Open or create the dynamic view or SQL view record definition.
2. Click on the **Record Type** tab.

ACCOMPLISHMT_VW (Record)

Record Fields Record Type

Record Type

- ☐ SQL Table
- ☒ SQL View
- ☐ Dynamic View
- ☐ Derived/Work
- ☐ SubRecord
- ☐ Query View
- ☐ Temporary Table

Non-Standard SQL

Table Name:

Build Sequence No:

[Click to open SQL Editor](#)

Record Type Tab of SQL View Record Definition

3. Click on the Click to Open SQL Editor button.

ACCOMPLISHMT_VW.2 (SQL Object)

[default]

```
SELECT A.emplid
,A.accomplishment
,A.org
,A.descr
,A.dt_issued
,A.yr_acquired
,B.accomp_category
,A.score
,A.passed
,A.LICENSE_NBR
,A.Issued_by
,A.State
,A.country
,A.expiratn_dt
,A.License_verified
,A.renewal
,A.native_language
,A.Translator
,A.speak_proficiency
,A.read_proficiency
```

SQL Editor for SQL View Record Definition

You can select a database type, but not an effective date, from the SQL editor for dynamic view and SQL view record definitions.

Application Engine Programs

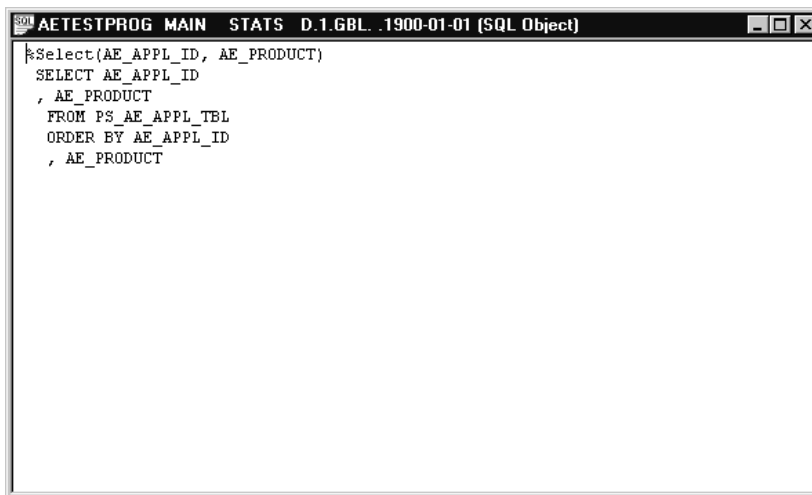
You can access the SQL editor from the following action types:

- Do Select
- Do Until
- Do When

- Do While
- SQL

To access the SQL editor in an Application Engine program:

1. Open the Application Engine program.
2. Select the action.
3. Either right-click and select **View SQL**, or select **View, SQL**.



Application Engine SQL Editor window

Select the database type and effective date for this SQL in the section, not in the SQL editor.



For more information see Application Engine.

Using the SQL Editor

The SQL Editor works much like any other text editor. You can use the same functions with it as with the PeopleCode editor: cut, paste, find, replace, and so on.



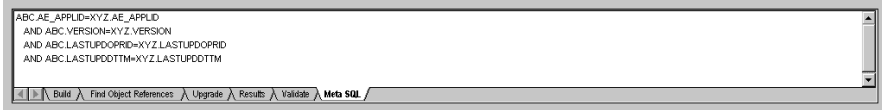
For more information see Editing Functions.

When you right click in an open SQL editor window, the pop-up menu lists all the available functions for the SQL editor:

Format Display	
Validate Syntax	
Resolve Meta SQL	
Delete Statement	F8
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Undo	Ctrl+Z
Find...	Ctrl+F
Replace...	Ctrl+H
Object Properties...	

SQL Editor pop-up menu

The following functions are available for the SQL editor, but are **not** available for the PeopleCode editor.

<i>Function</i>	<i>Description</i>
Format Display	You do not need to format your SQL statements; you only need to use the correct syntax. When you save or validate, the system formats the code according to the rules in the PeopleCode tables—no matter how you entered it originally. It automatically converts field names to uppercase and indents statements for you. Your SQL then looks consistent with other programs in the system.
Resolve Meta-SQL	<p>If there is meta-SQL in the SQL, select Resolve Meta-SQL to expand the meta-SQL statement in the Output Window, under the Meta-SQL tab.</p> <p>For example, the following code expands as follows:</p> <pre>%Join(COMMON_FIELDS, PSAEAPPLDEFN ABC, PSAESECTDEFN XYZ</pre>  <p>Meta-SQL Expanded in Output window</p>
Delete Statement	You can delete stand alone SQL statements. This menu item isn't enabled with SQL statements that have a database type of Default with no effective date, or a database type of Default and an effective date of 01/01/1900.

CHAPTER 5

PeopleCode Language

This section covers the syntax and fundamental elements of the PeopleCode language. It assumes that you have some familiarity with a structured programming language, such as C or Visual Basic.

In its fundamentals, PeopleCode syntax resembles that of other structured programming languages. Some aspects of the PeopleCode language, however, are specifically related to the PeopleTools environment. Definition name references, for example, provide a way of referring to PeopleTools definitions, such as record definitions or pages, without using hard-coded string literals. Other language features, such as the PeopleCode data types and metastrings, reflect the close interaction of PeopleTools and SQL.

Data Types

The conventional data types available in previous releases are the core of PeopleCode's functionality. The object-based data types are used to instantiate objects from the PeopleTools classes. The appropriate use of each data type is demonstrated where the documentation discusses PeopleCode that uses that data type.

PeopleSoft recommends that you declare your variables before you use them. For example, if you declare a variable of one data type, and assign it a value of a different type, the PeopleCode syntax checker catches that assignment as a design time error when you save your PeopleCode program. With an undeclared variable, the assignment error doesn't appear until runtime.

The following example produces a design time error when you try to save your program:

```
Local Field &DATE;  
  
&DATE = GetRecord(RECORD.DERIVED_HR);
```

Conventional Data Types

- ANY

When variables and function return values are declared as ANY, the data type is indeterminate, allowing PeopleTools to determine the appropriate type of value based on context. Undeclared local variables are ANY by default.

- BOOLEAN
- DATE
- DATETIME
- NUMBER
- OBJECT
- STRING
- TIME

Object-Based Data Types

For most classes in PeopleTools, you need a corresponding object-based data type to instantiate objects from that class.



For more information, see Understanding Objects and Classes in PeopleCode.

Data Buffer Access Types

- Field
- Record
- Row
- Rowset

Page Display Types

- Grid
- GridColumn
- Page

Internet Script Types

- Cookie
- Request
- Response

Miscellaneous Object-Based Types

- ASection
- Array
- File
- Interlink
- BIDocs



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.

- JavaObject



JavaObject objects can only be declared as type Local.

- Message
- ProcessRequest
- SQL

API Object Type

- ApiObject

Use this data type for any API object, such as a session object, a tree object, a Component Interface, a PortalRegistry, and so on.

The following ApiObject data type objects can be declared as Global:

- Session
- PSMessages collection
- PSMessage
- All Tree classes (trees, tree structures, nodes, levels, and so on.)

All other ApiObject data type objects *must* be declared as Local.

Comments and Statements

At the most general level, a PeopleCode program consists of comments and statements. Comments can be used to document your code. Control statements control the flow of the program's execution.

Comments

There are two ways to insert comments into your PeopleCode. You can surround comments with `/*` at the beginning and `*/` at the end. You can also use a **REM** (remark) statement for commenting. Put a semicolon at the end of a **REM** comment. If you don't, everything up to the end of the next statement will be treated as part of the comment.

```
REM This is an example of commenting PeopleCode;

/* ----- Logic for Compensation Change ----- */

/* Recalculate compensation change for next row. Next row is based on prior
value of EFFDT. */

calc_next_compchg(&OLDDT, EFFSEQ, 0);

/* Recalculate compensation change for current row and next row. Next row is
based on new value of EFFDT. */

calc_comp_change(EFFDT, EFFSEQ, COMP_FREQUENCY, COMPRATE, CHANGE_AMT,
CHANGE_PCT);

calc_next_compchg(EFFDT, EFFSEQ, 0);
```

Use comments to explain, preferably in language comprehensible to anyone reading your program, what your code does. Comments also allow you to differentiate between PeopleCode delivered with the product and PeopleCode that you add or change. This differentiation will help in your analysis for debugging, as well as upgrades.



We strongly suggest that you use comments to place a *unique identifier* marking any changes or enhancements that you have made to a PeopleSoft application. This will make it possible for you to search for all the changes you have made. This is particularly helpful when you are upgrading a database.

Statements

A statement can be a declaration, an assignment, a program construct (such as a **Break** statement or a conditional loop), or a subroutine call.

Separators

PeopleCode statements are generally terminated with a semicolon. The PeopleCode language accepts semicolons even if they're not required, such as after the last statement executed within an **If** statement. This allows you to consistently add semicolons after each statement.

Extra spaces and line breaks are ignored—and in fact are removed by the PeopleCode Editor when you save your code.

Assignment Statements

The assignment statement is the most basic type of statement in PeopleCode. It consists of an equal sign with a variable name on the left, and an expression on the right:

```
varname = expression;
```

The expression on the right is evaluated, and the result is placed in the variable named on the left. Depending on the data types involved, the assignment is passed either *by value* or *by reference*.

Passing by Value

In most types of assignment, the result of the right-hand expression is assigned to the variable as a newly created value, in its own allocated memory area. Subsequent changes to the value of that variable have no effect on any other data.

Passing by Reference

When both sides of an assignment statement are object variables, the result of the assignment is not a copy of the object. You're only making a copy of the reference. The variable names don't refer to separate and distinct objects; they both refer to the same object.

For example, Both &AN and &AN2 are arrays of number. Assigning &AN2 to &AN does **not** make &AN2 a distinct array. They still both point to the same information.

```
Local array of number &AN, &AN2;

Local number &NUM;

&AN = CreateArray(100, 200, 300);

&AN2 = &AN;

&NUM = &AN[1];
```

The assignment does not allocate any memory or copy any part of the original object. It simply makes &AN2 refer to the same array to which &AN refers. Any changes you make to the value of either variable will also affect the other. On the other hand, assigning &NUM to the first element in &AN (100), is **not** an object assignment. It's in effect passing by value.



In PeopleCode the equal sign can function as either an assignment operator or a comparison operator, depending on context.

Language Constructs

PeopleCode language constructs include:

- Branching structures: **If** and **Evaluate**.
- Loops and conditional loops: **For**, **Repeat**, and **While**.
- **Break** and **Exit** statements for escaping loops and terminating program execution.
- The **Return** statement for returning from functions.
- Variable and function declaration statements: **Global**, **Local**, and **Declare Function**.
- The **Function** statement for defining functions.



For more information see Control Statements.

Functions as Subroutines

PeopleCode, like C, doesn't have subroutines as such. PeopleCode subroutines are simply the subset of PeopleCode functions that are defined to return no value or to return a value optionally. Calling a subroutine is the same as calling a function with no return value:

```
function_name([param_list]);
```



For more information on function calls, see the section Functions. For information about declaring and defining functions see the built-in functions Function and Declare Function.

Control Statements

The following sections describe the following control statements, which control the flow of execution in a PeopleCode program.

Branching Statements

Branching statements control the flow of execution based on their evaluation of conditional expressions.

If...Then...Else

The syntax of the **If..Then...Else** statement is:

```

If condition Then

    [statement_list_1]

[Else

    [statement_list_2]]

End-if;

```

It evaluates the Boolean expression *condition*. If *condition* is TRUE, the **If** statement executes the statements in *statement_list_1*. If *condition* is FALSE, then it executes the statements in the **Else** clause; or, if there is no **Else** clause, it does nothing.



For more information see If.

Evaluate

The **Evaluate** statement is used in cases where you want to check multiple conditions. Its syntax is:

```

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;

```

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). For example, the following **Evaluate** statement executes only the first **When** clause. &USE_FREQUENCY in this example can only have one of three string values:

```

evaluate &USE_FREQUENCY
when = "never"
    PROD_USE_FREQ = 0;

```

```

when = "sometimes"
    PROD_USE_FREQ = 1;
when = "frequently"
    PROD_USE_FREQ = 2;
when-other
    Error "Unexpected value assigned to &USE_FREQUENCY."
end-evaluate;

```

To end the **Evaluate** after the execution of a **When** clause, you can add a **Break** statement at the end of the clause:

```

evaluate &USE_FREQUENCY
when = "never"
    PROD_USE_FREQ = 0;
    Break;
when = "sometimes"
    PROD_USE_FREQ = 1;
    Break;
when = "frequently"
    PROD_USE_FREQ = 2;
    Break;
when-other
    Error "Unexpected value assigned to &USE_FREQUENCY."
end-evaluate;

```

In some rare cases you may want to make it possible for more than one of the **When** clauses to execute:

```

evaluate &PURCHASE_AMT
when >= 100000
    BASE_DISCOUNT = "Y";
when >= 250000
    SPECIAL_SERVICES = "Y";
when >= 1000000
    MUST_GROVEL = "Y";
end-evaluate;

```



For more information see Evaluate.

For Loops

The **For** statement repeats a sequence of statements a specified number of times. Its syntax is:

```

For count = expression1 to expression2
    [Step i];
    statement_list
End-for;

```

The **For** statement initializes the value of *count* to *expression1*, then increments *count* by *i* each time after it executes the statements in *statement_list*. It continues until *count* is equal to

expression2. If the **Step** clause is omitted, then *i* equals one. If you want to count backwards from a higher value to a lower value, then use a negative value for *i*. You can exit a **For** loop using a **Break** statement.

The following example demonstrates the **For** statement:

```
&MAX = 10;
for &COUNT = 1 to &MAX;
    WinMessage("Executing statement list, count = " | &COUNT);
end-for;
```



For more information see For.

Conditional Loops

Conditional loops, **Repeat** and **While**, repeat a sequence of statements, evaluating a conditional expression each time through the loop. The loop terminates when the condition evaluates to True. You can exit from a conditional loop using a **Break** statement. If the **Break** statement is in a loop embedded in another loop, the break applies only to the inside loop.

Repeat

The syntax of the **Repeat** statement is:

```
Repeat
    statement_list
Until logical_expression;
```

The **Repeat** statement executes the statements in *statement_list* once, then evaluates *logical_expression*. If *logical_expression* is False the sequence of statements is repeated until *logical_expression* is True.



For more information, see Repeat.

While

The syntax of the **While** statement is:

```
While logical_expression
    statement_list
End-while;
```

The While statement evaluates *logical_expression* before executing the statements in *statement_list*. It continues to repeat the sequence of statements until *logical_expression* evaluates to False.



For more information, see [While](#).

Functions

PeopleCode supports the following types of functions:

Built-in	The standard set of PeopleCode functions described in PeopleCode Built-in Functions. These can be called without being declared.
Internal	Functions that are defined (using the Function statement) within the PeopleCode program in which they are called.
External PeopleCode	PeopleCode functions defined outside the calling program - generally in record definitions serving as function libraries.
External non-PeopleCode	Functions stored in external (C-callable) libraries.

In addition, PeopleCode also supports methods. The main difference between a built-in function and a method is:

- a built-in function, in your code, is on a line by itself, and doesn't (generally) have any dependencies. You don't have to instantiate an object before you can use the function.
- a method can only be executed from an object, using dot notation. You have to instantiate the object first.



For more information see [Function](#) and [Declare Function](#). For more information about methods, see [Understanding Objects and Classes in PeopleCode](#).

Defining Functions

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.

By convention, PeopleCode programs are stored in records whose names begin in `FUNCLIB_`, and they are always attached to the `FieldFormula` event (which is convenient because this event is no longer used for anything else).



You can't declare a variable within a function definition.



For more information see Function.

Declaring Functions

If you call an external function (that is, a function defined outside the program where it is called) from a PeopleCode program, you must declare the function at the top of the program. The syntax of the function declaration varies, depending on whether the external function is written in PeopleCode or compiled in a dynamic link library.



For more information see Declare Function.

Calling Functions

Functions are called with this syntax:

```
function_name([param_list])
```

The parameter list (*param_list*), is a list of expressions, separated by commas, that the function expects you to supply. Items in the parameter list can be optional, or required.

You can check the values of parameters that get passed to functions at runtime in the **Parameter** window of the PeopleCode debugger.



For more information see Debugging Your Application.

If the return value is required, then the function must be called as an expression, for example:

```
&RESULT = Product (&RAISE_PERCENT, .01, EMPL_SALARY);
```

If the function has an optional return value it can be called as a subroutine. If the function has no return value, it *must* be called as a subroutine:

```
WinMessage(64, "I can't do that, " | &OPER_NICKNAME | ".");
```

Parameters are always passed to internal and external *PeopleCode functions* by reference. If the function is supposed to be effecting a change in the data the caller passes, you must also pass in a variable.

Built-in function parameters can be passed by reference or by value, depending on the function. External C function parameters can be passed by value or by reference, depending on the declaration and type.



A built-in function is a function provided by PeopleCode, such as `RevalidatePassword`, `AddAttachment`, and so on. A PeopleCode function is one that you write yourself, or is written outside the PeopleSoft system.



For more information see [Passing Variables to Functions](#).



PeopleCode doesn't support true recursive functions. Though a function can call itself, the local variables used by the function keep their same values in all instances of the function, that is, local variables are scoped at the PeopleCode program level, **not** at the function level.

Function Return Values

Functions can return values of any supported data type; and some functions do not return any value.

Optional return values occur only in built-in functions—you can't define a function yourself that optionally returns a value. Optional return values are typical in functions that return a Boolean value indicating whether execution was successful. For example, the following call to **DeleteRow** ignores the Boolean return value and just deletes a row:

```
DeleteRow(RECORD.BUS_EXPENSE_PER, &L1_ROW, RECORD.BUS_EXPENSE_DTL, &L2_ROW);
```

While the following example checks the return value and returns a message saying whether it succeeded:

```
if DeleteRow(RECORD.BUS_EXPENSE_PER, &L1_ROW, RECORD.BUS_EXPENSE_DTL, &L2_ROW)
then
    WinMessage("Row deleted.");
else
    WinMessage("Sorry -- couldn't delete that row.");
end-if;
```

Expressions

Expressions evaluate to values of any of the PeopleCode data types. A simple PeopleSoft expression can consist of a constant, a temporary variable, a system variable, a record field reference, or a function call. Simple expressions can be modified by unary operators (such as a negative sign or logical NOT), or combined into compound expressions using binary operators (such as a plus sign or logical AND).

Definition name references evaluate to strings equal to the name of a PeopleTools definition, such as a record, page, or message. They provide a way of referring to definitions without using string literals, which are difficult to maintain.

Metastrings are special expressions used within SQL string literals. At runtime the metastrings expand into the appropriate SQL for the current database platform.

Constants

PeopleCode supports numeric, string, and Boolean constants.



You can express DATE, DATETIME, and TIME values by converting from STRING and NUMBER constants using the Date, Date3, DateTime6, DateTimeValue, DateValue, Time3, TimePart, and the TimeValue functions. You can also format a DATETIME value as text using FormatDateTime.

Numeric

Numeric constants can be any decimal number. Some examples are:

```
7
0.8725
-172.0036
```

String Constants

String constants can be delimited by using either single (') or double (") quote marks. If a quote mark occurs as part of a string, the string can be surrounded by the other delimiter type. As an alternative, you can include the delimiter twice. Some examples are:

```
"This is a string constant."
'So is this.'
'She said, "This is a string constant."'
"She said, ""This is a string constant."""
```

Boolean Constants

Boolean constants represent a truth value. The two possible values are:

```
TRUE
FALSE
```

Functions as Expressions

Any function that returns a value can be used as an expression. It can be used on the right side of an assignment statement, passed as a parameter to another function, or combined with other expressions to form a compound expression.



For more information, see Functions.

Variables

There are the following types of variables available in your program:

- System variables

System variables provide access to system information. System variables are prefixed with the '%' character, rather than the '&' character. Use these variables wherever you can use a constant, passing them as parameters to functions or assigning their values to fields or to temporary variables.



For more information, see System Variables.

- User-defined variables

These variable names are preceded by an "&" character wherever they appear in a program. Variable names can be 1 to 17 characters, consisting of letters A-Z and a-z, digits 0-9, and characters #, @, \$, and _.

User-Defined Variable Declaration and Scope

The difference between the type of variables has to do with their life spans:

Global	Valid for the entire session
Component	Valid while any page in the component in which it's defined stays active
Local	valid for the life of the PeopleCode program in which it's defined

You can declare variables using the **Global**, **Local** or **Component** statement, or you can use local variables without declaring them. Here are some examples:

```
Local Number &AGE;
```



```
Global String &OPER_NICKNAME;

Component Rowset &MY_ROWSET;

Local Any &SOME_FIELD;

Local ApiObject &MYTREE;
```

Variable declarations must be placed above the main body of a PeopleCode program (along with function declarations and definitions). Variables can be declared as any of the PeopleCode data types. If a variable is declared as an ANY data type, or if a variable is not declared, then PeopleTools will choose an appropriate data type based on context.



It's good practice to declare a variable as an explicit data type unless the variable is going to hold a value of an unknown data type. If you declare a variable of one data type, then assign to it a value of a different type, the PeopleCode editor will catch that assignment as a design time error when you try to save the program. With an undeclared variable, the assignment error won't appear until runtime.

Global variables remain defined and keep their values throughout a PeopleSoft session and can be accessed from different components and applications, including an Application Engine program. A Global variable must be declared, however, in each PeopleCode program where it's used. We recommend that you use Global variables rarely, because they are difficult to maintain.

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.

Component variables provide an intermediate ground between Global and Local scope. They remain defined and keep their values while any page in the component in which they're defined stays active. Like a Global variable, a Component variable must be declared in each PeopleCode program where it's used.

Component variables act the same as Global variables when an Application Engine program is called from a page (using **CallAppEngine**.)

Component variables remain defined after a **TransferPage**, **DoModal** or **DoModalComponent** function. However, variables declared as Component do **not** remain defined after using the **Transfer** function, whether you're transferring within the same component or not.

Local variables remain in scope for the life of a PeopleCode program.



You can't declare a variable within a function definition.

You can check the values of your Local, Global and Component variables at runtime in the different variable windows of the PeopleCode debugger.



For more information see [Debugging Your Application](#).

Restrictions on Use

The following data types can only be declared as Local. They can **not** be declared as either Global or Component:

- JavaObject
- Interlink



Interlink objects can only be declared as type Global in an Application Engine program.

The following ApiObject data type objects can be declared as Global:

- Session
- PSMessages collection
- PSMessage
- All Tree classes (trees, tree structures, nodes, levels, and so on.)

All other ApiObject data type objects, such as all Query objects, Component Interface objects, and so on, *must* be declared as Local.

Using User-Defined Variables

You should initialize user-defined variables by setting them equal to a constant or a record field before you use them. If you do not initialize them, strings are initialized as null strings, dates and times as nulls, and numbers as zero.

A user-defined variable can hold the contents of a record field for program code clarity. For example, you may give a variable a more descriptive name than a record field, based on the context of the program. If the record field is from another record, you may assign it to a temporary variable rather than always using the record field reference. This makes it easier to type the program, and can also make it easier to read.

Also, if you find yourself calling the same function repeatedly to get a value, you may be able to avoid some processing by calling the function once and placing the result in a variable.



PeopleCode doesn't support true recursive functions. Though a function can call itself, the local variables used by the function keep their same values in all instances of the function, that is, local variables are scoped at the PeopleCode program level, **not** at the function level.

Passing Variables to Functions

PeopleCode variables are always passed to functions by reference. This means, among other things, that the function can change the value of a variable passed to it so that the variable will have the new value on return to the calling routine. For example, the **Amortize** built-in function expects you to pass it variables into which it will place the amount of a loan payment applied towards interest (&PYMNT_INTRST), the amount of the payment applied towards principle (&PYMNT_PRIN), and the remaining balance (&BAL). It calculates these values based on information that the calling routine supplies in other parameters:

```
&INTRST_RT=12;

&PRSNT_BAL=100;

&PYMNT_AMNT=50;

&PYMNT_NBR=1;

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

System Variables

System variables are preceded by a percent (%) symbol whenever they appear in a program. You can use these variables to get the current date and time, information about the user, the current language, the current record, page, or component, and more.



For more information see System Variables.

Metastrings

Metastrings are special SQL expressions. The metastrings, also called meta-SQL, are prefixed with a percent (%) symbol, and can be included directly in the string literals. They expand at runtime into an appropriate substring for the current database platform. Meta-SQL 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 object methods (Select, SelectNew, Fill, etc.)
- with SQL objects

- in Application Engine
- with some record object methods (Insert, Update, etc.)
- with COBOL



For more information see Meta-SQL.

Record Field References

Record field references are used to retrieve the value stored in a record field, or to assign a value to a record field.

Record Field Reference Syntax

References to record fields have the following form:

[recordname.]fieldname

You need to supply the **recordname** only if the record field and your PeopleCode program are in different record definitions.

For example, suppose that in a database for veterinarians you have two records, PET_OWNER and PET. A program in the record definition PET_OWNER must refer to the PET_BREED record field in the PET record definition as:

PET.PET_BREED

However, a program in the PET record definition can refer to this same record field on a more intimate basis as simply:

PET_BREED

And, if the program is in the PET_BREED record field itself, it can refer to this record field using the caret (^) symbol:

^

The PeopleCode Editor replaces the caret symbol with the actual record field name.

You can also use object-based dot notation to refer to record fields, for example:

```
&FIELD = GetRecord(RECORD.PET_OWNER).GetField(FIELD.PET_BREED);
```



For more information about referencing record fields see Referencing Data in the Component Buffer.

Legal Record Field Names

A record field name consists of two parts, the record name and the field name, separated by a period.

The field names used in PeopleCode are consistent with the field names allowed in the field definition. Case is ignored, although the PeopleCode Editor will, for the sake of convention, automatically format field names in ALL CAPS. A field name can be 1 to 18 characters, consisting of alphanumeric characters determined by your current language setting in Windows, and characters #, @, \$, and _.

A record name can be 1 to 15 characters, consisting of alphanumeric letters determined by your current language setting in Windows, and characters #, @, \$, and _.

Definition Name References

Definition name references are special expressions that reference the name of a PeopleTools definition, such as a record, page, component, Business Interlink, and so on. Syntactically, a definition name reference consists of a reserved word indicating the type of definition, followed by a period, then the name of the PeopleTools definition. For example, the following definition name reference:

```
RECORD.BUS_EXPENSE_PER
```

refers to the definition name:

```
"BUS_EXPENSE_PER"
```

Generally, definition name references are passed as parameters to functions. If you attempt to pass a string literal instead of a definition name reference to such a function, you will get a syntax error.

You also use definition name references outside function parameter lists, for example in comparisons:

```
If (%Page = PAGE.SOMEPAGE) Then

    /* do stuff specific to SOMEPAGE */

End-If;
```

In these cases the definition name reference evaluates to a string literal. Using the definition name reference instead of a string literal makes it possible for PeopleTools to maintain the code if the definition name changes.

If you use the definition name reference, and the name of the definition changes, the change automatically "ripples" through the code, so you don't have to change it or maintain it.

In the PeopleCode editor, if you place your cursor over any definition name reference and right-click, you can select View Definition from the pop-up menu to open the definition.

Legal and Illegal Definition Names

Legal definition names, as far as definition name references are concerned, consist of alphanumeric letters determined by your current language setting in Windows, and the characters #, @, \$, and _.

In some cases, however, the definition supports the use of other characters. You can, for example, have a menu item named "A&M" stored in the menu definition even though "&" is an illegal character in the definition name reference. The illegal character will result in an error when you validate your syntax or attempt to save your PeopleCode.

You can get around this problem in two ways:

- Rename the definition so that it uses only legal characters.
- Enclose the name of the definition in quotes in the reference, for example:

```
ITEMNAME . "A&M"
```

The second solution is a commonly used workaround in cases where the definition name contains illegal characters. If you use this notation, the definition name reference is not treated as a string literal: PeopleTools maintains the reference the same way as it does other definition name references.



If your definition name begins with a number, you must enclose the name in quotation marks when you use it in a definition name reference. For example, `CompIntfc."1_DISCIPLIN_ACTN"`.

Reserved Word Summary Table

The following table summarizes the reserved words used in definition name references.

<i>Reserved Word</i>	<i>Common Usage</i>
BARNAME	Used with transfers and modal transfers.
BUSACTIVITY	Used with TriggerBusinessEvent.
BUSEVENT	Used with TriggerBusinessEvent.
BUSPROCESS	Used with TriggerBusinessEvent.
COMPINTFC	Used with the GetCompIntfc Session class method.
COMPONENT	Used with transfers and modal transfers.
FIELD	Used with the GetField Record class method.
FILELAYOUT	Used with the SetFileLayout File class method.
HTML	Used with the GetHTMLText function.
IMAGE	Used with the Add Image object method and the

<i>Reserved Word</i>	<i>Common Usage</i>
	ImageReference field.
INTERLINK	Used with the GetInterlink function.
ITEMNAME	Used with transfers and modal transfers.
MENUNAME	Used with transfers and modal transfers.
MESSAGE	Used with Application Messaging functions and methods.
PAGE	Used with transfers and modal transfers to pass the page item name (instead of the page name), and with grid and ActiveX controls to pass the page name.
RECORD	Commonly used in component buffer functions and methods to designate a record.
SCROLL	The name of the scroll area in the page. This name is always equal to the primary record of the scroll.
SQL	Used with the SQL Class.
STYLESHEET	Used with style sheets.

Operators

PeopleCode expressions can be modified and combined using math, string, comparison, and Boolean operators.

Math Operators

PeopleCode uses standard mathematical operators:

- + add
- subtract (or unary negative sign)
- * multiply
- / divide
- ** exponentiation

Exponentiation occurs before multiplication and division; multiplication and division occur before addition and subtraction. Math expressions otherwise are evaluated from left to right. You can use parentheses to force the order of operator precedence.

The minus sign can also, of course, be used as a negation operator, as in the following expressions:

```
-10
- &NUM
- Product (&PERCENT_CUT, .01, SALARY)
```

Operations on Dates and Times

You can add or subtract two date values or two time values, which gives you an NUMBER result. In the case of dates, the number represents the difference between the two dates in days. In the case of time, the number represents the difference in seconds. You can also add and subtract numbers to or from a time or date, which results in another date or time. Again, in the case of days the number represents days, and in the case of time the number represents seconds.

The following table summarizes these operations:

Operation	Result Type	Number Represents...
time + number	time	number of seconds
date + number	date	number of days
date - date	number	difference in days
time - time	number	difference in seconds
date + time	datetime	

String Concatenation

The string concatenation operator (||) is used to combine strings. For example, assuming &OPER_NICKNAME is "Dave", the following statement sets &RETORT to "I can't do that, Dave."

```
&RETORT = "I can't do that," || &OPER_NICKNAME || " ."
```

The concatenation operator automatically converts its operands to strings. This makes it easy to write statements that display mixed data types. For example:

```
&DAYS_LEFT = &CHRISTMAS - %Date;
WinMessage("Today is " || %Date || " . Only " || &DAYS_LEFT || " shopping days left!");
```

@ Operator

The @ operator converts a string storing a definition reference into the definition. This is useful, for example, if you want to store definition references in the database as strings and retrieve them

for use in PeopleCode; or if you want to get a definition reference in the form of a string from the operator using the **Prompt** function.

To take a simple example, if the record field EMPLID is currently equal to "8001", the following expression would evaluate to "8001":

```
@ "EMPLID"
```

The following example uses the @ operator to convert strings storing a record reference and a record field reference:

```
&STR1 = "RECORD.BUS_EXPENSE_PER";

&STR2 = "BUS_EXPENSE_DTL.EMPLID";

&STR3 = FetchValue(@(&STR1), CurrentLineNumber(1), @(&STR2), 1);

WinMessage(&STR3, 64);
```



String literals that reference definitions are not maintained by PeopleTools. Bear in mind that if you store definition references as strings, then convert them with the @ operator in your code, this will create maintenance problems whenever the definition names change.

The following function takes a rowset and a record, passed in from another program, and does some processing. The **GetRecord** method won't take a variable for the record, however, using the @ symbol you can dereference the record name. Because the record name is never hard-coded as a string, if the record name changes, this code will not have to change.

```
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;
```

Comparison Operators

Comparison operators are used to compare two expressions of the same data type. The result of the comparison is a Boolean value. The following table summarizes these operators:

<i>Operator</i>	<i>Meaning</i>
=	equal
!=	not equal
<>	not equal
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

You can precede any of the comparison operators with **Not**, for example:

- Not =
- Not <
- Not >=

Expressions formed with comparison operators form *logical terms* that can be combined using Boolean operators.

String comparisons are case-sensitive. You can use the Upper or Lower built-in functions to do a case-insensitive comparison.

Boolean Operators

The logical operators AND, OR, and NOT are used to combine Boolean expressions. The following table shows the results of combining two Boolean expressions with AND and OR operators:

<i>Exp1</i>	<i>Operator</i>	<i>Exp2</i>	<i>Result</i>
FALSE	AND	FALSE	FALSE
FALSE	AND	TRUE	FALSE
TRUE	AND	TRUE	TRUE
FALSE	OR	FALSE	FALSE
FALSE	OR	TRUE	TRUE
TRUE	OR	TRUE	TRUE

The NOT operator negates Boolean expressions, changing a TRUE value to FALSE and a FALSE value to TRUE.

In complex logical expressions using the operations AND, OR, and NOT, NOT takes the highest precedence, AND is next, and OR is lowest. Parentheses can be used to override precedence. (It's generally a good idea to use parentheses in logical expressions anyway, because it makes them easier to decipher.) If used on the right side of an assignment statement, Boolean expressions must be enclosed in parentheses.

The following are examples of statements containing Boolean expressions:

```
&FLAG = (Not (&FLAG)); /* toggles a Boolean */

if ((&HAS_FLEAS or &HAS_TICKS) and
    SOAP_QTY <= MIN_SOAP_QTY) then
    SOAP_QTY = SOAP_QTY + OrderFleaSoap(SOAP_ORDER_QTY);
end-if;
```


CHAPTER 6

Understanding Objects and Classes in PeopleCode

For this release, PeopleSoft introduces classes of objects that you can manipulate with PeopleCode. These objects may or may not have a GUI equivalent; some are only representations of data structures that occur at runtime. With PeopleCode you can manipulate data in the data buffer easily and consistently. These object classes enable you to write code that's more readable, more easily maintained, and more useful. Coding with objects is simply easier.

What is a Class?

A **class** is the formal definition of an object and acts as a template from which an instance of an object is created at runtime. The class defines the properties of the object and the methods used to control the object's behavior.

PeopleSoft delivers pre-defined classes (such as Array, File, Field, and SQL.) You cannot create custom classes.

What is an Object?

An **object** represents a unique instance of a data structure defined by the template provided by its class. Each object has its own values for the variables belonging to its class and responds to methods defined by that class.

The class is a template, or blueprint, from which you create an object. Once an object has been created (instantiated) from a class, you can change its properties. A **property** is an attribute of an object. Properties define object characteristics, such as name or value, or the state of an object, such as deleted or changed. Some properties are read-only and cannot be set, such as Name or Author. Other properties are read-write and can be set, such as Value or Label.

Objects are different from other data structures. They contain code (in the form of **methods**), not just static data. A **method** is a procedure or routine, associated with one or more classes, that acts on an object.

An analogy to illustrate the difference between an object and its class is the difference between a car and the blue Renault Citroen with license plate number TS5800B. A class is a general category, while the object is a very specific instance of that class. Each car comes with standard characteristics, such as four wheels, an engine or brakes, that define the class and are the template

from which the individual car is created. You can change the properties of an individual car by personalizing it with bumper stickers or racing stripes, which could be likened to changing the Name or Visible property of an object. The model and date it's created are like read-only properties because you can't alter them. A tune-up acts on the individual car and changes its behavior, much like a method acts on an object.

Instantiating Objects

A class is the blueprint for something, like a bicycle, a car, or a data structure. An object is the actual thing that's built using that class (or blueprint.) From the blueprint for a bicycle, you can build a specific mountain bike with 23 gears and tight suspension. From the blueprint of a data structure class, you build a specific instance of that class. *Instantiation* is the term for building that copy, or an instance, of a class.

Working with Objects

Generally you instantiate an object (create them from their class) using built-in functions or methods of other objects. Some objects are instantiated from data already existing in the data buffer. Think about this kind of object instantiation as taking a chunk of data from the buffer, encapsulating it in code (methods and properties), manipulating it, then freeing the references. Some objects can be instantiated from a previously created definition, such as a page or file layout definition.

The following example creates a field object:

```
Local field &MyField  
  
&MyField = GetField();
```

Getxxx built-in functions generally provide access to data that already exists, whether in the data buffers or from an existing definition.

Createxxx functions generally create defined objects that do not yet exist in the data buffer. *Createxxx* functions only create a buffer structure. They don't populate it with data. For example, the following function returns a record object for a record that already exists in the component buffer:

```
&REC = GetRecord();
```

The following example creates a stand-alone record. However, there is no data in &REC2. The specified record definition must be created previously, but the record doesn't have to exist in the buffer:

```
&REC2 = CreateRecord(EMP_CHKLIST_ITM);
```

The exceptions to the general Get/Create rule are objects that have no built-in functions and can only be instantiated from a session object (such as Tree classes, Component Interfaces, and so on). For *most* of these classes, when you use **Getxxx**, all you get is an *identifier* for the object. To fully instantiate the object, you must use an **Open** method.



For more information see Search Classes.

Object Properties

To set or get characteristics of an object, or to determine the state of an object, you must access its properties through **dot notation** syntax. Follow the reference to the object with a period, followed by the property, and assign it a value. The format is generally as follows:

```
Object.Property = Value
```

The following example hides the field &MYFIELD:

```
&MYFIELD.Visible = False
```

You can return information about an object by returning the value of one of its properties. In the following example, &X is a variable that is assigned the value found in the field &MYFIELD:

```
&X = &MYFIELD.Value
```

In the following example, a property is used as the test for a condition:

```
If &ROWSET.ActiveRowCount <> &I Then
```

Object Methods

You can also use dot notation to execute methods. Follow the reference to the object with a period, followed by the method name and any parameter the method may take. The format is generally as follows:

```
Object.method();
```

You can string methods and property values together into one statement. The following example strings together the GetField() method with the Name property:

```
If &REC_BASE.GetField(&R).Name = &REC_RELLANG.GetField(&J).Name Then
```

Some methods return a Boolean value: True if the method execute successfully; False if it doesn't. The following method compares all like-named fields of the current record object with the specified record. This method returns as True if all like-named fields have the same value:

```
If &MYRECORD.CompareFields(&OTHERRECORD) Then
```

Other methods return a reference to an object. The **GetCurrEffRow** method returns a row object:

```
&MYROW = &MYROWSET.GetCurrEffRow();
```

Some methods don't return anything. Each method's documentation tells you what it returns.

Many objects have default methods. Instead of typing the name of the method explicitly, you can use that method's parameters. Objects with default methods are composite objects. They contain additional objects within them. The default method is generally the method used to get the lower-level object.

A good example of a composite object is a record object. Record definitions are comprised of field definitions. The default method for a record object is **GetField**.

The following lines of code are equivalent:

```
&FIELD = &RECORD.GetField(FIELD.EMPLID);

&FIELD = &RECORD.EMPLID;
```



If the field you're accessing has the same name as a record property (such as, Name) you can't use the default method for accessing the field. You must use the **GetField** method.

Another example of default methods concerns rowsets and rows. Rowsets are made up of rows, so the default method for a rowset is **GetRow**. The two specified lines of code are equivalent: They both get the fifth row of the rowset:

```
&ROWSET = GetRowSet();

/*the next two lines of code are equivalent */

&ROW = &ROWSET.GetRow(5);

&ROW = &ROWSET(5);
```

The following example illustrates the long way of enabling the NAME field on a second level scroll (the code is executing on the first level scroll):

```
GetRowset(Scroll.EMPLOYEE_CHECKLIST).GetRow(1).GetRecord(EMPL_CHKLIST_ITM).GetField(FIELD.NAME).Enabled = True;
```

Using default methods enables you to shorten the above code to the following:

```
GetRowset(Scroll.EMPLOYEE_CHECKLIST)(1).EMPL_CHKLIST_ITM.NAME.Enabled = True;
```

Expressions of the form *class.name.property* or *class.name.method(..)* are converted to a corresponding object. For example:

```
&temp = RECORD.JOB.IsChanged;
```

is evaluated as if it were

```
&temp = GetRecord(RECORD.JOB).IsChanged;
```

Furthermore,

```
JOB.EMPLID.Visible = False;
```


is evaluated as if it were

```
GetField(JOB.EMPLID).Visible = False;
```

Object Assignment

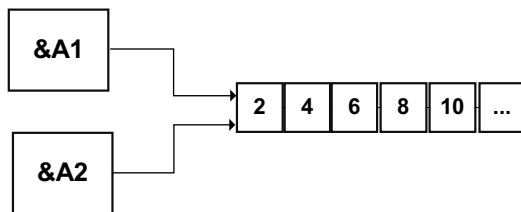
When you assign one object to another, you are not creating a copy of the object, but are only making a copy of the reference.

In the following example, &A1 and &A2 do not refer to separate and distinct objects, but refer to the same object. The assignment of &A1 to &A2 does not allocate any database memory or copy any part of the original object. It simply makes &A2 refer to the same object to which &A1 refers.

```
Local Array of Number &A1, &A2;
```

```
&A1 = CreateArray(2, 4, 6, 8, 10);
```

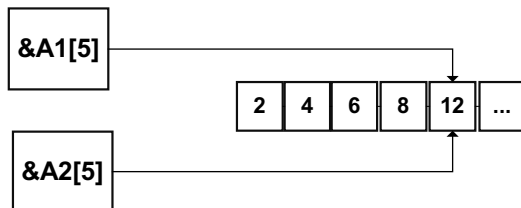
```
&A2 = &A1;
```



If the next statement is

```
&A2[5] = 12;
```

then &A1[5] also equals 12.



The following example is **not** considered an object assignment:

```
Local number &NUM;
```

```
Local Array of Number &A1;
```

```
&A1 = CreateArray(2, 4, 6, 8, 10);
```

```
&NUM = &A1[3];
```

&NUM is of data type Number, which isn't an object type. If you later change the value of &NUM in your program, you won't change the element in the array.

Passing Objects

All PeopleCode objects can be passed as function parameters. The application developer can pass complex data structures between PeopleCode functions (as opposed to passing long lists of fields). If a function is passed an object, the function will work on the actual object, not on a copy of the object.



For more information, see Object Assignment.

In the following simple example, a reference to the visible property is passed, not the value of visible. This allows the MyPeopleCodeFunction either to get or set the value of visible:

```
MyPeopleCodeFunction(&MyField.Visible);
```

In the following example, the function Process_Rowset loops through every row and record in the rowset it is passed and executes an UPDATE statement on each record in the rowset. This function can be called from any PeopleCode program and can process any rowset that is passed to it.

```
Local Rowset &RS;

Local Record &REC;

Function Process_RowSet(&ROWSET as Rowset);

    For &I = 1 To &ROWSET.Rowcount

        For &J = 1 To &ROWSET.Recordcount

            &REC = &ROWSET.GetRow(&I).GetRecord(&J);

            &REC.Update();

        End-For;

    End-For;

End-Function;

&RS = GetLevel0();

Process_RowSet(&RS);
```

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 de-reference the 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;
```


CHAPTER 7

Using Methods and Built-in Functions

This chapter covers a number of issues related to the use of PeopleCode methods and built-in functions. It discusses common restrictions on the use of methods and functions in certain PeopleCode events. It examines groups of methods or functions that are related by common syntactic complexities (such as functions that access data in multiple-scroll pages). And it views specific methods or functions in the context of a complex task (such as implementing a dynamic tree control, an ActiveX control, or a remote call).



For reference information on individual built-in functions, see PeopleCode Built-in Functions. For more information on individual methods, see PeopleCode Classes.

Restrictions on Method and Function Use

This summarizes general restrictions on the use of PeopleCode built-in functions and methods.

Think-Time Functions

"Think-time" functions suspend processing either until the user has taken some action (such as clicking a button in a message box), or until an external process has run to completion (for example, a remote process).

Think-time functions should be avoided in any of the following PeopleCode events:

- SavePreChange
- Workflow
- RowSelect
- SavePostChange
- Any PeopleCode event that fires as a result of a ScrollSelect, ScrollSelectNew, RowScrollSelect, RowScrollSelectNew function call
- Any PeopleCode event that fires as a result of a Select or SelectNew rowset method

Violation of this rule can result in application failure.

The following are think-time functions in PeopleTools 7.5 and PeopleTools 8.0:

- Call to an external DLL
- DoCancel
- DoModal
- DoModalComponent
- Exec (think-time only when synchronous)
- File attach functions.
- InsertImage
- OLE functions (think-time only when object requires user action) (CreateObject, ObjectDoMethod, ObjectSetProperty, ObjectGetProperty)
- Prompt
- RemoteCall
- RevalidatePassword
- WinExec (think-time only when synchronous)
- WinMessage and MessageBox (depending on *style* parameter)

WinMessage and MessageBox

The **WinMessage** and **MessageBox** functions sometimes behave as think-time functions, depending on the value passed in the function's *style* parameter, which controls, among other things, the number of buttons displayed in the message dialog box.



In PeopleSoft Internet Architecture *style* is ignored if the message has any severity other than Message.

Here is the syntax of both functions:

```
MessageBox(style, title, message_set, message_num, default_txt [, paramlist])
```

```
WinMessage(message [, style] [, title])
```



The **WinMessage** function is supported for compatibility with previous releases of PeopleTools. Future applications should use MessageBox instead.

If the *style* parameter specifies more than one button, the function behaves as a think-time function and is subject to the same restrictions as other think-time functions (that is, it should never be used from SavePreChange through SavePostChange PeopleCode, or in RowSelect).

If the **style** parameter specifies a single button (that is, the **OK** button), then 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 this situation, you should always pass an appropriate value in the **WinMessage** style parameter.

The following table shows the values that can be passed in the **style** parameter. To calculate the value to pass make one selection from each category in the table, then add the selections:

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 about these functions see MessageBox and WinMessage.

Program Execution with Fields not in the Data Buffer

Accessing a field that isn't in the data buffer under certain conditions causes a portion of your PeopleCode program to be skipped. The skip only occurs:

- in the Import Manager or if the reference is from the FieldDefault or FieldFormula events, and
- the reference executes on a field that doesn't exist in the data buffer

After the call to the invalid field, execution skips to the next "top-level" statement. Top-level statements are statements that aren't nested inside other statements. The start of a PeopleCode program is a top-level statement. Nesting begins with the first conditional statement (such as While or If) or the first function call.

For example, if your code is executing in a function *and* inside an If...then...end-if statement, and it runs into the skip conditions, the next statement executed is the one *after* the end-if, still inside the function.

Errors and Warnings

Error and **Warning** should not be used in FieldDefault, FieldFormula, RowInit, FieldChange, RowInsert, SavePreChange, WorkFlow and SavePostChange PeopleCode. An **Error** or **Warning** in these events causes a runtime error that forces cancellation of the component.



For more information about these functions, see Warning and Error.

DoSave

DoSave can be used only in FieldEdit, FieldChange, or MenuItemSelected PeopleCode.



For more information about this function see DoSave.

Record Object Database Methods

The following record object methods are used to update the database.

- Delete
- Insert
- Update

These methods should only be used in events that allow database updates, that is, in the following events:

- SavePreChange
- WorkFlow
- SavePostChange
- Message Subscription
- FieldChange
- Application Engine PeopleCode action



For more information about these methods, see ProcessRequest Class.

SQL Object Methods and Functions

The SQL object can be used to update the database. These functions and methods should only be used in events that allow database updates, that is, in the following events:

- SavePreChange
- WorkFlow
- SavePostChange
- Message Subscription
- FieldChange
- Application Engine PeopleCode action

Component Interface Restricted Functions

PeopleCode events and functions that relate exclusively to GUI and online processing can't be used by Component Interface. 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.

For the unsupported functions, you should put a condition around them, testing whether there's an existing Interface Component or not.

```

If %CompIntfcName Then

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

    /* do CI specific processing */

Else

    /* do regular processing */

    . . .

End-if;
```

CallAppEngine

The **CallAppEngine** function should only be used in 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.

ReturnToServer

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, not in an Component Processor flow event.



For more information see PeopleSoft Application Messaging.

GetPage

The **GetPage** function can't be used until after the page processor has loaded the page. You shouldn't use this function in an event prior to the PostBuild event.



For more information about events, see PeopleCode and the Component Processor. For more information about this function, see GetPage.

GetGrid

PeopleSoft builds a grid one row at a time. Because the Grid class applies to a complete grid, you can't use the **GetGrid** function in an event prior to the Activate Event.



For more information about events, see PeopleCode and the Component Processor. For more information about this function, see GetGrid.

GetControl

The **GetControl** function returns a reference to an ActiveX control. You can't access any controls until after the page processor has loaded the page. You shouldn't use this function in an event prior to the Activate Event.



For more information about events, see PeopleCode and the Component Processor. For more information about this function, see GetControl.

Publish Method

If you are using Application Messaging, your **Publish** PeopleCode should go in the SavePostChange event, for either the record or the Component.



For more information see PeopleSoft Application Messaging.

Implementing Modal Transfers

Modal transfers allow you to transfer from one component (the *originating* component) to another component (the *modal* component) modally; that is, requiring the user to OK or Cancel the modal component before returning to the originating component.

Modal transfers give you some control over the order in which the user fills in pages, which is particularly useful in cases where data in the originating component can be derived from data entered by the user into the modal component.

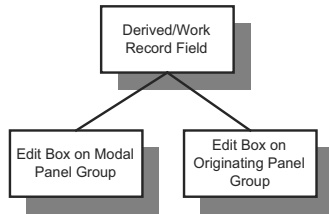
Be careful to not overuse this feature as it restricts user freedom by forcing users to complete interaction with the modal page before returning to the main component.

A modal component resembles, in many respects, a Windows modal dialog. It displays three buttons: OK, Cancel, and Apply. No toolbars or windows are available while the modal component has the focus.

- The OK button saves changes to the modal component and returns the user to the originating component.
- The Apply button saves changes to the modal component without returning to the originating component.
- The Cancel button returns the user to the originating component without saving changes to the modal component.

Modal components are generally smaller than the page from which they are invoked. Remember that "OK" and "Cancel" buttons are added at runtime, thus increasing the size of the page(s).

The originating component and the modal component share record fields in a Derived/Work record called a *shared work record*. The Derived/Work fields of this record provide the two components with an area in memory where they can share data. Edit boxes in both components are associated with the same Derived/Work field, so that changes made to this field in the originating component are reflected in the modal component, and vice versa.



Edit Boxes on the Originating and Modal Components Share the Same Data

Edit boxes associated with the same Derived/Work fields must be placed at level zero in both the originating component and the modal component.

You can use the shared fields to:

- Pass values assigned to the search keys in the modal component search record. If these fields are missing or invalid, the search dialog appears, allowing the user to enter search keys.
- Pass other values from the originating component to the modal component.
- Pass values back from the modal component to the originating component.

Considerations Before Implementing a Modal Transfer

Any component accessible through an application menu system can be accessed via a modal transfer. However, to implement a modal transfer, you need to make some modifications to pages in both the originating component and the modal component. Once these modifications are complete, you can implement the modal transfer using the **DoModalComponent** PeopleCode function from a page in the originating component.

Before beginning this process, you should answer the following questions:

- Should the originating component provide search key values for the modal component? If so, what are the search keys? (Check the modal component's search record.)
- Does the originating component need to pass any data to the modal component? If so, what record fields are needed to store this data?
- Does the modal component need to pass any data back to the originating component? If so, what record fields are needed to store this data?

To implement a modal transfer

1. Create Derived/Work record fields for sharing data between the originating and modal components.

Create a new Derived/Work record or open an existing Derived/Work record. If suitable record fields exist, you can use them; otherwise create new record fields for any data that needs to be shared between the components. These can be search keys for the modal component, data to pass to the modal component, or data to pass back to the originating component.

2. Add derived work fields to the level-zero area of the originating component.

Add one edit box for each of the Derived/Work fields that you need to share between the originating and modal components to the level-zero area of the page from which the transfer will take place. You will probably want to make the edit boxes invisible.

3. Add these same derived work fields to the level-zero area of the modal component.

Add one edit box for each of the edit boxes that you added in the previous step to the level-zero area of the page that you are transferring to. You will probably want to make the edit boxes invisible.

4. Add PeopleCode to pass values into the Derived/Work fields in the originating component.

If you want to provide search key values or pass data to the modal page, write PeopleCode that assigns appropriate values to the Derived/Work fields at some point before **DoModalComponent** is called.

For example, if the modal component search key is PERSONAL_DATA.EMPLID, you could place the following assignment statement in the Derived/Work field's RowInit event:

```
EMPLID = PERSONAL_DATA.EMPLID
```

You also might assign these values in the same program where **DoModalComponent** is called.

5. Add any necessary PeopleCode to access and change the Derived/Work fields in the modal component.

No PeopleCode is required to pass search key values during the search. However, if other data has been passed to the modal component, you may need PeopleCode to access and use the data. You may also need to assign new values to the shared fields so that they can be used by the originating component.

It is possible that the component was accessed through the menu system and not via a modal transfer. To write PeopleCode that runs only in the component when it is running modally, use the **IsModalComponent** built-in function:

```
If IsModalComponent() Then

    /* PeopleCode for modal execution only. */

End-If
```

6. Add any necessary PeopleCode to access changed Derived/Work fields in the originating component.

If the modal component has altered the data in the shared work fields, you can write PeopleCode to access and use the data after **DoModalComponent** has executed.



For more information on the PeopleCode functions, see **DoModalComponent** and **IsModal**.

Using the ImageReference Field

After you create an image definition in Application Designer, if you want to associate it with a field at runtime, the field has to be of type ImageReference. An example of this is referencing a red, yellow, or green light on a page, depending on the context.



For Window Client, you can **not** use a ImageReference field in a grid. For PeopleSoft Internet Architecture, you can.

To change the image value of a ImageReference field

1. Create a field of type ImageReference.
2. Create the images you want to use.

These images must be saved in Application Designer, as Image definitions.

3. Add the field to a record that will be accessed by the page.
4. Add an image control to the page and associate the image control with the ImageReference field.



For more information on creating fields, creating images, adding fields to records and associating a record field with an image control, see [Creating Field Definitions](#).

5. Assign the field value.

Use the keyword **Image** to assign a value to the field. For example:

```
Local Record &MyRec;

Global Number &MyResult;

&MyRec = GetRecord();

If &MyResult Then

    &MyRec.MyImageField.Value = Image.THUMBSUP;

Else

    &MyRec.MyImageField.Value = Image.THUMBSDOWN;

End-If;
```

Using PeopleCode with PeopleSoft Internet Architecture

If you're building pages using **View Internet Options**, there are a few considerations you'll have to take into account when you're creating your PeopleCode programs.

Internet Scripts

An Internet Script is a specialized PeopleCode function that generates dynamic web content. Internet Scripts interact with web clients (browsers) via a request-response paradigm based on the behavior of the Hypertext Transfer Protocol.



For more information see Internet Script Classes.

Unsupported Functions

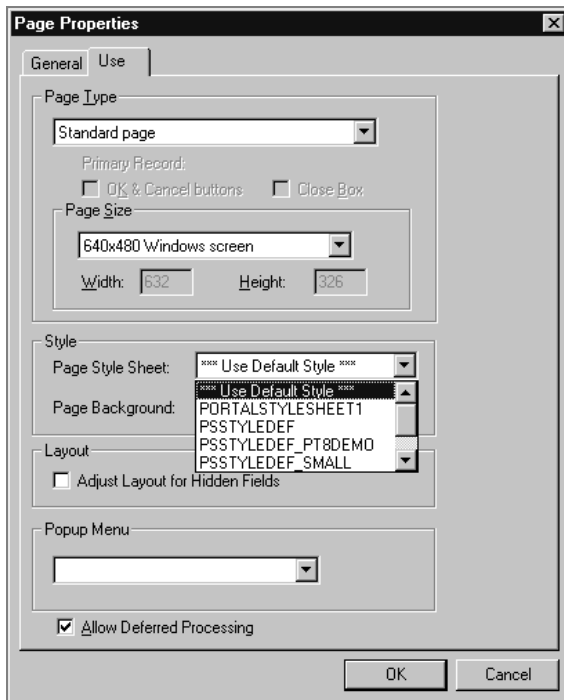
Certain PeopleCode functions and methods aren't supported in the PeopleSoft Internet Architecture. There is no client in a PeopleSoft Internet Architecture application, only the browser. This means client-only PeopleCode **isn't supported** in a PeopleSoft Internet Architecture application. You can use the Validate feature to check your application for client-only PeopleCode.



For more information see Client-Only PeopleCode and Validating Projects.

Using the Field Object Style Property

On the **Use** tab of the page properties, you can associate a page with a style sheet component.



Use tab of Page Properties



For more information on creating Style Sheets, see [Creating Style Sheet Definitions](#).

The style sheet has several classes of styles defined for it. You can edit each style class to change the font, the color, the background, etc. Then you can dynamically change the style of a field using the **Style** property on a field object. This isn't changing the style sheet: just the style class associated with that field.

The following example changes the style class of a field depending on the value entered by the user. This code is in the FieldChange event.

```
Local Field &field;

&field = GetField();

If TESTFIELD1 = 1 Then;

    &field.Style = "PSHYPERLINK";

End-If;

If TESTFIELD1 = 2 Then;
```

```
&field.Style = "PSIMAGE";

End-If;
```

TESTFIELD1: 

Field with PSHYPERLINK style

TESTFIELD1: 

Field with PSIMAGE style



For more information about the field class, see [Field Class](#).

HTML Area

An HTML area control can only be populated in the PeopleSoft Internet Architecture. If you're running in two-tier, the area isn't populated.

There are two ways to populate an HTML area control. One is to select **Constant**, and enter your HTML directly into the control.

The other is to associate the control with a record field, then populate that field with the text you want displayed in the HTML area.



When you associate an HTML area control with a field, make sure the field is long enough to contain the data you want to pass to it. For example, if you associate an HTML area control with a field that is only 10 characters long, only the first 10 characters of your text is displayed.

The following code populates an HTML area with a very simple bulleted list. This code is in the RowInit event of the record field associated with the HTML control.

```
Local Field &HTMLField;

&HTMLField = GetField();

&HTMLField.Value = "<ul><li>Item one</li><li>Item two</li></ul>";
```

The following code is in the FieldChange event of a pushbutton. It populates an HTML area (associated with the record field CHART_DATA.HTMLAREA) with a simple list.

```
Local Field &HTMLField;
```

```
&HTMLField = GetRecord(Record.CHART_DATA).HTMLAREA;  
  
&HTMLField.Value = "<ul><li>Item one</li><li>Item two</li></ul>";
```

The following code populates an HTML area (associated with the record DERIVED_HTML, the field HTMLAREA) with the output of the GenerateTree function:

```
DERIVED_HTML.HTMLAREA = GenerateTree(&TREECTL);
```



For more information see Using the GenerateTree Function.

The following tags are unsupported by the HTML area control:

```
<body>  
<frame>  
<frameset>  
<form>  
<head>  
<html>  
<meta>  
<title>
```



For more information about placing an HTML area control on your page, see HTML Area Control.

Using HTML Definitions and the GetHTMLText Function

If you're using the same HTML text in more than one place, or if it's a large, unwieldy string, you can create an HTML definition in Application Designer, then use the GetHTMLText function to populate an HTML area control.

The following is the HTML string to create a simple table:

```
<P>  
  
<TABLE>
```

```

<TR bgColor=#008000>

  <TD>

    <P><FONT color=#f5f5dc face="Arial, Helvetica, sans-serif"
      size=2>message 1 </FONT></P></TD></TR>

<TR bgColor=#0000cd>

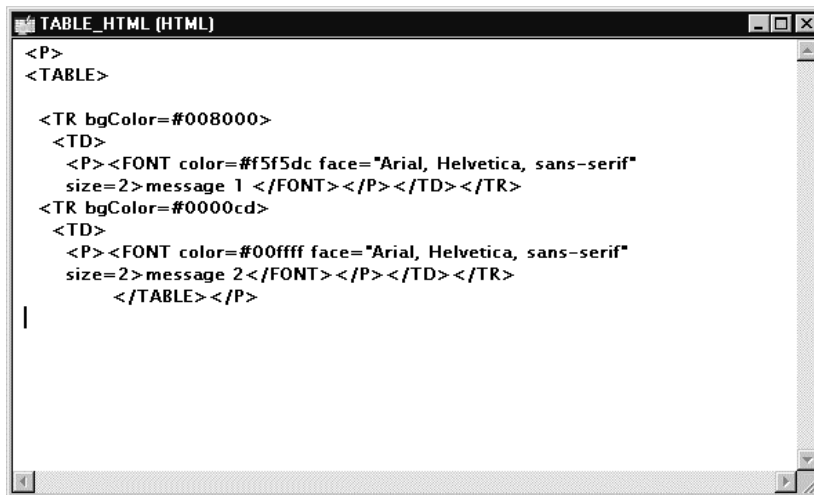
  <TD>

    <P><FONT color=#00ffff face="Arial, Helvetica, sans-serif"
      size=2>message 2</FONT></P></TD></TR>

</TABLE></P>

```

This HTML is saved to an HTML definition called TABLE_HTML:



HTML definition TABLE_HTML

This code is in the RowInit event of the record field associated with the HTML area control:

```

Local Field &HTMLField;

&HTMLField = GetField();

&string = GetHTMLText (HTML.TABLE_HTML);

&HTMLField.Value = &string;

```

This produces the following:

```
message 1
message 2
```

Example



For more information about GetHTMLText, see GetHTMLText.

Using HTML Definitions and the GetJavaScriptURL Method

HTML definitions can hold JavaScript programs in addition to HTML. If you have an HTML definition that contains a JavaScript, use the GetJavaScriptURL Response method to access (and execute) this JavaScript.



Example JavaScript HTML definition

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;
```



For more information, see GetJavaScriptURL and Creating HTML Definitions.

Increasing the Internet/Tuxedo Timeout

If you have a large component that's timing out, you can increase the Internet/Tuxedo timeout.

Modify the `pstools.properties` file (if you choose the defaults when you set up your web site, this file resides in the `c:\Program Files\Apache Group\Apache\htdocs\peoplesoft8` directory.)

Increase the values for the following lines:

- `tuxedo_network_disconnect_timeout=0`
- `tuxedo_send_timeout=50`
- `tuxedo_receive_timeout=600`



You need to recycle your web server for this to take effect.

Inserting using PeopleCode

When inserting rows using PeopleCode, you can either use the `Insert` method with a record object, or create a SQL Insert statement using the SQL object.

- If you're doing a single insert, use the `Record Insert` method
- If you're in a loop, and therefore calling the insert more than once, use the SQL object.

Why? Because the SQL object uses *dedicated cursors* and if the database you're working with supports it, *bulk insert*.

A *dedicated cursor* means that the SQL only gets compiled once on the database, so PeopleTools only looks for the meta-SQL once. This can mean better performance.

For *bulk insert*, inserted rows are buffered and only sent to the database server when the buffer is full or a COMMIT occurs. This cuts down on the number of roundtrips to the database. Again, this can mean better performance.

The following is an example of using the record insert method:

```
&REC = CreateRecord(Record.GREG) ;

&REC.DESCR.Value = "Y" | &I;

&REC.EMPLID.Value = &I;

&REC.Insert() ;
```

The following is an example using a SQL object for doing an insert:

```
&SQL = CreateSQL("%INSERT(:1)");  
  
&REC = CreateRecord(Record.GREG);  
  
&SQL.BulkMode = True;  
  
For &I = 1 to 10  
  
    &REC.DESCR.Value = "Y" | &I;  
  
    &REC.EMPLID.Value = &I;  
  
    &SQL.Execute(&REC);  
  
End-For;
```



For more information see Insert record method and SQL Class.

Using the GenerateTree Function

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 HTML area control and the GenerateTree function are only supported in the PeopleSoft Internet Architecture, not in Windows Client.

The GenerateTree function displays data from a rowset. You can populate this rowset using existing record data. You can also use the Tree Classes to display data from trees created using Tree Manager.

In order to use this function you must set up a page for displaying the data, as well as populating a standalone rowset with the data to be displayed.

Tree Control Test

SetID:

Set Control Value:

Tree Name: DEPT_SECURITY

Effective Date: 01/01/1996

[First](#) | [Previous](#) | [Next](#) | [Last](#) | [Left](#) | [Right](#)

00001 - Corporate Headquarters

FIN - Financial Services

HLC - Health Care Services

MFG - Manufacturing

M-AMERICAS - North and South America

M-ASIAPAC - Asia Pacific

M-EUR-ALL - Europe-Africa-Middle East

LOC - Local Counties

UNV - Higher Education

UTIL - Utilities

T000

T001

T002

T003

T004

Return to Search

HTML Tree Example

The positional links at the top of the page (First, Previous, Next, Last, Left, Right) allow the user to navigate around the tree. These links are automatically generated as part of the execution of GenerateTree.

The icon next to a node can have a + sign or a - sign in it, depending on whether the node is collapsed or expanded. When a node is collapsed, none of the nodes that report to the collapsed node are displayed, and the icon has a + sign. When a node is expanded, all the nodes that report to it are displayed, and the icon has a - sign. You can collapse and expand a node by clicking in the icon. This enables you to view the tree at different levels of detail. When the icon for a node has no + or - sign in it, it is a terminal node, and cannot be expanded or collapsed.

Building your HTML Tree Page

The page you use to display the HTML tree must contain the following:

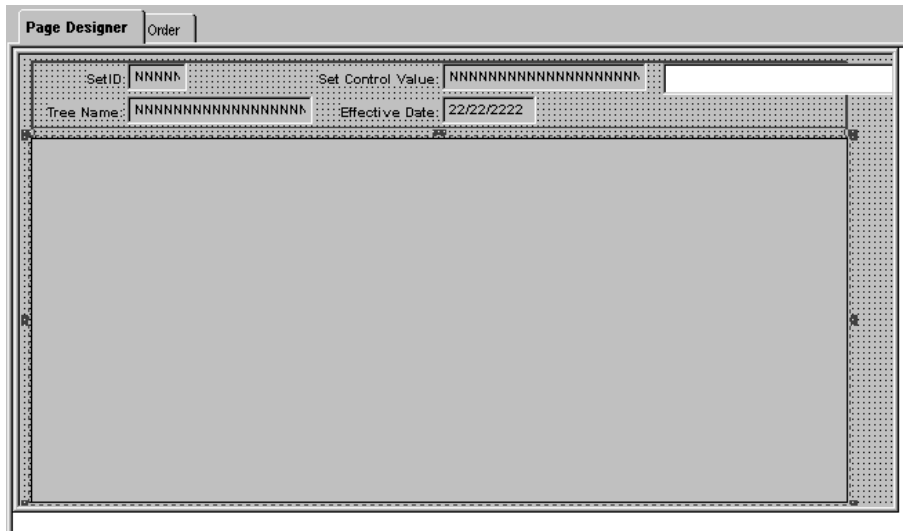
- An HTML area used to display the HTML tree
- An invisible character field that:
 - has a Page Field name
 - is at least 46 characters long
 - is invisible



The edit box should be invisible, but **not** display-only. If the field is display-only, it can't be written to. Making it invisible makes it not visible to the user, but it still has a buffer that can be written to.

Events are sent to the application from the HTML tree using the invisible field. The events are processed by FieldChange PeopleCode that is attached to the invisible field.

The following is an example page for an HTML tree:



Example Application Designer HTML tree page

The large area that is selected in the screen shot above is the HTML Area that displays the HTML tree. The HTML Area is attached to the `DERIVED_HTML.HTMLAREA` field for this example.

The white edit box is the invisible field used to pass events from the HTML tree to the application and is attached to the `DERIVED_HTML.TREECTLEVENT` field for this example.

The edit box must have a Page Field Name. In this example, the Page Field Name is `TREECTLEVENT`.

The HTML Tree Rowset Records

The `GenerateTree` function takes a pre-built and populated rowset as a parameter. This rowset must have a certain structure and contain certain fields. In this example, it's a stand-alone rowset, that is, the rowset is created using the `CreateRowset` function. The fields necessary for the rowset are contained in the following record definitions:

- The header record `TREECTL_HRD`, containing the subrecord `TREECTL_HDR_SBR`.
- The node record `TREECTL_NDE`, containing the subrecord `TREECTL_NDE_SBR`.

The header record is the level 0 record of the HTML tree rowset. It contains the options for the HTML tree, such as the name of the collapsed node image, the height of the images, the number of pixels to indent each node, and so on.

The node record is the level 1 record of the HTML tree rowset. It contains the tree data, as well as information about the data: is it a dynamic range leaf, what is the level, and so on.

There is a row in the level 1 scroll for each node or leaf in the tree data.

If you would like to store additional application data with each node in the tree, you can incorporate the TREECTL_NDE_SBR into a record of your definition and use your record to define the HTML tree rowset.

For example, you might want to store application key values with each node record, so that when a user selects a node you will have the data you need to perform the action that you wish to perform.

The following are the relevant fields in TREECTL_HDR_SBR:

PAGE_NAME	Name of the page that contains the HTML Area and the invisible field used to process the HTML tree events.
PAGE_FIELD_NAME	Page field name of the invisible field used to process the HTML tree events.
PAGE_SIZE	Number of nodes or leaves to send to the browser at a time. Set to 0 to send all the visible nodes or leaves to the browser. Default value is 0.
DISPLAY_LEVELS	Number of levels to display on the browser at a time. Default value is 8.
COLLAPSED_IMAGE	Collapsed node image name. Default value is PT_TREE_COLLAPSED.
EXPANDED_IMAGE	Expanded node image name. Default value is PT_TREE_EXPANDED.
END_NODE_IMAGE	End node image name. Default value is PT_TREE_END_NODE.
LEAF_IMAGE	Leaf image name. Default value is PT_TREE_LEAF.
IMAGE_WIDTH	Image width in pixels. All four images need to be the same width. Default value is 15 pixels.
IMAGE_HEIGHT	Image height in pixels. All four images need to be the same height. Default value is 12 pixels.
INDENT_PIXELS	Number of pixels to indent each level. Default value is 20 pixels.
TREECTL_VERSION	Version of the HTML tree. Default value is 812. Used with the DESCR_IMAGE field in TREECTL_HDR_SBR.

TREECTL_HDR_SBR (Record)						
Record Fields		Record Type				
Num	Field Name	Type	Len	Format	Short Name	Long Name
1	PAGE_NAME	Char	18	Upper	Page Name	Page Name
2	PAGE_FIELD_NAME	Char	18	Upper	Page Field Name	Page Field Name
3	PAGE_SIZE	Nbr	3		Page Size	Page Size
4	DISPLAY_LEVELS	Nbr	2		Display Levels	Display Levels
5	COLLAPSED_IMAGE	Char	30	Upper	Collapsed Image	Collapsed Image Name
6	EXPANDED_IMAGE	Char	30	Upper	Expanded Image	Expanded Image Name
7	END_NODE_IMAGE	Char	30	Upper	End Node Image	End Node Image Name
8	LEAF_IMAGE	Char	30	Upper	Leaf Image Name	Leaf Image Name
9	IMAGE_WIDTH	Nbr	2		Image Width	Image Width
10	IMAGE_HEIGHT	Nbr	2		Image Height	Image Height
11	INDENT_PIXELS	Nbr	2		Indent Pixels	Indent Pixels
12	TREECTL_VERSION	Nbr	4		HTML Tree Vers	HTML Tree Version
13	TOP_NODE_NUM	Nbr	7		Top Node Num	Top Node Num
14	LEFT_LEVEL_NUM	Nbr	2		Left Level Num	Left Level Num

TREECTL_HDR_SBR record definition

The following are the relevant fields in TREECTL_NDE_SBR:

LEAF_FLAG	If this is a leaf set to "Y". Default value is N
TREE_NODE	Node name.
DESCR	Node description. (optional)
RANGE_FROM	Leaf's range from value.
RANGE_TO	Leaf's range to value.
DYNAMIC_FLAG	If this leaf has a dynamic range, set to "Y". Default value is N
ACTIVE_FLAG	Set to "N" for the node or leaf not to be a link. Default value is Y
DISPLAY_OPTION	Set to "N" to display the name only. Set to "D" to display the description only. Set to "B" to display both the name and the description. Only used for nodes. Default value is B
STYLECLASSNAME	Used to control the style of the link associated with the node or leaf. Default value is PSHYPERLINK.
PARENT_FLAG	If this node is a parent and its direct children will be loaded now, set to "Y". If this node is a parent and its direct children are to be loaded on demand, set to "X". If this node is not a parent, set to "N". Default value is N.
TREE_LEVEL_NUM	Set to the node's level. Default value is 1.
LEVEL_OFFSET	If a child node is to be displayed more than one level to the right of its parent, specify the number of additional levels. Default value is 0.

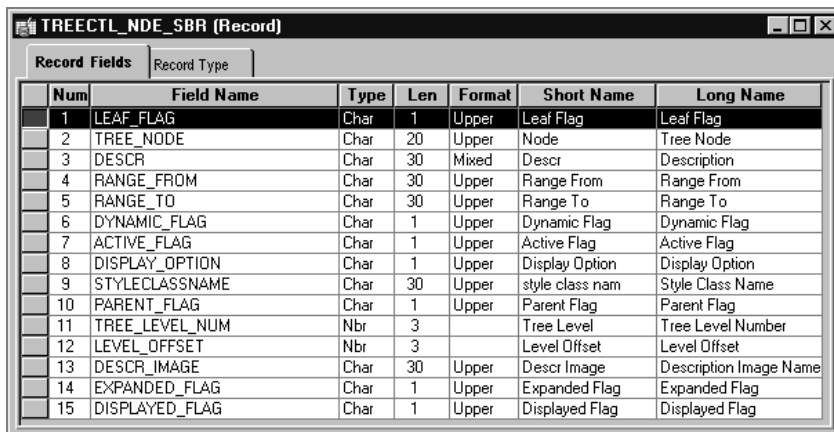
DESCR_IMAGE

Use this field to display an image after the node or leaf image and before the name or description. There is a space between the two images. The new image isn't scaled. This field takes a string value, the name of an image definition created in Application Designer.

This field is only recognized if the TREECTL_VERSION field is greater than or equal to 812.

EXPANDED_FLAG

When a node's EXPANDED_FLAG is "Y", the GenerateTree function expects the node's immediate children to be loaded into the &TREECTL Rowset (such as in PostBuild) and GenerateTree generates HTML such that the node is expanded and its immediate children are displayed.



Num	Field Name	Type	Len	Format	Short Name	Long Name
1	LEAF_FLAG	Char	1	Upper	Leaf Flag	Leaf Flag
2	TREE_NODE	Char	20	Upper	Node	Tree Node
3	DESCR	Char	30	Mixed	Descr	Description
4	RANGE_FROM	Char	30	Upper	Range From	Range From
5	RANGE_TO	Char	30	Upper	Range To	Range To
6	DYNAMIC_FLAG	Char	1	Upper	Dynamic Flag	Dynamic Flag
7	ACTIVE_FLAG	Char	1	Upper	Active Flag	Active Flag
8	DISPLAY_OPTION	Char	1	Upper	Display Option	Display Option
9	STYLECLASSNAME	Char	30	Upper	style class nam	Style Class Name
10	PARENT_FLAG	Char	1	Upper	Parent Flag	Parent Flag
11	TREE_LEVEL_NUM	Nbr	3		Tree Level	Tree Level Number
12	LEVEL_OFFSET	Nbr	3		Level Offset	Level Offset
13	DESCR_IMAGE	Char	30	Upper	Descr Image	Description Image Name
14	EXPANDED_FLAG	Char	1	Upper	Expanded Flag	Expanded Flag
15	DISPLAYED_FLAG	Char	1	Upper	Displayed Flag	Displayed Flag

TREECTL_NDE_SBR record definition

HTML Tree End-User Actions (Events)

The GenerateTree function works with an HTML area control and an invisible field. 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.

The FieldChange PeopleCode Example program checks for expanding (or collapsing) a node, as well as selecting a node, by checking the first character in the invisible field. The following example is just checking for whether a node is selected or not:

```
If Left(TREECTLEVENT, 1) = "S" Then
```

In your application, you can check for the following end-user actions.

<i>Event</i>	<i>Description</i>
Tn	Toggle the node. Expand or collapse, the opposite of the previous state.

Event	Description
	<i>n</i> is the node's row number in the TREECTL_NODE rowset
<i>Xn</i>	Expand the node, but first load the children. The children are loaded in PeopleCode, then the event is passed to GenerateTree, so that the HTML can be generated with the node expanded. <i>n</i> is the node's row number in the TREECTL_NODE rowset
F	Display the first page.
P	Display the previous page.
N	Display the next page.
L	Display the last page.
Q	Scroll the display left one level.
R	Scroll the display right one level.
<i>Sn</i>	Select the node or leaf. <i>n</i> is the node's or leaf's row number in the TREECTL_NODE rowset



Drag-and-drop is **not** supported.

Customizing the PeopleCode for the HTML Tree

The PeopleCode for initializing the HTML tree for this example was put into the PostBuild event of the component that contained the page with the HTML area used with the HTML tree.

The PostBuild PeopleCode Example program is an example of how to initialize the HTML tree using the Tree classes and load just the root node into the HTML tree rowset.

The first time a user expands a node, the node's direct children are loaded into the HTML tree rowset by the FieldChange PeopleCode Example program, listed further down. This chunking functionality allows the HTML tree to support trees of any size with good performance.

You can't just copy either the PostBuild or FieldChange PeopleCode example programs into your application. You must make some changes to them to make them work with your data. Changes that you will need to make to the PostBuild PeopleCode are as follows.

To modify the PeopleCode for your HTML tree (part one)

1. Set the PAGE_NAME and PAGE_FIELD_NAME fields.

The PAGE_NAME field contains the name of the page that contains the HTML Area and the invisible field that will be used to process the HTML tree events. The PAGE_FIELD_NAME field is the page field name of the invisible field that is used to process the HTML tree events.



This is the Page Field name of the invisible field, **not** the invisible field name.

2. Set the tree specific variables.

The &SET_ID, &USERKEYVALUE, &TREE_NAME, &TREE_DT, and &BRANCH_NAME variables contain the specific information about your tree. Set these values to the tree you want to open. In the example PeopleCode, they are set as follows:

```
&SET_ID = PSTREEDEFN_VW.SETID;

&USERKEYVALUE = "";

&TREE_NAME = PSTREEDEFN_VW.TREE_NAME;

&TREE_DT = PSTREEDEFN_VW.EFFDT;

&BRANCH_NAME = "";
```

3. Set the PAGE_SIZE field.

If you don't want the page to expand vertically to display the tree, set the PAGE_SIZE to a number of rows that will fit inside the HTML Area. If some vertical expansion is okay, but you don't want the page to get too big, set the PAGE_SIZE to whatever value you like. Set the PAGE_SIZE to 0 if you don't care how big the page gets.

4. Set the DISPLAY_LEVELS field to the number of levels that will fit inside the HTML Area.

If this field is set too large, wrapping may occur. Positional links at the top of the HTML area allow the user to navigate as the tree expands.

5. Set the DISPLAY_OPTION field (optional)

The default for the DISPLAY_OPTION field is to display both the node name and the description. You can chose to display just the node name or just the description. The values for this field are:

Value	Description
N	Display name only
D	Display description only
B	Display both name and description

6. Set the STYLECLASSNAME field for the root node (optional)

The STYLECLASSNAME field controls the style of the link associated with a node or leaf. The default for the STYLECLASSNAME is PSHYPERLINK. If PSHYPERLINK isn't the style you want to use, change this field value to the style you want.

7. Change the last line to assign the output of GenerateTree() to the field attached to the HTML Area that will display the tree.

In this example, the HTML Area control is the DERIVED_HTML.HTMLAREA. You need to specify the record and field name associated with the HTML area control on your page.

PostBuild PeopleCode Example

```

Component Rowset &TREECTL;

&NODE_ROWSET = CreateRowset (Record.TREECTL_NODE);

&TREECTL = CreateRowset (Record.TREECTL_HDR, &NODE_ROWSET);

&TREECTL.InsertRow(1);

&REC = &TREECTL.GetRow(2).GetRecord(1);

/* Set the HDR options:

1) PAGE_NAME - Name of the page that contains the HTML Area and the invisible
field that will be used to process the HTML tree events.

2) PAGE_FIELD_NAME - Page field name of the invisible field that will be used to
process the HTML tree events.

3) PAGE_SIZE - Number of nodes or leaves to send to the browser at a time. Set
to 0 to send all of the visible nodes or leaves to the browser. Default value:
0

4) DISPLAY_LEVELS - Number of levels to display on the browser at a time.
Default value: 8

5) COLLAPSED_IMAGE - Collapsed node image name. Default value:
PT_TREE_COLLAPSED

6) EXPANDED_IMAGE - Expanded node image name. Default value: PT_TREE_EXPANDED

7) END_NODE_IMAGE - End node image name. Default value: PT_TREE_END_NODE

8) LEAF_IMAGE - Leaf image name. Default value: PT_TREE_LEAF

9) IMAGE_WIDTH - Image width. All four images need to be the same size.
Default value: 15

10) IMAGE_HEIGHT - Image height. Default value: 12

11) INDENT_PIXELS - Number of pixels to indent each level. Default value: 20

*/

```

```

&REC.GetField(Field.PAGE_NAME).Value = "TREECTL_TEST";

&REC.GetField(Field.PAGE_FIELD_NAME).Value = "TREECTLEVENT";

&REC.GetField(Field.PAGE_SIZE).Value = 15;

&REC.GetField(Field.DISPLAY_LEVELS).Value = 8;

&REC.GetField(Field.COLLAPSED_IMAGE).Value = "PT_TREE_COLLAPSED";

&REC.GetField(Field.EXPANDED_IMAGE).Value = "PT_TREE_EXPANDED";

&REC.GetField(Field.END_NODE_IMAGE).Value = "PT_TREE_END_NODE";

&REC.GetField(Field.LEAF_IMAGE).Value = "PT_TREE_LEAF";

&REC.GetField(Field.IMAGE_WIDTH).Value = 15;

&REC.GetField(Field.IMAGE_HEIGHT).Value = 12;

&REC.GetField(Field.INDENT_PIXELS).Value = 20;


&SET_ID = PSTREEDEFN_VW.SETID;

&USERKEYVALUE = "";

&TREE_NAME = PSTREEDEFN_VW.TREE_NAME;

&TREE_DT = PSTREEDEFN_VW.EFFDT;

&BRANCH_NAME = "";


&MYSESSION = %Session;

&SRC_TREE = &MYSESSION.GetTree();

&RES = &SRC_TREE.OPEN(&SET_ID, &USERKEYVALUE, &TREE_NAME, &TREE_DT,
&BRANCH_NAME, False);


/* Just insert the root node into the &TREECTL Rowset. If the root node has
children, set the &PARENT_FLAG to 'X', so that its children will be loaded on
demand. */


&ROOT_NODE = &SRC_TREE.FindRoot();


If &ROOT_NODE.HasChildren Then

    &PARENT_FLAG = "X";

```



```

Else

    &PARENT_FLAG = "N";

End-If;

&NODE_ROWSET = &TREECTL.GetRow(2).GetRowset(1);

&NODE_ROWSET.InsertRow(1);

&REC = &NODE_ROWSET.GetRow(2).GetRecord(1);

/* Set the NODE values:

1) LEAF_FLAG - If this is a leaf set to "Y". Default value: N

2) TREE_NODE - Node name.

3) DESCR - Node description. (optional)

4) RANGE_FROM - Leaf's range from value.

5) RANGE_TO - Leaf's range to value.

6) DYNAMIC_FLAG - If this leaf has a dynamic range, set to "Y". Default value:
N

7) ACTIVE_FLAG - Set to "N" for the node or leaf not to be a link. Default
value: Y

8) DISPLAY_OPTION - Set to "N" to display the name only. Set to "D" to display
the description only. Set to "B" to display both the name and the description.
Only used for nodes. Default value: B

9) STYLECLASSNAME - Used to control the style of the link associated with the
node or leaf. Default value: PSHYPERLINK

10) PARENT_FLAG - If this node is a parent and its direct children will be
loaded now, set to "Y". If this node is a parent and its direct children are to
be loaded on demand, set to "X". Default value: N

11) TREE_LEVEL_NUM - Set to the node's level. Default value: 1

12) LEVEL_OFFSET - If a child node is to be displayed more than one level to the
right of its parent, specify the number of additional levels. Default value: 0

*/

&REC.GetField(Field.LEAF_FLAG).Value = "N";

&REC.GetField(Field.TREE_NODE).Value = &ROOT_NODE.NAME;

```

```

&REC.GetField(Field.DESCR).Value = &ROOT_NODE.DESCRPTION;

&REC.GetField(Field.RANGE_FROM).Value = "";

&REC.GetField(Field.RANGE_TO).Value = "";

&REC.GetField(Field.DYNAMIC_FLAG).Value = "N";

&REC.GetField(Field.ACTIVE_FLAG).Value = "Y";

&REC.GetField(Field.DISPLAY_OPTION).Value = "B";

&REC.GetField(Field.STYLECLASSNAME).Value = "PSHYPERLINK";

&REC.GetField(Field.PARENT_FLAG).Value = &PARENT_FLAG;

&REC.GetField(Field.TREE_LEVEL_NUM).Value = 1;

&REC.GetField(Field.LEVEL_OFFSET).Value = 0;


&SRC_TREE.Close();

DERIVED_HTML.HTMLAREA = GenerateTree(&TREECTL);

```

The FieldChange PeopleCode below is used to process the events passed from the HTML tree to the application. The code that processes the load children event loads the direct children of a node the first time the node is expanded by the user. Changes that you will need to make to the FieldChange PeopleCode are as follows.

To modify the PeopleCode for your HTML tree (part two)

1. Globally change TREECTLEVENT to the name of the invisible field that will be used to process the events.
2. Set the tree specific variables.

The &SET_ID, &USERKEYVALUE, &TREE_NAME, &TREE_DT and &BRANCH_NAME variables contain the specific information about your tree. Set these values to the tree you want to open. In the example PeopleCode, they are set as follows:

```

&SET_ID = PSTREEDEFN_VW.SETID;

&USERKEYVALUE = "";

&TREE_NAME = PSTREEDEFN_VW.TREE_NAME;

&TREE_DT = PSTREEDEFN_VW.EFFDT;

&BRANCH_NAME = "";

```

3. Set the DISPLAY_OPTION field (optional)

The default for the DISPLAY_OPTION field is to display both the node name and the description. You can choose to display just the node name or just the description. The values for this field are:

Value	Description
N	Display name only
D	Display description only
B	Display both name and description

4. Set the STYLECLASSNAME field for the root node (optional)

The STYLECLASSNAME field controls the style of the link associated with a node or leaf. The default for the STYLECLASSNAME is PSHYPERLINK. If PSHYPERLINK isn't the style you want to use, change this field value to the style you want.

5. Change the assignment of the output of every GenerateTree call to the field attached to the HTML Area that will display the tree.

In this example, the HTML Area control is the DERIVED_HTML.HTMLAREA. You need to specify the record and field name associated with the HTML area control on your page.

6. Change the code that processes the select event to perform the action you want when the user selects a node or leaf.

You will see this section marked as Process Select Event in the code sample, below. Also see HTML Tree End-User Actions (Events).

FieldChange PeopleCode Example

```

Component Rowset &TREETL;

/* process load children event */

If Left(TREETLEVENT, 1) = "X" Then

    &ROW = Value(Right(TREETLEVENT, Len(TREETLEVENT) - 1)) + 1;

    &NODE_ROWSET = &TREETL.GetRow(2).GetRowset(1);

    &PARENT_REC = &NODE_ROWSET.GetRow(&ROW).GetRecord(1);

    &PARENT_LEVEL = &PARENT_REC.GetField(Field.TREE_LEVEL_NUM).Value;

    &ROW = &ROW + 1;

    &SET_ID = PSTREEDEFN_VW.SETID;

```

```

&USERKEYVALUE = "";

&TREE_NAME = PSTREEDEFN_VW.TREE_NAME;

&TREE_DT = PSTREEDEFN_VW.EFFDT;

&BRANCH_NAME = "";

&MYSESSION = %Session;

&SRC_TREE = &MYSESSION.GetTree();

&RES = &SRC_TREE.OPEN(&SET_ID, &USERKEYVALUE, &TREE_NAME, &TREE_DT,
&BRANCH_NAME, False);

/* Find the parent node and expand the tree one level below the parent.
Insert just the direct children of the parent node into the &TREECTL Rowset. If
any of the child nodes have children, set their PARENT_FLAG to 'X', so that
their children are loaded on demand. */

&PARENT_NODE =
&SRC_TREE.FindNode(&PARENT_REC.GetField(Field.TREE_NODE).Value, "");

If &PARENT_NODE.HasChildren Then

    &PARENT_NODE.Expand(2);

If &PARENT_NODE.HasChildLeaves Then

    /* Load the child leaves into the &TREECTL Rowset. */

    &FIRST = True;

    &CHILD_LEAF = &PARENT_NODE.FirstChildLeaf;

    While &FIRST Or

        &CHILD_LEAF.HasNextSib

        If &FIRST Then

            &FIRST = False;

        Else

            &CHILD_LEAF = &CHILD_LEAF.NextSib;

        End-If;

    If &CHILD_LEAF.Dynamic = True Then

```

```

&RANGE_FROM = "";

&RANGE_TO = "";

&DYNAMIC_RANGE = "Y";

Else

&RANGE_FROM = &CHILD_LEAF.RangeFrom;

&RANGE_TO = &CHILD_LEAF.RangeTo;

&DYNAMIC_RANGE = "N";

End-If;

&NODE_ROWSET.InsertRow(&ROW - 1);

&REC = &NODE_ROWSET.GetRow(&ROW).GetRecord(1);

/* Set the NODE values:

```

- 1) LEAF_FLAG - If this is a leaf set to "Y". Default value: N
- 2) TREE_NODE - Node name.
- 3) DESCR - Node description. (optional)
- 4) RANGE_FROM - Leaf's range from value.
- 5) RANGE_TO - Leaf's range to value.
- 6) DYNAMIC_FLAG - If this leaf has a dynamic range, set to "Y". Default value: N
- 7) ACTIVE_FLAG - Set to "N" for the node or leaf not to be a link. Default value: Y
- 8) DISPLAY_OPTION - Set to "N" to display the name only. Set to "D" to display the description only. Set to "B" to display both the name and the description. Only used for nodes. Default value: B
- 9) STYLECLASSNAME - Used to control the style of the link associated with the node or leaf. Default value: PSHYPERLINK
- 10) PARENT_FLAG - If this node is a parent and its direct children will be loaded now, set to "Y". If this node is a parent and its direct children are to be loaded on demand, set to "X". Default value: N
- 11) TREE_LEVEL_NUM - Set to the node's level. Default value: 1

12) LEVEL_OFFSET - If a child node is to be displayed more than one level to the right of its parent, specify the number of additional levels. Default value: 0

*/

```

&REC.GetField(Field.LEAF_FLAG).Value = "Y";

&REC.GetField(Field.TREE_NODE).Value = "";

&REC.GetField(Field.DESCR).Value = "";

&REC.GetField(Field.RANGE_FROM).Value = &RANGE_FROM;

&REC.GetField(Field.RANGE_TO).Value = &RANGE_TO;

&REC.GetField(Field.DYNAMIC_FLAG).Value = &DYNAMIC_RANGE;

&REC.GetField(Field.ACTIVE_FLAG).Value = "Y";

&REC.GetField(Field.DISPLAY_OPTION).Value = "B";

&REC.GetField(Field.STYLECLASSNAME).Value = "PSHYPERLINK";

/* Leaves never have children. */

&REC.GetField(Field.PARENT_FLAG).Value = "N";

&REC.GetField(Field.TREE_LEVEL_NUM).Value = &PARENT_LEVEL + 1;

&REC.GetField(Field.LEVEL_OFFSET).Value = 0;

&ROW = &ROW + 1;

End-While;

End-If;

If &PARENT_NODE.HasChildNodes Then

    /* Load the child nodes into the &TREECTL Rowset. */

    &FIRST = True;

    &CHILD_NODE = &PARENT_NODE.FirstChildNode;

    While &FIRST Or

        &CHILD_NODE.HasNextSib

        If &FIRST Then

            &FIRST = False;

        Else

```

```

        &CHILD_NODE = &CHILD_NODE.NextSib;

End-If;

If &CHILD_NODE.HasChildren Then

    &PARENT_FLAG = "X";

Else

    &PARENT_FLAG = "N";

End-If;

/* If the tree uses strict levels, set the &LEVEL_OFFSET to the
number of levels that the child node is to the right of its parent minus 1. */

If &SRC_TREE.LevelUse = "S" Then

    &LEVEL_OFFSET = &CHILD_NODE.LevelNumber -
&PARENT_NODE.LevelNumber - 1;

Else

    &LEVEL_OFFSET = 0;

End-If;

&NODE_ROWSET.InsertRow(&ROW - 1);

&REC = &NODE_ROWSET.GetRow(&ROW).GetRecord(1);

&REC.GetField(Field.LEAF_FLAG).Value = "N";

&REC.GetField(Field.TREE_NODE).Value = &CHILD_NODE.Name;

&REC.GetField(Field.DESCR).Value = &CHILD_NODE.Description;

&REC.GetField(Field.RANGE_FROM).Value = "";

&REC.GetField(Field.RANGE_TO).Value = "";

&REC.GetField(Field.DYNAMIC_FLAG).Value = "N";

&REC.GetField(Field.ACTIVE_FLAG).Value = "Y";

&REC.GetField(Field.DISPLAY_OPTION).Value = "B";

&REC.GetField(Field.STYLECLASSNAME).Value = "PSHYPERLINK";

&REC.GetField(Field.PARENT_FLAG).Value = &PARENT_FLAG;

&REC.GetField(Field.TREE_LEVEL_NUM).Value = &PARENT_LEVEL + 1;

```

```

        &REC.GetField(Field.LEVEL_OFFSET).Value = &LEVEL_OFFSET;

        &ROW = &ROW + 1;

    End-While;

End-If;

/* change the parent's PARENT_FLAG from 'X' to 'Y' */
&PARENT_REC.GetField(Field.PARENT_FLAG).Value = "Y";

HTMLAREA = GenerateTree(&TREECTL, TREECTLEVENT);

End-If;

&SRC_TREE.Close();

Else

    /* Process select event. */

    /* As an example, just display the selected node name or leaf range as a
    MessageBox. */

    If Left(TREECTLEVENT, 1) = "S" Then

        &ROW = Value(Right(TREECTLEVENT, Len(TREECTLEVENT) - 1)) + 1;

        &NODE_ROWSET = &TREECTL.GetRow(2).GetRowset(1);

        &REC = &NODE_ROWSET.GetRow(&ROW).GetRecord(1);

        If &REC.GetField(Field.LEAF_FLAG).Value = "N" Then

            MessageBox(0, "", 0, 0, "The selected node is %1.",
            &REC.GetField(Field.TREE_NODE).Value);

        Else

            If &REC.GetField(Field.DYNAMIC_FLAG).Value = "N" Then

                If &REC.GetField(Field.RANGE_FROM).Value =
                &REC.GetField(Field.RANGE_TO).Value Then

```



```

        &TEMP = "[" | &REC.GetField(Field.RANGE_FROM).Value | "];

    Else

        &TEMP = "[" | &REC.GetField(Field.RANGE_FROM).Value | " - " |
&REC.GetField(Field.RANGE_TO).Value | "];

    End-If;

    Else

        &TEMP = "[ ]";

    End-If;

    MessageBox(0, "", 0, 0, "The selected leaf is %1.", &TEMP);

    End-If;

    Else

        /* process all other events */

        HTMLAREA = GenerateTree(&TREECTL, TREECTLEVENT);

    End-If;

End-If;

/* done processing the event, so clear it */

TREECTLEVENT = "";

```

Using the Attachment Functions

PeopleTools supplies a subrecord and a work record you should include in your component when using the file attachment functions.



You don't have to include the subrecord in your component: you could just include the fields. However, PeopleSoft recommends including the subrecord.

The subrecord is called FILE_ATTACH_SBR. It contains the following fields:

ATTACHSYSFILENAME	The system file name, that is, the name of the file as it's stored on the ftp archive.
ATTACHUSERFILE	The user file name, that is, the name of the file that the user selects.

No PeopleCode is associated with this subrecord. Developers should include this subrecord in target records for using attachments.

The work record is called FILE_ATTACH_WRK, and contains the following fields:

ATTACHADD	Contains a PeopleCode program used for adding attachments (the AddAttachment built-in function.)
ATTACHDELETE	Contains a PeopleCode program used for deleting attachments (the DeleteAttachment built-in function.)
ATTACHVIEW	Contains a PeopleCode program used for viewing attachments (the ViewAttachment built-in function.)

The PeopleCode program is associated with the FieldChange events for each, as well as the RowInit event for ATTACHADD. The PeopleCode is actually in the form of functions so this work record acts as a container for the appropriate methods related to attachments. You can choose to include this work record in your component and use component PeopleCode to invoke the functions, or you can use your own work record and just call these functions as needed from within your work record.

To use the Attachment functions:

1. Set up the URL for the ftp archive

For example, suppose the system is fileserver.ps.com, the user account is "joe", and the password is "secret". The full URL would be as follows:

```
ftp://joe:secret@fileserver.ps.com/
```

You can either enter create an entry for this URL in the URL Maintenance page, or specify it in your PeopleCode.



For more information about using the URL Maintenance page, see URL Maintenance.

If you specify the URL in your PeopleCode, you don't have to hardcode the user name and password. You can de-reference them using a variable (or record field). This enables you to use encryption with the username and password.

2. Insert the subrecord (FILE_ATTACH_SBR) into a record used with your page.

Typically, this record is inserted as part of the primary database record.



You don't have to include the subrecord in your component: you could just include the fields. However, PeopleSoft recommends including the subrecord.

3. Add buttons to add, delete, view attachments as needed.

You can either use the FILE_ATTACH_WRK record for the necessary record fields, or you can use your own work fields.

4. Modify the PeopleCode to support your application.

If you use the supplied PeopleCode programs, you must specify the URL. You may also want to modify the extension of the expected files (the default is "" which is any file or *.* You may want to indicate ".doc", ".xl", "*.html", "*.exe", "*.tar"). Comments in ATTACHADD.FieldChange tell what to do.

If you choose to use the FILE_ATTACH_WRK for buttons, you may want to use Component PeopleCode for your application specific code. This enables you to keep the code in FILE_ATTACH_WRK generic. If you use your own work fields, you can write your own code and call the functions in FILE_ATTACH_WRK as needed.

The following shows an attach resume page:

Attach Resume page

Using the Select and SelectNew Methods

Select and SelectNew, like the ScrollSelect functions, allow you to control the process of selecting data into a page scroll. You can only use these methods with a rowset. A rowset can be thought of as a page scroll.

A level0 rowset is a rowset that starts at the level0 of the page, and contains all the data in the component buffers.

A child rowset is a rowset that is contained by an upper level rowset, also called the parent rowset. For example, a level1 rowset could be considered the child rowset of a level0, parent, rowset. Or a level2 rowset could be the child rowset of a level1 rowset.

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.

The data contained in a child rowset depends on the row of the parent rowset.



For more information see Data Buffer Access.



In addition to these methods there is the record object method `SelectByKey`, which allows you to select into a record object. If you're only interested in selecting a single row of data, you may want to consider this method instead.

What Select Does

Select selects rows from a table or view and adds the rows to either a rowset or a row. Let's call the record definition of the table or view that it selected from the ***select record***. Let's call the primary database record of the top level rowset object executing the method 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 the same key fields as 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 you actually need.

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.

Select also accepts an optional SQL string that can contain a WHERE clause restricting the number of rows selected into the scroll area. The SQL string can also contain an ORDER BY clause, enabling you to sort the rows.

Select and SelectNew generate an SQL SELECT statement at runtime, based on the fields in the select record and WHERE clause passed to them in the function call. This gives Select and SelectNew 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.

Also, if you use one of the meta-SQL constructs or shortcuts in the WHERE clause, such as `%KeyEqual` or `%List`, even if a field has changed, you won't have to change your code.

Unlike the ScrollSelect functions, neither **Select** or **SelectNew** allow you to operate in **turbo** mode.

Select Syntax

The syntax of **Select** is:

```
Select([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 object executing the method, the second *scrollname* must be a child of the first child, etc.

This syntax does the following:

- Specifies an optional child rowset into which to read the selected rows
- Specifies the select record from which to select rows
- Passes a string containing a SQL WHERE clause to restrict the selection of rows and/or an ORDER BY clause to sort the rows

Let's examine the different parts of this syntax one at a time.

Specifying Child Rowsets

The first part of the Select syntax specifies a child rowset into which rows will be selected. This parameter is optional.

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. For example, suppose you've instantiated a level 1 rowset &BUS_EXPENSES_PER. The following would select into this rowset:

```
Local Rowset &BUS_EXPENSES_PER;

&BUS_EXPENSES_PER = GetRowset (SCROLL.BUS_EXPSNESE_PER) ;

&BUS_EXPENSES_PER.Select (RECORD.BUS_EXPENSE_VW, "WHERE SETID = :1 and CUST_ID = :2", SETID, CUST_ID) ;
```

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.



Note to developers familiar with previous releases of PeopleCode: In this situation, the Select method is acting like the RowScrollSelect function.

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.

In the following example, rows are selected into a child rowset BUS_EXPENSE_DTL, matching level-one keys, and with the charge amount equal to or exceeding 200, sorting by that amount:

```
Local Record &REC_EXP;

Local Rowset &BUS_EXPENSE_PER;
```

```

&REC_EXP = GetRecord(RECORD.BUSINESS_EXPENSE_PER;

&BUS_EXPENSE_PER = GetRowset(Scroll.BUS_EXPENSE_PER);

&BUS_EXPENSE_PER.Select(Scroll.BUS_EXPENSE_DTL, RECORD.BUS_EXPENSE_DTL, "WHERE
%KeyEqual(:1) AND EXPENSE_AMT >= 200 ORDER BY EXPENSE_AMT", &REC_EXP);

```

Specifying the Select Record

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 executing the method, or it can be a different record definition that has compatible fields.

The select record must be defined in Application Designer and be a built SQL table or view (using **Build, Project**), unless the select record is the same record as the primary database record associated with the rowset.

The select record 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.

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 you actually need.

The WHERE Clause

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 **SelectNew** 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.

To avoid errors, the WHERE clause should explicitly select matching key fields on parent and child rows. This is easy to do using the %KeyEqual meta-SQL.



For more information about the meta-SQL, see %KeyEqual.

Using Select like RowScrollSelect

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.



Note to developers familiar with previous releases of PeopleCode: In this situation, the Select method is acting like the RowScrollSelect function.

If you qualify the lower level rowset such that it only returns one row, it will be acting like a RowScrollSelect.

```
&RSLVL1 = GetRowset (SCROLL.PHYSICAL_INV) ;

&RSLVL2 = &RSLVL1 (&PHYSICAL_ROW) .GetRowset (SCROLL.PO_RECEIVED_INV) ;

&REC2 = &RSLVL2.PO_RECEIVED_INV;

If &PO_ROW = 0 Then

    &RSLVL2.Select (PO_RECEIVED_INV, "WHERE %KeyEqual (:1) and qty_available > 0",
    &REC2) ;

End-if;
```

Using Standalone Rowsets

Standalone Rowsets are like regular Rowsets except they aren't tied to a Component or Page. Use them when you need to work on data that isn't tied to a component or page buffer. Prior to this release, this was done using work records. You still must build work pages.



Standalone rowsets are *not* connected to the Page Processor, 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.

Just like any PeopleTools object the scope of standalone rowsets can be Local, Global or Component. Consider the following code:

```
Local Rowset &MYRS;

&MYRS = CreateRowset (RECORD.SOMEREC) ;
```

This creates a Rowset with SOMEREC as the Level 0 record. The Rowset is unpopulated. Functionally it is the same as an array of records.

To populate the standalone Rowsets you can use the Fill rowset method, the CopyTo rowset method or the record methods.

The Fill Method

The Fill method fills the rowset by reading records from the database, by first flushing out all the contents of the rowset. A where clause needs to be provided to get all the relevant rows.

```
Local Rowset &MYRS;  
  
Local String &EMPLID;  
  
&MYRS = CreateRowset (RECORD.SOMEREC) ;  
  
&EMPLID = '8001';  
  
&MYRS.Fill("where EMPLID = :1", &EMPLID);
```

The CopyTo Method

The CopyTo method copies like-named fields from a source rowset to a destination rowset. To perform the copy it uses like-named records for matching, unless specified. It works on any rowset except the Application Engine state records.

```
Local Rowset &MYRS1, MYRS2;  
  
Local String &EMPLID;  
  
&MYRS1 = CreateRowset (RECORD.SOMEREC) ;  
  
&MYRS2 = CreateRowset (RECORD.SOMEREC) ;  
  
&EMPLID = '8001';  
  
&MYRS1.Fill("where EMPLID = :1", &EMPLID);  
  
&MYRS1.CopyTo (&MYRS2);
```

Now &MYRS2 contains that same data as &MYRS1. In this case both &MYRS1 and &MYRS2 were build using like named records.

If you wish to use the CopyTo method where there are no like-named records, you need to specify the source and destination records. The following code will copy only like-named fields.


```

Local Rowset &MYRS1, MYRS2;

Local String &EMPLID;

&MYRS1 = CreateRowset (RECORD.SOMEREC1);

&MYRS2 = CreateRowset (RECORD.SOMEREC2);

&EMPLID = '8001';

&MYRS1.Fill("where EMPLID = :1", &EMPLID);

&MYRS1.CopyTo(&MYRS2, RECORD.SOMEREC1, RECORD.SOMEREC2);

```

Adding Child Rowset

The first parameter of the CreateRowset determines the top-level structure. In the previous examples we passed in the name of the record as the first parameter, thus the rowset was based on a record. You can also base the structure on a different rowset. In the following example &MYRS2 inherits the structure of &MYRS1.

```

Local Rowset &MYRS1, MYRS2;

&MYRS1 = CreateRowset (RECORD.SOMEREC1);

&MYRS2 = CreateRowset (&MYRS1);

```

To add a child rowset let us consider the following records to describe a relationship. The structure is made up of three records.

PERSONAL_DATA

BUS_EXPENSE_PER

BUS_EXPENSE_DTL

Here's how you build rowsets with child rowsets

```

Local Rowset &rsBusExp, &rsBusExpPer, &rsBusExpDtl;

&rsBusExpDtl = CreateRowset (Record.BUS_EXPENSE_DTL);

```

```

&rsBusExpPer = CreateRowset (&rsBusExpDtl, Record.BUS_EXPENSE_PER);

&rsBusExp = CreateRowset (&rsBusExpPer, Record.PERSONAL_DATA);

```

Another variation is

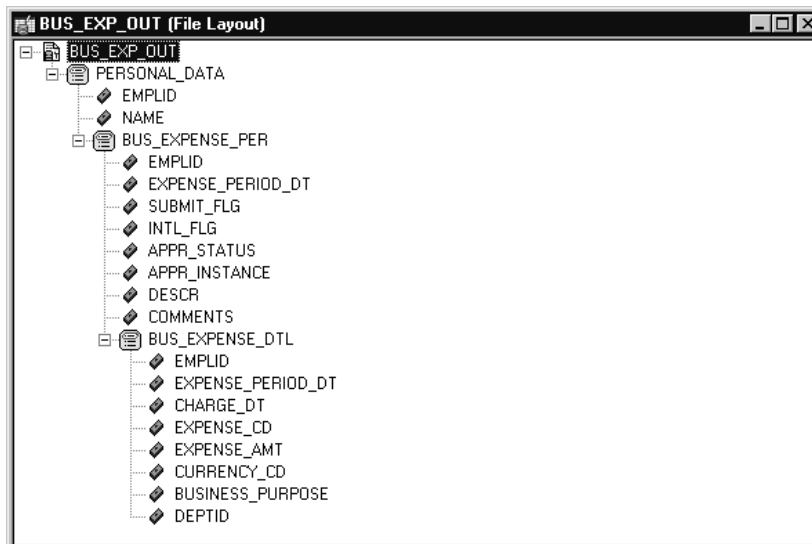
```

&rsBusExp = CreateRowset (Record.PERSONAL_DATA,
CreateRowset (Record.BUS_EXPENSE_PER, CreateRowset (Record.BUS_EXPENSE_DTL)));

```

Using Standalone Rowsets to Write a File

The following example writes a file using a file layout that contains parent-child records.



File Layout for example

```

Local File &MYFILE;

Local Rowset &rsBusExp, &rsBusExpPer, &rsBusExpDtl;

Local Record &rBusExp, &rBusExpPer, &rBusExpDtl;

Local SQL &SQL1, &SQL2, &SQL3;

&rBusExp = CreateRecord(Record.PERSONAL_DATA);

&rBusExpPer = CreateRecord(Record.BUS_EXPENSE_PER);

&rBusExpDtl = CreateRecord(Record.BUS_EXPENSE_DTL);

```

```

&rsBusExp = CreateRowset (Record.PERSONAL_DATA,
CreateRowset (Record.BUS_EXPENSE_PER, CreateRowset (Record.BUS_EXPENSE_DTL)));

&rsBusExpPer = &rsBusExp.GetRow(1).GetRowset(1);


&MYFILE = GetFile("c:\temp\BUS_EXP.out", "W", %FilePath_Absolute);

&MYFILE.SetFileLayout (FileLayout.BUS_EXP_OUT);


&EMPLID = "8001";


&SQL1 = CreateSQL("%selectall(:1) where EMPLID = :2", &rBusExp, &EMPLID);
&SQL2 = CreateSQL("%selectall(:1) where EMPLID = :2", &rBusExpPer, &EMPLID);

While &SQL1.Fetch(&rBusExp)

    &rBusExp.CopyFieldsTo(&rsBusExp.GetRow(1).PERSONAL_DATA);

    &I = 1;

    While &SQL2.Fetch(&rBusExpPer)

        &rBusExpPer.CopyFieldsTo(&rsBusExpPer(&I).BUS_EXPENSE_PER);

        &J = 1;

        &SQL3 = CreateSQL("%selectall(:1) where EMPLID = :2 and EXPENSE_PERIOD_DT
= :3", &rBusExpDtl, &EMPLID,
&rsBusExpPer(&I).BUS_EXPENSE_PER.EXPENSE_PERIOD_DT.Value);

        &rBusExpDtl = &rBusExpPer.GetRow(&I).GetRowset(1);

        While &SQL3.Fetch(&rBusExpDtl)

            &rBusExpDtl.CopyFieldsTo(&rsBusExpDtl(&J).BUS_EXPENSE_DTL);

            &rsBusExpDtl.InsertRow(&J);

            &J = &J + 1;

        End-While;

        &rsBusExpPer.InsertRow(&I);

        &I = &I + 1;

    End-While;

```

```

        &MYFILE.WriteRowset (&rsBusExp);

End-While;

&MYFILE.Close();

```

The above code generates the following output file.

AA8001	Schumacher, Simon	
BB8001	06/11/1989YNA0	Customer Go-Live Celebration
CC8001 10100	06/11/198906/01/198908226.83	USDEntertain Clients
BB8001	08/31/1989YNA0	Customer Focus Group Meeting
CC8001 10100	08/31/198908/11/1989012401.58	USDCustomer Visit
CC8001 10100	08/31/198908/12/198904250.48	USDCustomer Visit
CC8001 10100	08/31/198908/12/198902498.34	USDCustomer Visit
BB8001	03/01/1998YYP0	Attend Asia/Pacific Conference
CC8001 00001	03/01/199802/15/1998011200	USDConference
CC8001 00001	03/01/199802/16/19980220000	JPYConference
BB8001	05/29/1998NNP0	Annual Subscription
CC8001 10100	05/29/199805/29/199814125.93	USDSoftware, Inc.
BB8001	08/22/1998NNP0	Regional Users Group Meeting
CC8001 10100	08/22/199808/22/19981045.69	USDDrive to Meeting
CC8001 10100	08/22/199808/22/19980912.44	USDCity Parking
BB8001	12/12/1998NNP0	Customer Visit: Nevco
CC8001 00001	12/12/199812/02/199801945.67	USDCustomer Feedback
CC8001 00001	12/12/199812/02/19981010.54	USDTo Airport

CC8001 00001	12/12/199812/03/19980610	USDAirport Tax
CC8001 00001	12/12/199812/03/199804149.58	USDCustomer Feedback
CC8001 00001	12/12/199812/04/1998055.65	USDCheck Voicemail
CC8001 00001	12/12/199812/04/19980988	USDAirport Parking
CC8001 00001	12/12/199812/04/199802246.95	USDCustomer Feedback
CC8001 00001	12/12/199812/04/199803135.69	USDCustomer Feedback

Using Standalone Rowsets to Read a File

The following code can be used to read in the file created above and insert the rows into the database.

```

Local File &MYFILE;

Local Rowset &rsBusExp, &rsBusExpPer, &rsBusExpDtl;

Local Record &rBusExp, &rBusExpPer, &rBusExpDtl;

Local SQL &SQL1;

&rBusExp = CreateRecord(Record.PERSONAL_DATA);

&rBusExpPer = CreateRecord(Record.BUS_EXPENSE_PER);

&rBusExpDtl = CreateRecord(Record.BUS_EXPENSE_DTL);

&rsBusExp = CreateRowset(Record.PERSONAL_DATA,
CreateRowset(Record.BUS_EXPENSE_PER, CreateRowset(Record.BUS_EXPENSE_DTL)));

&MYFILE = GetFile("c:\temp\BUS_EXP.out", "R", %FilePath_Absolute);

&MYFILE.SetFileLayout(FileLayout.BUS_EXP_OUT);

&SQL1 = CreateSQL("%Insert(:1)");

```

```

&rsBusExp = &MYFILE.ReadRowset();

While &rsBusExp <> Null;

    &rsBusExp.GetRow(1).PERSONAL_DATA.CopyFieldsTo(&rBusExp);

    &rsBusExpPer = &rsBusExp.GetRow(1).GetRowset(1);

    For &I = 1 To &rsBusExpPer.ActiveRowCount

        &rsBusExpPer(&I).BUS_EXPENSE_PER.CopyFieldsTo(&rBusExpPer);

        &rBusExpPer.ExecuteEdits(%Edit_Required);

        If &rBusExpPer.IsEditError Then

            For &K = 1 To &rBusExpPer.FieldCount

                &MYFIELD = &rBusExpPer.GetField(&K);

                If &MYFIELD.EditError Then

                    &MSGNUM = &MYFIELD.MessageNumber;

                    &MSGSET = &MYFIELD.MessageSetNumber;

                End-If;

            End-For;

        Else

            &SQL1.Execute(&rBusExpPer);

            &rsBusExpDtl = &rsBusExpPer.GetRow(&I).GetRowset(1);

            For &J = 1 To &rsBusExpDtl.ActiveRowCount

                &rsBusExpDtl(&J).BUS_EXPENSE_DTL.CopyFieldsTo(&rBusExpDtl);

                &rBusExpDtl.ExecuteEdits(%Edit_Required);

                If &rBusExpDtl.IsEditError Then

                    For &K = 1 To &rBusExpDtl.FieldCount

                        &MYFIELD = &rBusExpDtl.GetField(&K);

                        If &MYFIELD.EditError Then

                            &MSGNUM = &MYFIELD.MessageNumber;

                            &MSGSET = &MYFIELD.MessageSetNumber;

                        End-If;

                    End-For;

                End-If;

            End-For;

        End-If;

    End-While;

```

```

        End-For;

    Else

        &SQL1.Execute (&rBusExpDtl);

    End-If;

End-For;

End-If;

End-For;

&rsBusExp = &MYFILE.ReadRowset();

End-While;

&MYFILE.Close();

```

Errors and Warnings

For the most part, errors and warnings display messages to users informing them about invalid data. For this reason, they are almost always placed in FieldEdit or SaveEdit PeopleCode, or in SearchSave PeopleCode for validation during search processing. In conjunction with edits, errors stop processing, while warnings allow processing to continue. When errors and warnings appear in places other than FieldEdit or SaveEdit, their effects vary.

Syntax of Errors and Warnings

Errors and warnings only require a message that the Component Processor displays to users. You can code the message into the error or warning statement, or you can use Message Catalog. Peoplesoft recommends using Message Catalog with the MsgGet, MsgGetExplainText, and so on.

Error and warning use the same syntax. For example:

```

Error MsgGet(11100, 180, "Message not found.");
Warning MsgGet(11100, 180, "Message not found.");

```

Errors, Warnings, and Edits

You can use the following PeopleCode events for validation edits: FieldEdit and SaveEdit. The Component Processor applies FieldEdit when the user changes a field, and SaveEdit when the user saves the component. Errors and warnings in these events display a message to the user. Most errors and warnings appear in these event types, although you can use them elsewhere.

For FieldEdit, you can use either the record field or component record field event. Remember that the record field event for each record fires before the component record field event for that

record. For SaveEdit, you can use the record field or the component record event. Remember that all record field events for a record fire before the component record events.

Errors and Warnings in FieldEdit

An error in FieldEdit prevents the system from accepting the new value of a field. The Component Processor highlights the offending field. The user must either change the field back to its original value or to something else which does not trigger the error. A warning allows the Component Processor to accept the new data. The Component Processor does not highlight any field that has warnings.

Errors and Warnings in SaveEdit

An error in SaveEdit prevents the system from saving any row of data. The Component Processor does not update the database for any field if one field has an error. Although the Component Processor displays an error message, it does not turn any field red. Unlike FieldEdit errors, SaveEdit errors can happen anywhere on the page or component, for any row of data. The data causing the error may appear on a different page within the same group, or a row of data not currently displayed. If this is the case, the field in error is brought into view by the system.

A warning in SaveEdit also gets applied to all data in the page or component, but the Component Processor will accept the data, if told to by the user. In a FieldEdit warning, the Component Processor displays a message box with the text and two push buttons – OK and the standard Explain (the Explain push button will return an explanation for the last message retrieved with the **MsgGet** function). In a SaveEdit warning, though, the message box contains an additional push button, Cancel. OK accepts the data, overriding the warning and continuing the save process. Cancel aborts the save process.

Because errors and warnings apply to all rows of data and all pages in a group, you must provide the user explicit information about what caused the error. Typically, you'll use the message catalog function to store messages and substitute variables into them. However, you can also facilitate this by concatenating in a field value. For example, if you have a stack of historical data on the page, you could use the following error statement:

```
Error ("The value exceeds the maximum on "|effdt|".");
```

Errors and Warnings in RowSelect



Errors and warnings should no longer be used in RowSelect processing: instead, use **DiscardRow** and **StopFetching**. The behavior of **Error** and **Warning** in RowSelect PeopleCode, as described here, is retained for compatibility with previous releases of PeopleTools.

RowSelect PeopleCode filters out rows of data after the system applies search record criteria. It also can stop the Component Processor from reading additional rows of data.

A warning causes the Component Processor to reject the current row, but the Component Processor does continue reading more data. An error prevents any more data coming into the page or component. The Component Processor accepts the row that causes the error, but doesn't read any more data. If you want to reject the current row and stop loading additional rows, you must issue a warning and an error.

You must specify text for an error or warning, but the Component Processor does not display messages from RowSelect. You can still use the message text as a way of documenting your program.



For more information see Understanding PeopleCode and Events, DiscardRow and StopFetching.

Errors and Warnings in RowDelete

When you delete a row of data (F8), the system prompts you to confirm. If you do confirm, RowDelete PeopleCode takes place, that is, any record field RowDelete PeopleCode will fire, and any component record RowDelete PeopleCode. Errors and warnings in RowDelete display a message box.

A warning from RowDelete presents two choices – accept the RowDelete (the OK push button), or cancel the RowDelete (the Cancel push button). Maybe after they read the warning message, the user will think better of the RowDelete. An error from RowDelete PeopleCode prevents the Component Processor from removing that row of data from the page.

Errors and Warnings in Other Events

You should not put errors or warning in PeopleCode attached to any of the remaining events (FieldDefault, FieldFormula, RowInit, FieldChange, RowInsert, SavePreChange, WorkFlow, and SavePostChange). However, the Component Processor may issue its own errors and warnings when performing PeopleCode. When this happens, it means the Component Processor tried to do something and failed, and it cannot recover from the error.

These Event types activate processing that an user has no direct control over. So, if the Component Processor comes across an error condition, the user cannot fix it. The Component Processor cancels the transaction to avoid unpredictable results.



For more information about these functions, see Warning and Error.

Using RemoteCall

RemoteCall is a PeopleTools feature that provides a means of executing a COBOL program remotely from within a PeopleSoft application. Remote calls are made using the **RemoteCall** PeopleCode function.

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 useful, because it provides a way to take advantage of existing COBOL processes.

There have been no changes to the syntax of the **RemoteCall** function, but the rules about where **RemoteCall** runs have been greatly simplified:

- In three-tier mode, **RemoteCall** always runs on the application server.
- In two-tier mode, **RemoteCall** always runs on the client.

As a result of this simplification, some features of PeopleTools 6 have been eliminated:

- There will be no more Location setting for remote calls in Configuration Manager.
- There is no longer a need to set or test network connections in the PeopleTools Options utility.
- There is no longer a capability of running in a "mixed mode" where there is a database connection and a Tuxedo connection on the client.



For more information see PeopleCode and PeopleSoft Internet Architecture.

The **RemoteCall** function:

- Is a synchronous call. The client passes parameters to the remote program, and then waits while the program runs. When the remote program is done, it returns any results or status information to the client, which then resumes execution. This means that **RemoteCall** is a "think-time" function and subject to certain restrictions.



For more information see Think-Time Functions.

- Is designed for fast response time, and has an API that provides programs with the response time needed for transaction processing. It is especially useful for SQL-intensive processes that do not run efficiently on the client. (Note that PeopleCode **SQLExec** and other SQL intensive PeopleCode can also be run on the application server in PeopleTools 7 or greater.)
- Can be executed with a command push button or from a pop-up menu.
- **RemoteCall** has no scheduling or multi-step job capabilities. Each execution of RemoteCall is independent of others.
- Allows you to reuse existing COBOL code.
- For PeopleTools 8.0, you can no longer use **RemoteCall** to execute an Application Engine program. Use the **CallAppEngine** function instead.

RemoteCall Components

The RemoteCall PeopleTools feature consists of the following components:

- **PeopleCode API.** This interface consists of the **RemoteCall** PeopleCode function. It is used from PeopleCode to start a remote program and handle any results. The PeopleCode client program does not include any special code to specify where the remote program is executed. TUXEDO can be configured to locally execute the program for testing purposes.
- **Remote Program API.** This is used by the remote COBOL program to receive or pass parameters and return status information.
- **PeopleSoft RemoteCall Service.** The PeopleSoft application server, PSAPPSRV, advertises the RemoteCall service. The service receives requests from clients and starts the requested program. When the program is completed, it passes the parameters and status code back to the client.
- **TUXEDO.** TUXEDO is a message-based transaction monitor for distributed applications. No direct TUXEDO calls need to be implemented in PeopleCode or remote programs.

PeopleCode API

You can execute the **RemoteCall** function from PeopleCode associated with any Component Processor event except SavePostChange, SavePreChange, Workflow, RowSelect, or in any PeopleCode event resulting from a **ScrollSelect** or related function call. However, remote programs that change data should not be run as part of a SaveEdit process, because the remote program may complete successfully even though an error occurs later in the save process.

The preferred method of calling a remote program that changes data is in FieldChange PeopleCode in a record field associated with a command push button, or from a pop-up menu item.

You should not use **RemoteCall** if you expect the remote program to return a large amount of data to the client, because data is passed back only through the parameters of the PeopleCode API.



For more information about the function see RemoteCall.

Authorization

Authorization to run a remote program is like authorization to run a PeopleCode program. Because a remote program is started from PeopleCode, the user has authorization in User Security to use the page that executes the PeopleCode.

Unit of Work

The remote program runs in a different unit of work from the page. A commit will be issued by PeopleTools if needed on the client before **RemoteCall** is called. This means that, by default, the remote program is not privy to any database changes unless the page is saved before the program is called. Once the remote program starts, it will run to completion and commit or abort before returning to the page. In this way, the remote program and the page will not have locking contention. To ensure that the save has actually been done, use the **DoSaveNow** built-in function.

Error Processing

When using RemoteCall to execute a COBOL program, two types of errors can occur:

- **PeopleTools errors.** An error could occur in PeopleTools or TUXEDO, or the service might not be found. These are treated as hard errors by PeopleCode. An error message box will be displayed, and that piece of PeopleCode will be terminated. So, in the case of a PeopleTools error, the remote program will always either return a Return code of zero, or terminate with a message due to a system error.
- **Application-specific errors.** Any error information specific to the remote application must be passed back in regular data variables, and your application can handle these in an application-specific way. If you have a status code on which your application depends, you should initialize it to an invalid value to be sure the COBOL program does indeed return the status code.

Client Execution Environment

Because the remote program is executed synchronously, clients will get an hourglass icon and will not be able to do anything in the current window until the remote application completes. They could move to another window and do processing there; or they could start up another PeopleSoft window. They will not be able to cancel the remote program once it starts. If the program does not terminate in a timely fashion (as determined by the RemoteCall timeout set with Configuration Manager), RemoteCall attempts to terminate the process and returns an error indicating that the program was terminated.

Remote Program API

The Remote Program API provides the functions to get and put data between the network and the COBOL program. These functions are implemented in C, but are callable from COBOL through the PTPNETRT program. For an example, see the PTPNTEST.CBL program.



If these APIs are called when the program is not running as a remote program, ACTION-GET and ACTION-PUT will return an error. All other actions will just return without doing anything.

Error Processing

If an unexpected error is found, call PTPNETRT with ACTION-RESET, then with ACTION-PUT to send back any error status variables, then with ACTION-DONE to send the buffer.

PeopleSoft RemoteCall Service

The RemoteCall Service serves as a bridge between the PeopleCode API and the remote COBOL programs. In PeopleTools 6, RemoteCall was the only PeopleSoft TUXEDO service. In PeopleTools 7, RemoteCall is one of many services advertised from the PSAPPSRV TUXEDO server, and can be configured as part of the standard domain setup and administration.

The client sends the RemoteCall service request, consisting of the connect information and the program name, as well as any other parameters for the program, to the application server. The RemoteCall service then executes the program and passes it the connect string.

RemoteCall and Process Scheduler

COBOL application programs initiated by the RemoteCall service use the same COBOL application architecture used by Process Scheduler. Once initiated by the Dispatcher, COBOL application programs call the COBOL SQL API program, PTPSQLRT, to connect to the RDBMS to compile and execute SQL statements. It is therefore fairly easy to design and implement COBOL programs to be "bilingual" with respect to Process Scheduler and RemoteCall.

The following guidelines will help you to choose the optimal method for running a particular COBOL program:

- Use Process Scheduler for asynchronous processes, or processes that can be scheduled, are multi-step, or that require printed output.
- Use RemoteCall for synchronous processes that are quick ("transaction processing" types of processes).

Modifying a Process Scheduler Program to Use RemoteCall

To enable an existing program that runs under the Process Scheduler to run under RemoteCall as well, make the following changes:

- Include the PTCNETRT copy member.
- Include the PTCNCHEK member before the connection call to PTPSQLRT.
- Add the call to PTPNETRT ACTION-DONE just before the program terminates (after the call to disconnect from the database). This should be conditional on whether you are RUNNING-REMOTE-CALL.
- If you are running as a RemoteCall, you should be sure that PROCESS-INSTANCE OF PRUNSTATUS is not set. Otherwise your calls to PTCSTAT will try to update the PSPRCSRQST table. This will not cause an error, but it is unnecessary processing.

This program can now run from the process scheduler or from RemoteCall. Note that if a program really wants to pass parameters, it has to have RemoteCall-specific ACTION-GET and ACTION-PUT calls.

Programming Guidelines

Bear the following points in mind when using RemoteCall:

- Do not use RemoteCall for long-running batch jobs. As a rule of thumb, if you think execution will take more than 15 seconds, you should not be using RemoteCall, but should instead use the Process Scheduler.
- RemoteCall is meant for running jobs on the server. It should not be used to invoke client-only programs. Support for local calling with RemoteCall is provided solely as a debugging and development aid. For client-only programs, use **Declare Function**, then call the external function from a library.
- If you do not want to modify an existing program, then pass only the program name and run control, and do not return any parameters. This way, the program requires few changes to run as a remote function.

CHAPTER 8

Referencing Data in the Component Buffer

PeopleCode frequently needs to refer to data in the Component Buffer—the area in memory that stores data for the currently active component.

This chapter provides some basic information about the structure and contents of the Component Buffer, then describes two means of specifying a piece of data in the Component Buffer from within PeopleCode:

- contextual references, which refer to data relative to the location of the currently executing PeopleCode program.
- references using scroll path syntax, which provide a complete—or absolute—path through the Component Buffer to the referenced component.

In addition to the built-in functions used to access the Component Buffer, PeopleCode provides enhanced access to structured data buffers using the new object syntax. You can use the object based PeopleCode to resolve contextual ambiguities when you reference a non-primary record field that appears on more than one scroll level in a component. Like built-in functions, object syntax provides for both relative and absolute references to Component Buffer data.



For more information about object based programming, see Understanding Objects and Classes in PeopleCode.

Component Buffer Structure and Contents

The Component Buffer is a structure in memory that stores all data for the currently active component. The Component Buffer consists of rows of buffer fields that hold data for the various records associated with page controls, including primary scroll records, related display records, derived/work records, and translate table records. PeopleCode can reference buffer fields associated with page controls as well as other buffer fields from the primary scroll record and related display records: for details see What Record Fields Are in the Component Buffer?

Primary scroll records are the principal SQL table or view associated with a page scroll level. A primary scroll record uniquely identifies a scroll level in the context of its page: each scroll level can have only one primary scroll record; and the same primary scroll record cannot occur on more than one scroll at the same level of the page. Parent-child relations among primary scroll records determine the dependency structure of the scrolls on the page. That is, the primary record

on a level one scroll must be a child of the primary record on level zero; the primary record on a level two scroll must be a child of the primary record on its enclosing level one scroll; and the primary record on a level three scroll must be a child of the primary record on its enclosing level two scroll.



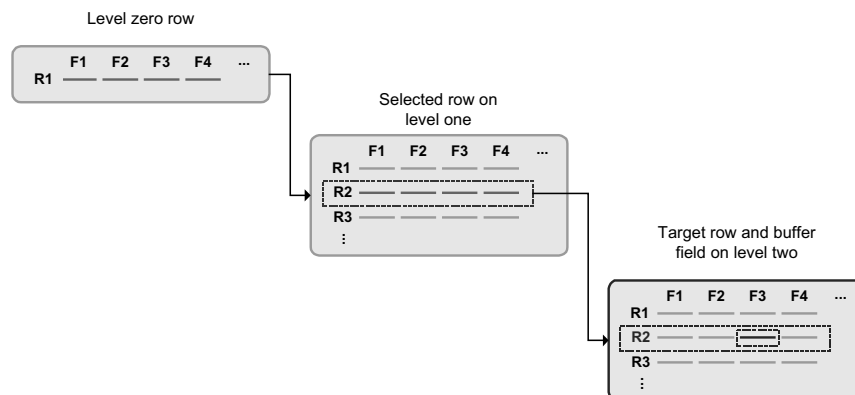
Level 0 may have multiple records.

The hierarchical relations among scrolls, controlled by hierarchical relations among primary scroll records, enable the end-user and PeopleCode to drill down through the scroll hierarchy to access any specific buffer field, including related display, derived/work, and translate table buffer fields, which occupy space on the same rows as the primary scroll record buffer fields with which they are associated.

For example, to access a page field on level two of a page, the end-user must:

- Select a field on level one of the page.
- Scroll to and select the desired field on level two of the page.

The following diagram illustrates this *scroll path* taken by the user:



Scroll Path to a Buffer Field

To access the same field in the Component Buffer, PeopleCode would need to:

1. Specify a scroll and row on scroll level one: this selects a subset of dependent rows on level two.
2. Specify a scroll and row on scroll level two
3. Specify the specific recordname.fieldname on the level two row.

PeopleCode Component Buffer functions use a common *scroll path syntax* for locating scrolls, rows, and fields in multiple-scroll pages.



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

Comparing Rowsets to Scrolls

The rowset is a new programming construct in PeopleTools 8 which enables more consistent, more convenient and less ambiguous manipulation of buffer data than existing built-in functions can achieve. It's a hierarchical data object that can represent an entire scroll and all of its subordinate scrolls.

A rowset can contain the entire contents of a Component Buffer, or the contents of any lower level scroll plus all of its subordinate buffer data. The hierarchical structure of component levels—scroll, row, record, field—is provided by the new object data types, **Rowset**, **Row**, **Record** and **Field** respectively.



For more information about rowsets and related objects, see [Data Buffer Access](#) and the [Rowset Class](#).

You can access any specific rowset, row, record or field within the buffer using the dot notation inherent in PeopleTools 8 object based programming. This enables you to reference fields within a record object, records within a row object, and rows within a rowset object as properties of the parent objects.



For more information about object based programming, see [Understanding Objects and Classes in PeopleCode](#).

What Record Fields Are in the Component Buffer?

The record fields in the Component Buffer are a superset of those accessible to the end user via page controls. In most cases PeopleCode can reference any record field in a scroll's primary scroll record or in a related display record—not just those fields that are associated with page controls. The following table provides specific information for various record types and locations:

Type and Location of Record	Presence in Component Buffer
Primary record on scroll levels greater than zero	On scroll levels greater than zero all record fields from the primary scroll record are in the Component Buffer. This means that on levels greater than zero PeopleCode can refer to any record field on the primary scroll record, even if it is not associated with a page control.

Primary record on scroll level zero	If scroll level zero of a page contains only controls associated with primary scroll record fields that are search keys or alternate search keys, then only the search key and alternate search key fields will be in the Component Buffer: not the entire record. The values for the field(s) come from the keylist, and the record won't run RowInit PeopleCode. If level zero contains at least one record field from the primary scroll record that is not a search key or alternate search key, then all the record fields from the primary scroll record will be available in the buffer. (For this reason you may sometimes need to add one such record field at level zero of the page to make sure that all the record fields of the level-zero primary record can be referenced from PeopleCode.)
Related display record fields	The buffer contains the related display record field, plus any record fields from the related display record that are referenced by PeopleCode programs. This means that you can reference any record field in a related display record.
Derived/work record fields	Only derived/work record fields associated with page controls are in the Component Buffer. Other record fields from the derived/work record cannot be referenced from PeopleCode.
Translate Table record fields	Only Translate Table fields associated with page controls are available in the Component Buffer. Other fields from the Translate Table cannot be referenced from PeopleCode.



Special limitation on RowSelect PeopleCode. In RowSelect PeopleCode, you can refer only to record fields on the record that is currently being processed.

Contextual References

In a **contextual reference** PeopleCode refers to a row or buffer field determined by the current context; that is, the context in which the PeopleCode program is currently executing.

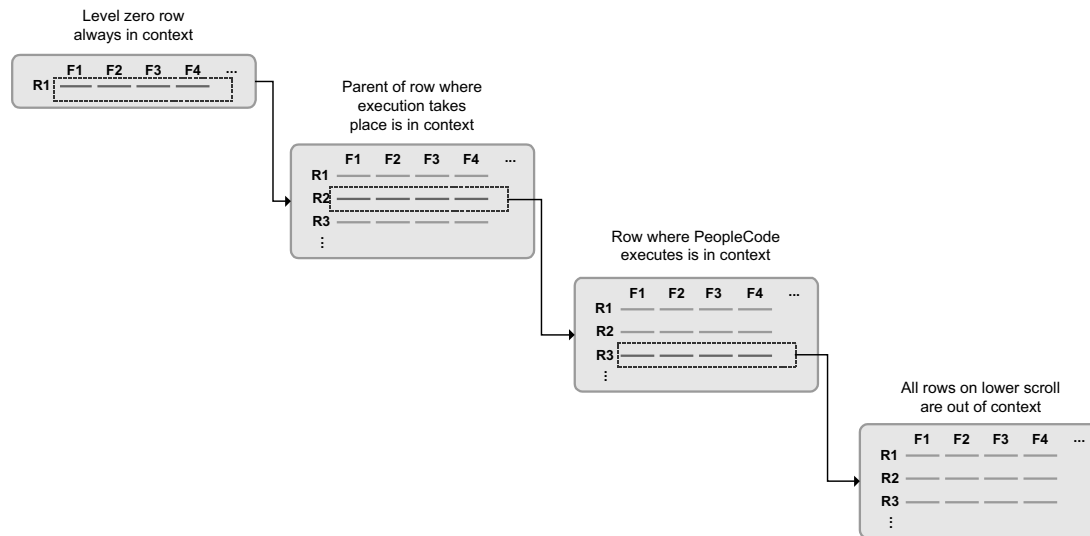
Understanding Current Context

All PeopleCode programs, with the exception of programs associated with standard menu items, execute in a **current context**. The current context determines which buffer fields can be contextually referenced from PeopleCode, and which row of data is the "current row" on each scroll level at the time a PeopleCode program is executing.

The current context is comprised of a subset of the buffer fields in the Component Buffer, determined by the row of data where a PeopleCode program is executing. The current context includes:

- All buffer fields in the row of data where the PeopleCode program is executing.
- All buffer fields in rows that are hierarchically superior to the row where the PeopleCode program is executing.

In the following diagram all rows enclosed in dotted rectangles are part of the current context:



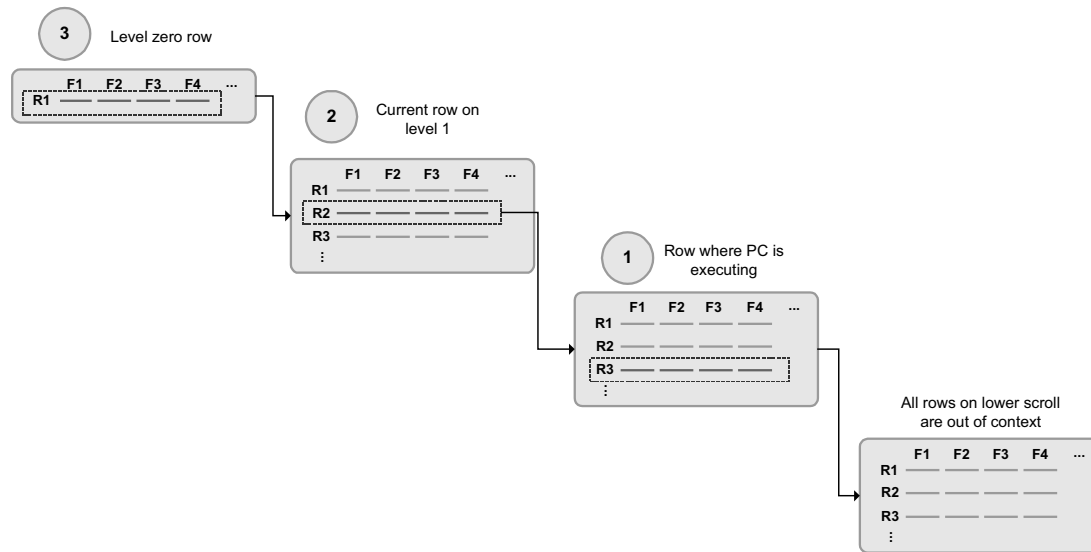
Context of PeopleCode Executing on a Level-Two Scroll

In this example a PeopleCode program is executing in a buffer field on row R3 on scroll level two. The rows in scroll level two are dependent on row R2 on scroll level one. The rows in scroll level one are dependent on the single row at scroll level zero. The current context consists of all the buffer fields at level two row R3, level one row R2, and level 0 row R1. The rows in the current context on levels one and two are the **current rows** on their respective scrolls. The single row on level zero is always current, and is included in any current context. All rows other than the current rows and the level zero row are outside the current context. Note that no current row can be determined on scrolls below the one where the PeopleCode is executing.

With PeopleTools 8, contextual references work within the structure of a rowset object, and can include references to all the field objects, record objects, row objects, and rowset objects in the current context.

Contextual Reference Processing Order

PeopleCode resolves contextual references at runtime by first looking in the row where the PeopleCode program is executing; if it doesn't find an appropriate buffer field, it looks in progressively higher rows in the current context. The following diagram shows this processing order:



Processing Order of a Contextual Reference

In typical pages this processing order is not significant; however, there are some cases where the same record occurs on more than one level of a page, and in these cases it's important to understand exactly how the direct reference is resolved.



For more information see Contextual Buffer Field Reference Ambiguity.

Contextual Row References

A contextual row reference refers to a row in the current context on level one or lower in the page. Because each scroll uses a unique primary record, the name of that record uniquely identifies whichever row is in the current context for that scroll level. A contextual row reference uses a **RECORD.recordname** component name reference to specify the scroll level of the intended row, resulting in a reference to the current row at the specified scroll level.

For example, you can use contextual row references with the **RecordDeleted**, **RecordNew**, and **RecordChanged** functions:

```
If RecordDeleted(RECORD.SOME_REC) Then...
```

With PeopleTools 8 object based programming, the desired row can be referenced by specifying parent rows or rowsets of the current rowset:

```
If GetRowSet().ParentRowset.ParentRow.IsDeleted Then...
```

In early versions of PeopleTools you could make contextual row references using a **recordname.fieldname** expression, like so:

```
HideRow(SOME_REC.ANY_FIELD)
```

```
If RecordDeleted(SOME_REC.ANY_FIELD) Then...
```

This syntax is still supported.



For more information see Understanding Current Context.

Contextual Buffer Field References

A *contextual buffer field reference* is a type of PeopleCode expression that refers to a buffer field by specifying a record field. The row of the buffer field is determined by the current context of the PeopleCode program where the reference is made (see Understanding Current Context). You can use a contextual buffer field reference to retrieve or update the value in the buffer field, to pass the buffer field value to a function, or to reference an instance of a page control associated with the buffer field. The following statements use contextual buffer field references:

```
SOME_RECORD.SOME_FIELD = &VAL; /* Assigns value of variable to buffer field */
```

```
&VAL = SOME_RECORD.SOME_FIELD; /* Assigns value of buffer field to variable */
```

```
Hide(SOME_RECORD.SOME_FIELD); /* Hides instance of control associated with  
buffer field */
```

With PeopleTools 8 object based programming, a field object incorporates information about both the record field on which the buffer field is based, and the page control with which the buffer field is associated. By referring to the field object, you're either making a contextual buffer field reference or you're changing an interface attribute of the associated page control, depending on the object property you use. The following example has the same effect as a contextual buffer field reference:

```
&MYFIELD.Value = &SOMEVAL; /* Assigns value of a variable to a buffer field */
```

Contextual Buffer Field Reference Ambiguity

Non-primary record fields may appear on more than one scroll level in a page. For example, a page may use a derived/work field `DERIVED_JS.CALC_1` as a work field on level one and level two of the same page. This creates distinct `DERIVED_JS.CALC_1` buffer fields for rows on both levels. Because of the order in which PeopleCode resolves contextual buffer field references (see Contextual Reference Processing Order), if the following contextual reference executes in a PeopleCode program on a level-two row:

```
&VAL = DERIVED_JS.CALC_1;
```

it will always retrieve the buffer field value on the current row on level two. PeopleCode on level two would be unable to retrieve the value of the `DERIVED_JS.CALC_1` on level one using a contextual reference.

To explicitly reference the `DERIVED_JS.CALC_1` buffer field on level one, you would need to use a Component Buffer function with a scroll path:

```
&VAL = FetchValue(Scroll.level1_scrollname, CurrentRowNumber(1),
DERIVED_JS.CALC_1);
```

The **CurrentRowNumber** function would return the current row on level one: that is, the parent row of the level two row where the PeopleCode program is executing.



For more information see [References Using Scroll Path Syntax and Dot Notation](#).

Ambiguous Contextual References to Buffer Fields on Level Zero

Level zero of a page contains only a single row of data, and the buffer fields in this row are always in the current context. For this reason you can *almost* always refer to a level-zero buffer field using a contextual reference. However, there are unusual cases when referential ambiguity makes it impossible to reference a buffer field on level zero contextually. For example, a page may use a derived/work field `DERIVED_JS.CALC_1` as a work field on level zero and level one of the same page. This creates distinct `DERIVED_JS.CALC_1` buffer fields for rows on both levels. Because of the order in which PeopleCode resolves contextual field references (see [Contextual Reference Processing Order](#)), if the following contextual reference executes in a PeopleCode program on a level-one row:

```
&VAL = DERIVED_JS.CALC_1;
```

it will always retrieve the buffer field value on the current row on level one.

To explicitly reference the `DERIVED_JS.CALC_1` buffer field on level zero, you must use a Component Buffer function with this syntax:

```
Function([recordname.]fieldname, rownum)
```

Here **rownum**, since it is on level zero, is always equal to 1. In the example you would use this statement:

```
&VAL = FetchValue(DERIVED_JS.CALC_1, 1);
```

Resolving Ambiguous References with Objects

With PeopleTools 8 object based programming, even if two field objects which are in different rowsets contain buffer data that's based on the same underlying record field, references to those objects are inherently unique, because each is instantiated with respect to a specific point in the hierarchy of the buffer. Any manipulation of a field object's interface properties will affect only the page control with which it's associated.

The following example instantiates a field object using the "long" form, to emphasise the hierarchical form of the data. It then hides the field's associated page control. Because this is a

unique instance of the field, based on its hierarchy, hiding this field won't affecting the visibility of any other page control associated with the same record field:

```
&MYFIELD =
GetRowset (SCROLL.EMPL_CHECKLIST) .GetRow (&I) .GetRecord (RECORD.EMPL_CHECKLIST) .Get
Field (EMPL_CHECKLIST.EMPLID) ;

&MYFIELD.Visible = False;

/* the same code, using the "short" form */

&MYFIELD = GetRowset (SCROLL.EMPL_CHECKLIST) .GetRow (&I) .EMPL_CHECKLIST.EMPLID;
```



Any change in a field object's **value** will affect both the underlying record field and the value of any other field object based on the same record field. This behavior is the same as the behavior of contextual buffer field references that alter the field value.

References Using Scroll Path Syntax and Dot Notation

A *scroll path* is a construction found in the parameter lists of many Component Buffer functions, which specifies a scroll level in the currently active page. Additional parameters are required to locate a row or a buffer field at the specified scroll level.

The scroll path construction was enhanced in PeopleTools 5 and again in PeopleTools 7.5. PeopleTools 7.5 and 8 support all previous versions of the scroll path syntax, and there is no requirement to update the syntax in existing applications. The older variants of this construction are covered under Scroll Path Syntax prior to PeopleTools 7.5.

PeopleTools 7.5 scroll path syntax provides the ability to eliminate ambiguous references, which, although rare, do sometimes occur in complex components. The problem of scroll path ambiguity is explained in more detail under Contextual Buffer Field References.

PeopleTools 8 adds the convenience of object based dot notation and default methods, which produce inherently non-ambiguous references, to your PeopleCode programs. You'll find examples of dot notation in this section along with examples of the scroll path syntax available in PeopleTools 7.5, which is still valid in PeopleTools 8.



For more information about object based access to data, see Data Buffer Access.

Scroll Path Syntax in PeopleTools 7.5

PeopleTools 7.5 offers two constructions for scroll paths: a standard scroll path syntax, which in all cases except data buffer references is identical to the syntax in PeopleTools 6 and 7; and an alternative syntax using a **SCROLL.scrollname** expression. The latter is more robust in that it


can handle some rare cases where a **RECORD.recordname** expression results in an ambiguous reference.

Scroll Path Syntax with RECORD.recordname

Here is the standard scroll path syntax:

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

This scroll path syntax is the same as in PeopleTools 6 and 7, except when the referenced component is a buffer field.




For more information on earlier versions of scroll path syntax, see Scroll Path Syntax prior to PeopleTools 7.5.

If the target level (the level you want to reference) is one, you need to supply only the **RECORD.target_recname** parameter. If the target scroll level is greater than one, you need to provide scroll name and row level parameters for all hierarchically superior scroll levels, beginning at level one. The following table shows scroll path syntax for the three possible target scroll levels:

Target Level	Scroll Path Syntax
1	RECORD.target_recname
2	RECORD.level1_recname, level1_row, RECORD.target_recname
3	RECORD.level1_recname, level1_row, RECORD.level2_recname, level2_row, RECORD.target_recname

If you are referring to a row or a buffer field, additional parameters are required after the scroll path.



For more information see Scroll Level, Row, and Buffer Field References.

Standard Scroll Path Syntax Parameters

RECORD.level1_recname

Specifies the name of a record associated with scroll level one, normally the primary scroll record. This parameter is required if the target scroll level is two or three.

level1_row

An integer that selects a row on scroll level one. This parameter is required if the target scroll level is two or three.

RECORD.level2_recname	Specifies the name of a record associated with scroll level two, normally the primary scroll record. This parameter is required if the target row is on scroll level three.
level2_row	An integer that selects a row on scroll level two. This parameter is required if the target row is on scroll level three.
RECORD.target_recname	Specifies a record associated with the target scroll level, generally the primary scroll record. The scroll can be on level one, two, or three of the active page.

Scroll Path Syntax with **SCROLL.scrollname**

As an alternative to **RECORD.recordname** expressions in scroll path constructions, PeopleTools 7.5 permits use of a new **SCROLL.scrollname** expression. Scroll paths using **SCROLL.scrollname** are functionally identical to those using **RECORD.recordname**, except that **SCROLL.scrollname** expressions are more strict: they can refer only to a scroll level's primary record; whereas **RECORD.recordname** expressions can refer to any record in the scroll level, which in some rare cases can result in ambiguous references. (This can occur, for example, if the **RECORD.recordname** expression inadvertently references a related display record in another page in the component.) Use of **RECORD.recordname** is still permitted, and there is no requirement to use the **SCROLL.scrollname** alternative unless it is needed to avoid an ambiguous reference.

The *scrollname* is the same as the scroll level's primary record name

The scroll name *cannot* be viewed or changed through the Application Designer interface. Here is the complete scroll path syntax using **SCROLL.scrollname** expressions:

```
[SCROLL.level1_scrollname, level1_row, [SCROLL.level2_scrollname, level2_row,
], SCROLL.target_scrollname
```

The target scroll level in this construction is the scroll level that you want to specify. If the target level is one, you only need to supply the **SCROLL.target_scrollname** parameter. If the target scroll level is greater than one, you need to provide scroll name and row level parameters for hierarchically superior scroll levels, beginning at level one. The following table shows scroll path syntax for the three possible target scroll levels:

Target Level	Scroll Path Syntax
1	SCROLL.target_scrollname
2	SCROLL.level1_scrollname , level1_row, SCROLL.target_scrollname
3	SCROLL.level1_scrollname , level1_row, SCROLL.level2_scrollname , level2_row, SCROLL.target_scrollname

If the component you are referring to is a row or a buffer field, additional parameters are required after the scroll path.



For more information see Scroll Level, Row, and Buffer Field References.

Alternative Scroll Path Syntax Parameters

SCROLL.level1_scrollname Specifies the name of the page's level-one scroll. This is always the same as the name of the scroll level's primary scroll record. This parameter is required if the target scroll level is two or three.

level1_row An integer that selects a row on scroll level one. This parameter is required if the target scroll level is two or three.

SCROLL.level2_scrollname Specifies the name of the page's level-two scroll. This is always the same as the name of the scroll level's primary scroll record. This parameter is required if the target row is on scroll level three.

level2_row An integer that selects a row on scroll level two. This parameter is required if the target row is on scroll level three.

SCROLL.target_scrollname The scroll name of the target scroll level, which can be level one, two, or three of the active page.

Scroll Level, Row, and Buffer Field References

You can reference a scroll level using the *scrollpath* construct alone. Functions that reference rows for buffer fields require additional parameters. The following table summarizes the three types of Component Buffer reference:

Target Component	Reference Syntax	Example Function
Scroll level	scrollpath	HideScroll(scrollpath);
Row	scrollpath, row_number	HideRow(scrollpath, row_number);
Field	scrollpath, row_number, [recordname.]fieldname	FetchValue(scrollpath, row_number, fieldname);



For more information on scroll path constructions see Scroll Path Syntax in PeopleTools 7.5. For information on scroll path constructions before PeopleTools 7.5 see Scroll Path Syntax prior to PeopleTools 7.5.

PeopleTools 8 provides an alternative to the scroll level, row and field components in the form of the data buffer classes **Rowset**, **Row**, **Record** and **Field**, which you reference using dot notation with object methods and properties. The following table demonstrates the syntax for instantiating and manipulating objects in the current context from these classes:

Target Object	Example Instantiation	Example Operation
Rowset	<code>&MYROWSET = GetRowset();</code>	<code>&MYROWSET.Refresh();</code>
Row	<code>&MYROW = GetRow();</code>	<code>&MYROW.CopyTo(&SOMEROW);</code>
Record	<code>&MYRECORD = GetRecord();</code>	<code>&MYRECORD.CompareFields(&SOMERECORD);</code>
Field	<code>&MYFIELD = GetRecord().fieldname;</code>	<code>&MYFIELD.Label = "Last Name";</code>

The following sections provide examples of functions using scroll path syntax, which refer to an example page from a fictitious veterinary clinic database. The page has three scroll levels, shown in the following table:

Level	Scroll Name (= Primary Scroll Record Name)
0	VET
1	OWNER
2	PET
3	VISIT

The examples given for PeopleTools 8 object based syntax assumes that the following initializing code has been executed:

```
Local Rowset &VET_SCROLL, &OWNER_SCROLL, &PET_SCROLL, &VISIT_SCROLL;
```

```
&VET_SCROLL = GetLevel0();
```

```
&OWNER_SCROLL = &VET_SCROLL.GetRow(1).GetRowSet(SCROLL.OWNER);
```

```
&PET_SCROLL = &OWNER_SCROLL.GetRow(2).GetRowSet(SCROLL.PET);
```

```
&VISIT_SCROLL = &PET_SCROLL.GetRow(2).GetRowSet(SCROLL.VISIT);
```

Referring to Scroll Levels

The **HideScroll** function provides an example of a reference to a scroll level. The syntax of the function is:

```
HideScroll(scrollpath)
```

where *scrollpath* is

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

To reference the level-one scroll in the example, you would use this syntax:

```
HideScroll (RECORD.OWNER) ;
```

This would hide the OWNER, PET, and VISIT scrolls on the example page.

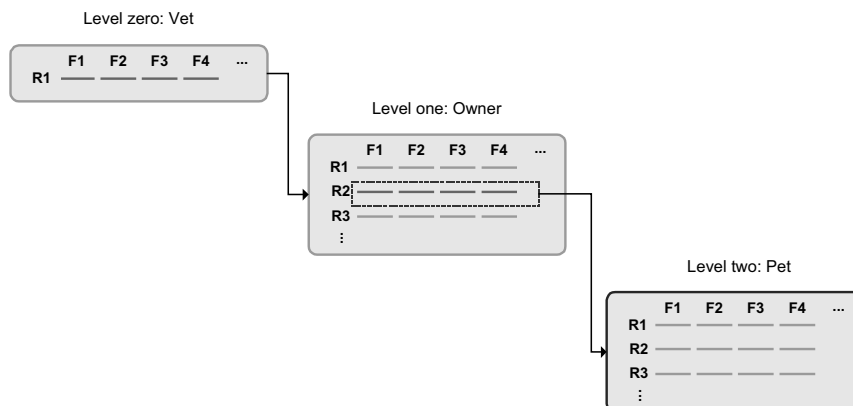
In PeopleTools 8, the object based version of this would be:

```
&OWNER_SCROLL.HideAllRows() ;
```

To hide scroll levels two and below, supply the primary record and row in scroll level one, then the record identifying the target scroll:

```
HideScroll (RECORD.OWNER, &L1ROW, RECORD.PET) ;
```

The following diagram shows the scroll path of this statement, assuming that the value of &L1ROW is 2:



Sample Scroll Path

Similarly, to hide the VISIT scroll on level three, you would specify rows on scroll levels one and two.

```
HideScroll (RECORD.OWNER, &L1ROW, RECORD.PET, &L2ROW, RECORD.VISIT) ;
```

If you wanted to use the new *SCROLL.scrollname* syntax, the above example could be written as this:

```
HideScroll (SCROLL.OWNER, &L1ROW, SCROLL.PET, &L2ROW, SCROLL.VISIT) ;
```

In PeopleTools 8, the object based version of this would be:

```
&VISIT_SCROLL.HideAllRows() ;
```

Referring to Rows

Referring to rows is precisely the same as referring to scroll areas, except that you need to specify the row you want to pick out on the target scroll. As an example, let's look at the **HideRow** function, which hides a specific row in the level-three scroll of the page:

```
HideRow(scrollpath, target_row)
```

To hide row number &ROW_NUM on level one:

```
HideRow(RECORD.OWNER, &ROW_NUM);
```

To do the same using the **SCROLL.scrollname** syntax:

```
HideRow(SCROLL.OWNER, &ROW_NUM);
```

In PeopleTools 8, the object based version of this for the OWNER rowset would be:

```
&OWNER_SCROLL(&ROW_NUM).Visible = False;
```

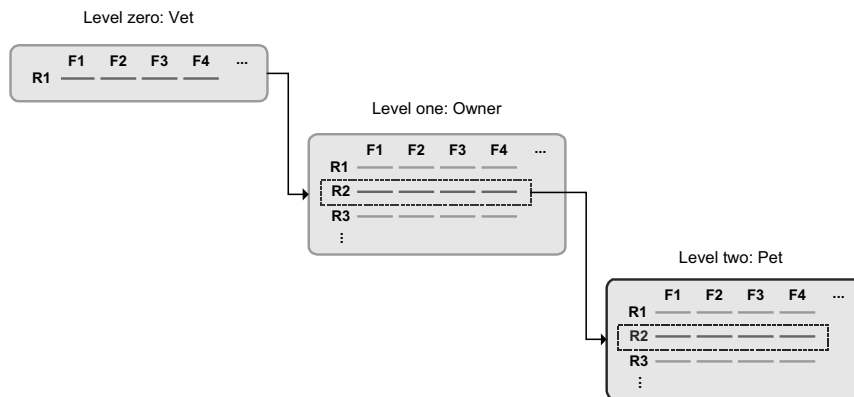
On level two:

```
HideRow(RECORD.OWNER, &L1_ROW), RECORD.PET, &ROW_NUM);
```

In PeopleTools 8, the object based version of this for the PET rowset would be:

```
&PET_SCROLL(&ROW_NUM).Visible = False;
```

The following diagram shows the scroll path of this statement, assuming that the value of &L1_ROW is 2 and that &ROW_NUM is equal to 2:



Scroll Path Statement

On level three:

```
HideRow(RECORD.OWNER, CurrentRowNumber(1), RECORD.PET, CurrentRowNumber(2),  
RECORD.VISIT, &ROW_NUM);
```

In PeopleTools 8, the object based version of this for the VISIT rowset would be:

```
&VISIT_SCROLL(&ROW_NUM).Visible = False;
```

Referring to Buffer Fields

Buffer field references require a *[recordname.]fieldname* parameter to specify a record field: the combination of scroll level, row number, and record field name uniquely identifies the buffer field:

```
FetchValue(scrollpath, target_row, [recordname.]fieldname)
```

Assume, for example, that record definitions in the veterinary database have the following fields that you want to reference:

Record	Sample Field
OWNER	OWNER_NAME
PET	PET_BREED
VISIT	VISIT_REASON

You could use the following examples to retrieve values on levels 1, 2, or 3 from a PeopleCode program executing on level zero.

To fetch a value of the OWNER_NAME field on the current row of scroll area one:

```
&SOMENAME = FetchValue(RECORD.OWNER, &L1_ROW, OWNER.OWNER_NAME);
```

In PeopleTools 8, the object based version of this for the OWNER rowset would be:

```
&SOMENAME = &OWNER_SCROLL(&L1_ROW).OWNER.OWNER_NAME;
```

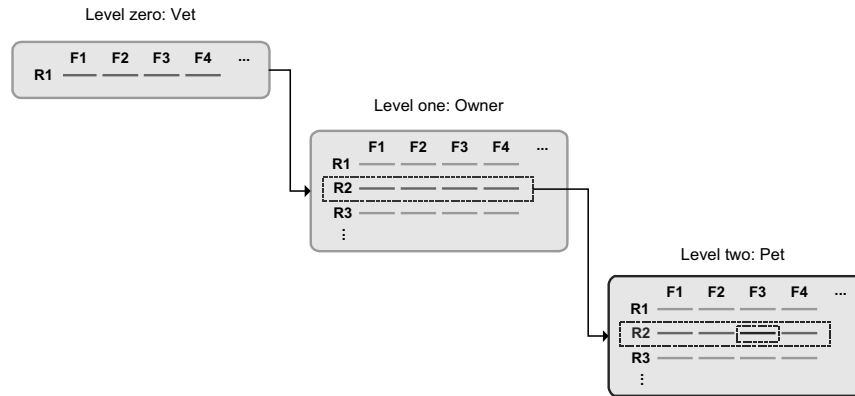
To fetch PET_BREED on level two:

```
&SOMEBREED = FetchValue(RECORD.OWNER, &L1_ROW, RECORD.PET, &L2_ROW,
PET.PET_BREED);
```

In PeopleTools 8, the object based version of this for the PET rowset would be:

```
&SOMEBREED = &PET_SCROLL(&L2_ROW).PET.PET_BREED;
```

The following diagram shows the scroll path to the target field, assuming that &L1_ROW equals 2, &L2_ROW equals 2, and field F3 is PET.PET_BREED:



Scroll Path to Target Field

To fetch VISIT_REASON on level three:

```
&SOMEREASON = FetchValue(RECORD.OWNER, &L1_ROW, RECORD.PET, &L2_ROW,
RECORD.VISIT, &L3_ROW, VISIT.VISIT_REASON);
```

To do the same thing using the *Scroll.scrollname* syntax:

```
&SOMEREASON = FetchValue(SCROLL.OWNER, &L1_ROW, SCROLL.PET, &L2_ROW,
SCROLL.VISIT, &L3_ROW, SCROLL.VISIT_REASON);
```

In PeopleTools 8, the object based version of this would be:

```
&SOMEREASON = &VISIT_SCROLL(&L3_ROW).VISIT.VISIT_REASON;
```

Using CurrentRowNumber

The **CurrentRowNumber** function returns the current row, as determined by the current context, for a specific scroll level in the active page. **CurrentRowNumber** is often used to determine a value for the *level1_row* and *level2_row* parameters in scroll path constructions. Because current row numbers are determined by the current context, **CurrentRowNumber** cannot determine a current row on a scroll level outside the current context (that is, a scroll level below the level where the PeopleCode program is currently executing).

For example, you could use a statement like this to retrieve the value of a buffer field on level three of the PET_VISITS page, in a PeopleCode program executing on level two:

```
&VAL = FetchValue(RECORD.OWNER, CurrentRowNumber(1), RECORD.PET,
CurrentRowNumber(2), RECORD.VISIT, &TARGETROW, VISIT_REASON);
```

Because the PeopleCode program is executing on level two, **CurrentRowNumber** can return values for levels one and two, but not three, because level three is outside of the current context and has no current row number.



For more information see Understanding Current Context and CurrentRowNumber.

Looping through Scroll Levels

Component Buffer functions are often used in **For** loops to loop through the rows on scroll levels below the level where the PeopleCode program is executing. The following loop, for example could be used in PeopleCode executing on a level-two record field to loop through rows of data on level three:

```
For &I = 1 To ActiveRowCount(RECORD.OWNER, CurrentRowNumber(1), RECORD.PET,
CurrentRowNumber(2), RECORD.VISIT)
    &VAL = FetchValue(RECORD.OWNER, CurrentRowNumber(1), RECORD.PET,
CurrentRowNumber(2), RECORD.VISIT, &I, VISIT_REASON);

    If &VAL = "Fleas" Then

        /* do something about fleas */

    End-If;

End-For;
```

A similar construct may be used in accessing other level 2 or level 1 scrolls, such as work scrolls.

In these constructions the **ActiveRowCount** function is often used to determine the upper bounds of the loop. When **ActiveRowCount** is used for this purpose, the loop goes through all of the active rows in the scroll (that is, rows that have not been flagged as deleted). If you use **TotalRowCount** to determine the upper bounds of the loop, the loop goes through all of the rows in the scroll: first those that have not been flagged as deleted, then those that have been flagged as deleted.

Scroll Path Syntax prior to PeopleTools 7.5

In scroll path field references in PeopleTools releases 5 through 7, a *recordname.fieldname* expression does double-duty, identifying both the target scroll and the target field. For example, in this statement:

```
Hide(RECORD.VET, CurrentRowNumber(1), OWNER.OWNER_NAME, &ROWNUM);
```

the expression `OWNER.OWNER_NAME` serves both to identify the scroll level and to specify a record field. Compare the PeopleTools 7.5 syntax, where a **RECORD.recordname** expression identifies the scroll level by its primary record:

```
Hide(RECORD.VET, CurrentRowNumber(1), RECORD.OWNER, &ROWNUM, OWNER.OWNER_NAME);
```

The pre-7.5 syntax is supported in PeopleTools 7.5, and there is no need to update programs to the new syntax if they are already working correctly. However, the old syntax can sometimes result in ambiguous references if the target record is a Derived/Work or related display field that occurs on more than one level on the page. You can avoid this problem by updating to the new syntax. The surest way to avoid ambiguity is to use the alternative **SCROLL.scrollname** construction (see Scroll Path Syntax with **SCROLL.scrollname**).

CHAPTER 9

Data Buffer Access

In addition to the built-in functions used to access the Component Buffer, there are classes of objects that provide access to structured data buffers using the new PeopleCode object syntax.

The data buffers accessed by these classes are typically the Component Buffers which are loaded when you open a component. However, these classes may also be used to access data from general data buffers, loaded by an Application Message, an Application Engine program, a Business Components, and so on.

The methods and properties of these classes provide functionality that is similar to what has been available previously using the built-in functions. However, they also provide improved consistency, flexibility, and new functionality.

Access Classes

There are four new data buffer classes: Rowset, Row, Record, and Field. These four classes are the foundation for accessing Component Buffer data through the new object syntax.

A **field** object, which is instantiated from the Field class, is a single instance of data within a record and is based on a field definition.

A **record** object, which is 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.

A **row** object, which is 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 may have one to n child rowsets. For example, a row in a level two scroll may have n level three child rowsets.

A **rowset** object is a data structure used to describe hierarchical data. It is made up of a collection of rows. A component scroll is a rowset. You can also have a level 0 rowset.

Data Buffer Model and Data Access Objects

The data model assumed by these classes is that of a PeopleTools component, where scrollbars or grids are used to describe a hierarchical, multiple-occurrence data structure. These four classes are built on the data model of a PeopleTools component, in which scrollbars or grids are used to describe a hierarchical, multiple-occurrence data structure. You can access these classes using dot notation.

The four data buffer classes relate to each other in a hierarchical manner. The main thing to remember when learning these relationships is:

- A *record* contains one or more *fields*.
 - Records contain the fields that make up that record.
- A *row* contains one or more *records* and zero or more child *rowsets*
 - A row contains the records that make up that row. It may also contain child rowsets.
- A *rowset* contains one or more *rows*
 - A rowset is a data structure that describes hierarchical data. For component buffers, think of a rowset as a scroll on a page that contains all of that scroll's data. A level 0 rowset contains all the data for the entire component.
 - You can use rowsets with application messages, file layouts, Business Interlinks, and other definitions besides components.
 - A level 0 rowset from a component buffer only contains one row, that is, the keys that the user specifies to initiate that component. A level 0 rowset from data that isn't a component, such as a message or a file layout, might contain more than one level 0 row.

The following is some simple PeopleCode that traverses through a two level Component Buffer using the new dot notation syntax. Level zero is based on record QA_INVEST_HDR and level one is based on record QA_INVEST_LN.

```

Local Rowset &HDR_ROWSET, &LINE_ROWSET;

Local Record &HDR_REC, &LINE_REC;

&HDR_ROWSET = GetLevel0();

For &I = 1 to &HDR_ROWSET.RowCount

    &HDR_REC = &HDR_ROWSET(&I).QA_INVEST_HDR;

    &EMPLID = &HDR_REC.EMPLID.Value;

    &LINE_ROWSET = &HDR_ROWSET(&I).GetRowset(1);

    For &J = 1 to &LINE_ROWSET.RowCount

        &LINE_REC = &LINE_ROWSET(&J).QA_INVEST_LN;

        &LINE_SUM = &LINE_SUM + &LINE_REC.AMOUNT.Value;

    End-For;

End-For;

```

You will notice that each rowset is declared and instantiated. In general, your code will be easier to read and maintain if you follow this practice.

Data Buffer Classes Examples

Most of the examples in this section use the page EMPLOYEE_CHECKLIST.

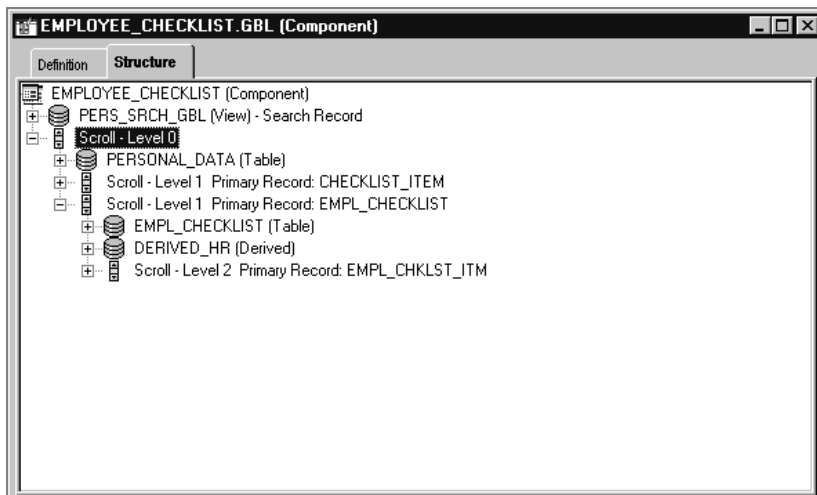
*Chklist Seq	*Chklist itm	*Briefing Status	*Status Date	
100	000015	Briefing with Human Resources	Initiated	08/11/2000
200	000025	Repatriation Discussion	Initiated	08/11/2000
300	000029	Career/Placement discussion	Initiated	08/11/2000

Employee Checklist Page

This page has the following record structure:

Scroll Level	Associated Primary Record	Rowset and Variable Name
Level 0	PERSONAL_DATA	Level 0 rowset: &RS0
Level 1 scroll	EMPL_CHECKLIST	Level 1 rowset: &RS1
Level 1 hidden work scroll	CHECKLIST_ITEM	Level 1 rowset: &RS1H
Level 2 scroll	EMPL_CHKLST_ITM	Level 2 rowset: &RS2

Another way of looking at the structure of a component is to use the Structure View. All of the scrolls are labeled, as well as the primary record associated with each:



Component Structure View showing data hierarchy

In the following example the visible level 1 scroll also only has one row. That row is made up of the following records:

- EMPL_CHECKLIST
- DERIVED_HR
- CHECKLIST_TBL
- PERSONAL_DATA

You can see which records are associated with a scroll by looking at the Order view for a page:

The screenshot shows the 'Order' tab of the 'EMPLOYEE_CHECKLIST.ENG (Page)' window. The table lists records and their associated fields, with checkboxes for 'Display Control' and 'Related Field'.

	Lv	Label	Type	Field	Record	Display Control	Related Field
1	0	Frame	Frame			<input type="checkbox"/>	<input type="checkbox"/>
2	0	Frame	Frame			<input type="checkbox"/>	<input type="checkbox"/>
3	0	Frame	Frame			<input type="checkbox"/>	<input type="checkbox"/>
4	0	Employee Name	Edit Box	NAME	PERSONAL_DATA	<input type="checkbox"/>	<input type="checkbox"/>
5	0	ID	Edit Box	EMPLID	PERSONAL_DATA	<input type="checkbox"/>	<input type="checkbox"/>
6	1	Checklist Item Tbl	Scroll Bar			<input type="checkbox"/>	<input type="checkbox"/>
7	1	Checklist Sequen	Edit Box	CHECKLIST_SEQ	CHECKLIST_ITEM	<input type="checkbox"/>	<input type="checkbox"/>
8	1	Scroll Bar 1	Scroll Bar			<input type="checkbox"/>	<input type="checkbox"/>
9	1	Checklist Date	Edit Box	CHECKLIST_DT	EMPL_CHECKLIST	<input type="checkbox"/>	<input type="checkbox"/>
10	1	derived_hr_effdt	Edit Box	EFFDT	DERIVED_HR	<input type="checkbox"/>	<input type="checkbox"/>
11	1	Checklist	Edit Box	CHECKLIST_CD	EMPL_CHECKLIST	<input checked="" type="checkbox"/>	<input type="checkbox"/>
12	1	Checklist Descripti	Edit Box	DESCR	CHECKLIST_TBL	<input type="checkbox"/>	<input checked="" type="checkbox"/>
13	1	Responsible ID	Edit Box	RESPONSIBLE_I	EMPL_CHECKLIST	<input checked="" type="checkbox"/>	<input type="checkbox"/>
14	1	Responsible Nam	Edit Box	NAME	PERSONAL_DATA	<input type="checkbox"/>	<input checked="" type="checkbox"/>
15	1	Comment	Long Edit Box	COMMENTS	EMPL_CHECKLIST	<input type="checkbox"/>	<input type="checkbox"/>
16	2	Scroll Bar 2	Scroll Bar			<input type="checkbox"/>	<input type="checkbox"/>
17	2	Chklist Seq	Edit Box	CHECKLIST_SEQ	EMPL_CHKLSL_ITM	<input type="checkbox"/>	<input type="checkbox"/>
18	2	Chklist Itm	Edit Box	CHKLSL_ITEM_C	EMPL_CHKLSL_ITM	<input checked="" type="checkbox"/>	<input type="checkbox"/>
19	2	Briefing Descriptio	Edit Box	DESCR	CHKLSL_ITEM_TBL	<input type="checkbox"/>	<input checked="" type="checkbox"/>
20	2	Briefing Status	Drop Down List	BRIEFING_STAT	EMPL_CHKLSL_ITM	<input type="checkbox"/>	<input type="checkbox"/>

EMPLOYEE_CHECKLIST page Order view showing records

The level 2 rowset has three rows in it. Each row is made up of two records: the primary record, EMPL_CHKLIST_ITM, and CHKLST_ITM_TBL, the record associated with the related display field DESCR.

Employee Checklist

Schumacher, Simon ID: 8001

*Checklist Date: 08/11/2000 Checklist: 000003 Repatriation Checklist

Responsible ID: 6602 Peppen, Jacques

Comment:

Level2 Rowset

*Chklist Seq	*Chklist Itm	*Briefing Status	*Status Date
100	Briefing with Human Resources	Initiated	08/11/2000
200	Repatriation Discussion	Initiated	08/11/2000
300	Career/Placement discussion	Initiated	08/11/2000

Row2 for level2 rowset

Field

Save Return to Search

Rowset and Rows

Every record has fields associated with it, like NAME, EMPLID, CHECKLIST_SEQ, and so on. These are the fields associated with the record definitions, not the fields displayed on the page.

Object Creation Examples

Each data buffer access class has its own data type which is the same name as the class, that you should use when declaring your variables (Rowset objects should be declared as type Rowset, field objects as type Field, and so on.) Data buffer access class objects can be of type Local, Global, or Component.

The following declarations are assumed throughout the examples that follow:

```
Local Rowset &LEVEL0, &ROWSET;

Local Row &ROW;

Local Record &REC;

Local Field &FIELD;
```

Accessing Level 0

The following code instantiates a rowset object, from the Rowset class, that references the level 0 rowset, containing all the page data. It stores the object in the &LEVEL0 variable.

```
&LEVEL0 = GetLevel0();
```

The level 0 rowset contains all the rows, rowsets, records and fields underneath it.

If the level 0 rowset is formed from Component Buffer data, the level 0 rowset has one row of data, and that row contains all the child rowsets, which in turn contain rows of data, that contain other child rowsets.

If the level 0 rowset is formed from buffer data, such as from an Application Message, the level 0 rowset may contain more than one row of data. Each row of the level 0 rowset contains all the child rowsets associated with that row, which in turn contain rows of data, that contain other child rowsets.

You would use a level 0 rowset when you wanted an absolute path to a lower level object or when you wanted to do some processing on the entire data buffer. For example, suppose you loaded all new data into the Component Buffers, and wanted to redraw the page. You could use the following code:

```
/* Do processing to reload Component Buffers */

&LEVEL0 = GetLevel0();

&LEVEL0.Refresh();
```



For more information see `GetJavaClass`.

Rowset Object

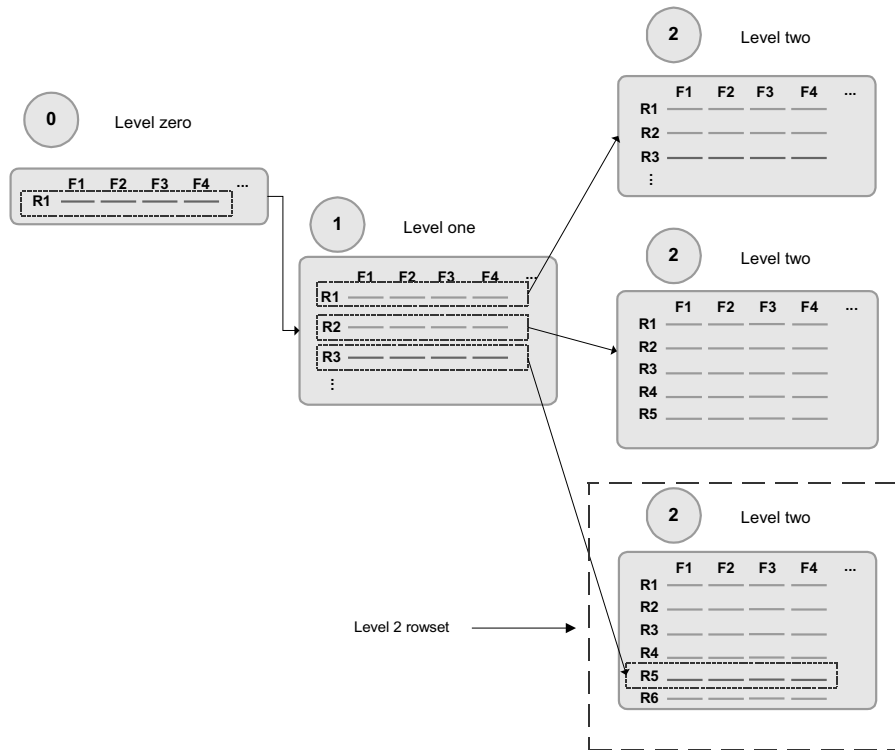
The following code instantiates a rowset object that references the rowset that contains the currently running PeopleCode program.

```
&ROWSET = GetRowset();
```

You might later use the `&ROWSET` variable 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 hide a child rowset (by setting the `Visible` property.)

The level one rowset will contain all the level two rowsets. However, the level two rowsets can only be accessed using the different rows of the level one rowset. From the level zero or level one rowset, you can only access a level two rowset by using the level one rowset **and** the appropriate row.

For example, suppose your program is running on some field of row five of a level two scroll, which is on row three of its level one scroll. The resulting rowset will contain all the rows of the level two scroll that are under the row three of the level one scroll. The rowset will **not** contain any data that is under any other level two scrolls.



Level 2 Rowset from Level 1 Row

Let's illustrate this further with an example from the Employee Checklist page.

Suppose that one employee was associated with three different checklists: Foreign loan departure, Foreign loan arrival, and Foreign loan host. The checklist code field (CHECKLIST_CD) on the first level of the page drives the entries on the second level. Each row in the level one rowset produces a *different* level two rowset.

The Foreign Loan Departure checklist (000001) produces a checklist that contains such items as Briefing with Human Resources, Apply for Visas/Work permits, and so on.

Employee Checklist

Schumacher, Simon ID: 8001

*Checklist Date: 08/11/2000 Checklist: 000001 Foreign Loan Departure Checklist

Responsible ID: 6602 Peppen, Jacques

Comment:

<Previous + 1 of 3 - Next>

*Chklist Seq	*Chklist Item	*Briefing Status	*Status Date
100	000015 Briefing with Human Resources	Initiated	08/11/2000
200	000030 Apply for Visas/Work Permits	Initiated	08/11/2000
300	000009 Reconfirm Relocation Package	Initiated	08/11/2000
400	000001 Select moving/storage company	Initiated	08/11/2000

Save Return to Search

Foreign Loan Departure Checklist

The Foreign Loan Arrival Checklist (0000004) produces a checklist that contains items as Register at Consulate, Open new foreign bank accounts, and so on.

Employee Checklist

Schumacher, Simon ID: 8001

*Checklist Date: 08/11/2000 Checklist: 000004 Foreign Loan Arrival Checklist

Responsible ID: 7705 Holt, Susan

Comment:

<Previous + 2 of 3 - Next>

*Chklist Seq	*Chklist Item	*Briefing Status	*Status Date
100	000022 Register at Consulate	Initiated	08/11/2000
200	000008 Open new foreign bank accounts	Initiated	08/11/2000
300	000018 Register children in school	Initiated	08/11/2000
400	000019 Join Newcomer's Club	Initiated	08/11/2000

Save Return to Search

Foreign Loan Arrival Checklist

The Foreign Loan Host Checklist (0000005) produces a checklist that contains items such as Arrange Meeting with Mentor, Clear Local tax filing reqmnts, and so on.

Employee Checklist

Schumacher,Simon

ID: 8001

*Checklist Date: 08/11/2000

Checklist: 000005 Foreign Loan Host Checklist

Responsible ID: 7705 Holt,Susan

Comment:

<Previous

+

3 of 3

-

Next>

*Chklist Seq	*Chklist Itm		*Briefing Status	*Status Date
100	000015	Briefing with Human Resources	Initiated	08/11/2000
200	000017	Arrange meeting with Mentor	Initiated	08/11/2000
300	000023	Clear local tax filing reqmnts	Initiated	08/11/2000
400	000027	Complete foreign tax forms	Initiated	08/11/2000

Save

Return to Search

Foreign Loan Host Checklist



For more information about the methods and properties of rowsets, see Rowset Class.

Row Object

When you create a page you put fields from different records onto the page. You can think of this as creating a type of pseudo-SQL join. The row returned from this pseudo join is a **row object**.

For example, the first level scroll of the EMPLOYEE_CHECKLIST page contains the following fields, associated with these records:

Field	Record
CHECKLIST_DT	EMPL_CHECKLIST
CHECKLIST_CD	EMPL_CHECKLIST
COMMENTS	EMPL_CHECKLIST
DESCR	CHECKLIST_TBL
NAME	PERSONAL_DATA
RESPONSIBLE_ID	EMPL_CHECKLIST

The pseudo-SQL join might look like the following:

```
JOIN A.CHECKLIST_DT, A.CHECKLIST_CD, A.COMMENTS, B.DESCR, C.NAME,
A.RESPONSIBLE_ID

FROM PS_EMPL_CHECKLIST A, PS_CHECKLIST_TBL B, PS_PERSONAL_DATA C, WHERE. . .
```

What goes into the WHERE clause is determined by the level 0 of the page. For our example, it's WHERE EMPLID=8001.

When the component is opened, data gets loaded into the Component Buffers. Any row returned by the pseudo-SQL statement is a level 1 **row object**.

<i>CHECKLIST _DT</i>	<i>CHECKLIST _CD</i>	<i>COMMENTS</i>	<i>DESCR</i>	<i>NAME</i>	<i>RESPONSIBLE _ID</i>
12/03/98	000001		Foreign Loan Department Checklist	Peppen, Jacque	6602



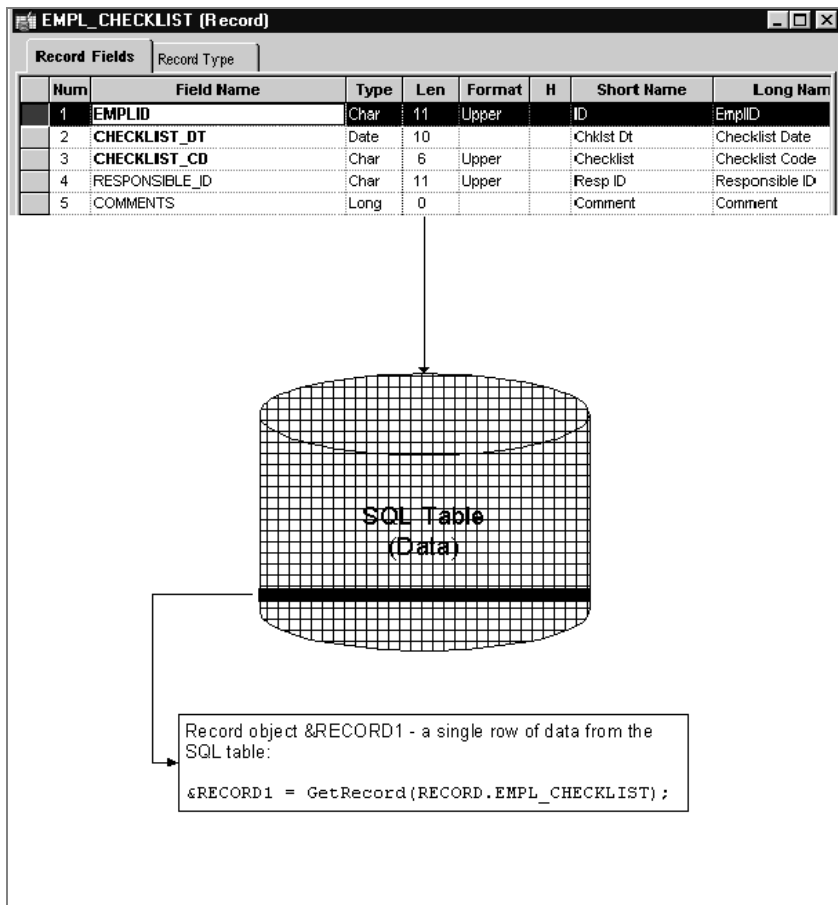
For more information on the row methods and properties, see Row Class.

Record Object

A record ***definition*** is a definition of what your underlying SQL database tables will look like, and how they will process data. After you create your record definitions, you build the underlying SQL tables that will actually house the application data your users will enter online in your production environment.

When you create a record object using the **CreateRecord** function, you're creating an area in the data buffers that has the same *structure* as the record definition, but no data.

When you instantiate a record object from the Record class using some variation of **GetRecord**, that record object references a single row of data in the SQL table.



Record Object



Remember, the data in the record that you retrieve is based on the **row**. This is analogous to setting keys to return a unique record.

The following code instantiates a record object for referencing the EMPL_CHECKLIST record of the specified row.

```
&REC = &ROW.GetRecord(RECORD.EMPL_CHECKLIST);
```

Using the "short" method, the following line of code is identical to the above line:

```
&REC = &ROW.EMPL_CHECKLIST;
```

You might later use the &REC variable and the CopyFieldsTo property to copy all like named fields from one record to another. In the following example, two row objects are created, the copy from row (COPYFRMROW) and the copy to row (COPYTROW). Using these rows, like-named fields are copied from CHECKLIST_ITEM to EMPL_CHKLIST_ITM.

```
For &I = 1 To &ROWSET1.ActiveRowCount
```

```
&COPYFRMROW = &ROWSET1.GetRow(&I);
```

```

&COPYTROW = &RS2.GetRow(&I) ;

&COPYFRMROW.CHECKLIST_ITEM.CopyFieldsTo(&COPYTROW.EMPL_CHKLIST_ITM) ;

End-For;

```

A row may contain more than one record: besides the primary database record, you may have a related display record or a derived record. You can access these records as well. The level 1 rowset, &ROWSET1, is made up of many records. The following accesses two of them: EMPL_CHECKLIST and DERIVED_HR.

```

&REC1 = &ROW.EMPL_CHECKLIST;

&REC2 = &ROW.DERIVED_HR;

```



For more information on the methods and properties used with a record, see ProcessRequest Class.

Field Object

The following instantiates a field object, from the Field class, that is used to access a specific field in the record.

```
&FIELD = &REC.GetField(FIELD.CHECKLIST_CD) ;
```

You might later use the &FIELD variable to as a condition, like:

```
If ALL(&FIELD) Then
```

Or

```
If &FIELD.Value = "N" Then
```



Remember, the data in the field that you retrieve is based on the **record**, which is in turn based on the **row**.

You can also set the value of a field. Remember, using **GetField** does not create a copy of the data from the Component Buffer. Setting the value or a property of the field object will set the actual Component Buffer field or property (see object assignment.)

In the following example, the type of a field is checked, and the value is replaced with the tangent of that value if it is a number.

```

If &FIELD.Type <> "NUMBER" Then

    /* do error recording */

Else

```

```
&FIELD.Value = Tan(&FIELD.Value);

End-If;
```



For more information on the methods and properties used with a field, see Field Class.

Traversing the Data Buffer Hierarchy Example

Suppose you want to access the BRIEFING_STATUS field at level 2 of the following page:

*Chklist Seq	*Chklist Item	*Briefing Status	*Status Date
100	000015 Briefing with Human Resources	Initiated	08/11/2000
200	000025 Repatriation Discussion	Initiated	08/11/2000
300	000029 Career/Placement discussion	Initiated	08/11/2000

Employee_Checklist page

The first thing to ask is where is your code executing? Where are you starting from? For this example, the code is starting at a field on a record at level 0. However, you don't always have to start at the level 0.

Once you start with level 0, you'll need to traverse the data hierarchy, through the level 1 rowset, to the level 2 rowset, before you can access the record that contains the field. Here's the hierarchy once again:

A rowset contains one or more rows, a row contains one or more records and zero or more child rowsets, and a record contains one or more fields.

Rowset

The first thing to get is the level 0 rowset, which is the PERSONAL_DATA rowset. However, you don't need to know the name of the level 0 rowset in order to access it.

```
&LEVEL0 = GetLevel0();
```

Rowsets contain rows

The next object to get is a row. As this code is working with data that is loaded from a page, there will only ever be one row at level 0. However, if you're dealing with rowsets that are populated with data that isn't based on Component Buffers (i.e., an Application Message) you may have more than one row at level 0.

```
&LEVEL0_ROW = &LEVEL0(1);
```

Rows can contain child rowsets

We need to get to the level 2 rowset. To do that, we need to traverse **through** the level 1 rowset first. Therefore, the next object we want to get is the level 1 rowset.

```
&LEVEL1 = &LEVEL0_ROW.GetRowset(ROLL.EMPL_CHECKLIST);
```

Rowsets contain rows

If you're traversing a page, the first thing you'll always do after you get a rowset is to get the appropriate row. Because we plan on doing processing on all the rows of the rowset, we'll set this up in a loop.

```
For &I = 1 to &LEVEL1.ActiveRowCount
    &LEVEL1_ROW = &LEVEL1(&I);
    . . .
End-For;
```

Rows can contain child rowsets, rowsets contain rows

We need to traverse down another level in our page structure. This means accessing the second level rowset. Then we need to access the rows in the second level rowset, in another loop.

Because we're processing all the rows at the level 1, we're just adding code to the above For loop. As we're processing through all the rows at level 2, we're adding a second For loop. The new code is in bold.

```
For &I = 1 to &LEVEL1.ActiveRowCount
    &LEVEL1_ROW = &LEVEL1(&I);
    &LEVEL2 = &LEVEL1_ROW.GetRowset(ROLL.EMPL_CHKLIST_ITM);
    For &J = 1 to &LEVEL2.ActiveRowCount
        &LEVEL2_ROW = &LEVEL2(&J);
```

```

. . .

End-For;

End-For;

```

Rows contain records

Rows also contain records. In fact, a row will always contain a record, and only **may** contain a child rowset, depending on how your page is set up. GetRecord is the default method for a row, so all you have to specify is the record name.

Because we're processing all the rows at the level 2, we're just adding code to the above For loops. The new code is in bold.

```

For &I = 1 to &LEVEL1.ActiveRowCount

    &LEVEL1_ROW = &LEVEL1(&I);

    &LEVEL2 = &LEVEL1_ROW.GetRowset (SCROLL.EMPL_CHKLIST_ITM);

    For &J = 1 to &LEVEL2.ActiveRowCount

        &LEVEL2_ROW = &LEVEL2(&J);

        &RECORD = &LEVEL2_ROW.EMPL_CHKLIST_ITM;

        . . .

    End-For;

End-For;

```

Records contain fields

Records are made up of fields. GetField is the default method for a record, so all you have to specify is the field name.

Because we're processing all the rows at the level 1, we're just adding code to the above For loops. The new code is in bold.

```

For &I = 1 to &LEVEL1.ActiveRowCount

    &LEVEL1_ROW = &LEVEL1(&I);

    &LEVEL2 = &LEVEL1_ROW.GetRowset (SCROLL.EMPL_CHKLIST_ITM);

    For &J = 1 to &LEVEL2.ActiveRowCount

        &LEVEL2_ROW = &LEVEL2(&J);

        &RECORD = &LEVEL2_ROW.EMPL_CHKLIST_ITM;

        &FIELD = &RECORD.BRIEFING_STATUS;
    End-For;
End-For;

```

```

        /* Do processing */

    End-For;

End-For;

```

Using shortcuts

The above code is the long way of accessing this field. What if you wanted to use all the shortcuts, and access the field in one line of code? Here it is! The following code assumes all rows are 1.

Rowset
Row
Rowset
Row
Rowset
Row
Record
Field

```
&FIELD = GetLevel0() (1).EMPL_CHECKLIST(1).EMPL_CHKLIST_ITM(1).EMPL_CHKLIST_ITM.BRIEFING_STATUS;
```

Rowset Example

Here's another way of expressing the code:

Rowset	&LEVEL0 = GetLevel0();
Row	&LEVEL0_ROW = &LEVEL0(1);
Rowset	&LEVEL1 = &LEVEL0_ROW.GetRowset(SCROLL.EMPL_CHECKLIST);
	For &I = 1 to &LEVEL1.ActiveRowCount
Row	&LEVEL1_ROW = &LEVEL1(&I);
Rowset	&LEVEL2 = &LEVEL1_ROW.GetRowset(SCROLL.EMPL_CHKLIST_ITM);
	For &J = 1 to &LEVEL2.ActiveRowCount
Row	&LEVEL2_ROW = &LEVEL2(&J);
Record	&RECORD = &LEVEL2_ROW.EMPL_CHKLIST_ITM;
Field	&FIELD = &RECORD.BRIEFING_STATUS;
	/* Do processing */
	End-For;

	End-For;
--	----------

Traversing a Rowset Example

The following code example traverses up to 4 levels of rowsets and could easily be modified to do more. This example only processes the **first** record in every rowset. If you wanted to process every record, you'd have to set up another For loop (For &R = 1 to &LEVELX.RECORDCOUNT, and so on.) Notice the use of ChildCount (to process all children rowsets within a rowset), ActiveRowCount, IsChanged, and dot notation.

‘...’ is where application specific code would go.

```

&Level0_ROWSET = GetLevel0();

For &A0 = 1 To &Level0_ROWSET.ActiveRowCount

    ...

/*****/

/* Process Level 1 Records */

/*-----*/

    If &Level0_ROWSET(&A0).ChildCount > 0 Then

        For &B1 = 1 To &Level0_ROWSET(&A0).ChildCount

            &LEVEL1_ROWSET = &Level0_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 */

```

Using a Hidden Work Scroll Example

In the FieldChange event for the CHECKLIST_CD field on the EMPL_CHECKLIST record, there is a program which does the following:

1. Flushes the rowset/hidden work scroll.
2. Selects into the hidden work scroll based on the value of the CHECKLIST_CD field and the effective date.
3. Clears out the second level scroll.
4. Copies like fields from the hidden work scroll to the second level scroll.

The following is the code to do this using built-in functions.

```

&CURRENT_ROW_L1 = CurrentRowNumber(1);

&ACTIVE_ROW_L2 = ActiveRowCount(RECORD.EMPL_CHECKLIST, &CURRENT_ROW_L1,
RECORD.EMPL_CHKLIST_ITM);

If All(CHECKLIST_CD) Then

```

```

ScrollFlush(RECORD.CHECKLIST_ITEM);

ScrollSelect(1, RECORD.CHECKLIST_ITEM, RECORD.CHECKLIST_ITEM, "Where
Checklist_Cd = :1 and EffDt = (Select Max(EffDt) From PS_Checklist_Item Where
Checklist_Cd = :2)", CHECKLIST_CD, CHECKLIST_CD);

&FOUNDDOC = FetchValue(CHECKLIST_ITEM.CHLST_ITEM_CD, 1);

&SELECT_ROW = ActiveRowCount(RECORD.CHECKLIST_ITEM);

For &I = 1 To &ACTIVE_ROW_L2

    DeleteRow(RECORD.EMPL_CHECKLIST, &CURRENT_ROW_L1, RECORD.EMPL_CHKLST_ITM,
1);

End-For;

If All(&FOUNDDOC) Then

    For &I = 1 To &SELECT_ROW

        CopyFields(1, RECORD.CHECKLIST_ITEM, &I, 2, RECORD.EMPL_CHECKLIST,
&CURRENT_ROW_L1, RECORD.EMPL_CHKLST_ITM, &I);

        If &I <> &SELECT_ROW Then

            InsertRow(RECORD.EMPL_CHECKLIST, &CURRENT_ROW_L1,
RECORD.EMPL_CHKLST_ITM, &I);

        End-If;

    End-For;

End-If;

End-If;

```

This following program does the exact same thing as the previous code, only it uses the data buffer classes:

1. Flushes the rowset/hidden work scroll (&RS1H).
2. Selects into &RS1H based on the value of the CHECKLIST_CD field and the effective date.
3. Clears out the second level Rowset (&RS2).
4. Copies like fields from &RS1H to &RS1.

```
Local Rowset &RS0, &RS1, &RS2, &RS1H;
```

```

&RS0 = GetLevel0();

&RS1 = GetRowset();

&RS2 = GetRowset(SCROLL.EMPL_CHKLIST_ITM);

&RS1H = &RS0.GetRow(1).GetRowset(SCROLL.CHECKLIST_ITEM);

&MYFIELD = CHECKLIST_CD;

If All(&MYFIELD) 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);

    For &I = 1 To &RS2.ActiveRowCount

        &RS2.DeleteRow(1);

    End-For;

&FOUND = &RS1H.GetCurrEffRow().CHECKLIST_ITEM. CHKLIST_ITEM_CD.Value;

If All(&FOUND) Then

    For &I = 1 To &RS1H.ActiveRowCount

        &COPYFRMROW = &RS1H.getrow(&I);

        &COPYTROW = &RS2.getrow(&I);
&COPYFRMROW.CHECKLIST_ITEM.CopyFieldsTo(&COPYTROW.EMPL_CHKLIST_ITM);

        If &I <> &RS1H.ActiveRowCount Then

            &RS2.InsertRow(&I);

        End-If;

    End-For;

End-If;

End-If;

```

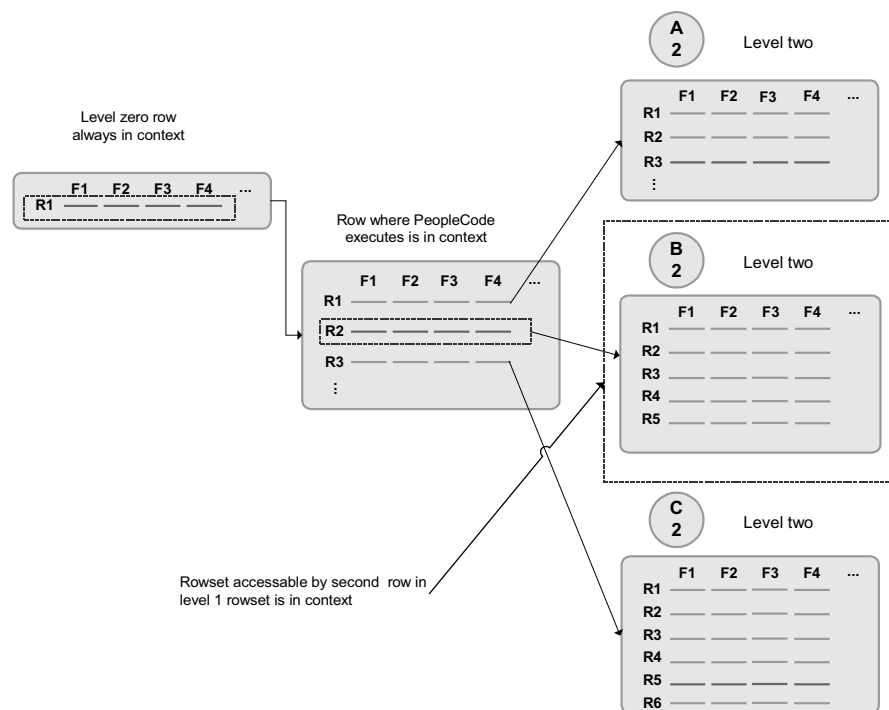
Current Context

Most PeopleCode programs execute in a **current context**. The current context determines which buffer fields can be contextually referenced from PeopleCode, and which row of data is the "current row" on each scroll level at the time a PeopleCode program is executing.

The current context for the data buffer access classes is similar to the current context for accessing the Component Buffer. However, with the Rowset class, a little additional explanation is necessary.



For more information about current context, see Contextual References.



Current Context for Rowsets

In this example a PeopleCode program is executing in a buffer field on the second row of the level one rowset. The following code will return a row object for the second row of the level one rowset, because that is the row that is the *current context*.

```
Local Row &ROW

&ROW = GetRow();
```

The following code will return the B2 level 2 rowset, because of the current context:

```
Local Rowset &ROWSET2
```

```
&ROWSET2 = &ROW.GetRowset (SCROLL.EMPL_CHKLIST_ITM) ;
```

This code will NOT return either the C2 or the A2 rowsets. It will only return the rowset associated with the second row of the level one rowset.

Creating Records or Rowsets and Current Context

When you instantiate a record object using the CreateRecord function, you are only creating an area in the data buffers that has the same *structure* as the record definition. It will **not** contain any data. This record object will not have a parent rowset or be associated with a row. It is a free-standing record object, and therefore is not considered part of the current context.

The same applies when you instantiate a rowset object using the CreateRowset function. You are only creating an area in the data buffers that has the same *structure* as the record(s) or rowset the new rowset is based on. It will **not** contain any data. This type of rowset will not have a parent rowset or row.

Accessing Secondary Component Buffer Data

When a secondary page is run, the data for its buffers is **copied** from the parent component to a buffer structure for the secondary page. This means there are two copies of this data. The data buffer classes give access to **both** of these copies of the data. Direct field references (recname.fieldname) will always use the current context to determine which value to access. So, in general, when using a secondary page, make sure that all your references are based on the secondary page.

Instantiating Rowsets using non-Component Buffer data

Both the application message and the file layout technologies represent hierarchical data, and use the rowset, row, record, field hierarchy. Though you use different methods to instantiate a rowset object for this data, you still use the same rowset, row, record and field methods and properties to manipulate the data. (Any exceptions are marked in the documentation.)

To instantiate a rowset for a message:

```
&MSG = CreateMessage (MESSAGE.EMPLOYEE_DATA) ;

&MYROWSET = &MSG.GetRowset () ;
```

To instantiate a rowset for a file layout:

```
&MYFILE = GetFile (&SOMENAME, "R") ;

&MYFILE.SetFileLayout (FILELAYOUT.SOMELAYOUT) ;

&MYROWSET = &MYFILE.ReadRowset () ;
```



For more information, see Using Standalone Rowsets.

In an Application Engine program, the default state record is considered the primary record, and the main record in context. You can access the default state record using the following:

```
&STATERECORD = GetRecord() ;
```

If you have more than one state record associated with an application engine program, you can access them the same way you would access other, non-primary data records, by specifying the record name. For example:

```
&ALTSTATE = GetRecord(RECORD.AE_STATE_ALT) ;
```



For more information, see PeopleSoft Application Messaging and Application Engine.

CHAPTER 10

PeopleCode and the Component Processor

The Component Processor is the PeopleTools runtime engine that controls processing of the application from the time the end-user requests a component from an application menu through the time that the database is updated and processing of the component is complete.

This chapter discusses the Component Processor and its complex interaction with PeopleCode programs. It describes the PeopleCode events that are generated during the Component Processor's flow of execution at runtime, and how PeopleCode events trigger PeopleCode programs.

Events Outside the Component Processor Flow

Application Messages also have events associated with them (OnPublishTransform, OnSubscribeTransform, OnRoutePublication, OnRouteSubscription and Subscription.) However, these events are **only** associated with the message definition, and are not associated with any page. Therefore, they aren't considered part of the Component Processor flow.



For more information about these events, see PeopleSoft Application Messaging.

ActiveX controls have two events that are part of the Component Processor flow (PSControlInit and PSLostFocus.) These events are described in this chapter. However, every ActiveX control also comes with its own set of events. These events are not considered part of the Component Processor flow.



For more information, see Implementing ActiveX Controls.

An Application Engine program can have a PeopleCode program as an action. Though the right-hand drop-down on the PeopleCode editor window shows the text "OnExecute" this really isn't an event. Any PeopleCode contained in an Application Engine action will only be executed when the action is executed.



For more information see PeopleCode Actions.

A Component Interface can have user-defined methods associated with it. These methods aren't part of any processor flow: they're called as needed by the program executing the Component Interface.



For more information see Component Interface Classes.

Security has a signon "event" during signon. This is actually PeopleCode programs on a record field that you've specified in setting up security.



For more information see Security.

How PeopleCode Programs Are Triggered

PeopleCode can be associated with a PeopleCode record field, a component record, and many other items. PeopleCode events fire at particular times, in particular sequences, during the course of the Component Processor's flow of execution. When an event fires, it triggers PeopleCode programs on specific objects.

The following items have events that are part of the Component Processor flow:

<i>Items</i>	<i>Events trigger</i>
Menu Items	programs associated with the menu item
Page fields (ActiveX controls)	programs associated with the page field ¹
Component record fields	programs on specific rows of data
Component records	programs on specific rows of data
Components	programs associated with the Component
Pages	programs associated with the page
Record fields	programs on specific rows of data

¹ ActiveX controls are the only page fields that have PeopleCode associated with them.

Let's look at two examples.

Suppose the end-user changes the data in a page field, then tabs out of the field. This end-user action causes the FieldEdit PeopleCode event to fire. The FieldEdit event affects only the specific field and row where the change took place. If a FieldEdit PeopleCode program is associated with that record field, the program will be executed. The program is executed just once, on the specific field and row of data.

If you have two FieldEdit PeopleCode programs, one associated with the record field and a second associated with the component record field, both programs will be executed, but only on the specific field and row of data. The FieldEdit PeopleCode program associated with the first record field fires first, then the FieldEdit PeopleCode program associated with the first component record field fires.

By contrast, suppose the end-user has opened a component for updating. As part of building the component the Component Processor fires the RowInit event. This event triggers RowInit PeopleCode programs on every record field on every row of data in the component. In a scroll area with multiple rows of data, every RowInit PeopleCode program is executed once for each row.

In addition, if you have RowInit PeopleCode associated with both the record field and the component record, both programs will be executed against every record field on every row of data in the component. The RowInit PeopleCode program associated with the first record field fires first, then the RowInit PeopleCode program associated with the first component record fires. This means if you've set the value of a field with the record field RowInit PeopleCode, then reset the field with the component record RowInit PeopleCode, the second value is the one that will be displayed to the end-user.

As you can see, when you develop with PeopleCode you need to consider when and where your programs will be triggered during the Component Processor's flow of execution.



For more information see Execution Order of Events and PeopleCode.

Accessing PeopleCode Programs

Every PeopleCode program is associated with a PeopleCode event, and is often referred to by that name, such as RowInit PeopleCode, or FieldChange PeopleCode. These programs are accessible from, and associated with, different items. The following table lists which event, and hence which types of PeopleCode programs, are accessible from which item.



Page Field events (PSControlInit and PSLostFocus) are only available for pages with ActiveX controls, which are not available in the PeopleSoft Internet Architecture. The SearchInit and SearchSave events (under Component Record) are only available for the search record associated with a component.

The SearchInit and SearchSave events are only available for the search record associated with a component.

Record Field	Component Record Field	Component Record	Component	Page	Menu
FieldChange	FieldChange	RowDelete	PostBuild	Activate	ItemSelected
FieldDefault	FieldDefault	RowInit	PreBuild		

Record Field	Component Record Field	Component Record	Component	Page	Menu
FieldEdit	FieldEdit	RowInsert	SavePostChg		
FieldFormula	PrePopup	RowSelect	SavePreChg		
PrePopup		SaveEdit	Workflow		
RowDelete		SavePostChg			
RowInit		SavePreChg			
RowInsert		SearchInit			
RowSelect		SearchSave			
SaveEdit					
SavePostChg					
SavePreChg					
SearchInit					
SearchSave					
Workflow					

The following table lists the different types of PeopleCode programs and where they're accessible from in the Application Designer.

PeopleCode Programs	In Application Designer
Record field	Record definitions and page definitions
Component record field, component record, and component	Component definitions
Page and page field (ActiveX control)	Page definitions
Menu item	Menu definitions

Execution Order of Events and PeopleCode

When you develop with PeopleCode you need to consider when and where your programs will be triggered during the Component Processor's flow of execution.

In PeopleSoft, the component is the representation of a transaction. Therefore, any PeopleCode that is associated with a transaction, should be in events associated with some level of the component. If you have code that should be executed every time a field is edited, you should put it at the record field level. If you associate code with the correct transaction, you don't have to check for the component that's issuing it (such as, surrounding your code with dozens of `IF %Component =`). Records become more reusable and code is more maintainable.

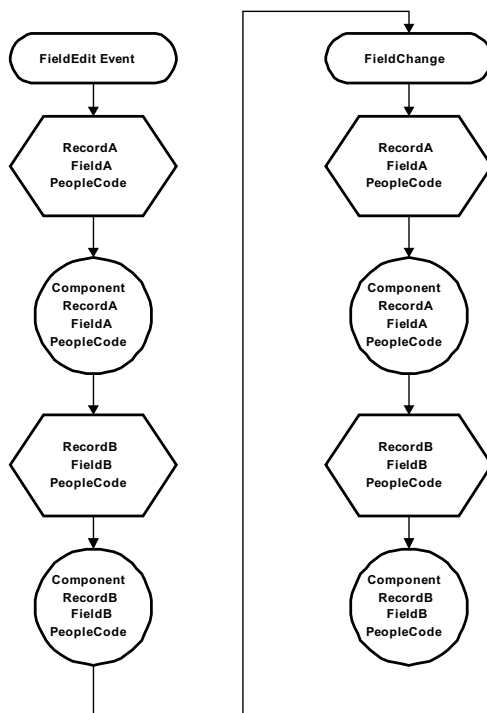
For example, if you have start and end dates for a course, you would always want to check and make sure the end date was after the start date. Therefore, your program to check the dates would go on the SaveEdit at the record field level.

All similarly named component events fire after the like-named record event. That is, the PeopleCode program associated with the record field event fires first, then the PeopleCode program associated with the like-named component event fires. This means that if you've set the value of a field with the record field PeopleCode, then reset the field with like-named component PeopleCode, the second value is the one that will be displayed to the end-user.

This is best illustrated with some examples:

Events after User Changes Field

```
Record.recordA.fieldA.FieldEdit -> Component.recordA.fieldA.FieldEdit ->
Record.recordB.fieldB.FieldEdit -> Component.recordB.fieldB.FieldEdit ->
Record.recordA.fieldA.FieldChange -> Component.recordA.fieldA.FieldChange ->
Record.recordB.fieldB.FieldChange -> Component.recordB.fieldB.FieldChange ->
```



Flow of Events and PeopleCode programs after end-user changes a field

Events after User Saves

```
Record.recordA.fieldA.SaveEdit -> Record.recordA.fieldB.SaveEdit ->
Record.recordA.fieldC.SaveEdit -> Component.recordA.SaveEdit
```

```
Record.recordB.fieldA.SaveEdit -> Record.recordB.fieldB.SaveEdit ->
Record.recordB.fieldC.SaveEdit -> Component.recordB.SaveEdit
```

```
Record.recordA.fieldA.SavePreChange -> Record.recordA.fieldB.SavePreChange ->
Record.recordA.fieldC.SavePreChange -> Component.recordA.SavePreChange
```

```
Record.recordB.fieldA.SavePreChange -> Record.recordB.fieldB.SavePreChange ->
Record.recordB.fieldC.SavePreChange -> Component.recordB.SavePreChange
```

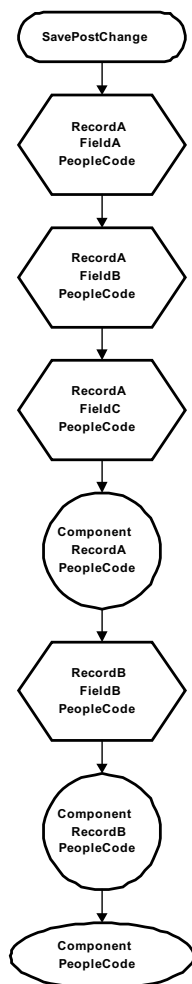
```
Record.recordA.fieldA.WorkFlow -> Record.recordB.fieldB.WorkFlow ->
Record.reocrdC.fieldC.WorkFlow
```

```
Component.Workflow
```

```
Record.recordA.fieldA.SavePostChange -> Record.recordA.fieldB.SavePostChange ->
Record.recordA.fieldC.SavePostChange -> Component.recordA.SavePostChange
```

```
Record.recordB.fieldA.SavePostChange -> Component.recordB.SavePostChange
```

```
Component.SavePostChange
```



Flow of PeopleCode programs after SavePostChange

Component Processor Behavior

This section takes a high-level look at the behaviors of the Component Processor from page startup to page display, as well as the processes initiated by end-user action after the page is displayed.



Keep in mind that this description is for components **not** running in **deferred** mode. If your component or page is running in deferred mode, refer to Deferred Processing Mode.

The next section, Processing Sequences, examines these processes in greater detail, showing the flow of system actions and PeopleCode events.

From Page Start to Page Display

Before the end-user chooses a component, the system is in reset state, in which no component is displayed. Component Processor's flow of execution begins when the end-user chooses a component from a PeopleSoft menu. The Component Processor then:

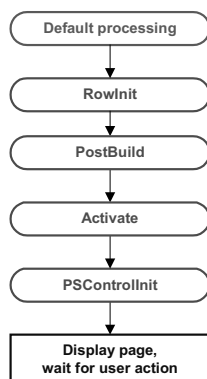
1. Performs search processing, in which it obtains and saves search key values for the component.
2. Retrieves from the database server any data needed to build the component, then builds the component, creating buffers for the component data.
3. Does any additional processing for the component, the page, or any ActiveX controls.



ActiveX control events are only applicable in the Windows client, not in the PeopleSoft Internet Architecture.

4. Displays the component and waits for end-user action.

The following flowchart shows the flow of execution at a high level.



Processing up to Page Display





PSControlInit is only applicable in the Windows client, not in the PeopleSoft Internet Architecture.

End-User Actions in the Component

Once the component is built and displayed, the Component Processor can respond to a number of possible end-user actions. The following table lists the end-user actions and briefly describes the resulting processing.



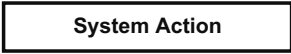



For more information on the processing sequences that result from the end-user action, follow the hypertext links in the table or read the Processing Sequences section of this chapter.

End-User Action	Description
Row Insert Processing	When the end-user requests a row insert, the Component Processor adds a row of data in the active scroll area, then displays the page again and waits for another action.
Row Delete Processing	When the end-user requests a row delete, the Component Processor flags the current row as deleted, then displays the page again and waits for another action.
Field Modification	If the end-user edits a page field, then leaves the field, the Component Processor performs standard edits (such as checking the data type and checking for values out of range). If the contents of the field do not pass the standard system edits, the Component Processor redisplay the page with an error or warning message and changes the field's color to the system color for field edit errors—usually red. Until the end-user corrects the error, the Component Processor will not let the end-user save changes or navigate to another field. If the contents of the field pass the standard system edits, the system redisplay the page and waits for further action.
Prompts	If the end-user clicks the prompt icon next to a field () a list of valid values for the prompt field displays. If the end-user clicks Return To Search or ALT+2, then press ENTER, a search dialog appears, allowing them to enter an alternate search key or partial value. If the end-user clicks the detail button next to a date field (XXX) a calendar displays.
Pop-up Menu Display	If the end-user clicks the pop-up icon next to a field () a pop-up menu appears. This can be a default pop-up menu or one that has been defined by the developer. If the end-user clicks the pop-up icon at the bottom of the page, the pop-up menu for the page displays.

ItemSelected Processing	The end-user can choose an item from a pop-up menu to execute a command.
PushButtons	The end-user can click a command push button to execute a command.
Save Processing	<p>The end-user can direct the system to save a component by clicking Save or by pressing ALT+1, then pressing ENTER. If any component data has been modified, PeopleSoft will also prompt the end-user to save a component when the Next or List icon button is selected, or when a new action or component is selected.</p> <p>The Component Processor first validates the data in the component then updates the database with the changed component data. After the update a SQL Commit command finalizes the changes.</p> <p>If a new component or new key has been requested, the Component Processor goes through Reset State to display a new component. If the end-user has not requested a new component, the Component Processor displays the page and waits for another end-user action.</p>
Exit Component	If the end-user clicks EXIT, the component goes into Reset State until the end-user requests another component.
ActiveX Control	Every ActiveX control comes with its own pre-defined events. If the end-user clicks on an ActiveX control, the Component Processor fires the relevant ActiveX control event(s). See Implementing ActiveX Controls.

Processing Sequences

This section looks at possible sequences of actions and PeopleCode events that can occur within the Component Processor's flow of execution. The logic of each sequence of actions is presented in a flow diagram with the following elements:

Symbol	Description
 System Action	Blue rectangles represent actions taken by the system.
 Decision Point	Dark red rhomboids represent branches (decision points) in the logic.
 PeopleCode Event	Green ellipses represent PeopleCode Events.
 Subsequence	Teal ellipses are subprocesses.

It's important to keep a clear distinction between the processing sequences described here and processing groups. Processing groups are units of processing that, as a whole, run either on the client or on the application server. Also, processing groups are not applicable when using the

PeopleSoft Internet Architecture: all processing runs on a server in PeopleSoft Internet Architecture.



For more information see Processing Groups.

Most of the sequences described in the following sections correspond to the high-level behaviors defined in Component Processor Behavior. However, two of the sections describe subsequences that occur only in the context of a larger sequence. These are Default Processing, which occurs in a number of different contexts, and Row Select Processing, which most commonly occurs as a part of component build in any of the Update action modes; RowSelect Processing also occurs when a ScrollSelect or related function is executed to load data into a scroll.



Keep in mind that the sequences described here are generalized, and that under some circumstances variations may occur, particularly when a PeopleCode function within a processing sequence initiates another processing sequence. For example, if a row of data is inserted or deleted programmatically during the Component Build sequence, this will set off a Row Insert or Row Delete sequence.

Also keep in mind that this description is for components **not** running in **deferred** mode. If your component or page is running in deferred mode, refer to Deferred Processing Mode.

Default Processing

In default processing, any "blank" fields in the component are set to their default value (if one is specified). The default value can be specified either in the Record Field Properties, or in FieldDefault PeopleCode. If no default value is specified, the field is left blank.



In the PeopleSoft Internet Architecture, if an end-user changes a field, but there is nothing to cause a trip to the server on that field, default processing and FieldFormula PeopleCode don't run. They only run when some other event causes a trip to the server.

Default processing is relatively complex. For the sake of clarity, the following two sections describe (1) how default processing works on the level of the individual field, and (2) how default processing works in the broader context of the component.

Field-Level Default Processing

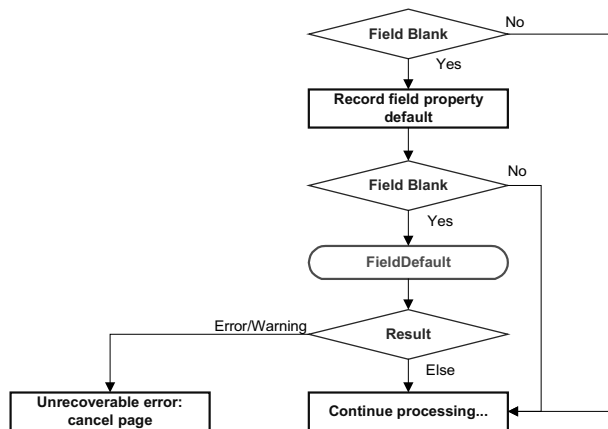
During default processing the Component Processor "looks at" all fields in all rows of the component. On each field, it does the following:

1. If the field is set to NULL (blank) for a character field or set to 0 for a numeric field the Component Processor sets the field to any default value specified in the record field properties for that field.
2. If no default value for the field is defined in the record field properties, then the Component Processor fires the FieldDefault event, which triggers any FieldDefault PeopleCode associated with the record field or the component record field.
3. If an **Error** or **Warning** executes in any FieldDefault PeopleCode a runtime error occurs that forces the end-user to cancel the page.



Important! Avoid using **Error** and **Warning** statements in FieldDefault PeopleCode.

The following flowchart shows this logic:



Field-Level Default Sequence Flow

Default Processing on Component Level

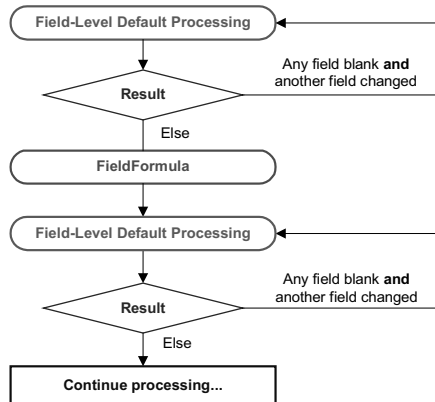
Under normal circumstances, default processing in a component is relatively simple: each field on each row of data undergoes Field-Level Default Processing. For typical development tasks, this is all you need to be concerned with. However, the complete context of default processing is somewhat more complex.

1. Field-Level Default Processing is done on all fields on all rows of data in the component.
2. If any field is still blank **and** any other field in the component has changed, Field-Level Default Processing may be repeated, in case a condition changed that causes default processing to now assign a value to something that was left blank previously.
3. The FieldFormula Event fires on all fields on all rows of data in the component. This PeopleCode event is now often used for FUNCLIB_ (function library) record definitions to store shared functions, so normally no PeopleCode programs execute.
4. **If** the FieldFormula Event changed anything, then Field-Level Default Processing is done again, in case FieldFormula PeopleCode blanked out a field or changed something that causes

default processing to now assign a value to something that was left blank previously. Since there shouldn't be any FieldFormula PeopleCode, this is unlikely to affect the development process or performance.

5. Once again, if any field is still blank **and** any other field in the component has changed, Field-Level Default Processing is repeated.

The following flowchart shows this logic:



Default Processing on Component Level

Search Processing in Update Modes

If the end-user chooses any of the Update action modes (Update, Update/Display All, or Correction), the Component Processor begins Update Mode search processing.

1. The SearchInit PeopleCode event fires, which triggers any SearchInit PeopleCode associated with the record field or the component search record, on the keys or alternate search keys in the component search record. This permits you to control the search dialog field values or the search dialog appearance programmatically, or to perform other processing prior to the search dialog display.



Set the search record for the component in the Component Properties.

For example, the following program in SearchInit PeopleCode on the component search key record field EMPLID sets the search key page field to the user's employee ID, grays out the page field, and enables the user to modify the user's own data in the component:

```

EMPLID = %EmployeeId;

Gray (EMPLID);

AllowEmplIdChg(true);

```



Effects of SetSearchDialogBehavior: Normally a search dialog is displayed. However, SetSearchDialogBehavior can be used to set the behavior of the search dialog before it is displayed. If SetSearchDialogBehavior is set to **Force display**, the dialog will be displayed even if all required keys have been provided. You can also set SetSearchDialogBehavior to **skip if possible**.

2. The search dialog and prompt list appears, in which the user can enter search keys, or select an Advanced search, to enter alternate search keys.



Effects of SetSearchDefault: Normally the values in the search dialog are not set to default values. However, if the SetSearchDefault function was executed in SearchInit PeopleCode for any of the search key or alternate search fields, those specific field in the dialog are set to their system default. No other default processing occurs (that is, the FieldDefault event does not fire).

3. The end-user enters a value or partial value in the search dialog, then clicks on Search.
4. The SearchSave PeopleCode event fires, which triggers any SearchSave PeopleCode associated with the record field or the component search record, on the search keys or alternate search keys in the search Record. This allows you to validate the user entry in the Search Dialog by testing the value in the search record field in PeopleCode and, if necessary, issuing an error or warning. If an **Error** statement is executed in SearchSave, the end-user is sent back to the search dialog. If a **Warning** is executed the end-user can click **OK** to continue or click cancel to return to the search dialog and enter new values.

If partial values are entered, such that the Component Processor can select multiple rows, then the Prompt List dialog is filled and the end-user can choose a value. If key value(s) from the search dialog are blank or if the system can't select any data based on the end-user entry in the search dialog, the system displays a message and re-displays the search dialog. If the values entered produce a unique value, the Prompt List isn't filled: instead, the end-user is taken directly to the page.

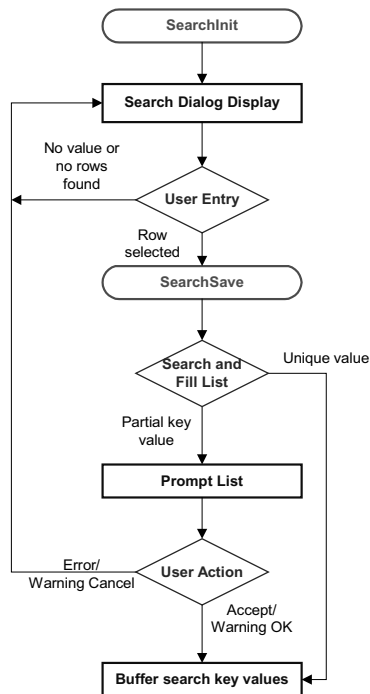


Effects of SetSearchEdit: Normally no system edits are applied when the end-user changes a field in the search dialog. However, if SetSeachEdit is executed for specific search dialog fields in SearchInit PeopleCode, the system edits will be applied to those fields after the end-user changes a field and either leaves the field or clicks Search. If the end-user entry in the field fails the system edits, the system displays a message, highlights the offending field, and returns the end-user to the dialog. The FieldEdit and SaveEdit PeopleCode events do not fire.

SearchSave does **not** fire after values are selected from the search list. If you need to validate data entered in the search dialog, use the Component PreBuild event to do so.

5. The Component Processor buffers the search key values. If the end-user then opens another component while this component is active, the Component Processor will use the same search key values and bypass the search dialog.

The following flowchart shows this logic. (It does not show the effects of executing the SetSearchDefault and SetSearchEdit functions.)



Search Processing Logic in Update Modes



You can use the IsSearchDialog function to create PeopleCode that runs only during search processing. To create processes that run only in a specific action mode, use the %Mode system variable. This could be useful in code that is part of a library function and that is invoked in places other than from the search dialog. It could also be used in PeopleCode associated with a record field that appears in pages as well as in the search dialog.

Search Processing in Add Modes

When the end-user starts up a component in Add or Data Entry mode, the Component Processor:

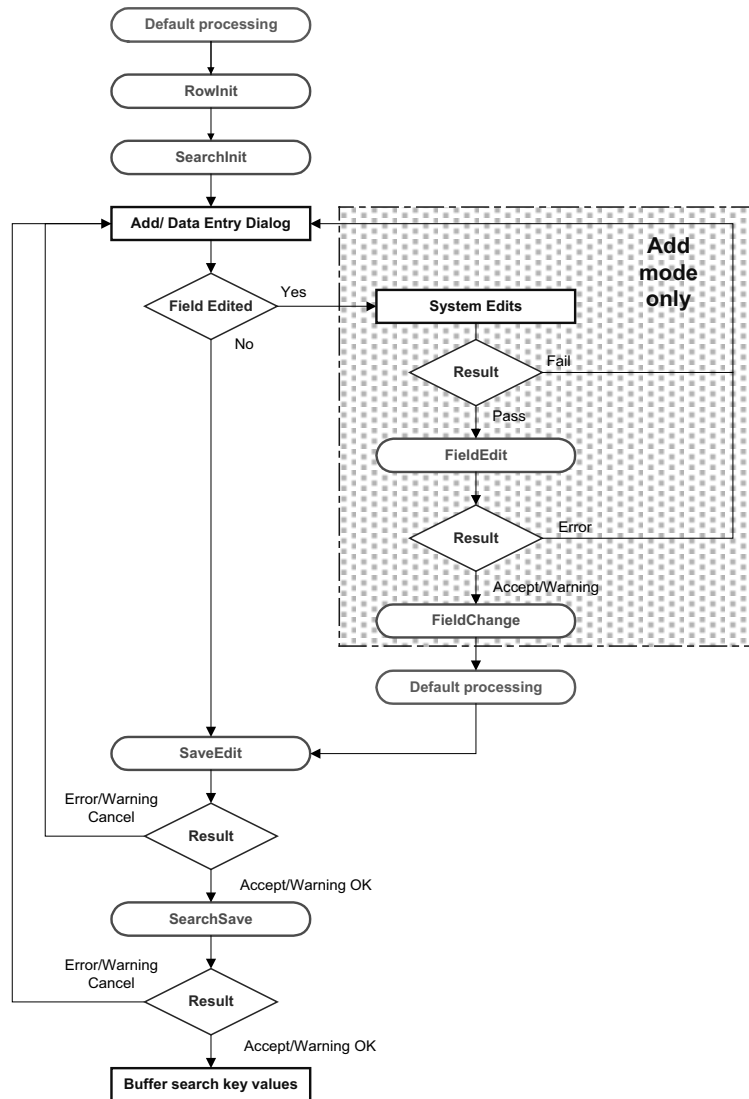
1. Runs default processing on the high-level keys to be displayed in the Add or Data Entry dialog.
2. Fires the RowInit event, which triggers any RowInit PeopleCode associated with the record field or the component record, on the Add or Data Entry dialog fields.

3. Fires the SearchInit event on dialog fields, which triggers any SearchInit PeopleCode associated with the record field or the component search record. This allows you to execute PeopleCode programs before the dialog is displayed.
4. Displays the Add or Data Entry dialog.
5. If the end-user changes a dialog field then leaves the field or clicks **OK**:
 - *In Add Mode only (not Data Entry mode)* a Field Modification processing sequence occurs.
 - Default processing is run on the Add or Data Entry dialog field(s). Normally this won't have any effect, because the field(s) will have a value.
6. When the end-user OK's the dialog, fires the SaveEdit event, which triggers any PeopleCode associated with the record field or the component record..
7. Fires the SearchSave event, which triggers any SearchSave PeopleCode associated with the record field or the component search record. This allows you to validate the end-user entry in the dialog. If an **Error** statement is executed in SearchSave, the end-user is sent back to the Add or Data Entry dialog. If a **Warning** is executed the end-user can click **OK** to continue or click cancel to return to the dialog and enter new value(s).
8. Buffers the search key values and continues processing.



It's simpler than it seems. If you compare the following diagram with Search Processing in Update Modes, you will notice that the add modes are considerably more complex and involve more PeopleCode events. However, in practice PeopleCode development is similar in both cases: PeopleCode that runs before the dialog appears (for example to control dialog appearance or set values in the dialog fields) generally goes in the SearchInit event; PeopleCode that validates end-user entry in the dialog goes in the SearchSave event.

The following flowchart shows this logic.



Search Processing Logic in Add and Data Entry Modes



You can use the `IsSearchDialog` function to create PeopleCode that runs only during search processing. To create processes that run only in a specific action mode, use the `%Mode` system variable. This could be useful in code that is part of a library function and that is invoked in places other than from the search dialog. It could also be used in PeopleCode associated with a record field that appears in pages as well as in the search dialog.

Component Build Processing in Update Modes

Once the Component Processor has saved the search keys values for the component, it uses the search key values to select rows of data from the database server using a SQL Select. After the rows are retrieved, the Component Processor:

1. Performs Row Select Processing, in which rows of data that have already been selected from the database server can be filtered before they are added to the component buffer.
2. Fires the PreBuild event, which triggers any PreBuild PeopleCode associated with the component record, giving you an opportunity to set global or component scope variables that can be used later by PeopleCode located in other events. PreBuild is also used to validate data entered in the search dialog, after a prompt list is displayed.



If a PreBuild PeopleCode program issues an error or warning, the end-user is returned to the search page. If there is no search page, that is, the search record has no keys, a blank component page displays.

3. Performs Default Processing on all the rows and fields in the component.
4. Fires the RowInit event, which triggers any RowInit PeopleCode associated with the record field or the component record. The RowInit event gives you an opportunity to programmatically initialize the values of non-blank fields in the component.
5. Fires the PostBuild event, which triggers any PostBuild PeopleCode associated with the component record, giving you an opportunity to set global or component scope variables that can be used later by PeopleCode located in other events.
6. Fires the Activate event, which triggers any Activate PeopleCode associated with the page about to be displayed, allowing you to programmatically control the display of that page.
7. Fires the PSControlInit event, which triggers any PSControlInit PeopleCode for any ActiveX controls on the page. This allows you to initially set the control and fill it with data.

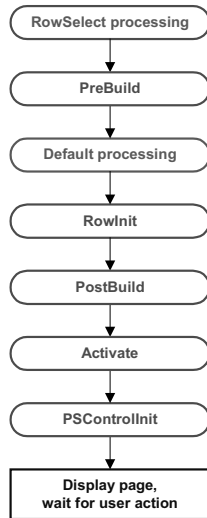


PSControlInit is only applicable in the Windows client, not in the PeopleSoft Internet Architecture.

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.

8. Displays the component and waits for end-user action.

The following flowchart shows this logic.



Component Build Processing in Update Modes



PSControllInit is only applicable in the Windows client, not in the PeopleSoft Internet Architecture.

Row Select Processing

Row Select processing provides an opportunity for PeopleCode to filter out rows of data after they have been retrieved from the database server via a SQL Select and before they are copied to the component buffers.

Row Select processing is a subprocess of Component Build Processing in Add Modes. It also occurs after a ScrollSelect or related function is executed.



This technique is not often used in recent applications, because it is far more efficient to filter out the rows using a search view, an effective dated record, the ScrollSelect or a related function, or the Select method, *before* they are brought down to the browser.

In Row Select processing the Component Processor:

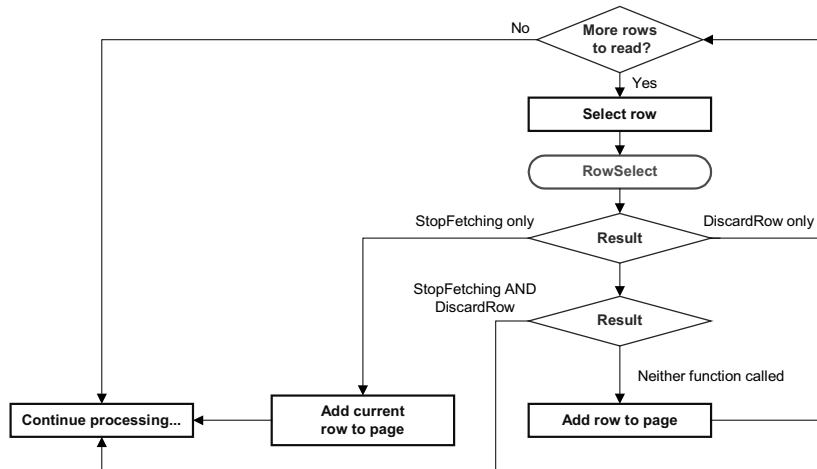
1. Checks whether there are any more rows to add to the component.
2. Fires the RowSelect event, which triggers any RowSelect PeopleCode associated with the record field or component record. This provides an opportunity for PeopleCode to filter rows using the StopFetching and DiscardRow functions. StopFetching causes the system to add the current row to the component, then stop adding rows to the component. DiscardRow filters out a current row, then continues the Row Select process.
3. If neither StopFetching nor DiscardRow is called, the Component Processor adds the rows to the page and checks for the next row. The process continues until there are no more rows to

add to the component buffers. If both StopFetching and DiscardRow are called, the current row is not added to the page, and no more rows are added to the page.



In RowSelect PeopleCode, you can only refer to record fields on the record that is currently being processed because the buffers are in the process of being populated. This means the data might not be present.

The following flowchart shows this logic:



Row Select Processing Logic

Component Build Processing in Add Modes

After search processing in Add or Data Entry modes, the Component Processor:

1. Runs default processing on all page fields. This gives you an opportunity to set default fields programmatically using FieldDefault PeopleCode.
2. Fires the RowInit event on all fields in the component, which triggers any RowInit PeopleCode associated with the record field or component record. This allows you to initialize the state of page controls using RowInit PeopleCode before they are displayed. (RowInit allows you to set the values of non-blank fields programmatically, whereas default processing is used to set blank fields to their default values.)
3. Fires the PostBuild event, which triggers any PostBuild PeopleCode associated with the component record, giving you an opportunity to set global or component scope variables that can be used later by PeopleCode located in other events.
4. Fires the Activate event, which triggers any Activate PeopleCode associated with the page about to be displayed, allowing you to programmatically control the display of that page.
5. Fires the PSControlInit event, which triggers any PSControlInit PeopleCode for any ActiveX controls on the page. This allows you to initially set the control and fill it with data.

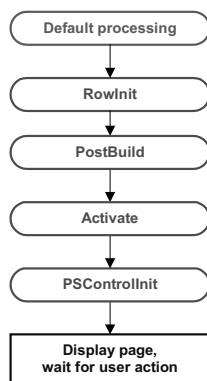


PSControlInit is only applicable in the Windows client, not in the PeopleSoft Internet Architecture.

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.

6. Displays a new component using the search key(s) obtained from the Add or Data Entry dialog with other fields set to their default values.

The following flowchart shows the logic:



Logic of Component Build Processing in Add Modes



PSControlInit is only applicable in the Windows client, not in the PeopleSoft Internet Architecture.

Field Modification

The Field Modification processing sequence occurs after the end-user does any of the following:

- Changes the contents of a field, then leaves the field.
- Changes the state of a radio button or checkbox.
- Clicks on a command push button.



Modifying an ActiveX control is not part of the Field Modification processing sequence. Any changes to an ActiveX control are handled by the events local to that Active X control. For more information see Implementing ActiveX Controls.

In this sequence the Component Processor:

1. Performs standard system edits.

To reduce trips to the server, some processing must be done locally on the machine where the browser is located, while some is performed on the server.

Standard system edits can be done either on the browser utilizing local JavaScript code or on the application server. The following chart outlines where these system edits are done.

System Edits	Where executed
checking data type	Browser
formatting	Application server / browser
updating current or history record	Browser
effective date	Browser
effective date/sequence	Browser
new effective date in range	Browser
duplicate key	Application server
current level is not effective-dated but one of its child scrolls is	Browser
required field	Browser
date range	Browser
prompt table	Application server
translate table	Browser
yes/no table	Depends on field type. Browser if the field is a checkbox. Application server if the field is an edit box and the values are Y or N.



Default processing for the field can be done on the browser **only** if the default value is specified as a constant in the Record Field Properties. If the field contains a default, these defaults will only occur upon Component initialization. Then, if a user then blanks out a default value, it will not be reinitialized.

The required fields check is *not* performed on derived work fields when you tab out of a field.

If the data fails the system edits, the Component Processor displays an error message and highlights the field in the system color for errors (usually red).

- If the field passes the system edits, Component Processor fires the FieldEdit PeopleCode event, which triggers any FieldEdit PeopleCode associated with the record field or the component record field. This permits you to perform additional data validation in PeopleCode. If an **Error** statement is called in any FieldEdit PeopleCode, Component Processor treats the error as it does a system edit failure: a message is displayed and the

offending field is highlighted. If a **Warning** statement is executed in any FieldEdit PeopleCode, a warning message appears alerting the end-user to a possible problem, but the system accepts the change to the field.

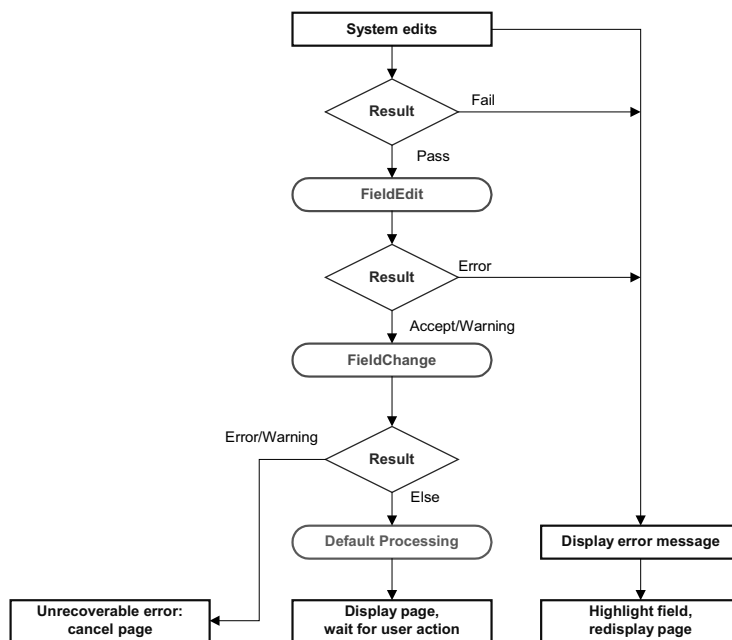
3. If the field change is accepted, the Component Processor writes the change to the component buffer, then fires the FieldChange event, which triggers any FieldChange PeopleCode associated with the record field or the component record field. This event allows you to add processes other than validation initiated by the changed field value, such as changes to page appearance or recalculation of values in other page fields. An **Error** or **Warning** statement in any FieldChange PeopleCode causes an unrecoverable runtime error and forces cancellation of the page.



Important! We recommend against putting an **Error** or **Warning** statement in any FieldChange PeopleCode. All data validation should be performed in FieldEdit.

After FieldChange processing, Component Processor runs default processing on all page fields, then redisplay the page. If the end-user has blanked out the changed field, or if **SetDefault** or a related function is executed, and the changed field has a default value specified in the record field definition or any FieldDefault PeopleCode, the field will be re-initialized to the default.

The following flowchart shows this logic:



Logic of Field Modification Processing



Note on PSControlInit. If the end-user changes a field using a drop down list or other prompt, the page will be redrawn and the PSControlInit event will fire after default processing. If the end-user scrolls down, the page will be redrawn and PSControlInit will fire. If the end-user tabs into a field, types a change, then tabs out of the field, PSControlInit will not fire. **Also note:** PSControlInit is only applicable in the Windows client, not in the PeopleSoft Internet Architecture.

Note on processing groups. The logic shown here occurs as one sequence initiated by a end-user changing a field value. However, for purposes of partitioning, the FieldEdit and FieldChange parts of the sequence form two different processing groups. In three-tier mode, FieldEdit PeopleCode always runs on the client (in Windows.) FieldChange and related processing can run on either the client or the application server. For more information, see Processing Groups. **Also note:** All events run on the application server in the PeopleSoft Internet Architecture. Processing groups only apply to Windows Client.

Row Insert Processing

Row Insert processing occurs when:

- The end-user requests a row insert in a scroll by pressing ALT+7 then pressing ENTER, clicking the Insert Row button, or the New button.
- A PeopleCode **RowInsert** function or a **InsertRow** method requests a row insert.

In either case the Component Processor:

1. Inserts a new row of data into the active scroll area. If the scroll area has a dependent scroll area, the system inserts a single new row into the blank scroll area—it continues until it reaches the lowest-level scroll area.
2. Fires the RowInsert PeopleCode event, which triggers any RowInsert PeopleCode associated with the record field or the component record. This event hits fields only on the inserted row, and any dependent rows that were inserted on lower-level scroll areas.
3. Runs default processing on all component fields. Normally this will affect only the inserted row fields, and fields on its dependent rows, since other rows will already have undergone default processing.
4. Fires the RowInit PeopleCode event, which triggers any RowInit PeopleCode associated with the record field or the component record. This event affects fields only on the inserted row and any dependent rows that were inserted.
5. Fires the PSControlInit event, which triggers any PSControlInit PeopleCode for any ActiveX controls on the page. This allows you to initially set the control and fill it with data.



PSControlInit is only applicable in the Windows client, not in the PeopleSoft Internet Architecture.

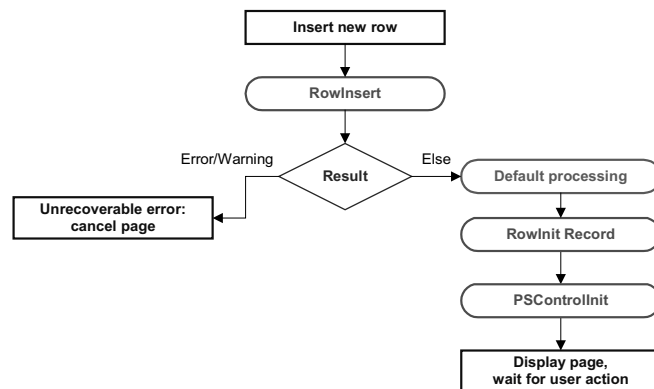
If you place an ActiveX control in a scroll that has an occurs count greater than one, the PSControlInit event will fire once for every occurrence. That is, if the scroll has an occurs count of 3, PSControlInit will fire 3 times.

6. Redisplays the page and waits for end-user action.



Important! We recommend against putting an **Error** or **Warning** statement in RowInsert PeopleCode. All data validation should be performed in FieldEdit or SaveEdit PeopleCode.

The following flowchart shows this logic:



Logic of Row Insert Processing



If none of the data fields in the new row are changed after the row has been inserted (either programmatically or by the end-user), when the page is saved, the new row isn't inserted into the database.



PSControlInit is only applicable in the Windows client, not in the PeopleSoft Internet Architecture.

Row Delete Processing

Row Delete processing occurs when:

- The end-user requests a row delete in a scroll by pressing ALT+8 then pressing ENTER, clicking the Delete Row button, or clicking the Delete button.

- A PeopleCode **RowDelete** function or a **DeleteRow** method requests a row delete.

In any case the Component Processor:

1. Fires the RowDelete PeopleCode event, which triggers RowDelete PeopleCode associated with the record field or the component record. This event hits fields on the deleted row and any dependent child scrolls. RowDelete PeopleCode allows you to check for conditions and control whether the end-user can delete the row. An **Error** statement displays a message and prevents the end-user from deleting the row. A **Warning** statement displays a message alerting the end-user about possible consequences of the deletion, but permits deletion of the row.
2. If the deletion is rejected the page is redisplayed after the error message.
3. If the deletion is accepted, the row, and any child scrolls dependent on the row, are flagged as deleted. It no longer appears in the page, but it is not physically deleted from the buffer and can be accessed by PeopleCode all the way through the SavePostChange event (note, however, that SaveEdit PeopleCode is not run on deleted rows).
4. Runs default processing on all component fields.
5. Fires the PSControlInit event, which triggers any PSControlInit PeopleCode for any ActiveX controls on the page. This allows you to initially set the control and fill it with data.



PSControlInit is only applicable in the Windows client, not in the PeopleSoft Internet Architecture.

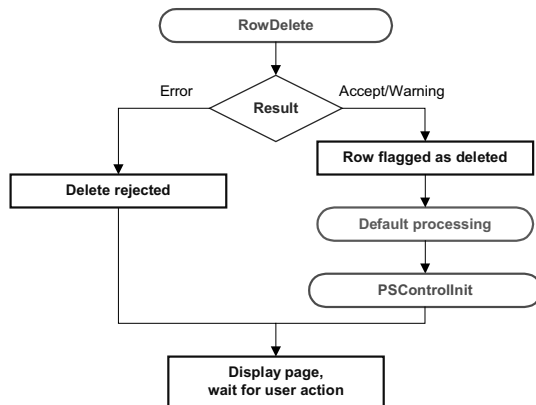
If you place an ActiveX control in a scroll that has an occurs count greater than one, the PSControlInit event will fire once for every occurrence. That is, if the scroll has an occurs count of 3, PSControlInit will fire 3 times.

6. Redisplays the page and waits for end-user action.
-



Note about deleted rows. PeopleCode programs are triggered on rows flagged as deleted in SavePreChange and SavePostChange PeopleCode. Use the RecordDeleted function, or the IsDeleted Row property, to test whether a row has been flagged as deleted. You can also access rows flagged as deleted by looping through the rows of a scroll area using a For loop delimited by the value returned by the TotalRowCount function or the RowCount Rowset property.

The following flowchart shows this logic:



Logic of Row Delete Processing



PSControlInit is only applicable in the Windows client, not in the PeopleSoft Internet Architecture.

PushButtons

When the end-user presses a pushbutton, this initiates the same processing as changing a field. Typically PeopleCode programs launched by push buttons are placed in the FieldChange event.



For more information see Field Modification.

Prompts

No PeopleCode event fires as a result of prompts, returning to the search dialog or displaying a calendar. This process is controlled automatically by the system.

Pop-up Menu Display

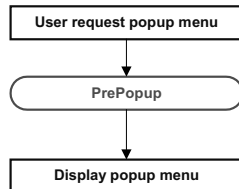
To display a pop-up menu, an end-user can click on the pop-up button, either next to a field or at the bottom of a page (if the page has a pop-up menu associated with it.) The end-user can display a standard pop-up menu on a page field if no pop-up menu has been defined by an application developer for that page field.

The PrePopup PeopleCode event fires only if the end-user displays a pop-up menu defined by an application developer on a page field. It doesn't fire before a pop-up menu attached to the page background.

The PrePopup PeopleCode event allows you to disable, check, or hide menu items in the pop-up.

PrePopup PeopleCode menu item operations (such as HideMenuItem, EnableMenuItem, and so on) work with pop-up menus attached to a grid, not a field in a grid, **only** if the PrePopup PeopleCode meant to operate on that pop-up menu resides in the record field that is attached to the **first** column in the grid. It doesn't matter if the first field is visible or hidden.

The following flowchart shows this logic:



Logic of PrePopup Processing

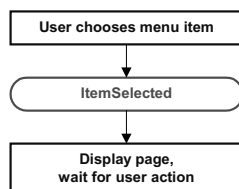
ItemSelected Processing

ItemSelected processing occurs when a end-user chooses a menu item from a pop-up menu. This fires the ItemSelected PeopleCode event, which is a menu PeopleCode event.



For more information see Menu Item PeopleCode.

The following flowchart shows this logic:



Logic of ItemSelected Processing

PSLostFocus Processing

PSLostFocus processing occurs after a end-user has selected an ActiveX control on a page, then leaves the ActiveX control. This fires the PSLostFocus PeopleCode event.

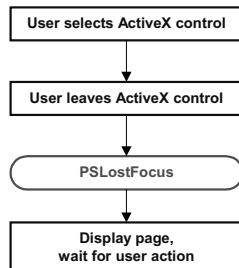


PSLostFocus is only applicable in the Windows client, not in the PeopleSoft Internet Architecture.



For more information see Implementing ActiveX Controls.

The following flowchart shows this logic:



Logic of PSLostFocus Processing

Save Processing

The end-user can direct the system to save a component by clicking Save or by pressing ALT+1, then ENTER. PeopleSoft will also prompt the end-user to save a component when the Next or List icon button is selected, or when a new action or component is selected. In all cases the Component Processor:

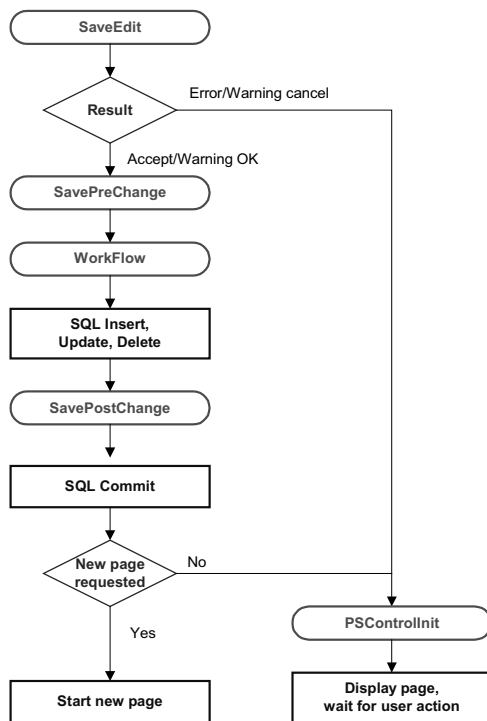
1. Fires the SaveEdit PeopleCode event, which triggers any SaveEdit PeopleCode associated with a record field or a component record. This gives you a chance to cross-validate the page fields before saving, checking consistency among the page field values. An **Error** statement in SaveEdit PeopleCode displays a message and then re-displays the page, aborting the save. A **Warning** statement allows the end-user to cancel save processing by pressing **Cancel**, or continue with save processing by pressing **OK**.
2. Fires the SavePreChange event, which triggers any SavePreChange PeopleCode associated with a record field, a component record, or a component. SavePreChange PeopleCode gives you a chance to process data after validation and before the database is updated.
3. Fires the Workflow event, which triggers any Workflow PeopleCode associated with a record field or a component. Workflow PeopleCode should be used only for workflow-related processing (**TriggerBusinessEvent** and related functions).
4. Updates the database with the changed component data, performing any necessary SQL Inserts, Updates, and Deletes.
5. Fires the SavePostChange PeopleCode event, which triggers any SavePostChange PeopleCode associated with a record field, a component record, or a component. SavePostChange PeopleCode can be used for processing that needs to occur after the database update, such as updates to other database tables not in the component buffer.
6. Issues a SQL Commit to the database server.
7. Depending on whether the end-user has requested a new component, either:

- fires the PSControlInit event and re-displays component
- or
- starts a new component.



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.

PSControlInit is only applicable in the Windows client, not in the PeopleSoft Internet Architecture. **Important!** Never use an **Error** or **Warning** statement in any Save Processing event other than SaveEdit. All component data validation should be performed in SaveEdit.



Logic of Save Processing



Note on processing groups. The logic shown here occurs as one sequence initiated by a end-user saving a component. The Component Save processing group, which, in three-tier mode, can run on either the client or the application server. Though SaveEdit is part of the Component Save processing group, individual SaveEdit PeopleCode programs can be specified to run on either the client or the server. For more information, see Processing Groups.

PSControlInit is only applicable in the Windows client, not in the PeopleSoft Internet Architecture.

Exit Component

When the end-user clicks EXIT, the component is canceled, the component buffer is cleared, the Component Processor returns to reset state and the user must login again. No PeopleCode events are fired.

PeopleSoft Internet Architecture Processing Considerations

- In the PeopleSoft Internet Architecture, if an end-user changes a field, but there is nothing to cause a trip to the server on that field, default processing and FieldFormula PeopleCode don't run. They only run when some other event causes a trip to the server.

This means other fields that depend on the first field using FieldFormula or default PeopleCode are not updated until the next time there is a server trip.

- In application that run on the PeopleSoft portal, external, dynamic hyperlink information **must** be placed in RowInit PeopleCode. If it's placed in FieldChange PeopleCode, it won't work.



This consideration doesn't apply to applications that run on Internet Client.

Deferred Processing Mode

When a component is running in deferred processing mode, trips to the server are reduced. When deploying some pages in the browser, you may want the user to be able to input data with minimal interruption or trips to the server. Each trip to the server results in the page being complete refreshed on the browser, which may cause the display to flicker. It can also slow down your application. By specifying a component as Deferred Processing Mode, you can achieve better performance.

If you've specified Deferred Processing Mode for a component, you can then specify whether a page within a component, or a field on a page, will also do their processing in deferred mode. The default is for all pages and fields to allow deferred processing.

Specifying that a field or page allows deferred processing, then **not** setting the component to Deferred Processing Mode, will **not** start deferred processing mode. You must set the component first.



For more information on how to set Deferred Processing Mode for components, pages or fields, see the appropriate sections in Application Designer.

The characteristics of this mode are:

1. Field modification processing is deferred.

No field modification processing is done on the browser. FieldEdit and FieldChange PeopleCode, as well as other edits, such as required field checks, formats, and so on, will not run until a specific user action occurs. Several actions cause field modification processing to execute, for example, clicking on a push button or hyperlink, navigating to another page in the component, and saving the page. The following actions will **not** cause field processing:

- launching an External Link
- clicking a List (Search)
- clicking a Process push button

Deferred processing mode affects the appearance of pages in significant ways. For example, related processing will not be done when the user tabs off a field. Consequently, PeopleSoft recommends avoiding related fields for components that use this mode.

2. Drop-down list values are static while the page is displayed on the browser.

Drop-down list values will be generated on the application server when generating the HTML for the page.

If translate values are used to populate the drop-down list and the current record contains an effective date, that date is static while the page is displayed. This means the drop-down list values may become out of date.

If prompt table values are used to populate the drop-down list, the high-order key field values for the prompt table are static while the page is displayed. This means the drop-down list values may become out of date.

PeopleSoft recommends that drop-down lists used on pages executed in deferred mode shouldn't have any interdependencies because the lists may become out of date so easily.

3. No field modification processing is done during prompt button processing.

When the user clicks a prompt button, a trip is made to the application server (if values weren't already downloaded) to select the search results from the database and to generate the HTML for the prompt dialog. During this trip to the application server, field modification processing is **not** performed because this may cause an error message for another field on the page, and this may confuse the user. While the page displays the high-order key field values for the prompt table should be static or not require field modification processing. Display-only, drop-down list, radio button, and check box fields do not require field modification processing. Field values that do not require field modification processing are temporarily written to the Component buffer, without any field modification processing being performed on them, including FieldEdit and FieldChange PeopleCode. The system restores the original state of the page processor before returning to the browser.

4. Field modification processing executes in field layout order.

The entire field modification processing sequence executes in field layout order for each field. If a field passes the system edits and FieldEdit PeopleCode, the field value is written to the component buffer. If an error occurs, field modification processing stops and the system generates new HTML for the page with the field in error highlighted and sent to the browser.

5. PeopleCode dependencies between fields on the page will not work as expected.

PeopleSoft recommends avoiding PeopleCode dependencies between fields on pages displayed in deferred processing mode. Also, avoid FieldChange PeopleCode that changes the display.

The following are examples of PeopleCode dependencies between fields on the page and the application server's action. In the following examples, field A comes before field B, which comes before field C.

- Field A has FieldChange PeopleCode that hides or grays field B. The value in field B of the page that was submitted from the browser will be discarded.
- Field B has FieldChange PeopleCode that hides or grays field A. The change made by the user for field A, if any, remains in the Component buffer.
- Field A has FieldChange PeopleCode that changes the value in the Component buffer for field B. If the value in field B of the page that was submitted from the browser passes the system edits and FieldEdit PeopleCode, it will be written to the Component buffer, overriding the change made by field A's FieldChange PeopleCode.
- Field B has FieldChange PeopleCode that changes the value in the Component buffer for field A. The change made by field B's FieldChange PeopleCode overrides the change made by the user to field A, if any.
- Field A has FieldChange PeopleCode that un-hides or un-grays field B. Field B has the value that was already in the Component buffer. If the user requests a different page or finishes, they may not have the opportunity to enter a value into field B and therefore the value may not be correct.
- Field B has FieldChange PeopleCode that changes the value in the Component buffer for field A, but field C has FieldChange PeopleCode that hides or grays field B. The change made by field B's FieldChange PeopleCode, a field that is now hidden or grayed, overrides the change made by the user to field A, if any.

There are other examples of PeopleCode dependencies between fields on the page. PeopleSoft recommends avoiding such dependencies by moving FieldChange PeopleCode logic from individual fields to save processing for the Component or FieldChange PeopleCode on a PeopleCode Command push button.

6. Not all push buttons cause field modification processing to execute.

Specifically, External Link, List (Search), and Process push buttons do not cause field modification processing to execute.

7. A PeopleCode Command push button can be used to cause field modification processing to execute.

An application can include to have a push button for the sole purpose of causing field modification processing to execute. The result is a new page showing any display changes that resulted from field modification processing.

8. A scroll push button (hyperlink) causes field modification processing to execute.

PeopleCode Events

The preceding sections discussed when sequences of PeopleCode events occur, and under what conditions they trigger PeopleCode programs. The following sections discuss the individual PeopleCode events.



Note on "PeopleCode Types." The term "PeopleCode type" is still frequently used, but it fits poorly into the new PeopleTools object-based, event-driven metaphor. The term *PeopleCode event* should now be used instead. However, it's often convenient to qualify a class of PeopleCode programs triggered by a specific event with the event name; for example, PeopleCode programs associated with the RowInit events are collectively referred to as *RowInit PeopleCode*.

Activate Event

The Activate event is fired every time the page is activated. This means when the page is first brought up by the end-user, or if a end-user tabs between different pages in a component. Every page has its own Activate event.

The main purpose of the Activate event is to segregate the PeopleCode that is related to a specific page from the rest of your application's PeopleCode. PeopleCode related to page display or page processing, such as enabling a field or hiding a scroll, is best put in this event. Also, you can use this event for security validation: if an user doesn't have clearance to view a page in a component, you would put the code for hiding it in this event.



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 Activate event. The Activate event isn't associated with a specific row and record at the point of execution. This means you can't use functions such as GetRecord, GetRow, and so on, that rely on context, without specifying more context.

Activate PeopleCode can only be associated with pages.

This event is only valid for pages that are defined as Standard or Secondary. This event is **not** supported for subpages.



For more information see Component Build Processing in Update Modes and Component Build Processing in Add Modes.

FieldChange Event

FieldChange PeopleCode is used to recalculate page field values, change the appearance of page controls, or perform other processing that results from a field change other than data validation. To validate the contents of the field, use FieldEdit Event.

The FieldChange event fires on the specific field and row that just changed.

Do not use **Error** or **Warning** statements in FieldChange PeopleCode: these statements cause a runtime error that forces the end-user to cancel the page without saving changes.

FieldChange PeopleCode is often paired with RowInit PeopleCode. In these RowInit/FieldChange pairs, the RowInit PeopleCode checks values in the component and initializes the state or value of page controls accordingly. FieldChange PeopleCode then rechecks the values in the component during page execution and resets the state or value of page controls.

To take a simple example, suppose you have a Derived/Work field called PRODUCT, the value of which is always the product of page field A and page field B. When the component is initialized you would use RowInit PeopleCode to initialize PRODUCT equal to $A * B$ when the component starts up or when a new row is inserted. You could then attach FieldChange PeopleCode programs to both A and B which also set PRODUCT equal to $A * B$. Whenever the end-user changes the value of either A or B, PRODUCT would be recalculated.

FieldChange PeopleCode can be associated with record fields and component record fields.



For more information see Field Modification.

FieldDefault Event

The FieldDefault PeopleCode event allows you to programmatically set fields to default values when they are initially displayed. This event is fired on all page fields as part of many different processes; however it only triggers PeopleCode programs when the following conditions are all true:

- The page field is still blank after applying any default specified in the record field properties. (This will be true if there is no default specified, if a null value is specified, or if a 0 is specified for a numeric field.)
- The field has a FieldDefault PeopleCode program.

In practice, FieldDefault PeopleCode normally defaults fields when new data is being added to the component; that is, in Add mode and when a new row is inserted into a scroll.

You must attach FieldDefault PeopleCode to the specific field that is being defaulted.



An **Error** or **Warning** issued from FieldDefault PeopleCode will cause a runtime error and force cancellation of the component.

FieldDefault PeopleCode can be associated with record fields and component record fields.



For more information see Default Processing.

FieldEdit Event

FieldEdit PeopleCode is used to validate the contents of a field, supplementing the standard system edits. If the data does not pass the validation, the PeopleCode program should display a message using the **Error** statement, which redisplay the page, showing an error message and turning the field red.

If you wish to permit the field edit, but alert the end-user to a possible problem, use a **Warning** statement instead of **Error**. A **Warning** statement displays a warning dialog with **OK** and **Explain** buttons. It permits field contents to be changed and continues processing as usual after the end-user clicks **OK**.

If your validation needs to check the contents of more than one field—that is, if the validation is checking for consistency across page fields—then you need to use SaveEdit PeopleCode instead of FieldEdit.

The FieldEdit event fires on the specific field and row that just changed.

FieldEdit PeopleCode can be associated with record fields and component record fields.



For more information see Field Modification.

FieldFormula Event

The FieldFormula event is a vestige of early versions of PeopleTools, and is not used in recent applications. Because FieldFormula PeopleCode fires in many different contexts and triggers PeopleCode on every field on every row in the component buffer, it can seriously degrade the performance of your application. In recent PeopleSoft applications, the RowInit and FieldChange events are used rather than FieldFormula.



In the PeopleSoft Internet Architecture, if an end-user changes a field, but there is nothing to cause a trip to the server on that field, default processing and FieldFormula PeopleCode don't run. They only run when some other event causes a trip to the server.

As a matter of convention, FieldFormula is now often used in FUNCLIB_ (function library) record definitions to store shared functions. This is purely a matter of convention, and in fact you can store shared functions in any PeopleCode event.

FieldFormula PeopleCode is only associated with record fields.



Important! Do not use FieldFormula PeopleCode in your components. Use it only to store external PeopleCode functions in FUNLIB_ record definitions.

ItemSelected Event

The ItemSelected event fires whenever the end-user chooses menu item from a pop-up menu. In pop-up menus ItemSelected PeopleCode executes in the context of the page field from where the pop-up menu is attached, which means that you can freely reference and change page fields, just as you could from a push button.



This event, and all it's associated PeopleCode, will not fire if run from a Component Interface.

ItemSelected PeopleCode is only associated with pop-up menu items.



For more information see ItemSelected Processing.

PostBuild Event

The PostBuild event fires after all the other component build events have fired. This event is often used to hide or unhide pages. It's also used to set component variables.

PostBuild PeopleCode is only associated with components.

PreBuild Event

The PreBuild event fires before the rest of the component build events. This event is often used to hide or unhide pages. It's also used to set component variables.



If a PreBuild PeopleCode program issues an error or warning, the end-user is returned to the search page. If there is no search page, that is, the search record has no keys, a blank component page displays.

The PreBuild event is also used to validate data entered in the search dialog, after a prompt list is displayed. For example, after the end-user selects key value(s) on the search, your PreBuild PeopleCode program fires, which catches the error condition and issues an error message. The end-user receives and acknowledges an error message. The component is cancelled (because of the error) and the end-user is returned to the Search dialog. PreBuild PeopleCode is only associated with components.

PrePopup Event

The PrePopup event fires just before the display of a pop-up menu.

You can use PrePopup PeopleCode to control the appearance of the Pop-up menu.



This event, and all its associated PeopleCode, will not fire if run from a Component Interface.

PrePopup PeopleCode can be associated with record fields and component record fields.



For more information see Pop-up Menu Display and CheckMenuItem, UnCheckMenuItem, DisableMenuItem, PeopleCode Built-in Functions and Language Constructs E-M, HideMenuItem.

PSControllnit Event

The PSControllnit event fires **every time** the page is redrawn. This means it fires after the component buffers are loaded and after a RowInsert or a RowDelete. It also fires after an end-user changes a field using a drop down or other prompt, or the end-user moves up or down a row on a scroll. It fires after an end-user clicks Next or Previous, or any other scroll movement controls.



PSControllnit is only applicable in the Windows client, not in the PeopleSoft Internet Architecture.

If you place an ActiveX control in a scroll that has an occurs count greater than one, the PSControllnit event will fire once for every occurrence. That is, if the scroll has an occurs count of 3, PSControllnit will fire 3 times.

Because PSControllnit can be fired so often, any PeopleCode associated with this event **must** be designed so it isn't sensitive to how often and when it gets run.

This event is only available with ActiveX controls. The PeopleCode placed in this event should primarily be used for synchronizing the control with the buffer data. It shouldn't be used for any other purpose.



If you need to load data into ActiveX control and want to place that code in the PSControlInit event, and only want it to run once in the entire session of the page, you should make sure the code is written in such a way that it is executed only on the first time the page is drawn but not in subsequent redraws.

This event, and all its associated PeopleCode, will not fire if run from a Component Interface.

PSControlInit PeopleCode is only associated with an ActiveX control.



For more information see Component Build Processing in Update Modes and Component Build Processing in Add Modes.

PSLostFocus Event

The PSLostFocus event fires for an ActiveX control page field when the end-user removes focus from the control. For example, the event fires when the end-user tabs off the control.

PSLostFocus does not fire when the end-user tabs from one field to another within the control. It only fires when the focus completely leaves the control. Use this event to move data from the control to the component data buffers.



PSLostFocus is only applicable in the Windows client, not in the PeopleSoft Internet Architecture.

This event, and all its associated PeopleCode, will not fire if run from a Component Interface.

PSLostFocus PeopleCode is only associated with an ActiveX control.



For more information see PSLostFocus Processing.

RowDelete Event

The RowDelete event fires whenever an end-user attempts to delete a row of data from a page scroll. You can use RowDelete PeopleCode to prevent the deletion of a row (using an **Error** or **Warning** statement) or to perform any other processing contingent on row deletion. For example, you could have a page field TOTAL on scroll level zero whose value is the sum of all the EXTENSION page fields on scroll level one. If the end-user deleted a row on scroll level one you could use RowDelete PeopleCode to recalculate the value of TOTAL.

The RowDelete event triggers PeopleCode on any field on the row of data that is being flagged as deleted.



RowDelete does not trigger programs on Derived/Work records.

RowDelete PeopleCode can be associated with record fields and component records.



For more information see Row Delete Processing and Errors and Warnings in RowDelete.

Considerations when Deleting all Rows from a Scroll

When the last row of a scroll is deleted, a new, dummy row is automatically added. As part of the RowInsert event, RowInit PeopleCode is run on this dummy row. If a field is changed by RowInit (even if it's left blank) the row is no longer New, and therefore won't be reused by any of the ScrollSelect functions or the Select method. In this case, you may want to move your initialization code from the RowInit event to FieldDefault.

RowInit Event

The RowInit event fires the first time the Component Processor encounters a row of data. It is used for setting the initial state of component controls. This happens during component build processing and row insert processing. It also happens after a ScrollSelect or related function is executed.

RowInit is not field-specific: it triggers PeopleCode on all fields and on all rows in the component buffer.

Do not use **Error** or **Warning** statements in RowInit PeopleCode: these cause a runtime error and force the end-user to cancel the component without saving.

RowInit PeopleCode is often paired with FieldChange PeopleCode. In these RowInit/FieldChange pairs, the RowInit PeopleCode checks values in the component and initializes the state or value of page controls accordingly. FieldChange PeopleCode then rechecks the values in the component during page execution and resets the state or value of page controls.

To take a simple example, suppose you have a Derived/Work field called PRODUCT, the value of which is always the product of page field A and page field B. When the component is initialized you would use RowInit PeopleCode to initialize PRODUCT equal to A * B when the component starts up or when a new row is inserted. You could then attach FieldChange PeopleCode programs to both A and B which also set PRODUCT equal to A * B. Whenever the end-user changes the value of either A or B, PRODUCT would be recalculated.

RowInit PeopleCode can be associated with record fields and component records.



For more information see Component Build Processing in Add Modes, Component Build Processing in Add Modes

Exception to RowInit Firing

There is a special instance when RowInit won't fire for a record. This will only occur if **all of the following** are true:

- the fields you've placed on the page for that record are all at level 0
- the values for every field you've placed on the page are available in the keylist
- every field you've placed on the page are display only

If all of these conditions are true, the record isn't loaded into the component buffer. The values for the fields come from the keylist. To make RowInit run, you need to add another field from the record that violates one of these conditions (such as, place an invisible field that isn't in the keylist on the page.)

RowInsert Event

When the end-user adds a row of data, the Component Processor generates a RowInsert event. You should use RowInsert PeopleCode for processing specific to the insertion of new rows. Do not put PeopleCode in RowInsert that already exists in RowInit, because a RowInit event always fires after the RowInsert event, which will cause your code to be run twice.



If none of the fields in the new row are changed after the row has been inserted (either by the end-user pressing ALT-7 and ENTER, or programmatically), when the page is saved, the new row isn't inserted into the database.

The RowInsert triggers PeopleCode on any field on the inserted row of data.

Do not use a **Warning** or **Error** in RowInsert: this will cause a runtime error and force cancellation of the component.

You can prevent the end-user from inserting rows into a scroll area by checking the **No Row Insert** box in the scroll bar's Page Field Properties; however, you can't prevent row insertion conditionally.

Page Field Properties

Label Use **General**

Scroll Attributes
Occurs Level: 1 Occurs Count: 1

Field Use Options
☐ Invisible ☒ Default Width
☐ No Auto Select ☐ No Auto Update
☒ No Row Insert ☐ No Row Delete

Scroll Action Buttons
☐ Previous Page ☐ Next Page
☐ Row Insert ☐ Row Delete
☐ Top ☐ Bottom
☐ Show Row Counter

Popup Menu

Field Help Context Number:
 < Auto Assign

☒ Allow Deferred Processing

OK Cancel

No Row Insert in Properties of the Scroll Bar



RowInsert does not trigger PeopleCode on Derived/Work fields.

RowInsert PeopleCode can be associated with record fields and component records.



For more information see Row Insert Processing.

RowSelect Event

The RowSelect event fires at the beginning of the Component Build process in any of the Update action modes (Update, Update/Display All, Correction). RowSelect PeopleCode is used to filter out rows of data as they are being read into the component buffer. This event also occurs after a ScrollSelect or related function is executed.

A **DiscardRow** function in RowSelect PeopleCode causes the Component Processor to skip the current row of data and continue to process other rows. A **StopFetching** statement causes the Component Processor to accept the current row of data, then stop reading additional rows. If both statements are executed, the program skips the current row of data, then stops reading additional rows.

PeopleSoft applications rarely use RowSelect, because it's inefficient to filter out rows of data after they've already been selected. Recent applications screen out rows of data using search

record views and effective-dated tables, which filter out the rows before they're selected. You could also use a `ScrollSelect` or related function to programmatically select rows of data into the component buffer.

In previous versions of PeopleTools the **Warning** and **Error** statements were used instead of **DiscardRow** and **StopFetching**. **Warning** and **Error** will still work as before in `RowSelect`, but their use is discouraged.



In `RowSelect` PeopleCode, you can only refer to record fields on the record that is currently being processed.

This event, and all its associated PeopleCode, will not fire if run from a Component Interface.

`RowSelect` PeopleCode can be associated with record fields and component records.



For more information see Row Select Processing.

SaveEdit Event

The `SaveEdit` event fires whenever the end-user attempts to save the component. You can use `SaveEdit` PeopleCode to validate the consistency of data in component fields. Whenever a validation involves more than one component field, you should use `SaveEdit` PeopleCode. If a validation involves only one page field, you should use `FieldEdit` PeopleCode.

`SaveEdit` is not field-specific: it triggers associated PeopleCode on every row of data in the component buffers, except rows flagged as deleted.

An **Error** statement in `SaveEdit` PeopleCode displays a message and redisplay the component without saving data. A **Warning** gives the end-user a chance to press OK and save the data, or press Cancel and return to the component without saving.

You can use the `SetCursorPos` function to set the cursor position to a specific page field following a **Warning** or **Error** in `SaveEdit`, to show the end-user the specific field (or at least one of the fields) that is causing the problem. Make sure to call **SetCursorPos** before the **Error** or **Warning** (because these may terminate the PeopleCode program).

`SaveEdit` PeopleCode can be associated with record fields and components.



For more information see Save Processing.

SavePostChange Event

After the Component Processor updates the database, it fires the SavePostChange event. You can use SavePostChange PeopleCode to update tables not in your component using the **SQLExec** built-in function.

An **Error** or **Warning** in SavePostChange PeopleCode will cause a runtime error, forcing the end-user to cancel the component without saving changes. Avoid **Errors** and **Warnings** in the this event.

The system issues a SQL commit after SavePostChange PeopleCode completes successfully.

If you are executing WorkFlow PeopleCode, bear in mind that if the WorkFlow PeopleCode fails, SavePostChange PeopleCode will not be executed. If your component has both WorkFlow and SavePostChange PeopleCode, consider moving the SavePostChange PeopleCode to SavePreChange or WorkFlow.

If you are doing application messaging, your **Publish()** PeopleCode should go into this event.



Caution! Never issue a SQL Commit or a Rollback manually from within a **SQLExec** function. Let the Component Processor issue these SQL commands.

SavePostChange PeopleCode can be associated with record fields, components and component records.



For more information see Save Processing.

SavePreChange Event

The SavePreChange event fires after SaveEdit completes without errors. SavePreChange PeopleCode gives you one last chance to manipulate data before the system updates the database; for instance, you could use SavePreChange PeopleCode to set sequential high-level keys. If SavePreChange runs successfully, a WorkFlow event is generated, then the Component Processor issues appropriate INSERT, UPDATE, and/or DELETE SQL commands.

SavePreChange PeopleCode is not field-specific: it triggers PeopleCode on all fields and on all rows of data in the component buffer.

SavePreChange PeopleCode can be associated with record fields, components and component records.



For more information see Save Processing.

SearchInit Event

The SearchInit event is generated just before a search dialog, add dialog, or data entry dialog is displayed. SearchInit triggers associated PeopleCode in the search key fields of the search record. This allows you to control processing before the end-user enters values for search keys in the dialog. In some cases you wish to set the value of the search dialog fields programmatically. For example, the following program in SearchInit PeopleCode on the component search key record field EMPLID sets the search key page field to the user's employee ID, grays out the page field, and enables the user to modify the user's own data in the component:

```
EMPLID = %EmployeeId;  
  
Gray (EMPLID);  
  
AllowEmplIdChg(true);
```

You can switch on system defaults and system edits during the search dialog by calling **SetSeachDefault** and **SetSearchEdit** in SearchInit PeopleCode. You can also control the behavior of the search dialog, either forcing it to display even if all the required keys have been provided, or skipping it if possible, with the **SetSeachDialogBehavior** function.



This event, and all its associated PeopleCode, will not fire if run from a Component Interface.

SearchInit PeopleCode can be associated with record fields and component search records.



For more information see Search Processing in Update Modes and Search Processing in Add Modes.

For more information about the functions, see SetSearchDefault, SetSearchEdit, and SetSearchDialogBehavior.

SearchSave Event

SearchSave PeopleCode is executed for all search key fields on a search dialog, add dialog, or data entry dialog after the end-user clicks Search. This allows you to control processing after search key values are entered, but before the search based on these keys is executed. A typical use of this feature is to provide cross-field edits for selecting a minimum set of key information. It is also used to force the user to enter a value in at least one field, even if it's a partial value to help narrow a search for tables with many rows.



SearchSave does **not** fire when values are selected from the search list. If you need to validate data entered in the search dialog, use the Component PreBuild event to do so.

You can use **Error** and **Warning** statements in SearchSave PeopleCode to send the end-user back to the search dialog if the end-user entry does not pass validations implemented in your PeopleCode.



This event, and all its associated PeopleCode, will not fire if run from a Component Interface.

SearchSave PeopleCode can be associated with record fields and component search records.



For more information see Search Processing in Update Modes and Search Processing in Add Modes.

Workflow Event

Workflow PeopleCode executes immediately after SavePreChange and before the database update that precedes SavePostChange. The main purpose of the Workflow event is to segregate PeopleCode related to Workflow from the rest of your application's PeopleCode. Only PeopleCode related to Workflow (such as **TriggerBusinessEvent**) should be in Workflow programs. Your program should deal with Workflow only after any SavePreChange processing is complete.

Workflow PeopleCode is not field-specific: it triggers PeopleCode on all fields and on all rows of data in the component buffer.

WorkFlow PeopleCode can be associated with record fields and components.



For more information see Save Processing and Writing Workflow PeopleCode.

PeopleCode Execution in Multiple Scroll Pages

Components with multiple occurs levels can have multiple rows of data from multiple primary record definitions. You need to know the order in which the system processes buffers for this data, because it applies PeopleCode in the same order.

The Component Processor uses a "depth-first" algorithm to process rows in multiple-scroll pages, starting with a row at level zero and drilling down to dependent rows on lower levels, then working its way up the hierarchy until it has processed all the dependent rows of the last row on the highest level.



For more information see Component Buffer Structure and Contents.

Scroll Level One

When pages have only one scroll bar, the Component Processor processes buffers in a straightforward manner. It processes record definitions at scroll level zero, then all rows of data at scroll level one.

Data is retrieved for all rows with a single SELECT statement, and then merged with buffer structures.

Scroll Level Two

With scroll bars at multiple scroll levels, the system does the same thing for each scroll-level-one row. It processes a single row of data at scroll level one, then processes all its subordinate rows of data at scroll level two. After processing all subordinate data at scroll level two, it processes the next row for scroll level one, and all the subordinate data for that row. It continues in this fashion until it processes everything.

Scroll Level Three

The Component Processor uses the same method for processing subordinate data at scroll level three. Data is retrieved for all rows with a single SELECT statement, and then merged with buffer structures. When it processes a single row of data at scroll level two, it processes all subordinate data before processing the next scroll-level-two row.

CHAPTER 11

PeopleCode and PeopleSoft Internet Architecture

The following sections discuss the PeopleCode considerations writing PeopleSoft Internet Architecture applications.

Using PeopleCode in the PeopleSoft Internet Architecture

You should take the following considerations into account when writing PeopleCode programs for the PeopleSoft Internet Architecture.

- To help your application run efficiently, you should avoid using field-level PeopleCode events (that is, **FieldEdit** and **FieldChange**).

If you need to use field-level PeopleCode events, such as **FieldChange** and **FieldEdit** in your applications, you should keep the following in mind: each time you execute a field-level PeopleCode program, it requires a trip to the application server to run the PeopleCode program.

However, the majority of your PeopleCode programs will naturally run on the application server as part of the Component build and save process. Don't hesitate to use PeopleCode for building and saving your components.

- Certain PeopleCode functions and methods are client-only. No PeopleCode runs on the client in a PeopleSoft Internet Architecture application, which means that client-only PeopleCode **isn't supported** in a PeopleSoft Internet Architecture application. You can use the Validate feature to check your application for PeopleCode that won't run in the PeopleSoft Internet Architecture.



For more information see Client-Only PeopleCode and Validating Projects.

- If an end-user changes a field, but there is nothing to cause a trip to the server on that field, default processing and FieldFormula PeopleCode don't run. They only run when some other event causes a trip to the server.

This means other fields that depend on the first field using FieldFormula or default PeopleCode are not updated until the next time there is a server trip.

- In application that run on the PeopleSoft portal, external, dynamic hyperlink information **must** be placed in RowInit PeopleCode. If it's placed in FieldChange PeopleCode, it won't work.
- Trips to the server are reduced when a component is running in deferred processing mode. Each trip to the server results in the page being complete refreshed on the browser, which may cause the display to flicker. It can also slow down your application. By specifying a component as Deferred Processing Mode, you can achieve better performance.



For more information see Deferred Processing Mode.

Avoiding Features Not Supported by PeopleSoft Internet Architecture

PeopleCode events and functions that relate exclusively to features not supported by the PeopleSoft Internet Architecture cannot, obviously, be used. The following functions and types of functions aren't supported:

- Menu PeopleCode. The **ItemSelected** PeopleCode event isn't not supported, as well as the **CheckMenuItem** and **UnCheckMenuItem** functions.
- Dynamic tree controls. Functions related to this control, such as **GetSelectedTreeNode**, **GetTreeNodeParent**, **GetTreeRecordName**, **RefreshTree** and **TreeDetailInNode** cannot be used.
- ActiveX controls. The **GetControl** and **GetControlOccurrence** functions cannot be used.
- Client-only functions and methods aren't supported.
- WinEscape



For more information see Client-Only PeopleCode.

Using PeopleCode to Populate Search Dialog Key Fields

In a PeopleSoft Internet Architecture application you typically want users to directly access their own data. To facilitate this, you may want to use SearchInit PeopleCode to populate and then gray out standard search dialog key fields. You might assign the search key field a default value based on the user ID or alias the user entered at logon.

You'll also need to call **AllowEmpIdChg**, allowing users to change their own data. This function takes a single Boolean parameter in which you pass True to allow employees to change their own data.

Here is a simple example of such a SearchInit program, using %EmployeeId to identify the user:


```
EMPLID = %EmployeeId;  
  
Gray (EMPLID);  
  
AllowEmplIdChg(true);
```

Client-Only PeopleCode

Certain PeopleCode built-in functions can run only on the client. If your PeopleSoft Internet Architecture application calls a client-only PeopleCode function at runtime, an error occurs.

Some built-in functions are always client-only, others are client-only under specific conditions.

Functions That Are Always Client-Only

The following built-in functions are client-only under all circumstances. There are two main reasons why this is the case. Some functions relate specifically to the user interface, so it only makes sense to run such functions on the client. Other functions, such as the DOS functions, could not run on a UNIX application server, and therefore need to run exclusively on the Windows client.



Client-only PeopleCode programs do not run on PeopleSoft Internet Architecture applications. Therefore the following PeopleCode functions are **not** supported on the PeopleSoft Internet Architecture.

- The DOS functions: ChDir, ChDrive, ExpandEnvVar, GetCwd, GetEnv.
- WinExec is Windows-specific and therefore client-only. The Exec function, which can call an executable on either a UNIX or NT application server, can be used instead.
- OLE automation functions: CreateObject, ObjectDoMethod, ObjectGetProperty, ObjectSetProperty.
- Functions that control menu appearance: CheckMenuItem and UnCheckMenuItem.
- ScheduleProcess and CreateProcessRequest. These functions, and the ProcessRequest object, may be client-only, depending on parameter settings.

Functions That Are Client-Only under Specific Conditions

The following built-in function is client-only under specific conditions.

ScheduleProcess

The **ScheduleProcess** function has a parameter, *run_location*, which 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.



In PeopleTools 8, the new ProcessRequest class' method and properties provide the same functionality as the ScheduleProcess function, and are subject to the same limitations regarding client/server operation. When the *Schedule* method is used in a program running on the application server, the *RunLocation* property cannot be set to "Client". As well, COBOL and SQR processes scheduled using the *Schedule* method from a program running on the server must also be set to run on the server.



For more information on ScheduleProcess, see ScheduleProcess.

Functions That Behave Differently in Three-Tier Mode

You can use the **SendMail** built-in function to send an email message from a PeopleSoft panel. The **SendMail** function only supports VIM and MAPI email subsystems for two-tier architecture, and only supports SMTP for three-tier.



For more information see SendMail.

Calling Executables on the Application Server

The **WinExec** built-in function is Windows-specific and therefore client-only. The **Exec** function is compatible with both UNIX and Windows, so it can be used to execute a program on either the client or the application server.



The **WinExec** function has a third parameter that controls the state of the window in which the called program runs. If you know that the called program is always going to run on the client, you might want to use **WinExec** rather than **Exec** so that you can take advantage of this functionality.



For more information see **Exec**.

Calling Dynamic Link Library Functions on the Application Server

To support processes running on an application server, it is possible to declare and call functions compiled in Windows dynamic link libraries and UNIX shared libraries (or shared objects, depending on the specific UNIX platform). You can do this either with a special PeopleCode declaration, or using the Business Interlink framework.



For more information see PeopleSoft Business Interlink Application Developer Guide.

When you call out to a DLL using PeopleCode, on Windows NT application servers, the DLL file has to be on the path. On UNIX application servers, the shared library file has to be on the *library path* (as defined for the specific UNIX platform).

The PeopleCode declaration and function call syntax remains unchanged. For example, the following PeopleCode could be used to declare and call a function **LogMsg** in an external library **Testdll.dll** on a Windows client or a Windows application server, or a **libtestdll.so** on an UNIX application server:

```
Declare Function LogMsg Library "testdll" (string, string)
    Returns integer;

&res = LogMsg("\temp\test.log", "This is a test");
```

Sample Cross-Platform External Test Function

Following is the C source code for a sample cross-platform test file. It is a bare-bones function that simply opens a log file and appends a line to it.

This file contains an interface function needed for non-Windows environments. This function is compiled conditionally: only if you are compiling for a non-Windows environment (i.e. UNIX). The interface function references a provided header file, **pcmext.h**. The interface function is passed type codes that can be optionally used for parameter checking.

```

/*
 * Simple test function for calling from PeopleCode.
 * This is passed two strings, a file name and a message.
 * It creates the specified file and writes the message
 * to it.
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef _WINDOWS
#define DLLEXPORT __declspec(dllexport)
#define LINKAGE __stdcall
#else
#define DLLEXPORT
#define LINKAGE
#endif

DLLEXPORT int LINKAGE LogMsg(char * fname, char * msg);

/*****
 * PeopleCode External call test function.
 *
 * Parameters are two strings (filename and message)
 * Result is 0 if error, 1 if OK
 *
 * To call this function, the following PeopleCode is
 * used
 *
 * Declare Function LogMsg Library "testdll"
 * (string, string)
 * Returns integer;
 *
 * &res = LogMsg("\temp\test.log", "This is a test");
 *****/

DLLEXPORT int LINKAGE LogMsg(char * fname, char * msg)
{
    FILE *fp;

    fp = fopen(fname, "a");          /* append */
    if (fp == NULL) return 0;

    fprintf(fp, "%s\n", msg);
    fclose(fp);
    return 1;
}

#ifdef _WINDOWS

/*****

```

```

*   Interface function.
*
*   This is not needed for Windows....
*
*****/

#include "pcmext.h"
#include "assert.h"

void LogMsg_intf(int nParam, void ** ppParams, EXTPARAMDESC * pDesc)
{
    int    rc;

    /* Some error checking */
    assert(nParam == 2);
    assert(pDesc[0].eExtType == EXTTYPE_STRING
        && pDesc[1].eExtType == EXTTYPE_STRING
        && pDesc[2].eExtType == EXTTYPE_INT);

    rc = LogMsg((char *)ppParams[0],
        (char *)ppParams[1]);
    *(int *)ppParams[2] = rc;
}

#endif

```

Updating the Installation and PSOPTIONS Tables

When an application updates either the PSOPTIONS or the Installation table it must call **UpdateSysVersion** from **SavePreChange** PeopleCode event. This way the updates will be take effect at the next panel load. Otherwise, the change will not take effect at the client workstation until the user logs off and logs back on.



Suggestions. Making changes to the Installation and PSOPTIONS tables is not a trivial matter. Only a database administrator or the equivalent person in your organization should make changes to these tables.

CHAPTER 12

Debugging Your Application

The PeopleCode Debugger is an integrated part of the Application Designer. The interface to the debugger has a visual indicator of breakpoints, an arrow indicating the current line and the ability to step through code. You can inspect the value of a variable by 'hovering' over it and reading the pop-up bubble help. The debugger also provides variable inspection windows for Globals, Locals, Function Parameters, and Component scoped variables. It also allows PeopleCode objects to be "expanded", so you can inspect their component parts.

Accessing the PeopleCode Debugger

You access the Debugger through the Application Designer. You can start a debugging session either before or after you start a PeopleSoft component.

To start the PeopleCode Debugger

1. Open the Application Designer.
2. Choose Debug, PeopleCode Debugger Mode.

The Local Variables watch window opens. PeopleCode programs that had breakpoints set from your previous debugging session are opened as well, and the breakpoints are restored.



If you've already opened the debugger, then closed it, the menu may not morph correctly to allow you to access the debugger a second time. If this occurs, click on the Local Variables window, then try the debug menu again.

You must make some adjustments to your system for running the debugger in PeopleSoft Internet Architecture or with Application Message Subscription PeopleCode.



For more information see Setting Up the Debugging Environment.



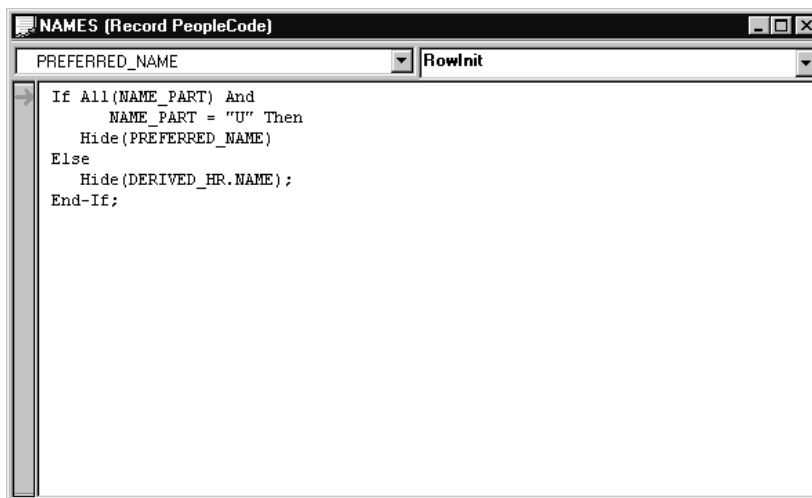
Your security administrator has options for allowing users to access different parts of the Application Designer, including the PeopleCode debugger. If you're having problems accessing the debugger, you may need to contact your system administrator concerning your security access.

PeopleCode Debugger Features

The following is a list of features for the PeopleCode debugger.

Visible Current Line of Execution

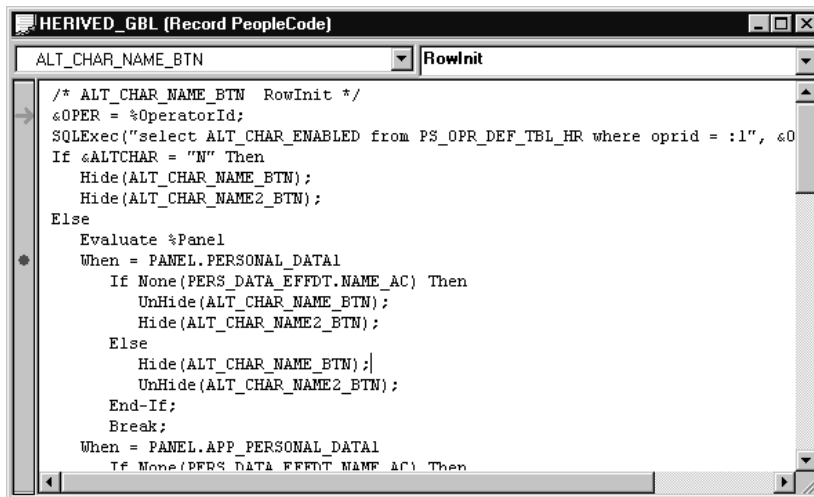
The current line indicator (green arrow displayed in left hand gutter) is illustrated below:



PeopleCode Debugger showing current line of execution

Visible Breakpoints

Visual indicators that signify breakpoint locations is supported. In the example below the current line indicator (green arrow) is shown at the first line, and the breakpoint (red dot displayed in left hand gutter) is on line 8:



PeopleCode Debugger showing breakpoint and current line of execution

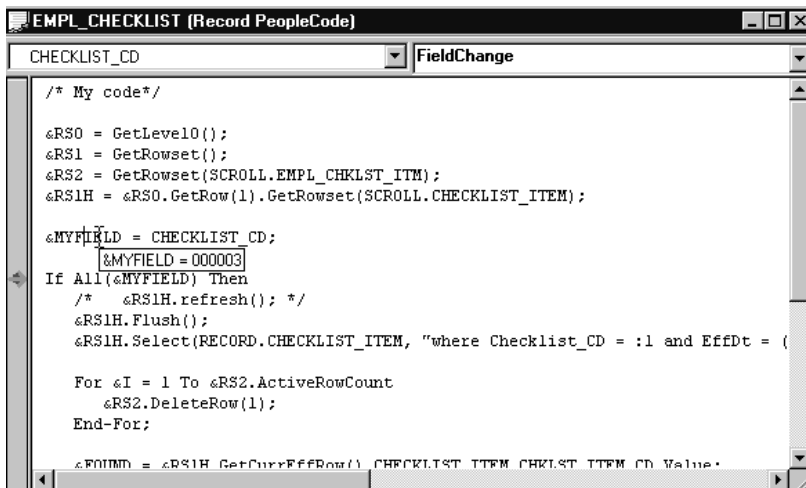
All breakpoints are saved when Exit Debug Mode is selected.



You can't set breakpoints on function declarations, variable declarations, or in comments.

Hover Inspect

If your program is already running, you can see the actual values for the variables by "hovering" over them. The current value will be displayed in a pop-up window.



PeopleCode Debugger showing hover inspect

Hover inspect is implemented only for simple variables and fields.

Hover inspect is **not** implemented for object expressions (for example, rowset assignments, array assignments, and so on.)

Single Debugger

Every PeopleSoft session you're running on a machine can have its own debugging session. However, there can only be one instance of the PeopleCode debugger **per session**. If more than one instance of Application Designer is running for a session, only one may be the active debugger at a given time.

From within a running instance of the Application Designer, any component in the same session is also placed into debug mode.

Once the session is in debug mode, any component that is started, that belongs to that session, will automatically go into debug mode.

Similarly, Application Engine PeopleCode, Component Interface PeopleCode, and Message Subscription PeopleCode can be debugged.

Once you exit debug mode, through the **Debug, Exit Debug Mode** menu or by exiting the Application Designer, all components in that session go out of debug mode.

If you exit a component, debugging continues with any remaining open and running components.

In the event there is more than one Application Designer session running, the Application Designer session that is used as a debugger is the first one that had been started.

If you're in debug mode, a PeopleCode editor window will be opened for every item (for example, record, component, page, and so on.) that has PeopleCode in it when that PeopleCode is executed. If a component has more than one event with a PeopleCode program, only one window will be opened per item.

For example, if you have a record that has PeopleCode in both the SearchSave and RowInit events, only one PeopleCode editor window will be opened: first it will contain the SearchSave PeopleCode program, then the RowInit program. If you have PeopleCode in the RowInit event for two different records that are part of the same component, two PeopleCode editor windows will be opened, one for each RowInit PeopleCode program.

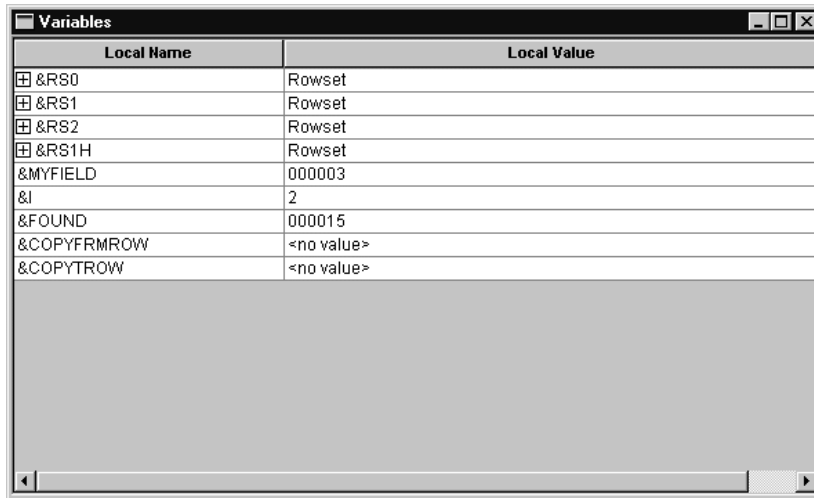
Variables Panes

There are the following types of variables panes:

- Local
- Global
- Component
- Parameter

The Local, Global, and Component variable panes show Local, Global, and Component variables, respectively. The Parameter variable pane shows the value of parameters passed in function declarations.

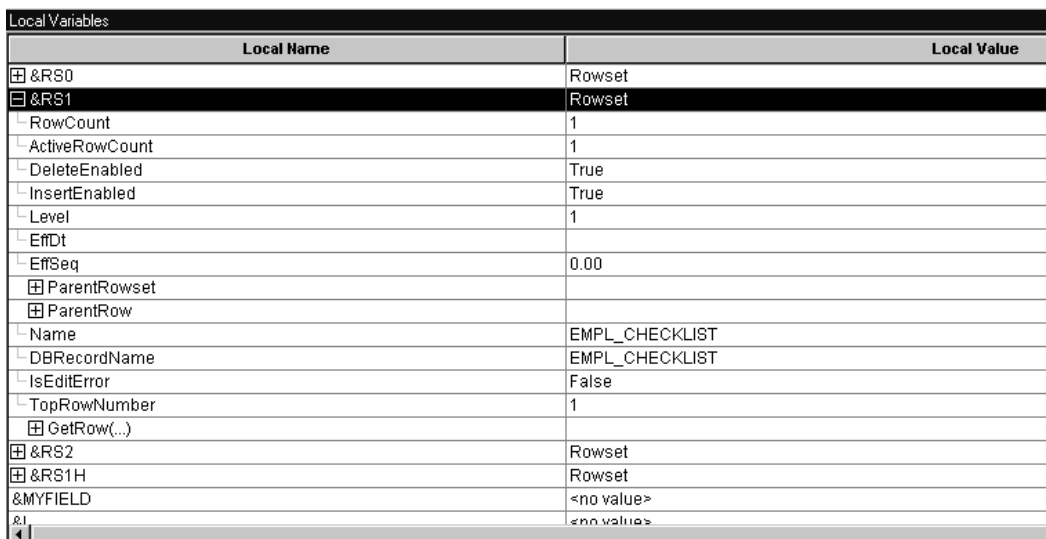
From the variables pane you can check the value of the variables you have in your program. These values are updated as your code executes.



Local Name	Local Value
&RS0	Rowset
&RS1	Rowset
&RS2	Rowset
&RS1H	Rowset
&MYFIELD	000003
&I	2
&FOUND	000015
©FRMROW	<no value>
©TROW	<no value>

Local Variables Pane

In addition, you can "expand" any of the objects to see its properties by clicking on the plus sign next to the variable name. In the following example, a level 1 rowset has been expanded. You can see the properties that are part of the rowset (such as ActiveRowCount or DBRecordName.)



Local Name	Local Value
&RS0	Rowset
&RS1	Rowset
RowCount	1
ActiveRowCount	1
DeleteEnabled	True
InsertEnabled	True
Level	1
EffDt	
EffSeq	0.00
ParentRowset	
ParentRow	
Name	EMPL_CHECKLIST
DBRecordName	EMPL_CHECKLIST
IsEditError	False
TopRowNumber	1
GetRow(...)	
&RS2	Rowset
&RS1H	Rowset
&MYFIELD	<no value>
&I	<no value>

Local variable pane with rowset object expanded

In addition, some objects "contain" other objects, like a rowset contains rows, rows contain records or child rowsets, and records contain fields. You can "expand" these secondary objects as

well, to see their properties. In the following example, the first row of a rowset has been expanded, as has the EMPL_CHECKLIST record.

Local Variables	
Local Name	Local Value
⊖ &RS1	Rowset
└─ RowCount	1
└─ ActiveRowCount	1
└─ DeleteEnabled	True
└─ InsertEnabled	True
└─ Level	1
└─ EffDt	
└─ EffSeq	0.00
⊕ ParentRowset	
⊕ ParentRow	
└─ Name	EMPL_CHECKLIST
└─ DBRecordName	EMPL_CHECKLIST
└─ IsEditError	False
└─ TopRowNumber	1
⊖ GetRow(...)	
⊖ (1)	
└─ RecordCount	4
└─ ChildCount	1
└─ RowNumber	1
└─ Visible	True
└─ Selected	False
└─ IsChanged	True
└─ IsDeleted	False
└─ IsNew	True
⊕ ParentRowset	
└─ IsEditError	False
└─ Style	
⊖ GetRecord(...)	
⊖ EMPL_CHECKLIST	
└─ IsDeleted	False
└─ IsChanged	True
└─ Name	EMPL_CHECKLIST
└─ FieldCount	5

Variable pane with rowset, row and record expanded (shown with condensed font)

Field Values

When you look at a field object in the debugger, the value of the field is listed under the Value column. This way you don't have to drill-down to the Value property to see the value of a field.

The following example shows the PERSONAL_DATA record, and the values of the fields.

Local Variables	
Local Name	
[-] PERSONAL_DATA	
[-] IsDeleted	False
[-] IsChanged	True
[-] Name	PERSONAL_DATA
[-] FieldCount	83
[-] ParentRow	
[-] RelLangRecName	
[-] IsEditError	False
[-] GetField(...)	
[-] EMPLID	8001
[-] NAME	Schumacher,Simom
[-] NAME_PREFIX	Mr
[-] NAME_SUFFIX	
[-] LAST_NAME_SRCH	SCHUMACHER
[-] FIRST_NAME_SRCH	SIMOM
[-] ADDRESS1	481 Haven Ct
[-] ADDRESS2	
[-] ADDRESS3	
[-] ADDRESS4	
[-] CITY	Moraga

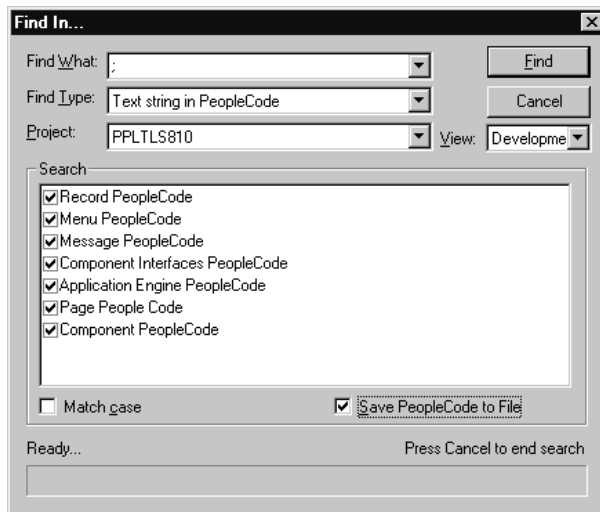
PERSONAL_DATA record field values

General Debugging Tips

- If you're having problems determining if the correct data is getting loaded into the component buffers, you can use the **View Component Buffers** view window to see all the values currently in the component buffer. (This is equivalent to putting a **GetLevel0** function at the start of a program.)

Using the &LEVEL0 variable, you can drill-down through all the levels of the rowset object, see the row, records, fields, and so on. This will show you everything that has been loaded into the component buffers for that component.

- While at a breakpoint, if you lose track of the window, or the location within the window, that is displaying the green execution location arrow, you can use the "Execution Location Properties" menu item's ViewCode button to find your current execution location again.
- Objects will remain expanded in the variable windows as you step through PeopleCode. This allows easy and fast inspection of the state of an object as you step through the PeopleCode. However, there is a performance cost for this feature. If you are finished examining an object, you may want to collapse it to improve the "stepping" response speed.
- If a database transaction has been started (either for you by PeopleTools, or by you in PeopleCode) other users of that database will be blocked from accessing that database until the transaction is complete. If you are stepping through PeopleCode while this transaction is open, you could potentially block other users for an extended period of time. You may want to use a private database for debugging to avoid blocking other users.
- If you want to create a file that contains all the PeopleCode for a project (or database) you can use the Find In. . utility, and search for ";". Be sure to check Save PeopleCode to File.



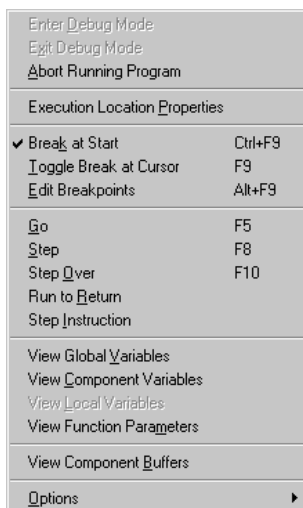
Find In . . . dialog box for finding all PeopleCode

DoModal Considerations

If you've set the PeopleCode Debugger to Break At Start, and you're using the DoModal PeopleCode function, the DoModal window may display **behind** the PeopleCode debugger window. This makes it seem as though the debugger has stopped, when it actually hasn't. Be sure to check that other windows haven't opened while you're debugging your code.

PeopleCode Debugger Options

After the debugger is running, if you select **Debug** again, you can select other options:



PeopleCode Debugger Options

Exit Debug Mode

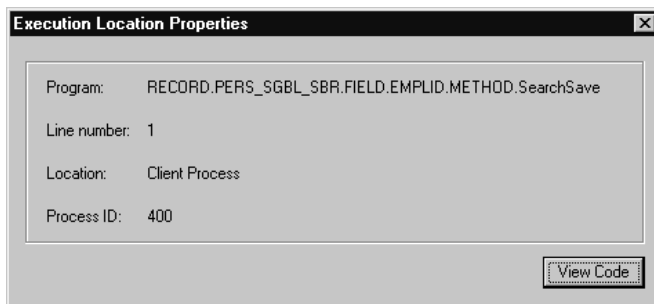
Quit debug mode. When you exit debug mode, all breakpoints are automatically saved. If you close Application Designer, you automatically exit debug mode.

Abort Running Program

Stop running the PeopleCode program that is currently running.

Execution Location Properties

Displays the location of the running code in a dialog box. This includes the record name, field name, event name, and line number of the code. It also indicates if the code is executing on the client or server. You can also view the exact code by selecting **View Code**.



Execution Location Properties dialog box

Break at Start

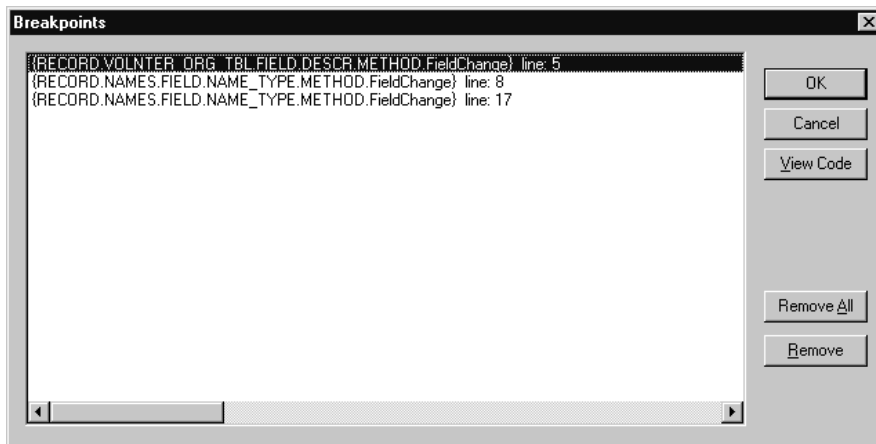
Pauses execution of the component on the first line of every PeopleCode program that executes in the component. If you've started a component with **Break at Start** selected, then start a second component, the PeopleCode associated with the second component will be stopped at the first line of the first PeopleCode program as well, as part of the same debugging session.

Toggle Break at Cursor

Remove the breakpoint if the line the cursor is currently on has a breakpoint. Add a breakpoint if the line the cursor is currently on does not have a breakpoint.

Edit Breakpoints

Brings up a menu that displays the lines that have breakpoints. From this menu you can bring up the code that contains the breakpoint, using **View Code**. You can also remove one or all of your breakpoints.



Edit Breakpoints Menu

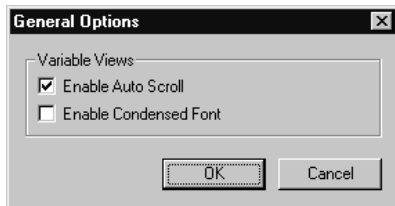
Go	Continue processing until the next breakpoint. If Break At Start is enabled, processing pauses at the next PeopleCode program.
Step	Executes the current line of the PeopleCode program, stepping into functions.
Step Over	Steps through each line of the PeopleCode program, one line at a time, but steps over the functions; the functions are executed, but not stepped into.
Run to Return	Processes past the return of the current function, then pauses.
Step Instruction	Processes low-level pseudo-machine code instructions internal to PeopleCode. This option is used in conjunction with Log Options .
View Global Variables	Opens a separate window for watching variables declared as Global.
View Component Variables	Opens a separate window for watching variables declared as Component.
View Local Variables	Opens a separate window for watching variables declared as Local.
View Function Parameters	Opens a separate window for watching user-specified parameters in Function calls.
View Component Buffers	Opens a separate window for viewing the current component buffers. This is equivalent to getting a level 0 rowset for the component.



These five windows are continuously updated as the program executes.

Options

Allows you chose between opening up a dialog box for general options or for specifying log options.



General Options Dialog Box

The General Options dialog lets you specify conditions of the View windows.

If you have Auto Scroll enabled, and you click on a plus symbol next to a variable name in a View window, the variable you clicked 'scrolls' to the top of the View window.

If you have Condensed Font enabled, all View windows are displayed with a smaller font.

The default is for both of these options to be selected.



For more information on log options, see Setting PeopleCode Debugger Log Options.

Additional Features

Break at Termination: Once you are in debug mode, generally, any PeopleCode program in the session that terminates abnormally will first break in the debugger. In addition, the error message will be displayed in the PeopleCode log in the bottom window of the Application Designer.

Setting Up the Debugging Environment

Before you can use the debugger in 3-tier and the PeopleSoft Internet Architecture, you must manually create a user account named "PeopleCodeDebugger", with a null password, and configure the DbgBrkr Class to use this account.



For more information on how to set up this account, see Domain Settings.

If you have problems debugging in a non-2-tier environment, consider the following:

- The Application Server must be on the same computer as PeopleTools and the debugger.
 - The Application Server must be setup for PeopleSoft Internet Architecture debugging.
-



Note for Windows Client: A dual machine/debugger configuration can be setup. One computer can run the Windows client and a debugger for it. Another computer can run the Application Server and a debugger for it.

Note for Internet Architecture: While the Application Designer and Application Server must be on the same computer, the browser may be on a different computer or the same computer.

Debugging Subscription PeopleCode

You can debug Subscription PeopleCode using the following instructions.

To debug Subscription PeopleCode

1. Start Application Designer on the server machine that will be processing the subscription

You must use the same User ID and database as that machine has had its Application Servers configured to.

2. Start debug mode.

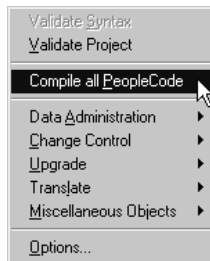
When the subscription server executes the subscription PeopleCode, the debugger will start.

Compiling all PeopleCode Programs

In addition to checking individual programs, you can compile all PeopleCode programs in a database to check for errors. This option opens and compiles every PeopleCode program. This option can be run after an upgrade to verify that all the programs were upgraded correctly. It could also be run on an as-needed basis to check for corruption in your programs.

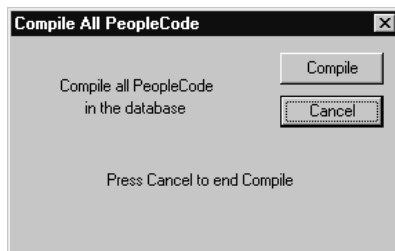
To compile all PeopleCode programs

1. Open Application Designer while accessed to the database that contains the PeopleCode you want to check.
2. Select Tools, Compile All PeopleCode.



Compile All PeopleCode

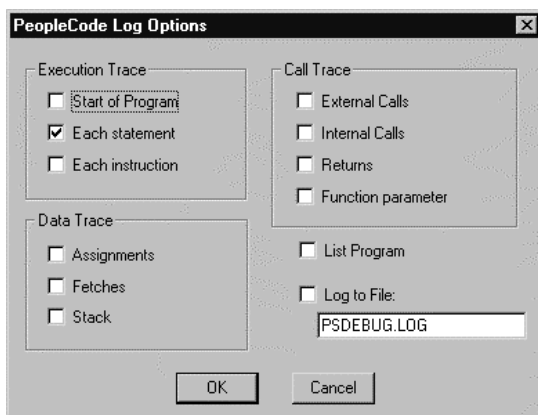
3. Select Compile on the Compile All PeopleCode dialog box.



Compile All PeopleCode dialog box

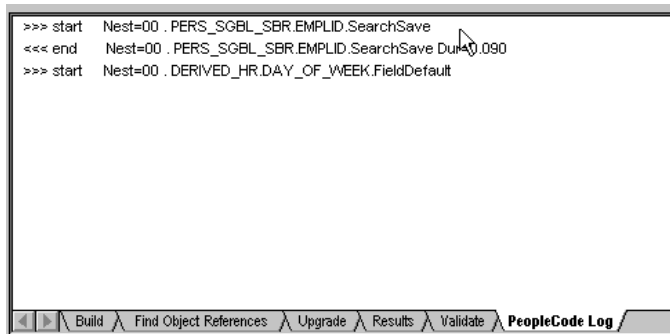
If you have any errors, they'll be displayed in the PeopleCode log display window.

Setting PeopleCode Debugger Log Options



Log Options Dialog

Use the PeopleCode debugger to view PeopleCode that is executed while you're stepping through your application (use **Debug, Log Options**.) All log information will be displayed in the PeopleCode Log window, at the bottom of the Application Designer.



PeopleCode Log Window

You can also record what you see in a log file. The log results can be customized to record a variety of online information—from as little as you like to probably more than you need.

If you exit debug mode, but do **not** close the Application Designer, all the log options you specified will still be there when you start debug mode again.

When you close the Application Designer, all log options are deselected. The next time you enter debugging mode, you will have to reselect your debug log options.

Some of the log options are described in the next few sections. For complete details about the log file, see *Interpreting the PeopleCode Debugger Log File*.

All the options available in the Log Options dialog are also available in the Configuration Manager, on the Trace tab, in the PeopleCode Trace section.



For more information see Trace.

Execution Trace Options

Execution Trace is set to trace each PeopleCode statement. You can also trace the start of each program or each program instruction.

Data Trace Options

The Data Trace options are Assignments, Fetches, and Stack. They have the following meanings:

- **Assignments.** Records each assignment made to a field.
- **Fetches.** Records the field values retrieved from a PeopleCode fetch.
- **Stack.** This option gives the contents of the internal machine stack. Typically, only PeopleSoft staff developing PeopleCode language enhancements use this option.

Call Trace Options

The Call Trace options enable you to record the values of External calls, Internal calls, Returns, and Function parameters. These options have the following meanings:

- **External calls.** This option traces each call to external (PeopleCode) functions.
- **Internal calls.** This option records each call to internal subroutines.
- **Returns.** Logs the occurrence of program returns.
- **Function parameters.** Logs the value of individual PeopleCode function parameters.

Log To File

When you chose this option you must specify the name of a file: you'll receive an error and logging to file will be disabled if you don't specify the name of a file.

If you don't specify a directory location, the file will be placed in the same directory as the directory you're running PeopleTools from.

If you specify the name of an already existing file, you'll get a warning message, asking whether to overwrite the file or not. At this point, you must go back into the Log Options and specify a different file name. If you do **not** specify a new log file name and start running an application, your log file will be overwritten.

If you run more than one application and do **not** exit out of the Application Designer between times, each trace will be appended the specified log file.

Interpreting the PeopleCode Debugger Log File

You can produce a trace log using any of the following methods:

- Log File option in the PeopleCode Debugger
- Configuration Manager Trace tab
- Built-in functions SetTracePC and SetTraceSQL
- PeopleTools Utilities (this is included for backward compatibility purposes only, and in general, shouldn't be used.)

The first option, using the Log File, produces essentially the same trace file as using any of the other options. The only difference is that the other trace files contain timing information, that is, when each line started processing, and how long it took to execute.



For more information see Performance Monitoring.

The log file produced by the latter options is specified by PeopleTools Trace File option on the Configuration Manager. All of these options write to the **same file**. The Log File option writes to the file you specify, that is, a different file.

Trace files are also produced by the Application Engine. These logs may contain more information.



For more information see Tracing Application Engine Programs.

Log File Contents

Unless you enjoy wading through program dumps, the log file probably contains some distracting information, but it does have a wealth of useful information for debugging PeopleCode.

You can view the log using any editor that displays ASCII text, such as Notepad. It has the following components.

Line Count	Specifies a line number within the file.
Internal Information	Contains reference numbers used for internal tracing. You can ignore this information.
Instruction Location	Address of an instruction processed in the program. You can follow programs and functions using this number.
Operation Code	The operation performed by the program.
Operation Operands	Contains information specific to each operation. The table below lists the possible operations and the operands that appear for the list and trace options.

The following tables describes how the operation and operands work together:

Operation	Operands	Description
ACCEPT		Causes control to transfer to the next level-1 statement.
ADD		Adds the contents of the top two items on the stack and pushes the result onto the stack.
BR TRUE	location	Tests the top item on the stack to determine if it contains a Boolean value of True. If true, control is passed to the specified instruction location.

Operation	Operands	Description
BR FALSE	location	Tests the top item on the stack to determine if it contains a Boolean value of False. If false, control is passed to the specified instruction location.
BRANCH	location	Control is passed to the specified instruction location.
BRANCH <	location	Tests the top two items on the stack to determine if the first is less than the second. If so, control is passed to the specified instruction location.
BRANCH <=	location	Tests the top two items on the stack to determine if the first is less than or equal to the second. If so, control is passed to the specified instruction location.
BRANCH <>	location	Tests the top two items on the stack to determine if the first is not equal to the second. If so, control is passed to the specified instruction location.
BRANCH =	location	Tests the top two items on the stack to determine if the first is equal to the second. If so, control is passed to the specified instruction location.
BRANCH >	location	Tests the top two items on the stack to determine if the first is greater than the second. If so, control is passed to the specified instruction location.
BRANCH >=	location	Tests the top two items on the stack to determine if the first is greater than or equal to the second. If so, control is passed to the specified instruction location.
BUILTIN	function	Executes the function specified.
	# Parm	The number of parameters passed to the function are also shown. The top elements of the stack are the parameters to the function.
CALL	DLL	Calls a dynamic link library module.

Operation	Operands	Description
	int <functionname> params = nn	Calls the internal PeopleCode function named <i>functionname</i> . The number of passed parameters, <i>nn</i> , is also shown.
	ext record.fieldname proctype fcn = function name params = nn	Call the external PeopleCode function name <i>functionname</i> that is located in the specified <i>record.fieldname</i> , with the specified <i>proctype</i> (that is, event type). The number of passed parameters, <i>nn</i> , is also shown.
CONCAT		Concatenates the top two items on the stack and places the result onto the stack. If the items are not strings, they will be converted to strings before concatenation.
DIVIDE		Divides the top two items on the stack and pushes the result onto the stack.
DUP		Duplicates the value of the top element or the stack.
ERROR		Stops execution of the PeopleCode program.
EXIT		Exits the currently executing PeopleCode program.
FETCH	record.field	Retrieves the value of record.field and pushes it onto the stack.
	Temp# NN	Retrieves the current value of temporary value NN and pushes it onto the stack. Temporary variables are assigned numbers sequentially starting with zero as they occur in a program.
	Builtin %function	Retrieves the value of the built-in function and pushes it onto the stack.
MULTIPLY		Multiplies the top two items on the stack and pushes the result onto the stack.
NEGATE		Negates the value of the top stack item.
POP		Removes the top item from the stack.
PUSH	constant	Pushes the specified constant onto the stack.

Operation	Operands	Description
RETURN		Returns control to a calling program or, if executed from the top-level PeopleCode program, exits the program.
SET TRUE	temp #nn	Sets the specified temporary variable to a TRUE value.
SET FALSE	temp #nn	Sets the specified temporary variable to a FALSE value.
START		The beginning of a PeopleCode program.
	Field=record name.field name-type	Specified record and field names of the program being executed. Type is one of the PeopleCode program types or: Assignment -- Used for all other type of PeopleCode programs
	Temps=NN	Specifies the number of temporary variables used by the program.
	Stack=NN	Specifies the maximum number of items placed onto the stack at any point in time. (that is, the maximum stack depth).
	Source=NN	The length of the PeopleCode program.
START INT START EXT	functionname params = nn	Indicates the beginning of a PeopleCode program. The number of passed parameters are shown.
STATEMENT	Next=location	Marks the beginning of a level-1 statement and specifies the location of the next statement.
STOP		Marks the end of a PeopleCode program.
STORE	record.field	Stores the contents of the top element of the stack into the specified record.field.
	Temp# NN	Stores the contents of the top element of the stack into the specified temporary variable.
SUBTRACT		Subtracts the top two elements of the stack and places the result onto the stack.

Operation	Operands	Description
WARNING		Denotes that a warning statement was issued. If this is the first warning issued by the program, the value on top of the stack is saved as the value returned by the program. If another warning has already occurred, then the top element of the stack is removed.

Sample Trace File

The following is a sample trace file:

```

At VOLUNTEER_ORG_TABL.ENG.Activate    PCPC:23  Statement:1

    0, start      Id=VOLUNTEER_ORG_TBL.GBL.PostBuild Temps=1 Stack=1

    1, statement Next=52

    20, builtin   - GetLevel0 #Parms=0

    32, store     &LVL0 (temp #0)

    52, statement Next=68

    68, stop

    0, start      Id=VOLUNTEER_ORG_TABL.ENG.Activate Temps=1 Stack=2

    1, statement Next=92

    20, fetch     PAGE.VOLUNTEER_ORG_TABL

    36, push      CHART1

    62, builtin   - GetControl #Parms=2

    72, store     &MYOBJ (temp #0)

    92, statement Next=108

    108, stop

    0, start      Id=VOLNTER_ORG_TBL.DESCR.FieldChange Temps=1 Stack=1

    1, statement Next=52

    20, builtin   - GetRecord #Parms=0

    32, store     &REC (temp #0)

    52, statement Next=68

```

```

68, stop

0, start      Id=VOLNTER_ORG_TBL.DESCR.SavePreChange Temps=0 Stack=3

1, statement Next=140

20, push      VOLNTER_ORG_TBL.DESCRSHORT

34, builtin    - None #Parms=1

48, br False 140

        Branch taken.

140, statement Next=156

156, stop

```

About Operations and Operands

The real information the trace provides lies in the operations and operands. Operations show what PeopleCode does, and operands show what the operations use to operate.

Stacks

The Component Processor works with data in buffers it allocates. PeopleCode works with data on the stack. Throughout the trace, you will see items pushed onto the stack and retrieved from the stack for operations.

We only mention stacks here for ease of explanation later. You do not need to worry about maintaining the stack; the PeopleCode evaluator does that for you.

If you really want to see the stack, you can turn that option on. But with a copy of the program in front of you, you should be able to figure out the appropriate items coming from and going to the stack.

START

The START operator indicates the beginning of a PeopleCode program. START shows some basic details of the program:

Id	Specifies the component containing the program and the event.
Temps	Number of temporary variables in the program
Stack	Maximum items you can place on the stack at any given time

This example indicates a PostBuild event on the component VOLUNTEER_ORG_TBL has started:

```
0, start      Id=VOLUNTEER_ORG_TBL.GBL.PostBuild Temps=1 Stack=1
```



PeopleCode programs can be associated with many types of components, such as components, component records, record fields, and so on. You must look carefully at the Id to verify where exactly the PeopleCode is located. For example, if the Id ends with a market, such as GBL, it's a component.

The following example indicates a FieldChange on the record field VOLNTER_ORG_TBL has started.

```
0, start      Id=VOLNTER_ORG_TBL.DESCR.FieldChange Temps=1 Stack=1
```

This component also had a PeopleCode program associated with the FieldChange event on the component record field.

```
0, start      Id=VOLUNTEER_ORG_TBL.GBL.VOLNTER_ORG_TBL.DESCR.FieldChange Temps=1
Stack=1
```

STOP

The STOP operator indicates the end of a PeopleCode program. STOP takes no operands.

BRANCH

BRANCH operations evaluate comparisons of the top two items on the stack. PeopleCode uses multiple flavors of BRANCH, depending on the comparison type. The trace identifies each Branch type with a comparison operator, such as =, <, or >.

Trace can also use BRANCH without a comparison operator. With some comparison operators (< and >), it makes a difference what item goes on the left and what item goes on the right. In those cases, the first item from the stack goes on the left and the second goes on the right.

All BRANCHes use location as an operand. When PeopleCode takes the branch (a true comparison), it goes to the specified location:

```
branch <> 428
```

In the example above, PeopleCode compares the top two items on the stack. If they don't equal each other, PeopleCode goes to the instruction at location 428. Location codes help you when you have a lot of code between two consecutively processed statements.

PUSH

The PUSH operator places a value on the stack. Typically, the operand is a constant. The trace treats any parameter passed to a PeopleCode function as a constant, so you may see a temporary variable pushed onto the stack:

```
push      1
```

In the example above, PeopleCode puts the value 1 onto the stack.

FETCH

The FETCH operator retrieves a value and pushes it onto the stack. FETCH uses one operand with three mutually exclusive values.

<i>Location</i>	The location of the value PeopleCode should retrieve. If you opt to show fetched values, you will see a Fetch Field issued with each fetch of each value.
<i>Temp</i>	The temporary variable whose value PeopleCode should retrieve. Trace does not show the value of temporary variables.

Builtin The system variable—such as %Page—whose value PeopleCode should retrieve. The example shows that PeopleCode fetched Country from PERSONAL_DATA, and the value was USA. It pushed the value USA onto the stack.

```
fetch      PERSONAL_DATA.COUNTRY
          Fetch Field:PERSONAL_DATA.COUNTRY Value=USA
```

In the following, the page VOLUNTEER_ORG_TBL was fetched.

```
fetch      PAGE.VOLUNTEER_ORG_TBL
          Fetch Field: PAGE.VOLUNTEER_ORG_TBL Value=VOLUNTEER_ORG_TBL
```

The example shows that PeopleCode retrieved the value of the current page, using the system variable %Component. The next line shows the component that was fetched.

```
fetch      Builtin - %Component
          fetch      COMPONENT.CHECKLIST_TABLE1
```

BUILTIN

The operator BUILTIN processes a built-in function, including any parameters which the function requires. BUILTIN uses two operands.

Function	The name of the built-in function.
# Pargs	The number of parameters the function requires. The top items of the stack get passed to the function.

The preceding example fetches the value of the system variable %Page and pushes it onto the stack. It then pushes two constants, 1 and 8, onto the stack. With those three stack items, it calls the built-in function Substring, which needs three parameters: a string, a starting point, and a length.

```

fetch      Builtin  - %Page
push       1
push       8
builtin    - Substring #Parms=3

```

The PeopleCode that generated this output is:

```
Substring(%Page, 1, 8)
```

CALL

The CALL operator indicates that PeopleCode will perform a function next. CALL has one operand with three mutually exclusive values.

DLL	Calls a dynamic link library.
Int	Calls an internal PeopleCode function (part of the same program). Includes the name of the function and number of parameters.
Ext	Calls an external PeopleCode function. Includes the record name, field name, program type, name of the function, and number of parameters.
Params	The number of parameters passed to the called function. The top items of the stack get passed to the function.

The example indicates that PeopleCode will call the function calc_fte on Funclib_HR.FTE. This function requires one parameter:

```
call ext FUNCLIB_HR.FTE FieldFormula fcn=calc_fte params=1
```

START EXT

Immediately after CALL with the Ext operand, you will see START EXT. This operation records the beginning of an external PeopleCode function. It uses two operands.

Function	The name of the function starting.
Params	The number of parameters passed to the function.

You would see this example at the beginning of processing for the PeopleCode function calc_fte when another program calls it.

```
start ext calc_fte params=1
```

RETURN

The RETURN operator, which takes no operands, indicates a return to a calling program. It typically represents the end of a called PeopleCode function. If RETURN appears in the top-level program, it means PeopleCode has terminated the program.

STORE

The STORE operator indicates that the value of the top item in the stack goes into a field or variable. STORE means something gets updated, so it's something you will want to pay attention to when you look through a big trace to find out how in the world a field or temporary variable could have possibly gotten a value. STORE uses one operand with two mutually exclusive values.

Field	The record.field where the system puts the value of the top item in the stack. If you opt to show assigned values, you will see a Store Field. Store Field shows the updated record.field and its current value.
Temp	The name of the temporary variable which receives the value of the top item in the stack.

The example gets today's date with a fetch. It then puts the value of today's date in JOB.EFFDT.

```
fetch    builtin    - %Date
store    JOB.EFFDT
Store Field:JOB.EFFDT Value=1997-11-11
```

ERROR

The ERROR operator, which takes no operands signifies that the Component Processor is terminating the program.

STATEMENT

This operator can mislead you at first until you get used to seeing it. Most of the time in a trace, you can ignore it. It marks the beginning of a top-level PeopleCode statement—one not part of a controlling statement like EVALUATE or IF. Its single operand shows the location of the next top-level statement.

Printed Data Values

When data item values are shown in the trace, the following formats are used:

<i>Data Value Item</i>	<i>Description</i>
Int= nnn	Displays the value of a numeric field.

Str = xxx	Displays the first 30 bytes of a string value, or the value of the function being passed.
Date = yyyy-mm-dd	Displays a date in Year-Month-Day format.
Bool = true Bool = false	Displays the value of a Boolean variable.
variable = record.field value = xxx	Displays the name of a record.field passed by reference, and its data value
Temp = #nnn Value = xxx	Displays the number of a temporary variable and its data value.
<i>datatype</i>	Displays the data type (such as Record, Row, Rowset, Array, and so on.) and an internal value.

Other Items in the Log File

There are several other items that can appear in a debugging trace. The following table describes those items.

<i>Trace Item</i>	<i>Description</i>
Store Field:record name.field name Value=xx	Issued when the assignments trace option is selected. It contains the record and field names and the value that is stored.
Fetch Field:recordname.fieldname Value=xx	Issued when the Fetch Fields option is selected. It contains the record and field name and the value that is retrieved.
Fetch Field:recordname.fieldname Contains Null Value	Issued when the Fetch Field option is selected and the selected record.field contains a null value.
Fetch Field:recordname.fieldname Does Not Exist	Issued when the Fetch Fields option is selected and when the field is not found.
Branch Taken	Displayed after a branch test when the branch is taken.
Field Not Found, Statement Skipped	Displayed whenever a "referenced field was not found" error causes the PeopleCode processor to skip to the next statement.

<i>Trace Item</i>	<i>Description</i>
vvvvvv PeopleCode Program Listing	Issued when the List Program option is selected. It marks the beginning of a PeopleCode program listing.
^^^^^ PeopleCode Program Listing End	Issued when the List Program option is selected. It marks the end of a PeopleCode program listing.
Error Return -> NNN	Issued when a fatal error condition terminates the PeopleCode program.

Find In . . .

You can use the Application Designer's Find In feature to search for strings, either in PeopleCode programs or in SQL definitions. This feature searches:

- All PeopleCode programs and all SQL statements
- Just PeopleCode programs
- Just SQL statements

You can further refine your search by specific project and by item searched, that is, you can specify if you want record PeopleCode, page PeopleCode, menu PeopleCode, and so on.

All output from the search is placed in an output window. You can save these results to a file, copy them, clear them or print them.

From the output window, you can immediately open any of the PeopleCode programs or SQL statements listed.

Also, from the output window, you can insert selected records into a project. Then, if you need to search those records again, you can search by project.

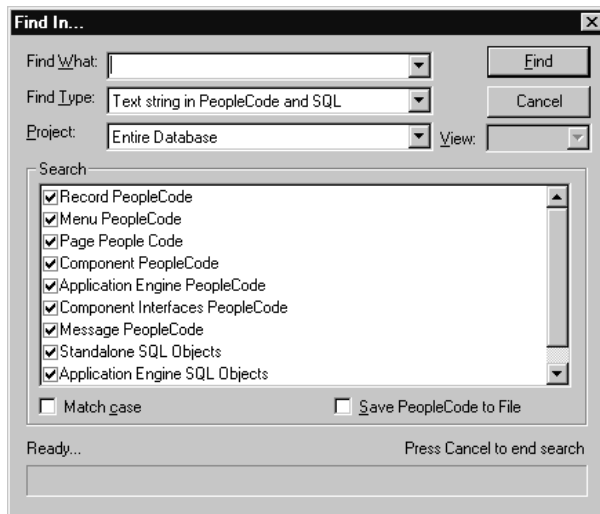


If you want to create a file that contains all the PeopleCode for a project (or database) you can use the Find In. . utility, and search for ";". Be sure to check Save PeopleCode to File.

To find a text string in PeopleCode or SQL

1. In Application Designer, choose **Edit, Find in**

This displays the Find in dialog.



Find in Dialog

Use this dialog to specify a string to be searched for.

2. Type the search string you want to find in the Find What edit box

If you only want returned those items that match the case of what you typed, check the Match Case checkbox at the bottom of the dialog.

3. Specify with the Find Type edit box whether you are searching in PeopleCode and SQL, just PeopleCode or just SQL.
4. Select the project you want to search.

You can search the entire data base, or any existing project.

5. Select the view you want to search (optional)

If you chose to not search the entire database, you can specify if you want to search the Development view or the Upgrade view. The default is the Development view.

6. Select the items you want to search

You can search all items that contain either PeopleCode or SQL, or just a subset of items.

7. Save the search results to a file (optional)

You can save the results of a PeopleCode search to text file, which you can view or print using a text editor or word processor. The text file will contain the entire PeopleCode program that contained the string.

If you want to save your results to a file, check the Save PeopleCode to File checkbox at the bottom of the dialog. The results will be saved to the file, as well as display in the Application Designer **Find In . . .** output window

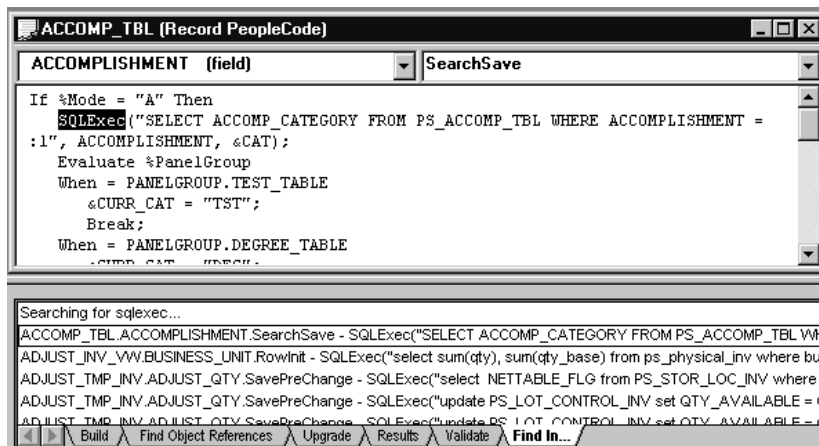
8. Click the **Find** button to start the search.

As Find In . . . searches the database, it displays a counter at the bottom of the Find In dialog indicating the number of PeopleCode programs searched.

You can click on the Cancel button to stop the process.

9. Check Find in Output Window for results.

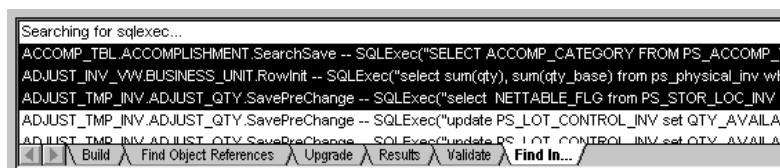
The results of the search appear in the **Find in . . .** tab of the output window. Each line shows where the string was found. You can open any of the programs listed by double-clicking on a line in the output window.



Opening a PeopleCode Program from the Find in Tab

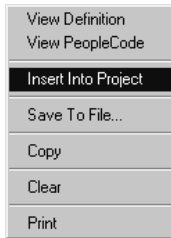
To save records in a project

1. Run Find In to search for a string.
2. Highlight the references you want saved in the output window using click-shift.



Find in output window with records selected

3. Right-click on the highlighted records and select **Insert Into Project**.



Find in pop-up menu

All the selected records will be inserted into the current open project. Save your project.

The next time you search, you can just search your project (select a Project on the Find In dialog box) instead of searching the entire database.

Cross Reference Reports

You may find a situation where a field value changes, and you don't know how it changed. There are two ways of finding all references to a field.

- Find Object References in Application Designer
- Cross reference reports



For more information about Find Object References, see Using Application Designer Projects.

PeopleTools is delivered with three PeopleCode cross reference reports:

- **XRFFLPC.** Reports on all fields in the system referenced by other PeopleCode programs. The report sorts by record names, then field names. XRFFLPC shows the records, fields, and PeopleCode program types that reference each field.
- **XRFPCL.** Reports on the fields that each program references. It sorts the report by record definition, field name, then PeopleCode type. It shows the records and fields referenced for each program. This report and XRFFLPC complement each other by using converse approaches to reporting the cross references.
- **XRFPNPC.** Reports on pages with PeopleCode. This report show pages containing fields with PeopleCode attached to them.

You can run these using PeopleSoft Query and either view the reports online or print them out.

PUBLIC.QUERY.XRFPCL - Query

File Edit View Go Favorites Criteria Help

Fields Criteria SQL Results

Record **Field Name** **Prog Type** **Record (Table) Name** **PeopleCode Reference**

ABSENCE_HIST	BEGIN_DT	3	ABSENCE_HIST	DURATION_DAYS
ABSENCE_HIST	BEGIN_DT	3	ABSENCE_HIST	RETURN_DT
ABSENCE_HIST	BEGIN_DT	3	DERIVED_HR	DAY_OF_WEEK
ABSENCE_HIST	BEGIN_DT	3	FUNCLIB_HR	DAY_OF_WEEK
ABSENCE_HIST	DURATION_DAYS	0	ABSENCE_HIST	BEGIN_DT
ABSENCE_HIST	DURATION_DAYS	0	ABSENCE_HIST	DURATION_HOURS
ABSENCE_HIST	DURATION_DAYS	0	ABSENCE_HIST	RETURN_DT
ABSENCE_HIST	RETURN_DT	0	ABSENCE_HIST	BEGIN_DT
ABSENCE_HIST	RETURN_DT	3	ABSENCE_HIST	BEGIN_DT
ABSENCE_HIST	RETURN_DT	3	ABSENCE_HIST	DURATION_DAYS
ACCOMPLISHMENTS	ACCOMPLISHMENT	3	ACCOMPLISHMENTS	DESCR
ACCOMPLISHMENTS	ACCOMPLISHMENT	3	ACCOMPLISHMENTS	ORG
ACCOMPLISHMENTS	ACCOMPLISHMENT	3	ACCOMP_TBL	DESCR
ACCOMPLISHMENTS	ACCOMPLISHMENT	6	ACCOMPLISHMENTS	DESCR
ACCOMPLISHMENTS	ACCOMPLISHMENT	6	ACCOMPLISHMENTS	ORG
ACCOMPLISHMENTS	ACCOMPLISHMENT	6	ACCOMP_TBL	DESCR

Ready Rows Fetched = 22844

XRFPCL Query Results



For more information about running queries, see [Using PeopleSoft Query](#).

CHAPTER 13

Using Three-Tier and Windows Client

This section covers a number of issues related to the use of PeopleCode methods and built-in functions when used in a three-tier or windows client architecture. PeopleSoft applications are written to work in the PeopleSoft Internet Architecture. However, you may have legacy applications or an environment that requires using this older architecture.

Implementing Dynamic Tree Controls

Dynamic tree controls are a search tool that can be embedded in a page.



Dynamic Tree controls are only available in Windows Client. They are **not** available in the PeopleSoft Internet Architecture.

Dynamic trees give the user a view of hierarchical data structures and enable them to drill down through the hierarchy to a particular row of data. The data in the row can be accessed by a PeopleCode program, which would most likely be run from a command push button or a pop-up menu item.

Dynamic trees are only available in Windows client. If you want to use trees in the PeopleSoft Internet Architecture, use the GenerateTree function.



For more information see Using the GenerateTree Function.



A dynamic tree control is **not** the same as an ActiveX tree view control or an HTML tree.

There are two types of dynamic tree controls:

- **multiple-table dynamic trees**, in which the user drills down through a hierarchy of parent and child records
- **recursive single-table dynamic trees**, in which the user drills down through rows that have internal dependencies within a single record

Dynamic tree controls must be at level zero of the page.

Although the nodes in a dynamic tree display a single field, they are in fact bound to an entire row of data. When the user selects a node in the tree, PeopleCode can retrieve the value of any record field on that row.

The root node of the tree takes its initial value from a level-zero record field to which it is bound. If the value of this record field changes, you can use the PeopleCode **RefreshTree** function to update the value in the tree's root node.



For more information see `GetSelectedTreeNode`, `GetTreeNodeRecordName`, `GetTreeNodeValue`, `GetSubContractInstance`, `RefreshTree`.

To build a multiple-table dynamic tree:

This procedure describes how to build a dynamic tree to access data on multiple tables that are related to one another hierarchically. The descriptions are illustrated with a simple example, in which we set up a hierarchy of animal categories, types, and breeds for a veterinary clinic database.

1. Make sure the tables for the tree are set up hierarchically.

Multiple-table dynamic trees are based on parent-child relationships among multiple records. Each child node of the tree must be bound to a child record of the record to which its parent node is bound.



Recall that a child record has all of the key fields belonging to the parent record, plus at least one more key.

For example, for the animal categories tree, we set up three tables: `ANIMALCATEGORY`, `ANIMALTYPE`, and `ANIMALBREED`. The key fields in these tables are as follows:

The `ANIMALCATEGORY` table, the highest level table, has this key field:

Key Field	Description
<code>ANIMALCATEGORY</code>	The highest level key, which can have values of either "Pet" or "Livestock".

The `ANIMALTYPE` table is a child of `ANIMALCATEGORY`:

Key Field	Description
<code>ANIMALCATEGORY</code>	The highest level key, which can have values of either "Pet" or "Livestock"
<code>ANIMALTYPE</code>	The general category of veterinary patient, such as "Dog", "Cat", or "Rodent".

The `ANIMALBREED` table is a child of `ANIMALTYPE`:

Key Field	Description
ANIMALCATEGORY	The highest level key, which can have values of either "Pet" or "Livestock"
ANIMALTYPE	The general category of veterinary patient, such as "Dog", "Cat", or "Rodent".
ANIMALBREED	The breed of animal, such as "German Shepherd" or "Persian".



Note on siblings. At each level of the hierarchy below the root node, you can have two or more "sibling" records, which must both be child records of the same parent. The example does not include any siblings.

2. Add the tree and other required objects to a page.

You'll need the following objects on the dynamic tree control's page:

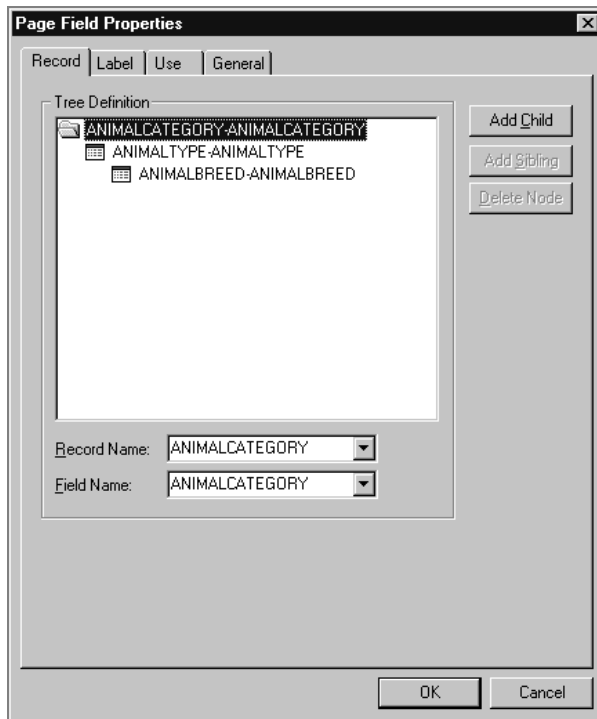
- **Dynamic tree control.** After you place and size this control on the page, it appears as a blank rectangle. The dynamic tree must be added at level zero of the page.
- **Push button or pop-up menu.** The data in the dynamic tree must be accessed by a PeopleCode program. This program will most likely be in a push button or pop-up menu.
- **Any other controls your implementation requires.** The other objects on the page will depend on your implementation.

In our example, we use a command push button to run the PeopleCode program, and add two Derived/Work fields to display and store values for ANIMALTYPE and ANIMALBREED.

3. Edit the dynamic tree control's properties.

Right-click on the dynamic tree control and choose Page Field Properties to display the Page Field Properties dialog. Link the root of the tree to the highest key field of the highest-level record in the record hierarchy. Add child records, binding them the highest key field that distinguishes them from the parent record.

In our example, we link the root of the tree to ANIMALCATEGORY.ANIMALCATEGORY, then click Add Child to add a new child node. We bind this child node to ANIMALTYPE.ANIMALTYPE (the child table of ANIMALCATEGORY). We then add a child node of ANIMALTYPE, binding it to ANIMALBREED.ANIMALBREED.



Page Field Properties of Dynamic Tree Control

4. Add PeopleCode to a push button or pop-up menu item for accessing data from the dynamic tree.

The PeopleCode program determines the node that the user has selected using the **GetSelectedTreeNode** function. This function returns an Object value that represents the node that the user has selected. You can pass this value to the **All** function to determine whether any node has been selected, and to other functions to retrieve the current nodes record, value, and parent record.

In our example, the program determines if the current node is bound to the ANIMALBREED record. If it is, then it copies the values from the currently selected row into the page's Derived/Work fields.

```
&TREENODE = GetSelectedTreeNode(RECORD.ANIMALCATEGORY);

/* make sure user has selected a node */

If All(&TREENODE) Then

/* if the user has selected a terminal node */

    &RECNAME = GetTreeNodeRecordName(&TREENODE);

    If &RECNAME = RECORD.ANIMALBREED Then

        ANIMALTYPE = GetTreeNodeValue(&TREENODE, ANIMALBREED.ANIMALTYPE);

        ANIMALBREED = GetTreeNodeValue(&TREENODE, ANIMALBREED.ANIMALBREED);

    End-If;
```

End-If;

In a multiple-record dynamic tree, you will probably want to take different actions, depending on the record to which the currently selected node is bound. For example, you may want to take action only if the user has selected one of the tree's terminal nodes. To accomplish this, you can use the **GetTreeNodeRecordName** function to determine the record to which the currently selected node is bound and test the value returned by the function in an **If** or **Evaluate** statement.

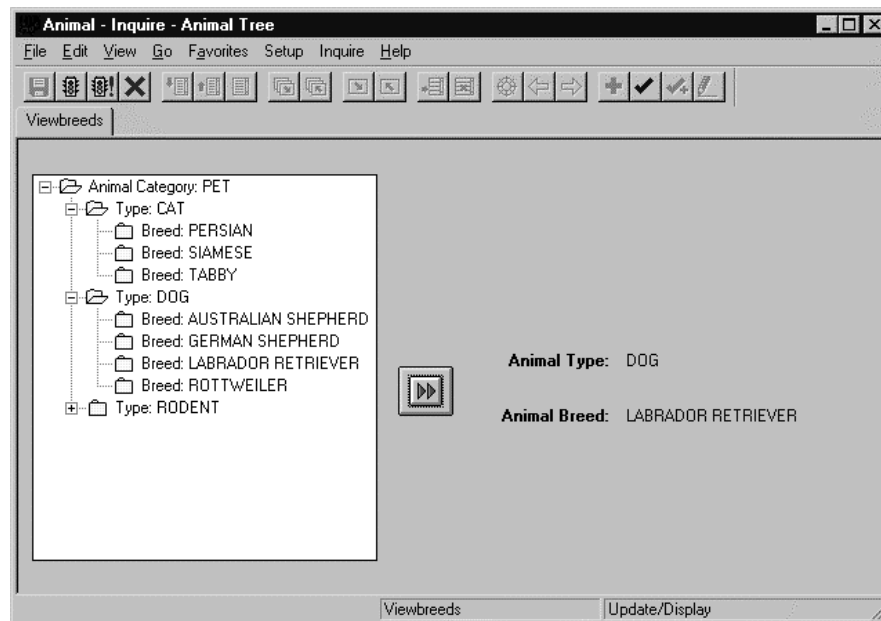


For more information on PeopleCode functions related to dynamic tree controls, see **GetSelectedTreeNode**, **GetTreeNodeRecordName**, **GetTreeNodeValue**, **GetSubContractInstance**, **RefreshTree**.

5. Test the control.

To test the page you will of course need to insert it into a component, then associate the component with a standard menu item. You will also need to assign yourself rights to the new menu item in Security Administrator.

The example dynamic tree control panel looks like this:



Dynamic Tree Control Example at runtime

The root node in the tree is populated the level-zero record field to which it is bound. When the user selects a terminal node in the tree (a Breed), then clicks the push button, PeopleCode copies values from the selected row into the Derived/Work fields on the right. If the user has not selected a terminal node, then the PeopleCode does nothing.

Recursive Single-Table Dynamic Trees

Recursive single-table dynamic trees access data on a table containing rows of data that have internal dependencies. These types of dynamic trees are intended for use in tables where the rows of data capture relationship between components and subcomponents, and where the same object can have subcomponents and be a subcomponent of other objects.

The classic application of this type of data structure and tree would be in a bill of materials. In manufacturing databases, tables of this type can contain very large numbers of rows, and the hierarchies of objects can be extremely deep. The dynamic tree control generates SQL to select specific rows from these large hierarchies quickly and efficiently.

A simple example of this type of structure, which is much smaller than a real-world application, but which will serve to illustrate the basic concepts, would be a database representing objects in the solar system and their satellites. A planet, for example, is a satellite of a star; a planet may have its own satellites, moons, which in turn may have their own satellites. For purposes of this example, let's assume that Jupiter's moon Ganymede has two "submoons."

To represent this structure in a single table, you could use these fields:

Field	Description
ORBITEE	The object around which the satellite revolves.
ORBITER	The satellite.

If you populated this table with some data about the solar system, it might look like this:

Orbitee	Orbiter
Sun	Saturn
Sun	Jupiter
Jupiter	Amalthea
Jupiter	Callisto
Jupiter	Ganymede
Ganymede	Wally
Ganymede	Beaver

A recursive single-table dynamic tree would enable a user to drill down through this hierarchical data structure to find a specific satellite.

To build a single-table dynamic tree control:

This procedure describes the steps in building a single-table dynamic tree control, illustrated by an example page that accesses data about celestial bodies and their satellites in the solar system.

1. Make sure the record is set up correctly.

The single record should be structured as described in Recursive Single-Table Dynamic Trees with fields for components and subcomponents that can store the same values. It can also have higher-level keys.

In our example, we'll use STARSYSTEM as a high-level key, ORBITEE as a second key, and ORBITER as a third field. We'll also add a field, ORBTYPE to classify the object type of ORBITEE. The example record, SATELLITES, has these field.

Field	Key	Search	Description
STARSYSTEM	Key	Search	The star system.
ORBITEE	Key	Search	The object about which the satellite revolves.
ORBITER	Non-key	Non-search	The satellite.
ORBTYPE	Non-key	Non-search	The object type of ORBITEE (star, planet, etc.)

Notice that SATELLITES has two search keys. The root node of the dynamic tree will take it's value from the second key field, ORBITEE. The child node of the dynamic tree will be bound to ORBITER. Notice that ORBITER is not a key: this is a requirement in a recursive single-table dynamic tree: *the child node of the tree cannot be bound to a key field.*

Because ORBITER is not a key, and the ORBITEE field will have duplicate values, the SATELLITES record would have to permit "duplicate" rows. For example, the following two rows would be treated as duplicates, because their key values are identical:

STARSYSTEM	ORBITEE	ORBITER	ORBTYPE
SUN	JUPITER	WALLY	PLANET
SUN	JUPITER	BEAVER	PLANET

This makes it difficult to populate the table with data in PeopleTools, which prevents you from adding rows with duplicate key field values.

There are two ways around this problem. One is to construct a table in which ORBITER is a key, then base the dynamic tree control on a view (derived from this table) in which ORBITER is not a key. In our example, assume that the SATELLITES record is such a view.

An alternate solution would be to add another key field to the table, such as ORBITER_ID, and make ORBITER a non-key field. This ensures that the rows are unique, permits data entry into the table from the page, and also provides a non-key field to which the child node in the dynamic tree can be bound.

2. Add controls to the page.

The page must contain:

- The dynamic tree control. The dynamic tree control must be at level zero.
- A command push button or pop-up menu item to contain a PeopleCode program that accesses the dynamic tree's data.

- Any other controls required by your implementation.

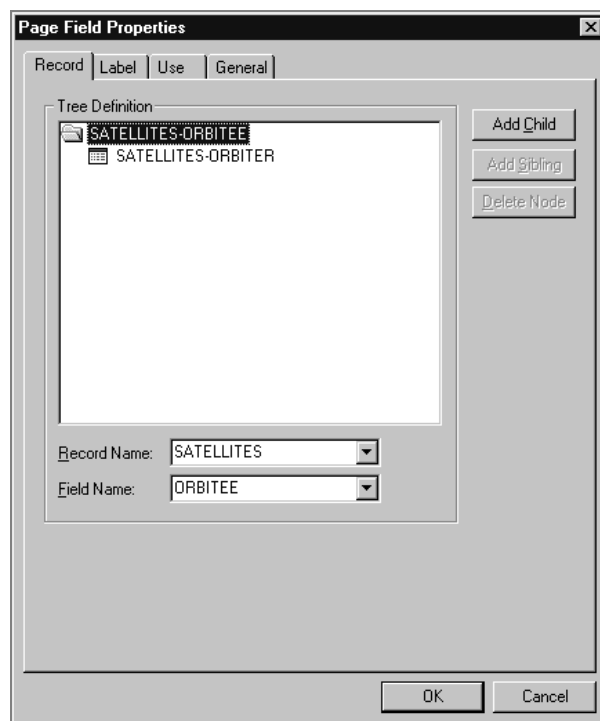
In our example, we will add a high-level key linked to SATELLITES.STARSYSTEM, a command push button, and Derived/Work fields to display values from ORBITEE, ORBITER, and ORBTYPE.

3. Edit the dynamic tree control's properties.

Right-click in the dynamic tree control and choose **Page Field Properties** to display the Page Field Properties dialog.

This type of tree only requires two nodes: one parent and one child. Bind the root of the tree to the record field that represents the component in the component-subcomponent pair. Then add a child to the tree and bind it to the subcomponent member of the pair.

In our example, we bind the parent to SATELLITES.ORBITEE and the child to SATELLITES.ORBITER.



Page Field Properties of Single-Table Tree Control

4. Add PeopleCode to access the dynamic tree data.

In the FieldChange event of the record field to which your push button is bound (or the ItemSelected event of the menu item if you are using a pop-up menu) add PeopleCode for accessing the dynamic tree's data.

In our example, we add the following program, which copies data from the selected row it into corresponding Derived/Work fields:

```

&TREENODE = GetSelectedTreeNode(RECORD.SATELLITES);

ORBITEE = GetTreeNodeValue(&TREENODE, SATELLITES.ORBITEE);

ORBITER = GetTreeNodeValue(&TREENODE, SATELLITES.ORBITER);

ORBTYP = GetTreeNodeValue(&TREENODE, SATELLITES.ORBTYPE);

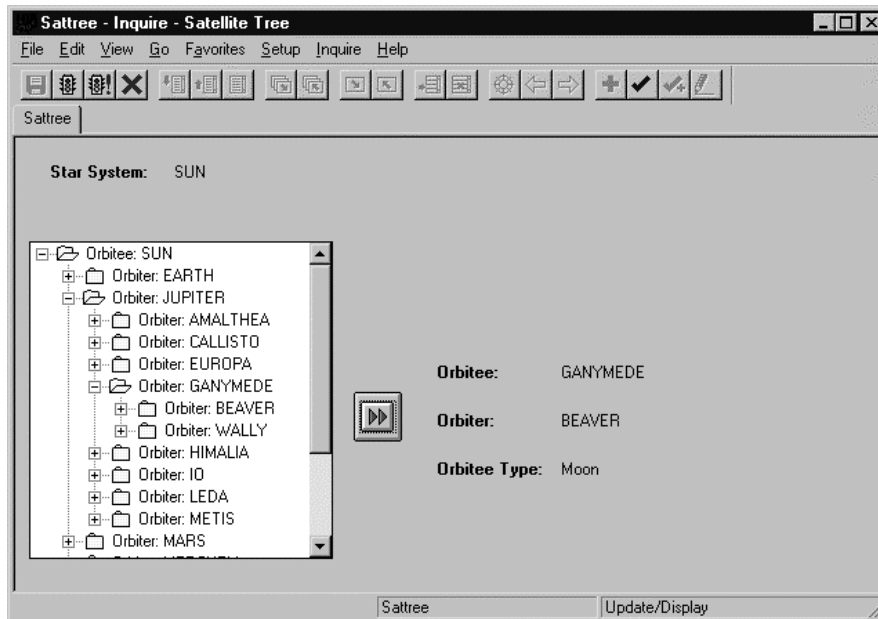
```

You can use the **GetSelectedTreeNode** function to return an Object value representing the currently selected node, then pass this value to the **GetTreeNodeValue** function to return the value of a field bound to the node. Notice that you can reference any field on the row, not just the field displayed in the control.

5. Test the page.

To test the page you will of course need to insert it into a component, then associate the component with a standard menu item. You will also need to assign yourself rights to the new menu item in the Maintain Security pages.

The example panel looks like this:



Single-Table Tree Control at Runtime

The root node in the tree is populated from the second key field ORBITEE. When the user expands the tree and selects a node, then clicks the command button, PeopleCode copies values from the selected row into the Derived/Work fields.

How It Works

A recursive single-table dynamic tree control is built using only two nodes, level 0 (the parent) and level 1 (the child).

Recall that in tables of this type, there are two fields, one representing a "component," the other a "subcomponent." In our example these fields are ORBITEE and ORBITER. The parent node of the tree is bound to the component field in the component-subcomponent pair, which in the case of our example would be ORBITEE. The child node is bound to the subcomponent field, which in the case of the example would be ORBITER.

When the user expands the root of the tree, an SQL statement uses the "component" record field value of the row to which the root node is bound, expanding the node by selecting all the subcomponent rows belonging to that component. In the example, it finds all the satellites of a celestial body: if the root node were SUN, the SQL statement would expand the node by selecting all rows where STARSYSTEM = 'SUN' and ORBITEE = 'SUN':

```
SELECT STARSYSTEM, ORBITEE, ORBITER, ORBTYPE FROM PS_SATELLITES WHERE
STARSYSTEM='SUN' AND ORBITEE='SUN' ORDER BY STARSYSTEM, ORBITEE
```

When the user selects a node on the next level down, the subcomponent value of that row becomes the component value in the SQL. For example, if the user expands the JUPITER node, the value of ORBITER in the row would determine the value for ORBITEE in the SQL query, which would expand the node by selecting all rows where STARSYSTEM = 'SUN' and ORBITEE = 'JUPITER'.

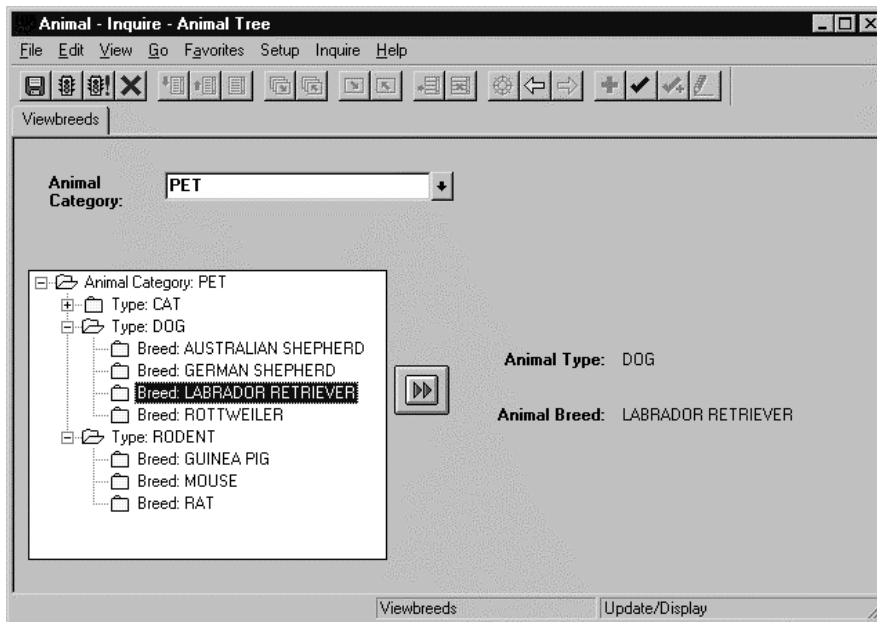
```
SELECT STARSYSTEM, ORBITEE, ORBITER, ORBTYPE FROM PS_SATELLITES WHERE
STARSYSTEM='SUN' AND ORBITEE='JUPITER' ORDER BY STARSYSTEM, ORBITEE
```

This process is repeated as the user expands each subordinate node. If the SQL query doesn't return any rows, the node is not expanded.

Controlling the Root Node of the Dynamic Tree

In the preceding examples the root node of the dynamic tree was bound to a record field that could not be controlled from within the page. If the value of this record field can be changed in the page, you can add a PeopleCode program that uses the **RefreshTree** function to update the value of the root node after changing the value.

To illustrate one way of doing this, suppose that in our first example (To Build a Multiple-Table Dynamic Tree) we added an editable edit box bound to the same record field as the root node of the dynamic tree. If this edit box prompted against another table containing the set of animal categories, the page would look like this:



Page Showing User Control of Tree Root Node

If the user changes the value in the edit box, PeopleCode can update the value in the root node of the dynamic tree using the **RefreshTree** function:

```
RefreshTree (RECORD.ANIMALCATEGORY) ;
```

The most logical place for this program would probably be the FieldChange event of the record field to which the Animal Category edit box is bound. This way the tree would be updated whenever the user changes the value in the edit box.

Implementing ActiveX Controls

ActiveX controls present both a visual and programmatic interface to application developers. The visual interface provides for the selection of controls, painting onto pages, and the setting of properties.



ActiveX controls are only available in Windows Client. They are **not** available in the PeopleSoft Internet Architecture.

The programmatic interface has three important aspects:

- properties (Color, Hide, Value, etc.)
- methods (RandomFillColumns, ExtractIcon, etc.)
- events (UserMovedBar, OnLeftClick, etc.)

Access to the properties and methods are through the PeopleCode **GetControl** function. The events are accessed through the PeopleCode editor.

The inclusion of an ActiveX control in an application requires PeopleCode programming. The amount of programming depends on the complexity of the control. A control used only for presentation (i.e., a graph) will require significantly less programming than one that requires a lot of event handling (i.e., an interactive tree.)

ActiveX controls are not automatically associated with any record. You must use PeopleCode programming to write any data entered in an ActiveX control to a record if you wish to preserve or use the information.

PeopleSoft supports the following ActiveX controls

- Microsoft Chart Control (version 6.0)
- Microsoft TreeView Control (version 6.0)
- Microsoft ImageList Control (version 6.0)



The ActiveX tree view control is **not** the same as the dynamic tree control or an HTML tree.



For more information see ActiveX Controls in PeopleTools.

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.)

The ActiveX controls supported by PeopleTools are **not** UNICODE compliant. This is a restriction of the controls themselves, not of PeopleTools. This means, if you had some non-ANSI text (such as Japanese) stored in your UNICODE database, you can only view/edit the text in a page containing an ActiveX control in machines set up for that language (such as a Japanese NT machine.)



For more information see Globalization.

Manipulating Events for ActiveX Controls

Every PeopleCode program is associated with one component (like an ActiveX control, a record field, a component, a page, etc.) and one event. To manipulate an ActiveX control during runtime, you can put your code in any of the existing PeopleCode events (RowInit, Activate, etc.), the PeopleCode ActiveX events (**PSControlInit** and **PSLostFocus**), or one of the specific events associated with that control.



After you put an ActiveX control on a page, you must **save** the page before you can access any of the ActiveX control events.

The **PSControlInit** and **PSLostFocus** events are only available for ActiveX controls.

PSControlInit

The **PSControlInit** event fires **every** time the page is redrawn. This means it fires after the component buffers are loaded and after a RowInsert or a RowDelete. It also fires after a user changes a field using a drop down or other prompt. It fires after a user scrolls up or down in a page.

This event is only available with ActiveX controls. Generally, you will place PeopleCode that will synchronize the control with the buffer data. Because this event can be fired many times during the normal running of a page, any PeopleCode in the PSControlInit event should be designed so it isn't sensitive to where and how often it gets run.

If you want to use this event for initially loading data into the control, you should write your code in such a way that it is only run once. The following code example shows a possible way of doing this:

```
Component boolean &RUN;

Local object &MYTREE;

Function PSControlInit()

    &MYTREE = GetControl();

    If Not (&RUN) Then

        /* do initial processing */

        &RUN = True;

    Else

        /* do other processing */

    End-If;

End-Function;
```

PSLostFocus

The **PSLostFocus** event fires for an ActiveX control page field when the user removes focus from the control. For example, this event fires when the user tabs off the control. PSLostFocus does **not** fire when the user tabs from one field to another within the control. It only fires when

the focus completely leaves the control. Use this event to move data from the control to the component data buffers.



For more information about these events see PeopleCode and the Component Processor.

Control Specific Events

In addition to PSControlInit and PSLostFocus, every ActiveX control also comes with its own set of events. A set of events is particular to that ActiveX control. For example, the Chart control has events like OnAxisTitleUpdated, while the TreeView control has events like OnCollapse.

When you put an ActiveX control on a page, PeopleTools creates a blank PeopleCode "template" with each event, in which the event handler signature is given. In the signature, the variables that get passed to the event, and what data type those variables are, is listed.

For example, the OnMouseDown event, which is specific to the TreeView control, has the following code created when the page with the control is first saved:

```
Function OnMouseDown(&BUTTON As number, &SHIFT As number, &X As any, &Y As any);  
  
    /* TODO: Add your code here */  
  
End-Function;
```

To access ActiveX control events

1. Register your ActiveX control (using Client Workstation Install) and place it ActiveX on your page.
2. Name the ActiveX control and save the page.
3. Select the ActiveX control, then access the PeopleCode associated with that control.

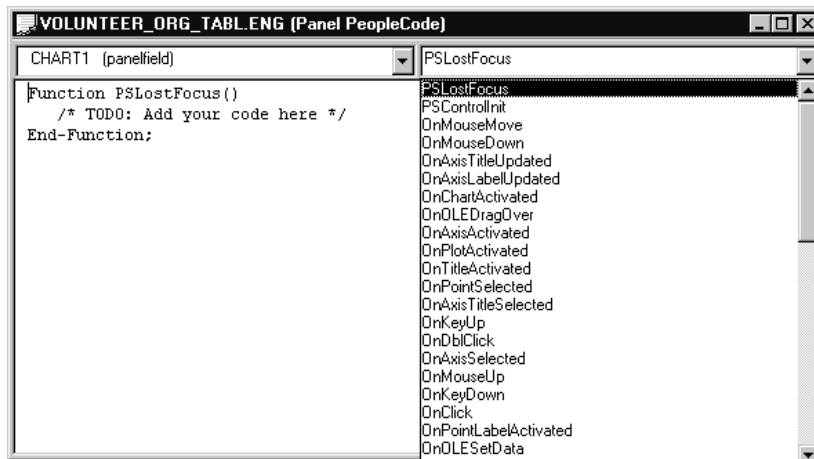
There are two ways to do this:

- Select View, View PeopleCode

OR

- Left-click and select **View PeopleCode** from the pop-up menu.
4. Select the right-hand dropdown button just under the title bar.

A list of all the events for that ActiveX control will be displayed.



List of events for Chart ActiveX control

Manipulating ActiveX Control Properties and Methods

You can manipulate an ActiveX control's properties and methods either at design time or at runtime. You can use the PeopleTools ActiveX control property dialog, or the control's built-in property dialog, if one is provided by the control vendor. Design time property settings are persistent, though they may be changed at runtime using PeopleCode, or a user interface provided by the control.

To access an ActiveX control from your PeopleCode program, you must use the **GetControl** function. The GetControl function returns a reference to an 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 or methods of that control.



The **GetControl** function returns a reference to an ActiveX control. You can't access any controls until after the page processor has loaded the page. You shouldn't use this function in an event prior to the Activate event.

For example, you can set the font size of a TreeView control with the following code:

```
&MYTREE = GetControl();

&MYTREE.Font.Size = 18;
```

Or you could retrieve a node that was highlighted by the user:

```
&MYTREE = GetControl();

&MYNODE = &MYTREE.SelectedItem;
```

Data Types for Declaring ActiveX Controls

You should use the data type **Object** when you declare your ActiveX controls, or any other object instantiated from an ActiveX control.

For example, you can instantiate a node object from a TreeView control, so both are declared as type **Object**.

```
Local Object &MYTREE, &MYNODE;
```

```
Component Object &MYCHART;
```

The purpose of the **Object** data type is to hold objects during the course of a session so that you can run its methods.



Important! ActiveX controls are **not** associated with any record field. **Object** is a valid data type for variables, but not for record fields. You can store data from the object, different values, but not the object itself. ActiveX Controls must be initialized every time the page is activated at runtime.

Initializing an ActiveX Control with Data

You don't just want to control what a control looks like. You also want to be able to fill it with data, both when the page is first displayed and when other fields on the page have been changed.

In the following code example, the chart control is loaded with data, based on the data in the component buffers. This code would be in the **PSControlInit** event for the ActiveX control:

```
Function PSControlInit()

&CONTROL = GetControl()

&CONTROL.ChartType = 3;

&MAX = ActiveRowCount(ROLL.ABSENCE_HIST);

&CONTROL.ColumnCount = 1;

&CONTROL.RowCount = &MAX;

For &I = 1 To ActiveRowCount(ROLL.ABSENCE_HIST)

    &VAL = ABSENCE_HIST.DURATION_DAYS.Value;

    &DATE = ABSENCE_HIST.BEGIN_DT.Value;

    &CONTROL.Row = &I;

    &CONTROL.Data = &VAL;

    &CONTROL.RowLabel = &DATE;
```

```
End-For;  
  
End-Function;
```

If the user types a new value into the **Days** field, you want the chart control to reflect this change. 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;
```

You don't need to redraw the page: the control is automatically updated when the code is executed.

Because this code is not in an event that's part of the ActiveX control, you must specify which page in the component the ActiveX control sits on, and you must specify the control name.



For more information see GetControl.

Using ScrollSelect Functions

Normally, when the user opens a page, the page scroll areas are filled with rows automatically, based on the structure and relationships of the page's record definitions. *ScrollSelect* functions are a special set of functions that enable you to control this process programmatically. These functions are:

- ScrollSelect
- ScrollSelectNew
- RowScrollSelect
- RowScrollSelectNew



The ScrollSelect functions are kept for backwards compatibility. New applications should use the data buffer access classes and the appropriate methods and properties instead. For more information see Data Buffer Access.

Although these four functions differ from one another significantly in their effect, their syntax and usage is similar. Once you have mastered one of the functions, you will be able to use all of them. As an example, let's look at how to use **ScrollSelect**, then examine the specific differences among the four functions.



All ScrollSelect functions work with grid controls.

In PeopleTools 8, the rowset class was introduced. This class has two methods, **Select** and **SelectNew**, which, depending on the parameters used with them, act as either **ScrollSelect** or **RowScrollSelect**, or as **ScrollSelectNew** or **RowScrollSelectNew**, respectively. The **ScrollSelect** functions are still supported, however, PeopleSoft recommends using the new rowset object with the new methods.



For more information, see [Using the Select and SelectNew Methods](#)

In addition to these methods there is the record class method **SelectByKey** that allows you to select into a record object. If you're only interested in selecting a single row of data, you may want to consider this method instead.



For more information see [SelectByKey](#).

What ScrollSelect Does

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 defined on the page) 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 the same key fields as 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 you actually need.

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. When a scroll is selected into, any autoselected child scrolls will also be read. The child scrolls will be read using a where clause that filters the rows according to the where clause used for the parent scroll, using a subselect.

ScrollSelect also accepts an optional SQL string that can contain a WHERE clause restricting the number of rows selected into the scroll area. The SQL string can also contain an ORDER BY clause, enabling you to sort the rows.

ScrollSelect functions generate an 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 statements 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.

ScrollSelect Syntax

The syntax of **ScrollSelect** is:

```
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]...
```

Although the syntax is complex, it really does only four things:

- Specifies a target scroll area into which to read the selected rows.
- Specifies the select record from which to select rows.
- Passes a string containing a SQL WHERE clause to restrict the selection of rows and/or an ORDER BY clause to sort the rows.
- Optionally switches on the Turbo ScrollSelect performance optimization.

Let's examine these parts of the syntax one at a time.

Specifying the Target Scroll Area

The first part of the argument list specifies the page scroll into which rows will be selected, which we'll call the *target scroll area*:

```
levelnum,
[RECORD.level1_recname, [RECORD.level2_recname,]]
RECORD.target_recname
```

The *levelnum* parameter specifies the scroll level of the target scroll area, which can be 1, 2, or 3. The **RECORD.target_recname** parameter specifies the default scroll record of the target scroll area.

The syntax of the two optional **RECORD.levelx_recname** parameters works as described in Scroll Path Syntax with RECORD.recordname.



In this function you don't specify row numbers for higher-level scrolls.

The SQL String

The *sqlstr* parameter is a string literal containing a SQL WHERE clause and/or ORDER BY clause:

```
[, sqlstr [, bindvars]]
```

The WHERE clause explicitly selects rows from the select record; the optional ORDER BY clause sorts the rows. For example, the following statement selects rows in which EMPLID matches the level-one key and the charge amount equals or exceeds 200, sorting the rows by EXPENSE_AMT:

```
ScrollSelect(2, RECORD.BUS_EXPENSE_PER, RECORD.BUS_EXPENSE_DTL,
RECORD.BUS_EXPENSE_DTL, "WHERE EMPLID=:1 AND EXPENSE_PERIOD_DT=:2 AND
EXPENSE_AMT >= 200 ORDER BY EXPENSE_AMT", EMPLID, EXPENSE_PERIOD_DT);
```

To avoid errors, the WHERE clause should explicitly select matching key fields on parent and child rows. The following statement will result in a "No matching buffer found for level" error:

```
ScrollSelect(2, RECORD.BUS_EXPENSE_PER, RECORD.BUS_EXPENSE_DTL,
RECORD.BUS_EXPENSE_DTL, "WHERE EXPENSE_AMT >= 200 ORDER BY EXPENSE_AMT");
```

You can use bind variables in the SQL string, just as you can in a **SQLExec** statement string. Bind variables are references within the *sqlstr* string to optional field specifiers listed in the *bindvars* list. Within *sqlstr*, the bind variables are represented by integers preceded by colons:

```
:1, :2,...
```

The integers must be in numerical order. Each of these :integers represents a field specifier in the *bindvars* list, so that :1 refers to the first field specifier in *bindvars*, :2 refers to the second field specifier, and so on. For example, in the following code:

```
"where deptid between :1 and :2", DEPTID_FROM, DEPTID_TO
```

:1 is replaced by the value contained in the page field DEPTID_FROM; :2 is replaced by the value contained in the page field DEPTID_TO.

Bind variables in ScrollSelect functions work the same way as bind variables in **SQLExec** statements.



For more information see SQLExec.



Be aware that the PeopleCode compiler knows nothing about the contents of the SQL string at compile time. It will not perform standard formatting on the string contents. If there is an error in your SQL you will get a runtime error.

Specifying the Select Record

The **RECORD.sel_recname** parameter specifies the select record, from which the **ScrollSelect** function will select data. The select record must have the same key fields as the default scroll record, to assure that dependencies with other records on the page will work correctly. The select record can be identical to the default scroll record. If it is not, it must be defined and built (that is, the SQL table created) in Application Designer. The select record cannot be a derived/work record.

Turbo ScrollSelect

Turbo ScrollSelect is a feature that addresses performance issues. This feature improves the performance of ScrollSelect verbs (**ScrollSelect**, **ScrollSelectNew**, **RowScrollSelect**, **RowScrollSelectNew**), in some cases by 300%. However, it affects the way that RowInit events are generated.

To turn on Turbo ScrollSelect, add the **turbo** parameter with a value of TRUE to the end of your ScrollSelect function parameter lists. Turbo mode will be invoked only if the optional **turbo** parameter is set. This means that existing code will continue to behave as before unless you add the new parameter. For example, if your code looks like this:

```
ScrollSelect(2, RECORD.CUST_CONVER, RECORD.CUST_CONVER_DTL,
RECORD.CUST_CONVER_DTL, "where setid=:1 and cust_id=:2",
CUSTOMER.SETID, CUSTOMER.CUST_ID);
```

You should add TRUE to the end, as shown here:

```
ScrollSelect(2, RECORD.CUST_CONVER, RECORD.CUST_CONVER_DTL,
RECORD.CUST_CONVER_DTL, "where setid=:1 and cust_id=:2",
CUSTOMER.SETID, CUSTOMER.CUST_ID, true);
```

You should exercise caution in implementing Turbo ScrollSelect in existing applications, because under some circumstances it may produce undesired changes in the behavior of the application. The areas to look out for are dependencies in the order of PeopleCode execution.

Without Turbo mode, when you call **ScrollSelect**, any RowInit in the component that has not yet executed will be executed immediately, even if it is not in the record specified by **ScrollSelect**. For example, let's say you have a component with four RowInit PeopleCode programs at Level 0. The first RowInit program ("A") executes. Then in the second RowInit program ("B"), you call **ScrollSelect** to load a level-one scroll. Before the next line in program B executes, all the remaining level-zero "C" and "D" RowInits are run, followed by the RowInits in the level-one records.

With Turbo mode on, when you call **ScrollSelect**, only RowInits on the records specified by **ScrollSelect** are executed. In the preceding example, the first RowInit program ("A") executes as

before. But when the second RowInit program ("B") calls **ScrollSelect** to load a level-one scroll, only the RowInits in the level-one records execute before the next line in Program B. The remaining level-zero "C" and "D" RowInits run after RowInit "B" finishes.

In the example, a behavior change would occur if you have logic in Program B after the **ScrollSelect** that assumes that Programs "C" and "D" have already been run.

Other ScrollSelect Functions

The three other ScrollSelect functions, **ScrollSelectNew**, **RowScrollSelect**, and **RowScrollSelectNew**, are syntactically similar to **ScrollSelect**, but exhibit one or more of two functional behaviors that we'll call New Data behavior and Specific Row behavior. You can think of these behaviors as features distributed in the following feature matrix:

	- <i>Specific Row</i>	+ <i>Specific Row</i>
- <i>New Data</i>	ScrollSelect	RowScrollSelect
+ <i>New Data</i>	ScrollSelectNew	RowScrollSelectNew

New Data Behavior

Functions that end in New mark any rows read into the scroll as new rows, so that when the user saves the scroll, the scroll record is updated with the new rows. This behavior is useful, for example, if you want to update rows in the scroll record with rows selected from a different select record.

Specific Row Behavior

Functions that begin with "Row" read data from the select record into a scroll under a specific parent row, rather than automatically distributing the rows under the correct parent rows throughout the buffer. With functions exhibiting this behavior, the programmer must use the WHERE clause in the SQL string to ensure that only rows that are dependent on the parent row are read into the scroll from the select record. Otherwise, all rows will be read in under the specified parent row.

To specify the parent row, you use the additional *levelx_row* parameter in the function call when specifying the target record:

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

This is a common construct in PeopleCode. Frequently the **CurrentRowNumber** function is used to determine the row value.



For more information see Referring to Scroll Levels.

Here is an example of RowScrollSelect using **CurrentRowNumber** to determine the parent row, and a WHERE clause to ensure that only matching child rows are read into the target scroll:

```
RowScrollSelect(2, RECORD.BUS_EXPENSE_PER, CurrentRowNumber(),  
RECORD.BUS_EXPENSE_DTL, RECORD.BUS_EXPENSE_DTL, "where EXPENSE_PERIOD_DT = :1",  
EXPENSE_PERIOD_DT);
```

Using OLE Functions

OLE automation is a Microsoft Windows protocol that enables one application to control another's operation. The applications communicate by means of an **OLE object**. One of the applications (called the **automation server**) makes available an OLE object that the second application (the **client** application) can use to send commands to the server application. The OLE object has **methods** associated with it, each of which corresponds to an action that the server application can perform. The client runs the methods, which cause the server application to perform the specified actions.

PeopleCode includes a set of functions that enable you to get OLE objects from automation servers and to run the objects' methods. In other words, your PeopleCode program can be an OLE client.



The PeopleCode OLE functions do not work in the PeopleSoft Internet Architecture.

PeopleCode includes a set of functions that enable you to control other Microsoft Windows applications through Object Linking and Embedding (OLE). You can connect to any application that's registered as an OLE automation server and invoke its methods.



Differences in Microsoft Windows applications from one release to the next (that is, properties becoming methods or vice versa) can cause problems with ObjectGetProperty, ObjectSetProperty() and ObjectDoMethod().

In addition, some of the objects associated with the ActiveX controls supported by PeopleTools have special properties that can't be set using the native property. These properties can only be set using the PeopleCode ObjectSetProperty built-in function.

This section describes the PeopleCode functions you use to communicate with other applications through OLE.



For more information about OLE automation and the methods available for a particular OLE automation server, refer to the documentation for the OLE-automated application. For more information on OLE functions, see `CreateObject`, `ObjectDoMethod`, `ObjectGetProperty`, `ObjectSetProperty`.

Data Types

To support OLE, PeopleCode has a special data type—`OBJECT`—which it uses for OLE objects. The purpose of the `OBJECT` data type is to hold OLE objects during the course of a session so that you can run its methods. You can't store `OBJECT` data for any extended period of time.



Important! `OBJECT` is a valid data type for variables, but not for record fields. Because OLE objects are by nature temporary, you can't store `OBJECT` data in a record field, including work record fields.

Some OLE object methods return data to the client. You can use such methods to get data from the automation server, as long as the method returns the data in a form that PeopleCode can handle; that is, the data must be in a PeopleCode-supported data type. If the method returns data in an spreadsheet, for example, you won't be able to accept the data because PeopleCode doesn't support spreadsheets.



ActiveX Controls should also be declared as type `Object`, but the OLE functions generally shouldn't be used with ActiveX controls, outside a few specific exceptions.

Sharing a Single Object Instance

When you need the services of an OLE automation server, you create an instance of its OLE object, using the `CreateObject` function. Once you have the object, you can run its methods as often as you like. You don't need to create a new instance of the object each time.

In a typical scenario, you have a PeopleSoft component that needs to access Microsoft Excel, or Word, or some other automation server, perhaps one you have created yourself. Various PeopleCode programs associated with the component need to run OLE object methods.

Rather than create a new instance of the OLE object in each PeopleCode program, you should create one instance of the OLE object in a PeopleCode program that runs when the component starts (such as `RowInit`) and assign it to a global variable. Then, any PeopleCode program can reference the object and invoke its methods.

OLE versus WinExec

The **WinExec** and **Exec** built-in functions provide another way you can start another application from PeopleCode. Unlike the OLE functions, however, **Exec** and **WinExec** don't allow you to control what actions the application takes after you start it. You can start the application—and if you use the synchronous option you can find out when it closes—but you can't affect its course or receive any data in return.

WinExec is appropriate in two situations:

- When you just want to start an application and continue processing.
- When you have a short, unvarying process that you want to run, such as copying a file.

The **Exec** function, unlike **WinExec** and the OLE functions, is not Windows-specific. This means that you can run it on an application server to call an executable on the application server platform, which in PeopleTools release 7 can be either Windows NT or UNIX.



Important! If you use the **WinExec** function with its synchronous option, your PeopleCode program (and the PeopleSoft application) remain paused until the called program is complete. If you start a program that waits for user input, such as Notepad, your application will appear hung until the user closes the called program. The synchronous option also imposes limits on the PeopleCode.



For more information about these functions, see **Exec** and **WinExec**.

Processing Groups

As of PeopleTools 7 some online processes will run on the client machine, and others on an application server. This division of labor is called **partitioning**: processes that exchange a significant amount of data with the database server run on an application server on or near the physical location of the database server; processes that do not involve a significant exchange of data with the database server run on the client computer. This widely accepted **three-tier** model provides excellent performance, scalability, and security.



In the PeopleSoft Internet Architecture, all processing occurs on the server, so there is no concept of partitioning or process groups.

All Search API PeopleCode must be run on a server. It cannot be run on a client.

In order to partition application processing between the client and the application server, it is necessary to define units that, as a whole, run in one location or the other. We call these units **processing groups**.

In this section we'll look at all processing groups, paying special attention to those that can be run in either location at the discretion of the application developer or administrator. These are the processing groups that you must take into consideration when designing or upgrading an application for three-tier architecture.

Processing groups can encompass one or more PeopleCode events. There are three processing groups that can run on either the client or on the application server (Component Build, Component Save, and FieldChange). Each SaveEdit PeopleCode program, which is normally part of the Component save processing group, can be specified to run either on the client or the server. All other PeopleCode processing in PeopleTools 7 and later takes places exclusively on the client.

Component Build

The Component Build processing group includes all processing done after the key list of a panel is selected and before the user can interact with the panel. This includes building panel buffers and running many types of PeopleCode. The types of People Code that can run here, in the following order, are:

- RowSelect
- PreBuild
- RowInit
- PostBuild
- Activate
- PSControlInit (if an ActiveX control is present within the Component)



For more information on these events, see Component Build Processing in Update Modes and Component Build Processing in Add Modes.

The following events do not normally run, but they can run if a **InsertRow** or **DeleteRow** PeopleCode function or method is called elsewhere in the processing group.

- RowInsert
- RowDelete
- PSControlInit

By default, all Component Build processing happens on the application server.

FieldChange PeopleCode

The FieldChange event fires after:

1. The user changes a field
2. System validations and FieldEdit PeopleCode validations of the new data have completed successfully.

FieldChange PeopleCode is logically distinct from FieldEdit PeopleCode:

- FieldEdit is used to handle single-field data validation
- FieldChange handles other processing triggered by a change to the field, such as recalculation of other panel field values.

By default, all FieldChange processing happens on the client. For typical FieldChange processes, such as graying or hiding a field or performing a simple calculation on a panel, the client is the most efficient location.

However, if a FieldChange program is SQL-intensive, you can improve performance by changing the program's location to the application server (see Controlling Process Location). If you do this, make sure to check the FieldChange program for client-only PeopleCode.

FieldChange programs are associated with either a record field or a Component record field. Every FieldChange program can have its run location set independently to **server**, **client**, or **default**.

If *either* a record field FieldChange or a Component record field FieldChange program is set to run on the **server**, and the other is specified as **default**, then both will run on the server as part of a single transaction. If one is specified as server and the other as client, then *both* specifications will be honored.

The FieldChange program associated with a record field is always run before the Component record field FieldChange program, regardless of run location.

Field default processing for a field always follows both FieldChange programs. This is generally performed on the server as part of the same processing group, unless there is a Component record field FieldChange program that is specified to run on the client. That is, if there is a record field FieldChange program marked **server**, and a Component record field FieldChange program marked **client**, processing returns to the client for the Component record field FieldChange program, and field defaults are also done on the client.

Component Save

The Components Save processing group involves all processing after the user has saved the Component and client-based SaveEdit PeopleCode validations have succeeded. It includes SavePreChange, WorkFlow, and SavePostChange PeopleCode, as well as updates to the database. By default, all Component Save processing happens on the application server. It may include SaveEdit.

SaveEdit PeopleCode

The SaveEdit event fires whenever the user attempts to save the Component. SaveEdit PeopleCode is used to validate the consistency of data in Component fields. If the validation involves more than one Component field, you should use SaveEdit PeopleCode. If a validation involves only one panel field, you should use FieldEdit PeopleCode.

SaveEdit PeopleCode is logically distinct from the PeopleCode in the Component Save processing group because if an error occurs, the Component is redisplayed without the data being saved. An error in any of the Component Save processing PeopleCode will cause a runtime error, forcing the user to cancel the Component without saving changes.

By default, all SaveEdit PeopleCode happens on client. For typical SaveEdit processes, such as performing a simple calculation on the sum of two fields, the client is the most efficient location.

However, if a SaveEdit program is SQL-intensive, you can improve performance by changing the program's run location to the application server. You can specify either the individual SaveEdit PeopleCode program, or that all SaveEdit PeopleCode for the Component be run on the application server (see Controlling Process Location). If you do this, make sure to check all SaveEdit PeopleCode programs for client-only PeopleCode.

SaveEdit programs are associated with either a record field or a Component record. Every SaveEdit program can have its run location set independently to **server**, **client**, or **default**.

A SaveEdit program set to run on the client always runs before server SaveEdit programs, regardless of whether the program is associated with a record field or a Component record.

If the Component Save processing group is set to **save on client**, the SaveEdit program run location is ignored. In all other cases, the specification of client or server is honored.

The meaning of **default** for a SaveEdit program depends on the Component Save processing group run location setting. If Component Save processing is **server with edits**, **default** for SaveEdit is **server**. If Component Save processing is **server**, **default** for SaveEdit is client.

All client SaveEdit programs are performed before the application server is invoked. Then all server SaveEdit programs are performed, then other save processing. This split into client SaveEdit programs and server SaveEdit programs occurs even if running in 2 tier mode.

Other Processing Groups

The following table summarizes the processing groups that run on the client only:

Processing Group	Description
SearchInit/SearchSave	PeopleCode processes that occur immediately before and after the search dialog, prior to Component Build.
Field Edit	PeopleCode that fires after the user changes a field then moves the focus from the changed field.
RowInsert	PeopleCode processes that occur after the user attempts to insert a row.

RowDelete	PeopleCode processes that occur after the user attempts to delete a row.
PrePopup	Processes that occur immediately before a pop-up menu is displayed.
F4 and alt F4 prompts	Processes that occur when the user presses F4 or Alt+F4 in a field that validates against a prompt table or against the translate table.
ItemSelected	Processes that occur immediately after the user chooses a menu item from a standard or pop-up menu.

Note on External Function Location

Declared PeopleCode functions run in the location from which they are called. In other words, the location of the processing group where the function call is made determines the location of the called function.

For example, suppose that a FieldChange PeopleCode program calls an external function **RecalculateTotal**, which is stored in the FieldFormula event in some record field in the record definition FUNCLIB_HR.

- If the FieldChange PeopleCode runs on the client, then **RecalculateTotal** runs on the client.
- If the FieldChange PeopleCode runs on the application server, then **RecalculateTotal** runs on the application server.

The fact that **RecalculateTotal** is stored in a FieldFormula event has no bearing on where the function runs.

Default Processing Locations

The following table summarizes the locations where the processing groups can occur in three-tier mode. (Remember that in two-tier mode there is no application server, so all application processes run on the client.)

Processing Group	Location (Default in Bold)
Component Build	Client or Application Server
Component Save	Client or Application Server
FieldChange	Client or Application Server
SaveEdit	Client or Application Server
All others	Client

Controlling Process Location

An application or developer can change the default location of a Component Build, Component Save, or FieldChange processing group. The locations for Component Build and Component Change are set for the Component as a whole. The location for FieldChange is set for the individual PeopleCode program. The location for SaveEdit processing is set for the individual PeopleCode program, or as part of the Component save specification.


Each of the following procedures involves accessing and setting a radio button that determines the location of a specific processing group. The radio buttons are:

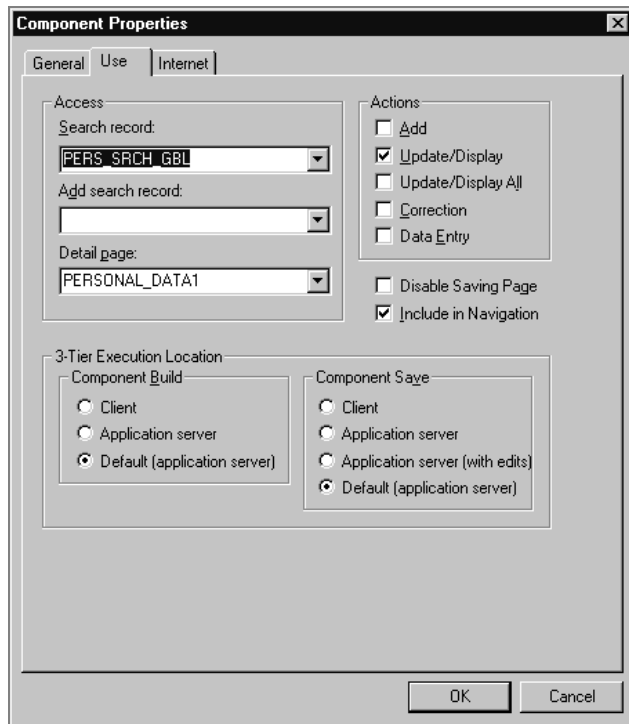
- **Default.** Specifies that this processing group runs in the default location. (See Default Processing Locations.)
- **Client.** Specifies that the processing group will run on the client workstation.
- **Application Server.** Specifies that the processing group will run on the application server, if the application is running in three-tier mode.
- **Application Server (with edits).** Specifies that the Component save processing group will run on the application server, including any SaveEdit PeopleCode programs (unless specifically marked to run on the client.)

To change the location of a processing group

1. Open the Component in Application Designer.

If the Component is in the current project you can open it from the Project View. Otherwise, use **File, Open**.

2. Click the **Properties** button  on the toolbar, or press ALT+ENTER, display the Component Properties dialog, then click the Use tab.



Component Properties Dialog

3. To set a location for the Component Build processing group, choose a radio button under Component Build. To set a location for the Component Save processing group, choose a button under Component Save.
4. Optionally document the change in the Comments field on the General dialog page, then accept the dialog.

To change the location of a FieldChange or SaveEdit processing group

1. From Application Designer, open the PeopleCode Editor for the record field that you want to change.

There are several ways to do this, but one convenient way is to right-click on the record field in the project workspace, then choose View PeopleCode.

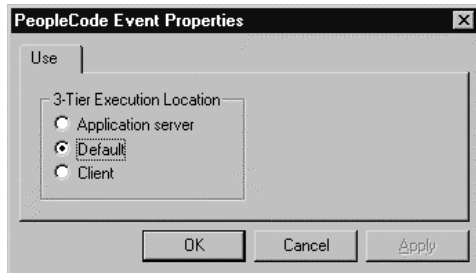
2. Access the FieldChange or SaveEdit Event for the record field that you want to change.

To do this, choose the FieldChange or SaveEdit event from the right dropdown list at the top of the PeopleCode Editor.

3. Right-click in the PeopleCode Editor, then choose Object Properties from the context menu.

The PeopleCode Event Properties dialog appears.

4. Click the Use tab.



Use Page of the PeopleCode Event Properties Dialog

5. Choose a radio button to set the 3-Tier Execution Location of the FieldChange or SaveEdit program, then accept the dialog.

Index

@

@ operator 5-22

A

access classes 9-1

accessing

- application message PeopleCode 2-17
- component PeopleCode 2-11
- component record field PeopleCode 2-8
- component record PeopleCode 2-9
- field object 9-12
- menu item PeopleCode 2-14
- page field PeopleCode 2-13
- page PeopleCode 2-12
- panel group record field PeopleCode 2-9
- PeopleCode Debugger 12-1
- PeopleCode programs 10-3
- record field PeopleCode 2-4
- record object 9-10
- row object 9-9
- rowset object 9-6
- SQL editor 4-2

accessing external functions 3-10

accessing level 0 9-5

Activate event 10-33

ActiveX controls

- data types for 13-16
- events 13-12
- implementing 13-11
- initializing data 13-16
- properties and methods 13-15

ambiguity in field references 8-7

application messages

- accessing PeopleCode 2-17
- debugging 12-12

application server

- DLL functions 11-5

assigning

- objects 6-5

attachment functions

- subrecord 7-37
- using 7-37
- work record 7-37

B

backing up PeopleCode programs 2-20

Boolean constants 5-13

Boolean operators 5-24

buffer fields

- referring to 8-16

Business Interlink

- generating PeopleCode 3-14

C

CD-ROM

- ordering iii

class

- definition of 6-1

client-only PeopleCode 11-3

comments 5-4

comparing rowsets to scrolls 8-3

comparison operators 5-24

compile all PeopleCode programs 12-12

component buffer

- contents 8-3
- contextual buffer field references 8-7
- contextual row references 8-6
- current context 8-4
- processing order 8-5
- record fields 8-3
- referring to fields 8-16
- resolving ambiguity 8-7
- scroll path syntax 8-9
- structure 8-1

component build processing

- add mode 10-19
- update mode 10-16

Component Interface

- generating PeopleCode 3-15

component processor 10-1

- behavior 10-7
- events outside flow 10-1

component processor events 10-33

- Activate event 10-33
- event order 10-4
- FieldChange event 10-34
- FieldDefault event 10-34
- FieldEdit event 10-35
- FieldFormula event 10-35
- ItemSelected event 10-36
- PostBuild event 10-36

- PreBuild event 10-36
- PrePopup event 10-37
- PSControlInit event 10-37
- PSLostFocus event 10-38
- RowDelete event 10-38
- RowInit event 10-39
- RowInsert event 10-40
- RowSelect event 10-41
- SaveEdit events 10-42
- SavePostChange event 10-43
- SavePreChange events 10-43
- SearchInit event 10-44
- SearchSave event 10-44
- Workflow event 10-45
- concatenating strings 5-22
- conditional statements 5-6
- content reference field 7-11
- contextual buffer field references 8-7
- contextual references
 - data buffer access classes 9-22
 - resolving ambiguity 8-7
- contextual row references 8-6
- control statements 5-6
- controlling process location 13-30
- copying PeopleCode programs 2-21
- cross reference reports 12-30
- current context of component buffer 8-4
- CurrentRowNumber
 - using 8-17

D

- data buffer classes
 - current context 9-22
 - example 9-3
 - traversing hierarchy example 9-13
- data buffer model 9-1
- data types 5-1
 - conventional 5-1
 - object-based 5-2
- date and time operators 5-22
- debugging
 - subscription PeopleCode 12-12
- default processing
 - field-level 10-10
 - panel group level 10-11
- deferred processing mode 10-30
- DLL functions
 - application server 11-5
- dot notation
 - syntax 8-9
- dynamic tree controls
 - implementing 13-1

E

- errors and warnings
 - restrictions 7-51
- Evaluate
 - more examples 5-7
- Exec function
 - OLE automation 13-25
- expressions 5-12

F

- field modification processing 10-20
- FieldChange event 10-34
 - processing groups 13-26
- FieldDefault event 10-34
- FieldEdit event 10-35
- FieldFormula event 10-35
- fields
 - converting strings to field references 5-22
 - naming 5-20
 - reference syntax 5-18
- Find In tool 12-27
- fonts
 - PeopleCode editor 3-12
- formatting
 - PeopleCode 3-10
- function execution location 13-29
- functions
 - calling 5-11
 - declaring 5-11
 - parameters 5-11
 - return values 5-12

G

- GenerateTree function
 - building HTML tree page 7-20
 - customizing PeopleCode 7-25
 - end-user actions 7-24
 - FieldChange example 7-31
 - HTML tree 7-19
 - PostBuild example 7-27
 - rowset records 7-21
 - using 7-19

H

- HTML tree 7-19

I

- implementing
 - ActiveX controls 13-11

- modal transfers 7-8
- tree controls 13-1
- inserting using PeopleCode 7-18
- ItemSelected
 - event 10-36
 - processing 10-27

L

- language constructs 5-6
- legal names
 - record fields 5-19
- logical operators 5-24
- loops
 - For 5-8
 - Repeat 5-9
 - scroll levels 8-18
 - While 5-9

M

- math operators 5-21
- Metastrings 5-17
- modal transfers
 - considerations 7-9
 - implementing 7-8
- multiple occurs levels 10-45
- multiple scroll levels
 - effects on PeopleCode execution 10-45

N

- name
 - references 5-19
 - reserved 5-20
- numeric constants 5-13

O

- object
 - assignment 6-5
 - definition of 6-1
 - methods 6-3
 - properties 6-3
 - working with 6-2
- object-based datatypes 5-2
- occurs levels
 - multiple 10-45
- OLE automation
 - data types 13-24
 - defined 13-23
 - sharing object instances 13-24
 - WinExec 13-25
- operators

- @ 5-22
- Boolean 5-24
- date and time 5-22
- math 5-21
- relational 5-24
- string concatenation 5-22

P

- passing objects 6-6
- PeopleBooks
 - CD-ROM, ordering iii
 - printed, ordering iii
- PeopleCode
 - accessing 2-2
 - accessing external functions 3-10
 - client-only 11-3
 - compiling all programs 12-12
 - component processor 10-1
 - current context 8-4
 - finding strings in 12-27
 - how programs are triggered 10-2
 - inserting 7-18
 - PeopleSoft Internet Architecture applications 11-1
 - using drag-and-drop 3-13
- PeopleCode Debugger
 - accessing 12-1
 - DoModal consideration 12-8
 - environment 12-11
 - features 12-2
 - log file interpretation 12-15
 - log options 12-13
 - messages 12-12
 - options 12-8
 - sample trace file 12-20
 - single debugger considerations 12-4
 - subscription PeopleCode 12-12
 - variables panes 12-4
- PeopleCode editor
 - choosing a font 3-12
 - context-sensitive help 3-12
 - definition list 3-2
 - drag-and-drop 3-10
 - event list 3-6
 - find and replace 3-8
 - formatting statements 3-10
 - functions 3-8
 - generating Business Interlink template 3-14
 - generating Component Interface template 3-15
 - generating definition references 3-13
 - replace dialog box 3-8
 - using 3-7
 - validating syntax 3-9
 - window 3-1
- PeopleCode programs
 - automatic backup 2-20

- copying 2-21
 - saving 2-20
- PeopleSoft Internet Architecture
 - deferred processing mode 10-30
 - processing considerations 10-30
 - search page 11-2
 - using GetHTMLText 7-15
 - using GetJavaScriptURL 7-17
 - using HTML area 7-14
 - using PeopleCode 7-12
 - using Style property 7-12
 - writing PeopleCode applications 11-1
- pop-up menus
 - processing 10-26
- PostBuild event 10-36
- PreBuild event 10-36
- PrePopup event 10-37
- processing
 - deferred mode 10-30
- processing groups 13-25
 - controlling location 13-30
 - FieldChange 13-26
 - other 13-28
 - Panel Group Build 13-26
 - Panel Group Save 13-27
 - SaveEdit 13-28
- processing sequences
 - component build in add mode 10-19
 - component build in update mode 10-16
 - default processing 10-10
 - field modification 10-20
 - ItemSelected 10-27
 - pop-up menu 10-26
 - PSLostFocus 10-27
 - pushbuttons 10-26
 - Row Delete 10-24
 - Row Insert 10-23
 - Row Select 10-18
 - Save 10-28
 - search in add mode 10-14
 - search in update mode 10-12
- PSControlInit event 10-37
- PSLostFocus event 10-38
 - processing 10-27
- PSOPTIONS table
 - considerations 11-7
- pushbuttons
 - processing 10-26

R

- record field
 - legal names 5-19
- relational operators 5-24
- Remote Call 7-53
 - and Process Scheduler 7-57
 - components of 7-55

- PeopleCode API 7-55
 - programming guidelines 7-58
 - remote program API 7-56
- reserved words 5-20
- restrictions on function and method use 7-1
 - CallAppEngine 7-6
 - Component Interface 7-6
 - data buffer fields 7-4
 - DoSave 7-4
 - Errors and warnings 7-4
 - GetControl 7-7
 - GetGrid 7-7
 - GetPanel 7-7
 - Publish method 7-8
 - record object 7-5
 - ReturnToServer 7-7
 - SQL object 7-5
 - think-time 7-1
 - WinMessage and MessgeBox 7-2
- Row Delete processing 10-24
- Row Insert processing 10-23
- Row Select processing 10-18
- RowDelete event 10-38
 - considerations 10-39
- RowInit event 10-39
 - exception 10-40
- RowInsert event 10-40
- rows
 - referring to 8-15
- RowSelect event 10-41
- rowset
 - comparing to scroll 8-3
 - example 9-17
 - instantiating using non-panel buffer data 9-23
 - standalone rowsets 7-43

S

- Save processing 10-28
- SaveEdit event 10-42
 - processing groups 13-28
- SavePostChange event 10-43
- SavePreChange event 10-43
- saving PeopleCode programs 2-20
- scroll levels
 - looping through 8-18
 - multiple 10-45
 - referring to 8-13
- scroll path syntax 8-9
- ScrollSelect functions
 - behaviors of 13-22
 - specifying select record 13-21
 - SQL string 13-20
 - syntax of 13-19
 - target scroll in 13-19
 - Turbo 13-21
 - using 13-17

- search PeopleCode 12-27
- search processing
 - add mode 10-14
 - update mode 10-12
- SearchInit event 10-44
- SearchSave event 10-44
- Select method
 - child rowsets 7-41
 - select record 7-42
 - syntax 7-40
 - using 7-39
- SQL
 - Metastrings 5-17
- SQL definitions 4-2
- SQL editor
 - accessing 4-2
 - using 4-1, 4-6
- standalone rowsets 7-43
 - adding child rowsets 7-45
 - CopyTo method 7-44
 - Fill method 7-44
 - reading from files 7-49
 - writing to files 7-46
- starting
 - third-party applications from PeopleCode 13-25
- statements 5-4
- string concatenation 5-22
- string constants 5-13
- subroutines 5-6
- syntax
 - scroll path 8-9
 - validating with PeopleCode editor 3-9
- system edits 10-21

T

- think-time functions 7-1
- three-tier architecture
 - processing groups 13-25

- time and date operators 5-22
- tree controls
 - controlling root node 13-10
 - implementing 13-1
 - multiple-table 13-2
 - single-table 13-6
- Turbo ScrollSelect 13-21
- Tuxedo *See Remote Call*

U

- using
 - attachment functions 7-37
 - content reference field 7-11
 - GenerateTree function 7-19
 - insert 7-18
 - select method 7-39
 - standalone rowsets 7-43

V

- validating syntax
 - PeopleCode editor 3-9
- variables
 - passing to functions 5-17
 - system 5-17
 - user-defined 5-16

W

- warnings and errors
 - restrictions 7-51
- WinExec function
 - OLE automation 13-25
- Workflow event 10-45

