



ALBPM Studio Help

Version: 6.0

Contents

Getting Started.....	8
What is Studio?.....	8
Documentation Roadmap.....	9
What's New in ALBPM 6.0 Studio.....	9
ALBPM Examples.....	10
User Interface Overview.....	11
What is Eclipse?.....	11
Profiles.....	11
ALBPM Perspective.....	12
Views.....	12
Resources.....	18
Editors.....	18
Common UI Tasks.....	19
Studio Preferences.....	20
Setting Studio Preferences.....	20
Setting Project Preferences.....	20
Setting Engine Preferences.....	20
Setting Eclipse Preferences.....	20
Migrating Project Code.....	22
Import 5.7 project into Studio 6.0.....	22
Fix External Resources.....	22
Changes in Standard Components.....	23
Changes in Dashboards.....	24
Changes in Implicit Java Classpath.....	24
Changes in PBL.....	25
Changes in Transformations.....	25
Process Designer.....	26
Projects.....	26
What is a Project?.....	26
Project Tasks.....	26
Project Properties Reference.....	28
Processes.....	29
What is a Process?.....	29
Process Instances.....	29
Process Tasks.....	31
Process Property Reference.....	33
Process Web Service Reference.....	33
Activities.....	34
What Is an Activity?.....	34
Activity Types.....	34
Adding an Activity.....	65

Adding a Global Activity.....	65
Editing Activity Properties.....	66
General Activity Property Reference.....	66
Activity Groups.....	67
Tasks.....	71
What is a Task?.....	71
Implementation Types.....	72
Transitions.....	79
What Is a Transition?.....	79
Transition Types.....	80
Adding a Transition.....	86
Variables.....	88
What is a Variable?.....	88
Variable Types.....	89
Creating Project and Instance Variables.....	99
Audit Events.....	99
Audit Events Overview.....	99
Configuring Audit Event Generation.....	101
Organization.....	103
Organization Overview.....	103
Importing an Organization.....	104
Exporting an Organization.....	104
Organizational Units.....	105
Roles.....	106
Groups.....	107
Participants.....	107
Holiday Rules.....	108
Calendar Rules.....	109
Business Parameters.....	109
Deleting an Organizational Element.....	110
Editing an Organizational Element.....	111
Dynamic Business Rules.....	112
When to use Dynamic Business Rules.....	112
Using Dynamic Business Rules.....	112
Defining a Business Rule.....	114
Letting Participants Edit Business Rules.....	115
Simulations.....	115
What is a Simulation?.....	116
Process Simulation Model.....	116
Project Simulation Models.....	118
Simulation View.....	119
Tasks.....	120
Sharing Files with Version Control.....	121
What is Version Control?.....	121
Sharing Files with Version Control.....	121

Extracting Files from Version Control.....	121
Localizing a Project.....	122
Localization Overview.....	122
Adding a Language to a Project.....	122
Localizing a Process Name.....	123
Localizing an Activity within a Process.....	123
Correlations Overview.....	123
Correlation Sets.....	124
Defining a Correlation Set.....	124
Creating a Correlation Set.....	124
Adding Correlation Properties.....	125
Correlation Property Data Types.....	125
Using BPEL.....	126
ALBPM BPEL Support.....	126
Enabling BPEL Process Design and Editing.....	126
BPEL Processes Execution.....	127
Creating a New BPEL Process.....	127
Importing a BPEL Process.....	128
Opening a BPEL Process.....	129
Using the BPEL Process Editor.....	129
BPEL Example Project.....	130
Exception Handling.....	131
Exception Handling in ALBPM.....	131
System Exceptions.....	131
Business Exceptions.....	132
Code-level Exception Handling.....	132
Process-level Exception Handling.....	133
Typical Exception Handling Flow.....	133
The Project Catalog.....	135
BPM Objects.....	136
What is a BPM Object?.....	136
What is an Attribute?.....	138
What is a Presentation?.....	139
Creating a BPM Object.....	139
Defining an Attribute.....	140
Creating a Presentation.....	143
BPM Object Catalog.....	144
What is the BPM Object Catalog?.....	144
Modules.....	145
Components.....	145
External Components.....	146
.NET Component.....	146
AquaLogic Service Bus.....	147
COM Component.....	149

CORBA.....	153
JNDI.....	162
SAP.....	164
SQL Database.....	167
Table (SQL Query).....	170
Web Service.....	171
XML Schema.....	171
Business Activity Monitoring (BAM).....	175
What is BAM?.....	175
Enabling and Configuring BAM in Studio.....	175
BAM Database.....	175
Using Variables in BAM.....	176
Creating a Predefined BAM Dashboard.....	177
Viewing BAM Dashboards in Studio.....	177
BAM Database Reference.....	177
External Resources.....	183
What is an External Resource?.....	183
External Resources and the Catalog.....	183
Creating an External Resource.....	183
External Resource Reference.....	183
SQL Database.....	183
SAP Service.....	189
Web Service.....	190
UDDI Registry.....	191
HTTP Server Configuration.....	191
Microsoft .NET Service.....	192
Mail Outgoing Service.....	192
J2EE Application Server.....	193
Enterprise JavaBean (EJB).....	194
Java Class Library.....	194
AquaLogic Service Bus.....	195
Mail Incoming Service.....	196
Microsoft COM Service.....	196
JMX Service.....	197
CORBA Service.....	197
JMS Messaging Service.....	198
JNDI Directory Server.....	198
Java Process Definition (JPD).....	200
Unit Tests.....	201
What is a Unit Test?.....	201
Creating a Unit Test.....	201
Running a Unit Test.....	201
Test Results View.....	202
Process Execution Engine.....	203

What is the Process Execution Engine?.....	203
Starting the Process Execution Engine.....	204
Setting Engine Preferences.....	204
Process Business Language (PBL).....	205
PBL Overview.....	205
PBL Methods.....	205
Comments.....	205
Expressions.....	206
Programming Styles.....	207
Data Types Overview.....	208
Type Conversion.....	209
Numbers Overview.....	209
Integers.....	210
Reals.....	210
Decimals.....	211
Decimal Arithmetic.....	211
Real and Decimal Numbers.....	213
Enumerations.....	213
Number Functions Reference.....	214
String Overview.....	219
String Functions.....	220
String Attributes.....	226
Time and Interval Overview.....	226
Time Attributes.....	229
Time Functions.....	235
Interval Attributes.....	246
Interval Functions.....	250
Array Overview.....	254
Indexed Arrays.....	255
Associative Arrays.....	256
Manipulating Arrays.....	257
Array Functions.....	258
Array Attributes.....	261
Array Procedures.....	262
Mapping Array Members.....	263
Variables Overview.....	263
Instance Variables.....	264
Project Variables.....	266
Argument Variables.....	267
Local Variables.....	268
Predefined Variables.....	269
Initializing Variables.....	274
Operator Types Overview.....	275
Arithmetic Operators.....	275
Relational Operators.....	276

Logical Operators.....	276
Statements Overview.....	276
Statement Timeout.....	277
Input Statement.....	278
Compound Statement.....	284
Simple Conditional Statements (if-then-else).....	285
Case Statement.....	287
Bounded Loops.....	287
Unbounded Loops.....	289
Exit Statement.....	289
Labeled Statement.....	290
Throw Statement.....	290
Logging Statement.....	291
Regular Expression Overview.....	291
Regular Expressions in Functions.....	292
Search and Replace.....	293
Modifiers.....	294
Metacharacters and Character Sets.....	294
Matching Repetitions.....	295
Anchors.....	297
Alternations.....	298
Grouping.....	298
Extraction.....	299
Backreferencing.....	302
Objects Overview.....	302
Creating an Object.....	303
Duplicating an Object.....	303
Calling an Object.....	304
Current and Default Instances.....	304
Object Cleanup.....	304
Code Conventions Overview.....	305
Improving Code Readability.....	305
General Naming Conventions.....	310
Specific Naming Conventions.....	312
Creating Statements.....	313
Code Layout and Comments.....	316
Embedded SQL Overview.....	319
SQL Operators.....	319
SQL Keywords.....	320
INSERT Statement.....	321
UPDATE Statement.....	322
DELETE Statement.....	322
SELECT Statement.....	323
Stored Procedures.....	324

Getting Started

The topics in this section provide general information about AquaLogic BPM Studio including tutorials, an overview of the Studio user interface, and reference information about Studio preferences.

What is Studio?

AquaLogic BPM Studio is a desktop application that allows you to model and implement business processes. Studio creates a common interface for business analysts and developers by providing common views into the same process model.

AquaLogic BPM Studio allows you to integrate, design, test, and evolve your business activities using a process driven method to coordinate and manage internal and external business services.

Process Design

ALBPM Studio provides a complete process modeling environment. Within an ALBPM project, you can create different process models that correspond to different areas of your business. This allows you to create models that account for all of the people, systems and organizations within your business. Each process contains activities, transitions, and roles that define the tasks and workflow.

In addition to the activities, transitions, and roles of your process, you can also create project variables that can be used to define Key Performance Indicators (KPI) for your business process.

Studio also allows you to comprehensively document how your process functions. Based on this documentation, developers can implement the process according to the specifications defined by the business analysts who created the process. This allows your process design to function as a contract between the process design and implementation stages.

After you have created a process model, ALBPM Studio allows you to run process simulations that mimic how your process behaves in production.

Process Development

ALBPM Studio also provides a complete process development environment that allows you to go from the process modeling stage to a functioning production environment.

ALBPM Studio allows you to define the business rules and logic that ties your business process together.

For processes requiring integration with back-end applications, AquaLogic BPM processes communicate with these underlying application services through components. Components are also cataloged for use with the AquaLogic BPM adaptors framework that interfaces with application APIs. These APIs can be implemented in various technologies, including Java, EJB, COM, CORBA/IDL, JDBC/ODBC, XML, JMS and other middleware.

Technology adaptors connect to this standard technology instead of a particular application. This allows the component Catalog to connect to any object. It has the ability to introspect any object technology and read its methods and properties to create a 'wrapper' or 'proxies' that directly interfaces with it

ALBPM Studio also provides an environment for testing your business processes before they are deployed in a production environment.

Profiles

To ensure that the appropriate level of functionality is presented to each type of users, AquaLogic BPM Studio provides different profiles based on each user type. See [Profiles](#) on page 11 for more information.

Eclipse Framework

AquaLogic BPM Studio is built on the Eclipse IDE framework. For information on how ALBPM Studio uses Eclipse, see [What is Eclipse?](#) on page 11.

Documentation Roadmap

The AquaLogic BPM Documentation Set provides comprehensive information for installing, configuring, and using each component of the ALBPM Product Suite.

The current version of the AquaLogic BPM documentation set is available at <http://edocs.bea.com>.

What's New in ALBPM 6.0 Studio

This topic provides an overview of the main new features, improvements and changes in this release of AquaLogic BPM Studio.

Standards Support

- Process models in ALBPM are now compliant with the XPD 2.0 standard.
- Support for BPEL 1.1. You can import BPEL 1.1 models into an ALBPM Project, and new models can be designed within ALBPM Studio. The Process Execution Engine is now capable of executing BPEL 1.1 natively.
- ALBPM Studio application is now built on top of the Eclipse platform.

Studio IDE

- Studio now includes a software agent for automatic problem reporting and feedback. In case of unexpected errors in Studio, an automatic report will be sent to BEA for analysis. Studio will prompt you for approval before enabling this feature. We also encourage you to send us feedback using the **Help ► Feedback...** menu option.
- When you first start ALBPM Studio, you have to select one of the available profiles according your skill set: Business Analyst, Business Architect, Developer. ALBPM Studio presents a different subset of features depending on the selected profile. This keeps the user interface uncluttered, hiding what you don't need. All available features are visible under the Developer profile. The on-line documentation in Studio is also filtered depending on the active profile. To switch profiles go to [Help ► Welcome](#).
- This new release introduces the concept of Project Variables, replacing the External and Business Variables of previous versions. Project Variables are functionally equivalent to the old External Variables but are simpler to use: they are available to all processes in the project, with no need to "promote" them from External to Instance. When the new property **Business indicator** is enabled, Project Variables behave as the old Business Variables (they are used for BAM reporting).
- ALBPM project directories do not use the .fpr extension anymore.
- The Organization data and Simulation definitions are now accessed as nodes in the project tree.
- On previous version of Studio the Business Parameters of the project were accessible from the Variables panel on right. Now you access them from the Business Parameters node under the Organization node of the project tree.
- Integration with Version Control System feature (VCS) was re-implemented to leverage the Eclipse platform. This paves the way for supporting virtually any Source Control systems compatible with Eclipse.
- Each resource that is independently stored as part of an ALBPM Project is modified using an "Editor" tabbed panel, and you must explicitly save your changes on each resource with File > Save . For example, on earlier versions of Studio you add or modify a Participant using a separate dialog window. Now a special Participants editor opens in a new tab of the edition area. This makes it easier to work with Version Control systems, as each resource is managed and saved independently.
- Some editors may open nested editors (accessible via smaller tabs at the bottom of the editor). For example, the editor for Process models uses independent sub-tabs for the process diagram and for each opened process method.

Process Designer

- You can now open several projects at the same time. Before opening a project, you first need to add it to your Studio workspace.
- Incremental compilation: There is no need for Publish&Deploy anymore. Once you start Studio's Process Execution Engine, the project is running. While it is running, the Execution Engine immediately applies changes you make to your project design and code.
- A new type of Interactive activity: Decision activities. This type of activity allows the end user to decide the next path a process instance will take (one of the possible outgoing transitions), based on the value of certain instance variables. The Process Execution Engine keeps track of those decisions over time and presents the end user with recommendations on what decision to take based on past experience.
- Business Rules: ALBPM Studio now provides a way of defining business rules using a graphical rules editor, without requiring any coding. After the project is deployed, authorized end users can also modify these rules on-the-fly, while the processes are executing. They can do so right from the ALBPM WorkSpace UI.
- Round-trip Simulation: You can now create Simulation models from the actual execution of the processes during a given period of time. This makes it easier to create realistic Simulation models.

User Interface

- ALBPM WorkSpace has been re-designed and re-implemented from the ground up. It is based on a modern modular architecture which makes it easier to customize and integrate naturally with AquaLogic UI and WebLogic Portal. The old WorkSpace is still provided for backward compatibility but may be removed in future versions.
- Dashboards provide better quality graphics and end user interaction (i.e. rotation, detaching of pie sections).

Integration

- Native integration with ALSB. You can now easily consume ALSB services from ALBPM and also register a business process in ALSB.
- Web Services in ALBPM now include support for WS-Security, Document-Literal style and WS-I compliance.
- ALBPM Studio now includes JDBC drivers for the most popular DBMS. This means you can integrate with Oracle, DB2 and Microsoft SQL Server right out of the box.
- PAPI has deprecated several methods in favor of new ones which follow a new naming convention. PAPI methods which were deprecated in ALBPM 5.7 have been deleted from the API.
- PAPI WebService 1.0 has been deprecated in favor of the new PAPI WebService 2.0. PAPI-WS 1.0 is accessible through ALBPM WorkSpace while PAPI-WS 2.0 is accessible through its own new Web Application (papiws). This new version is functionally equivalent to the native Java PAPI, and adheres to the WS-Security specification using the UserNameToken Profile implementation as well as HTTP Basic Authentication.

ALBPM Examples

AquaLogic BPM Studio contains several example projects. These are located in `BEA_HOME/albpm6.0/studio/samples`.

To view the sample projects, you can import them as an exported ALBPM Project. See [Importing a Project](#) on page 27.

The following table provides a brief description of each example:

Example	Description
BPELProject1.exp	Demonstrates how to use BPEL.
BPMUnitTestExample.exp	Demonstrates how to build automated tests for BPM Objects and Processes.
ExpenseManagement.exp	Provides an example of a basic business process used by the ALBPM Studio Basic Tutorial.

Example	Description
CorrelationCase04.exp	Provides an end-to-end example that demonstrates many of the features of ALBPM.
DashboardDrilldownInstanceData.exp	Demonstrates how to use Dashboards.
HROnboarding.exp	Provides a end-to-end example project that demonstrates many of the features of ALBPM.
OrderFullfillment.exp	Provides an end-to-end example project that demonstrates many of the features of ALBPM.

User Interface Overview

This section provides topics describing how to use Studio's Eclipse-based User Interface.

What is Eclipse?

AquaLogic BPM Studio is built on Eclipse. Eclipse is an open-source, industry-standard integrated development environment (IDE) used for creating software applications.

In addition to providing an IDE, Eclipse can also be used as a platform to develop custom applications. For more information on Eclipse see <http://eclipse.org>.

The following components of the Eclipse IDE are commonly used in AquaLogic BPM Studio:

Eclipse Component	Description
Workbench	Contains the basic desktop environment for Eclipse.
Perspective	Defines the layout of related editors and views.
Resources	Contains the files, folders, and projects.
Editors	Allow you to create and edit resources.
Views	Allow you to navigate, edit, and view information about resources.
Toolbars	Provide quick access to product features.

Profiles

AquaLogic BPM Studio provides three separate profiles. Each profile contains different levels of functionality which is targeted towards a specific user type.

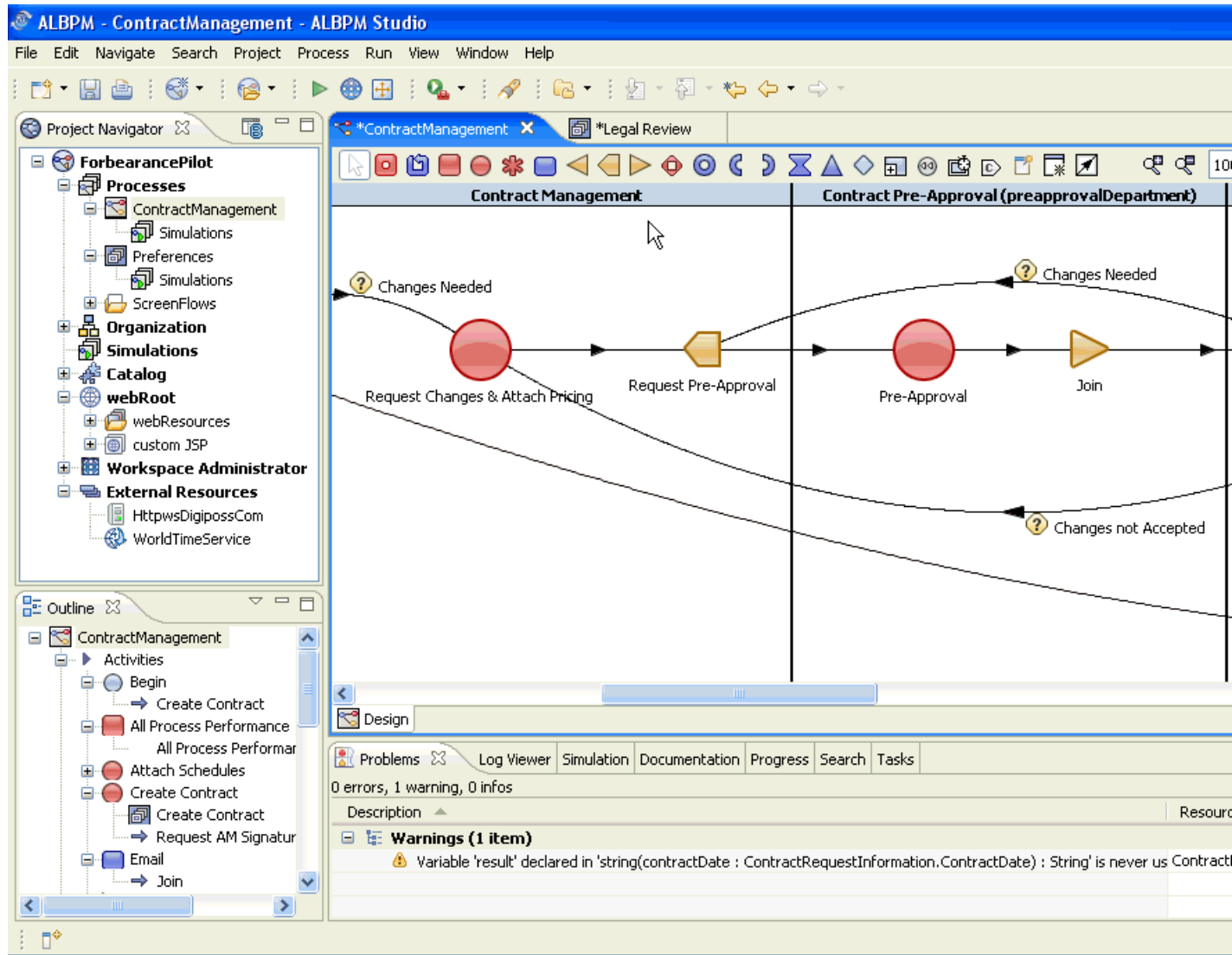
Profile	Description
Business Analyst	Provides access to process modeling functionality, but does not contain any coding elements.
Business Architect	Provides access to process modeling functionality as well as modeling and service mapping, module definitions, and access to the BPM Object Catalog. Some basic coding functionality is supported.
Developer	Provides access to process modeling and all development functionality.

Each AquaLogic BPM profile uses the same perspective so the layout of views and editors is identical. However, the contents of views and editors may be different.

ALBPM Perspective

The AquaLogic BPM Perspective provides a set of editors, views, and toolbars customized for modeling and implementing business processes.

The following image shows how a typical AquaLogic BPM Project appears within the Studio application.



Views

The topics in this section provide a general introduction to Views and provide information about ALBPM custom views.

What is a View?

Views provide multiple ways of navigating resources within your Project.

Eclipse Standard Views

The Eclipse IDE provides a standard set of views that are available in all perspectives. For general information on using Views and for specific information about the default Eclipse views see the *Workbench User Guide*.

ALBPM Custom Views

The AquaLogic BPM Perspective provides the following custom views:

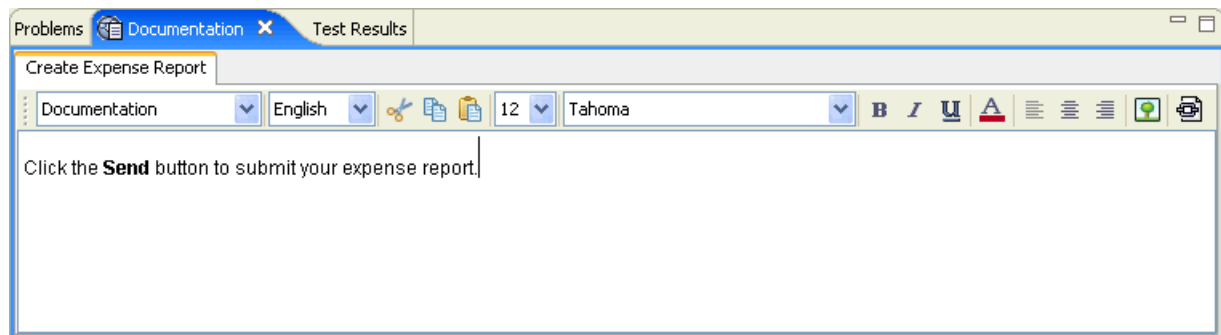
View	Description
Documentation	Allows you to create and edit documentation for a Process and its Activities.
Log Viewer	Displays the error log for the embedded Process Execution Engine.
Problems	Displays information about errors and warnings that occur within a Project.
Project Navigator	Provides a hierarchical view of resources within a Project.
Properties	Displays the properties of a BPM Object Presentation.
Simulation	Allows you to run and view Process simulations.
Variables	Displays a list of variables grouped by type.
Test Results	Displays the results of CUnit and PUnit tests.

 **Note:** Custom Views are only available from the ALBPM Perspective. They are not visible within other perspectives.

Documentation View

The Documentation View allows you to view, create and edit documentation for your Process and Activities.

This view provides a graphical text editor that allows you to perform basic text formatting, add images, and create hyperlinks.



Documentation Audience





ALBPM Studio allows you to create documentation for two difference audiences:

Audience	Description
Documentation	Provides information about a Project to end users. Content provided in this option appears in WorkSpace.
Use Case Documentation	Provides internal information about a Project that is useful to process architects and developers.

You can switch between audiences using the drop-down menu in the Documentation View toolbar. Both types of documentation appear in a generated Project Report.

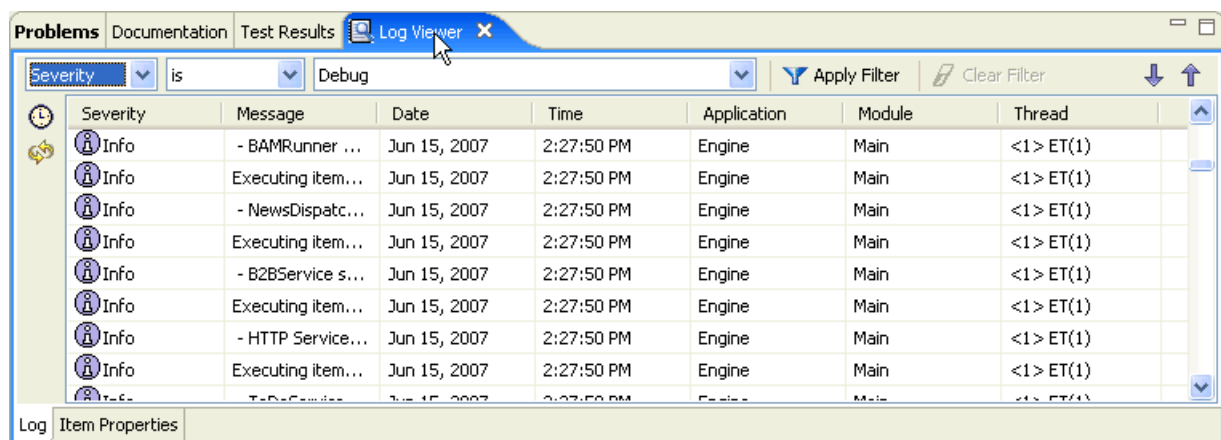
Documentation View Toolbar

The following table outlines the options available in the Documentation View toolbar:

Toolbar Element	Description
Audience Drop-down Menu	Selects the audience for the current content.
Language Drop-Down Menu	Specifies the language for the current audience.
	Cuts the current selection and copies it to the clipboard.
	Copies the current selection to the clipboard.
	Pastes the current selection at the cursor location.
Font Size	Defines the font size for the current selection.
Font Type	Defines the font type for the current selection. Available font types depend on the font styles installed on your system.
	Allows you to insert an image within the documentation.

Log Viewer View

The Log Viewer View provides logging information for the Embedded Process Execution Engine.



The Log Viewer only displays messages for the Project you use to start the Embedded Process Execution Engine. To view log messages for another Project, you must stop the engine and restart it using a different Project. See [Starting the Process Execution Engine](#) on page 204 .

Logging Information

The Log Viewer displays the following information for each log message:

Column	Description
Severity	Indicates the kind of message (FATAL, SEVERE, WARNING, INFO, DEBUG).
Message	Contains the message that the Engine sends to the log.
Date	The time that the message was logged.
Time	The date the message was logged.

Column	Description
Application	Application that sent the message. All BPM system applications can send log messages to the log files.
Module	Module that sent the message.
Thread	Thread that sent the message.

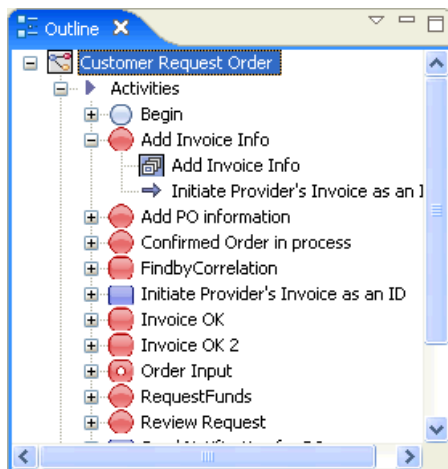
Item Properties Tab

The Item Properties displays detailed information about specific log entries.

Outline View

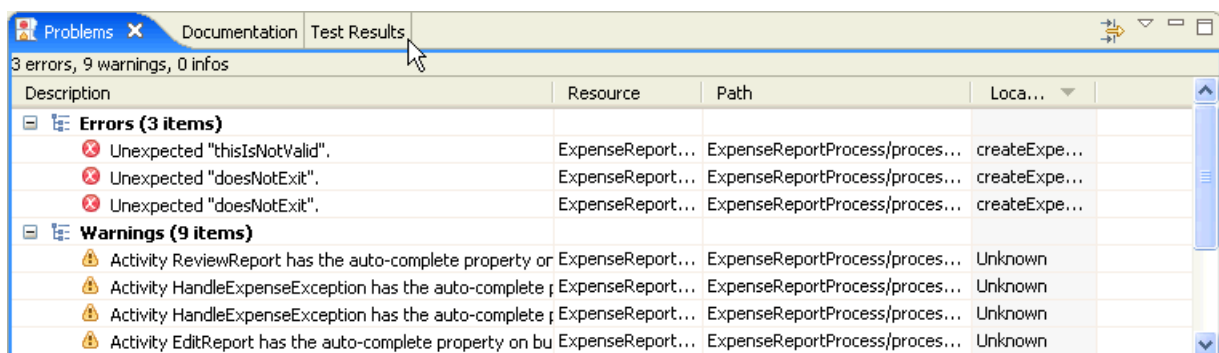
The Outline View displays an outline structure of a file that is currently open in an editor. The Outline View is a standard Eclipse View, but is frequently used within ALBPM Studio.

The contents of the Outline View depend on the contents of the currently highlighted editor. The following image shows an example of the Outline View of an ALBPM Process.



Problems View

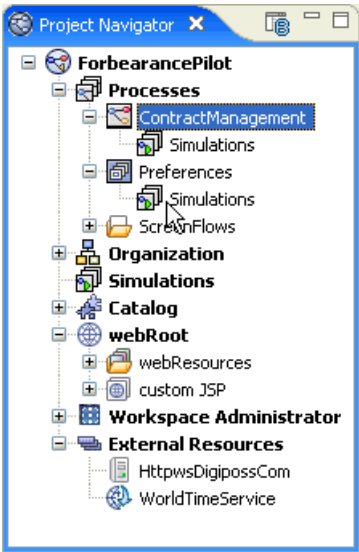
The Problems View displays information about errors and warnings that occur within the Project.



Project Navigator View

The Project Navigator View displays all of the Projects and project resources within your current workspace.

The following figure shows a typical ALBM Studio Project:

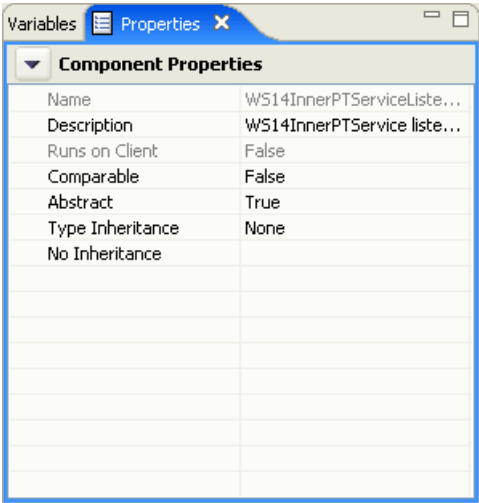


Setting ALBPM Studio Preferences

The  icon in the Project Navigator toolbar opens the Studio Preferences window. For more information on setting these preferences, see [Studio Preferences](#) on page 20.

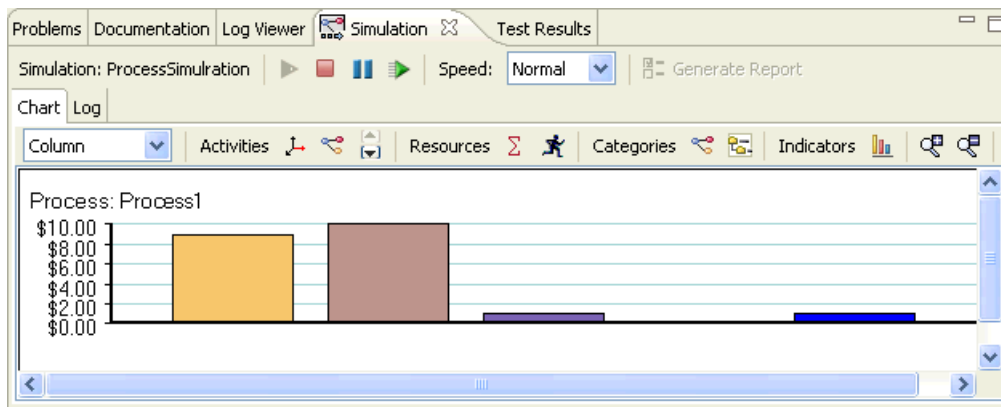
Properties View

The Properties View displays the properties of a BPM Object Presentation. This view is different from the standard Eclipse Properties view.



Simulation View

The Simulation View allows you to run and view Process simulations.



Variables View

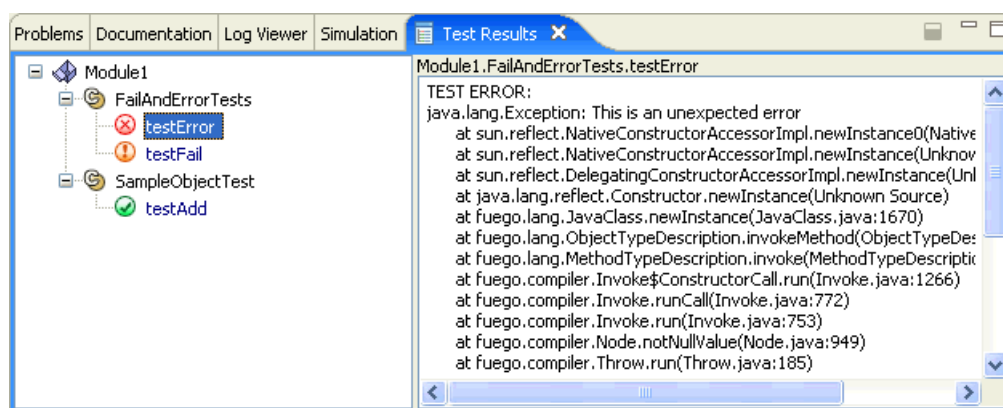
The Variables View displays the variables that are available within a Project.

The contents displayed in the Variable View depends on your current Profile and the context of the editor window you are viewing.

Project	
projectVar1	String(30)
projectVar2	String(30)
projectVar3	Time
Instance	
exceptionHolder	Any
Name	exceptionHolder
Type	Any
Description	
Arguments	
arg1	String
Local	
local1	String
local2	Real

Test Results View

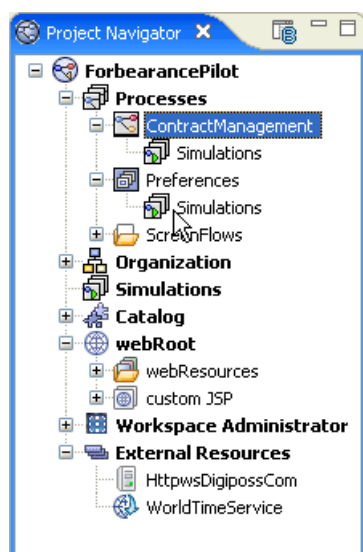
The Test Results View displays results from PUnit and CUnit tests.



Resources

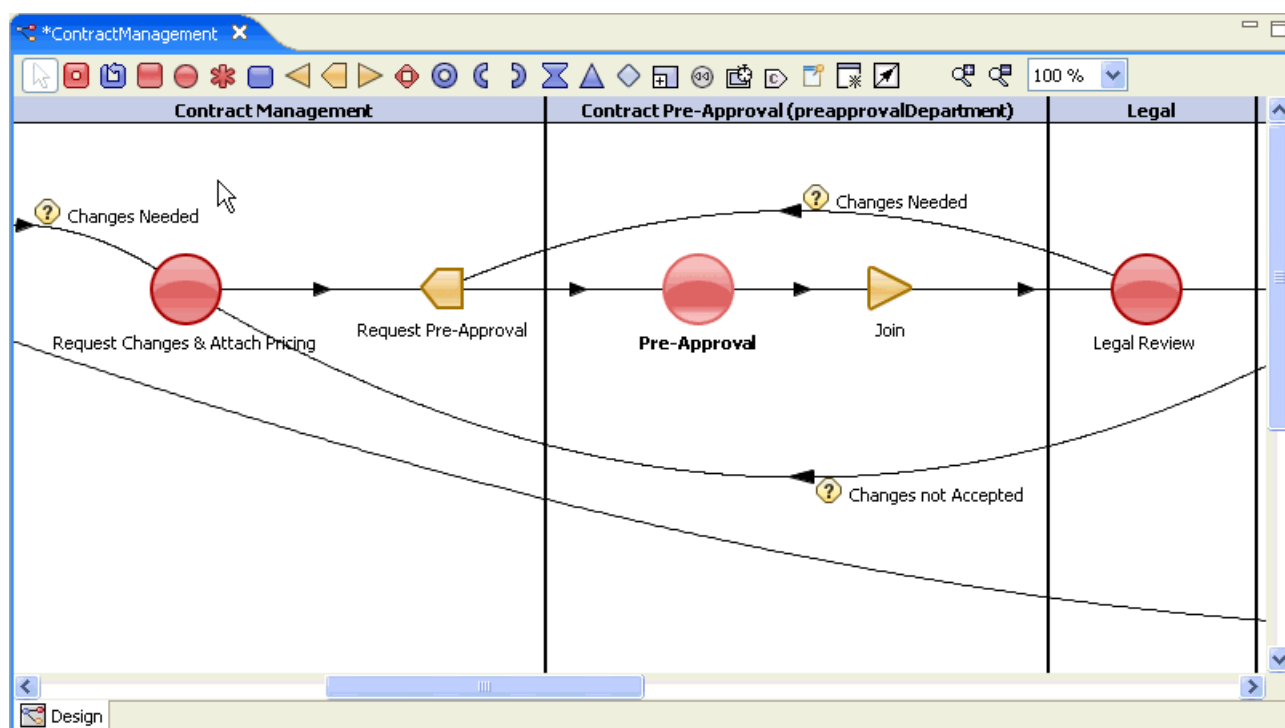
Resources are the files, folders, and projects that are part of your Eclipse workbench. Within AquaLogic BPM Studio, resources also include all of the Processes, BPM Objects, External Resources, etc. that are part of your business process.

The following image shows some of the Resources of a typical AquaLogic BPM Studio Project.



Editors

Editors allow you to create and edit resources within your Project. ALBPM provides different editors for different types of resources.



Common UI Tasks

This section contains basic tasks related to the ALBPM Studio user interface.

Changing Profiles

This task outlines the procedures for changing Profiles within ALBPM Studio.

ALBPM Studio provides different profiles for different types of users. See [Profiles](#) on page 11.

1. Select **Help** ► **Welcome**.
2. Click the profile you want to use.
See [Profiles](#) on page 11.

Changing Perspectives

This task shows you how to change perspectives in ALBPM Studio.

By default, Studio starts in the ALBPM perspective. To change perspectives:

1. Click the Perspective button in the upper right hand corner of the ALBPM Studio window.
2. Select **Other**.
3. Choose the Perspective you want to use.

Showing Views

This task outlines the procedures for showing Views in ALBPM Studio.

To show a view that is not visible within a perspective or to show a view that you have closed:

1. Select **Window** ► **Show View** ► **Other**.
2. Expand the folder of the type of View you want to open.
3. Select the View.

Views that are specific to the ALBPM Studio perspective are in the BPM folder.


Studio Preferences

ALBPM Studio provides different levels of customization preferences.

Setting Studio Preferences

ALBPM Studio preferences determine the general behavior of the Studio application.


To set Studio Preferences:

1. Ensure that the Project Navigator is visible.
See [Showing Views](#) on page 19.
2. Select the  icon.
The **Preferences** window appears.
3. Select a preference category from the list. The set of preferences for the selected category appears on the right.
4. Click **Ok**.

Setting Project Preferences


Project preferences allow you to customize ALBPM Project-specific behavior.

To set project preferences:

1. Right-click on the Project whose preferences you want to edit.
2. Select **Project Preferences** ().
3. Edit the Project preferences.
See [Project Properties Reference](#) on page 28 for detailed information on each Project preference.
4. Click **Ok**.

Setting Engine Preferences

Engine preferences allow you to customize the ALBPM Process Execution Engine behavior. Engine preferences are saved independently for each ALBPM project.

1. Ensure that the Project Navigator is visible.
See [Showing Views](#) on page 19.
2. Right-click on the ALBPM Project whose Engine preferences you want to edit.
3. Select **Engine Preferences** ().
The **Preferences** window appears.
4. Select a preference category from the list. The set of preferences for the selected category appears on the right.
5. Click **Ok**.

Setting Eclipse Preferences

Eclipse preferences allow you to customize general behavior of the Eclipse Platform.

1. From the main menu: **Windows ► Preferences**.
The **Preferences** window appears.

2. Select a preference category from the list. The set of preferences for the selected category appears on the right.
3. Click **Ok**.

Migrating Project Code

These sections provide specific procedures for migrating existing code based on ALBPM 5.7.2 into ALBPM 6.0.x. You must migrate your project code with ALBPM Studio before upgrading an existing ALBPM Enterprise environment.

Import 5.7 project into Studio 6.0

Follow this procedure to import an ALBPM project developed with version 5.7 into ALBPM Studio 6.0.

1. Open your project with ALBPM Studio 5.7.
2. Make sure the project is free from any errors. Select **Run ► Check All** on the menu to run the project consistency check.

You must fix all errors using Studio 5.7 before moving to 6.0.

3. Export your project from ALBPM 5.7.

From the menu, select **File ► Export Project....** Select the **Include all libraries** option and follow the next screens to generate an `.exp` file (the exported ALBPM project).

4. Start ALBPM Studio 6.0 and import the project you just exported.



Note: Opening an expanded 5.7 project from Studio 6.0 is not supported. You must first export the project to an `.exp` file with Studio 5.7 (see previous step).

- a) Select **File ► Import...** on the menu.
- b) Select **BPM ► Exported ALBPM Project into Workspace**. Push **Next** and follow the Wizard to select the exported project file and provide a name for the project.

As a result, your project is now included into your ALBPM Studio Workspace.

If there are any compilation errors, they will appear in the **Problems** view.

Fix External Resources

Due to improvements to the way ALBPM handles integration with other external systems, you might need to do some minor changes to the External Resources of your projects.

1. Open your project with ALBPM Studio 6.0.
2. Migrate to the new JDBC drivers provided by BEA.

ALBPM now supports several SQL databases out of the box (you don't need to provide an external JDBC driver anymore). If your project integrates with any of these databases, it is recommended that you migrate to the new drivers.

- a) Remove references to JDBC Java Libraries.

On the **Project Navigator** expand **External Resources** and remove those Java Libraries resources holding JDBC drivers (refer to the "External Resources" section of Studio's online help for details).

- b) Modify any existing SQL External Resources to use BEA's JDBC drivers

On the **Project Navigator** expand **External Resources** and double-click on a SQL database resource. In the **Edit External Resource** dialog change the value of the **Supported Type** drop-down to the corresponding BEA driver for your database.

For advanced JDBC URL configurations refer to the custom properties of the JDBC driver:

- For Oracle: http://edocs.bea.com/wls/docs100/jdbc_drivers/oracle.html#
- For Microsoft SQL Server: http://edocs.bea.com/wls/docs100/jdbc_drivers/mssqlserver.html
- For DB2: http://edocs.bea.com/wls/docs100/jdbc_drivers/db2.html
- For Informix: http://edocs.bea.com/wls/docs100/jdbc_drivers/informix.html
- For Sybase: http://edocs.bea.com/wls/docs100/jdbc_drivers/sybase.html

3. Re-catalog all components in the Catalog of your project.

This step is needed to re-generate the metadata associated with each component. You must re-catalog all types of components, including Java classes, SQL tables, Web Services, .Net, JNDI, CORBA, COM and SAP components.



Note: On version 5.x, Java type `java.util.Date` (when used in method arguments or as return type) was cataloged as `Fuego.Lang.Time`. On version 6 onwards, the type `java.util.Date` is maintained as such when cataloging. This doesn't cause incompatibilities.



Important: Cataloged Java classes do not include inherited members anymore (see next step).

4. Catalog additional Java classes as needed.

On version 5.7, a cataloged Java class included all members inherited from parent classes. Starting with version 6.0, cataloged Java classes only include methods and attributes explicitly defined by the class, excluding the ones inherited by its parent class. If there's code in your project using inherited members of a class, you must also catalog the parent Java classes that includes such members.

For example, if your project has cataloged EJBs (Enterprise JavaBeans), you must also to include the standard `javax.ejb.*` classes and interfaces into your catalog. Otherwise, standard methods like `getEJBHome()` and `getHandle()` do not show as members of your Bean.

Changes in Standard Components

Some standard components of the catalog are no longer available on this version, and others are now deprecated. This version also provides new components, which can cause conflicts when migrating old code.

Removed and Deprecated components

You have to adapt your code if your project depends on components which have been removed on this version of ALBPM. Refer to [Removed and Deprecated Components](#) *Some standard components of the catalog are no longer available on this version, and others are now deprecated.* to see which components have been removed or deprecated on each version of ALBPM.

Note about New Components

Standard components which are new to this version can cause naming conflicts with existing custom components in your project.

For example, a new component named `MailSender` has been added to the `Fuego.Net` module. If your project defines a BPM Object type also named `MailSender`, existing code referring to `MailSender` now becomes ambiguous.

To resolve this conflict, you should use the fully qualified names of those components:

```
sender = MailSender()           // ambiguous
sender = YourModule.MailSender() // explicitly refer to custom BPM Object
sender = Fuego.Net.MailSender() // explicitly refer to standard component
```



Important: This version of ALBPM supports Java 1.5. Therefore, modules under `Java.*` also include several new classes which may collide with your existing BPM Object names. Use fully qualified names in case of ambiguity.

Changes in Dashboards

The Dashboard components have been improved on ALBPM 6.0. They use a new graphics layer and provide new functionality. As a result, you might need to modify some of your existing 5.7 Dashboards to make them fully compatible with this new version.

Feature changes

Waterfall charts and Multiple-Pie charts are not available on version 6.0. When a project is imported from 5.7 they will be replaced by a bar chart.

Some properties of the Dashboard widgets are not available anymore.

- No canvas plot outline and plot background in pies.
- Horizontal layout is not available for area charts, stacked area charts and line charts.
- On Ctrl-Click. For all chart widget, this event cannot be captured anymore.

Dashboard Display Properties (on Global Activities) are only taken into account when displaying the Dashboards through the classic 5.7 Workspace.

Code changes

Existing 5.7 Dashboards featuring "drill down" from a chart to a list of process instances use code that needs to be changed.

The out-of-the-box Workload dashboards provide this feature. To make them compatible with 6.0 you need to change the auto-generated methods `drillDownTo*()` (generated by the dashboard editor) and `drillDownToInstancesView()` (generated with the out-of-the-box dashboards), replacing the following code:

```
...
result = getQueryStringFor(
    ClientBusinessProcess.processService,
    viewId : "viewId")

//Just to open the url
openURL this
using url = result,
    newWindow = false,
    standardBrowser = false
....
```

with:

```
broadcastViewChangeEvent this
using viewId = viewId
```

This new code is simpler and allows for displaying the dashboard and the list of instances view at the same time in the same page (in different *portlets*).

Changes in Implicit Java Classpath

Implicit directories added into Studio's Java "classpath" have changed in version 6.0.

Version 5.x of ALBPM Studio loaded Java classes and resources from the `studio/ext/` directory. This allowed your project code to load any resources (like property or configuration files) placed in this directory.

On ALBPM Studio version 6.0, directory `studio/ext/` was removed. You should place your resource files on your project's `lib/` directory instead. This is a cleaner solution, since each project has its own `lib/` directory and you don't need to modify your Studio installation directories anymore.

On ALBPM Enterprise, the same resource files should be placed in the `<engine_home_dir>/lib` directory, where `<engine_home_dir>` is the ALBPM Engines' Home directory, configured via Process Administrator.

On ALBPM Enterprise for J2EE, the resource files should be placed in a directory loaded by your application server's class loaders. For the case WebLogic, refer to

<http://edocs.bea.com/wls/docs92/programming/classloading.html#wp1096756>.



Important: When exporting a project from ALBPM Studio for deployment on ALBPM Enterprise, you should select option **Include Versionable Libraries Only**. This prevents non-versionable libraries and resource files from being included in the exported project file. If you don't select this option, resource files included in the project may be loaded before the files placed in `<engine_home_dir>/lib`.

Changes in PBL

Changes in Process Business Language

Comparison Operator

The comparison operator (`==`) is no longer allowed outside of boolean expressions.

For example, the following code was allowed on version 5.x, but will generate a **Statement is expected** error:

```
Boolean var;
var == false; // error reported on version 6
```

These kind of expressions do not product any effects. If your existing 5.x code fails on version 6 because of this error, you must revisit the code and understand what was the original intention behind it. Most commonly, it was a mistake of using the comparison operator (`==`) instead of the assignment operator (`=`).

Changes in Transformations

The deprecated feature of Transformations have been removed on ALBPM version 6.0.

Transformation methods existing on ALBPM 5.x projects are automatically converted to regular PBL code when the project is imported in ALBPM Studio 6.0.3+. Backwards compatibility is preserved, so no change is needed on your existing code base.

Each Transformation method is converted into three new methods:

- `<transformation>Method()` - Main method replacing transformation
- `<transformation>MethodBegin()` - helper method
- `<transformation>MethodEnd()` - helper method

For example, a Transformation method named `toMyObject()` on your 5.x project is converted into these three methods:

- `toMyObject()` - Main method replacing transformation
- `toMyObjectBegin()` - helper method invoked from `toMyObject()`
- `toMyObjectEnd()` - helper method invoked from `toMyObject()`

Process Designer

Projects

The topics in this section provide conceptual, task, and reference information about ALBPM Projects.

What is a Project?

Projects provide a way to organize, develop and manage different processes, users, components and systems catalogs.

A business project is the combination of a series of actions or operations pursuing a common business purpose. These activities, either human or automated, need to be executed in order to deliver a product or service. Business requirements may involve functional integration across the company or organization.

A Project involves not only the representation of all the elements that are part of a business, the human resources, the organization, the processes and the systems execution but also the way in which all of them interact.

Projects enable you to group processes that are related in some way and separate them from other groups.

Each project has its own component catalog so that you will be able to separate components used in some processes but not in others by grouping them in different projects. The project also contains all the abstract user roles used in it and its own Organization information required in order to deploy the project.

Project Resources

Each Project contains the following resources that are visible in the Project Navigator:

Resource	Description
Processes	Contains the Processes, Procedures, and Screenflows defined for the Project. Within the Processes resource you can create multiple folders to organize resources.
Organization	Contains the Organizational elements that are defined for the Project.
Simulations	Contains simulations defined for the entire Project.
Catalog	Displays the list of catalog resources accessible from the Project.
webRoot	Contains user interface resources such as HTML pages and Java Server Pages.
Custom Views	Contains Views and Presentations that are defined locally for testing within ALBPM Studio.
External Resources	Contains connectivity information for external resource such as databases.

Project Tasks

The following topics contain low-level tasks for creating and using Projects.

Creating a Project

This section describes the basic procedures for creating an ALBPM Project.

To create a Project:

1. Select **File ► New ► Project**
The **New Project** wizard appears.
2. Select **ALBPM ► ALBPM Project**.
3. Click **Next**.
4. Provide a Project name.
5. Choose a Project Root Directory.
The Project root directory contains all of the resources used by the Project.
6. Click **Next**.
7. Click **Finish**.

The new Project appears in the Project Navigator.

Exporting a Project

Exporting a Project allows you to create a self-contained, portable version of a Project.

To export a Project:

1. Right-click on the Project you want to export.
2. Select **Export Project**.
3. Choose which Java Class Libraries to include with the export file.
4. Click **Next**.
5. Provide the name and output directory of the exported Project file.
6. Click **Next**.
7. Click **Finish**.

Importing a Project

Importing a project allow you to share Project resources.

To import a previously exported project:

1. Select **File ► Import...**
The **Import** window appears.
2. Select **BPM ► Exported ALBPM Project into Workspace**.
3. Click **Next**.
4. Click **Browse** to locate the Project you want to import.
5. Click **Open**, then click **Next**.
6. Provide a name for the imported Project.



Note: The original name of the exported Project appears by default. However, if your workspace already contains a project with this name, you must specify a different one.

7. Click **Next**.

The Project files are expanded into your workspace.

8. Click **Next**, then click **Finish**.

The imported Project appears in the Project Navigator.

Importing Designs

The Import Designs command allows you to import Processes, Screenflows and Procedures from other common Business Process Modeling formats.

To import designs:

1. Right-click on the Project or Processes resource where you want to import Project Designs.
2. Select **Import Designs**.
3. Specify the type of file you want to import in the **Files of type** drop-down list.
4. Select the file you want to import.
5. Click **Open**.

Creating a Project Report

Creating a project report allows you to view general and summary information about your project.

To generate a Project Report:

1. Right-click on the Project you wish to create a report for.
2. Select **Project Report**.

The **Report Options** window displays.

3. Choose the elements you want to include in the report:

Report Elements
Include Use Cases
Include Implementation Source Code
Include Variables


4. Click **OK**.
5. Select either *HTML* or *PDF* from the report output type drop-down list.
6. Select the folder where you want to output the report.
7. Click **OK**.

The report is generated. In PDF format, this will be a single file. In HTML, a folder and subfolders are created. You start reading an HTML report at the `index.html` page in the top folder.

Setting Project Preferences

Project preferences allow you to customize ALBPM Project-specific behavior.

To set project preferences:

1. Right-click on the Project whose preferences you want to edit.
2. Select **Project Preferences** ()
3. Edit the Project preferences.
See [Project Properties Reference](#) on page 28 for detailed information on each Project preference.
4. Click **Ok**.

Project Properties Reference

Project properties allow you to define different aspects of an ALBPM Project.

General

Property	Description
Developing for J2SE Development	

Languages

Property	Description
Current Language	Specifies the language used by this project. This setting is used to determine the language of Process names. If the localized version of a label is available, this version is displayed to users. If no localized version is available, the default language is used.

You can also use this page to add languages to a project. After adding a language to a project, you have the option of creating localized Activity labels.

Exception Handling

Property	Description
Exception Handling	<p>Defines how exceptions are handled within the project. The following options are available:</p> <ul style="list-style-type: none"> • Propagate: Causes un-handled exceptions are propagated to the parent or invoking process. The instance of the child process is aborted without executing the End Activity. • Handle Exceptions: Allows you to explicitly define how exceptions are handled.
Automatically Generate Exception Handling Activity	Causes Studio to automatically generate an exception handling Activity and Role when creating a new process. This option is only available when Handle Exceptions is selected.

Processes

The topics in this section provide conceptual, task, and reference information about using Processes with ALBPM Studio.

What is a Process?

A Process is a logic representation of a business function broken into multiple steps that correspond to different tasks or functions. Each process models a specific business need

Processes are composed of logical steps called activities. For example, for an Order Management process you might create activities called Create Order, Check Inventory, Select Shipping Route, Check Customer Credit, Pick Product, Pack Product, Create Billing, Create Invoice, Print Invoice and Ship Product

Each activity is assigned to a role. Roles indicate who will perform the specific activity. You connect the activities to transitions in order to define the process workflow sequence from activity to activity. Transitions may be unconditional, conditional, due or exception depending on the type of workflow that is desired.

Version control built into the BPM system lets you modify processes on the fly, even if instances already exist in published and deployed processes.

Process Instances

A process instance is a single enactment of a process. The creation of instances is generally triggered by an event such as when an employee submits an expense report.

Some examples of process instances include:

- In a Hiring process, a prospective new hire.

- In an Order Management process, a new order.
- In a Patient Insurance Care process, a patient's insurance claim.

Within a Business Process, there must be a way to create an individual instance of a Process. There are several internal and external ways to create an instance. Instances are created in the Begin activity and stopped when they reach the End activity of the process. Once an instance has been created, it begins its flow through each activity in the process according to transition rules and Business Process Methods logic. The following sections describe how a Process Instance is created.

Internal Instance Creation

An internal instance creation is generally started by an activity type that is designed to create instances. The following table lists activities that can create instances in processes.

Activity	Creation Method
Global Creation	The Global Creation activity is the most common way an instance is created in a process. It appears in WorkSpace where an end user can manually click it, complete its BP-method and send the instance on in the process. No special BP-method is required to create an instance.
Global Automatic	The Global Automatic activity is another common way to create an instance. However, this activity type does not create instances automatically. The create method of the Process Instance component must be invoked from a BP-method in order to create an instance
Subflow	The Subflow activity creates instances in the subprocess indicated in its Activity Properties dialog box. Instance creation is automatic
Process Creation	The Process Creation activity also creates instances in the subprocess indicated in its Activity Properties dialog box. Instance creation is automatic

Process Instance Component Creation

The Process Instance component contains the method create, which allows you to create an instance in the process indicated by the parameters of the method's template. The create method can be called from any BP Method. This component comes with the Studio installation. It is located in the **fuego** module on the Component Browser in the BP Method Editor.

External Instance Creation

Instances can also be created by external events. The external event triggers a Global Creation activity in a process. One of the most common ways in which this happens is by integrating an existing Web application with a process. For example, if a customer places an order through a Web application, an instance can be created in the corresponding process. See Creating instances from a Web Page for further information.



Note: In Studio, the maximum number of instances to be created at a time is 5000. The engine is able to execute that maximum number simultaneously. Therefore, it cannot create more instances than that number.

API Instance Creation

Process Instances can also be created using the Process API (PAPI).

Web Service Instance Creation

When a Process is exposed as a Web Service, Process Instance can be created via a Web Service.

Process Tasks

This section contains tasks for creating Processes as well as using a Process as a Web Server, defining process simulations models, and others.

Creating a Process

Creating a new process allows you to begin modeling your business function.

Before performing the procedures in this task, ensure that you have created a Project. See [Creating a Project](#) on page 26.

To create a new Process:

1. In the Project Navigator, expand the Project where you want to create a new Process.
2. Right-click on **Processes** ➤ **New Process**
3. Provide a Process name and optional description.
4. Select the check box corresponding to how you want to generate Audit Trail Events for this Process.
See [Audit Events](#) on page 99 for more information.
5. Click **OK**.

The new process is opened in a Process Editor window. All new processes are created with a Begin and End Activity connected by a default Transition. The new Process appears in the Processes resource in the Project Navigator.

Importing Designs

The Import Designs command allows you to import Processes, Screenflows and Procedures from other common Business Process Modeling formats.

To import designs:

1. Right-click on the Project or Processes resource where you want to import Project Designs.
2. Select **Import Designs**.
3. Specify the type of file you want to import in the **Files of type** drop-down list.
4. Select the file you want to import.
5. Click **Open**.

Creating a Process Simulation Model

The following steps describe how to create a Process Simulation Model.

1. In the Project Navigator View, expand the Project where you want to create the Process Simulation Mode.
2. Expand **Processes**.
3. Right-click on the Process.
4. Select **New Process Simulation Model**.
5. Enter a name for you new Process Simulation Model.
6. Click **OK**.

The Process Simulation Model appears in the editor window. It also appears as a Resource in the Project Navigator View.

You can define the behavior of your Process Simulation Model.

Exposing a Process as a Web Service

Exposing a Process as a Web Service allows it to be accessed externally by other applications within your enterprise.

Before performing the procedures in this task, ensure that you have created a Process. See [Creating a Process Method](#).

1. In the Project Navigator, expand the Project containing the Process you are exposing as a Web Service.
2. Expand Processes.
3. Right-click on your Process, then select **Process Web Service**

The Process Web Services tab is displayed in the Process Editor.

4. Click **Add** to create a new Web Service operation.
5. Provide the required information for each field.
See [Process Web Service Reference](#) on page 33 for more information.
6. Click **Ok**.

The new Web Service operation appears in the table. You can more operations as required.

7. Configure the optional **Advanced Options**.
See [Process Web Service Reference](#) on page 33 for more information.
8. Select **File ► Save**

The process is exposed as a Web Service. If you start the Process Execution Engine using this process, you can view the deployed Web Service at: <http://localhost:9000/>. See [Starting the Process Execution Engine](#) on page 204 for information.

Publishing a Process to AquaLogic Service Bus

ALBPM Processes can be used within AquaLogic Service Bus.

Before performing the procedures in this task, you should ensure that AquaLogic Service bus is configured and running.

To publish an AquaLogic BPM process to AquaLogic Service Bus.

1. Create a Process.
See [Creating a Process](#) on page 31 for more information.
2. Expose the Process as a Web Service.
See [Exposing a Process as a Web Service](#) on page 31.
3. Start the Process Execution Engine.
See [Starting the Process Execution Engine](#) on page 204.
4. Create an ALSB External Resource of type Management Host.
See [Creating an External Resource](#) on page 183 and [AquaLogic Service Bus](#) on page 195 for more information.
5. Create a second ALSB External Resource.
 - a) Configure the External Resource as described in [Creating an External Resource](#) on page 183.
 - b) Set the type to Process Registration.
This can be the same External Resource you configured in the previous step.
 - c) Provide a project name.
This is the name of the ALSB project where the process will be published to. This can be a new or existing project.
 - d) After you have provided all of the information, click **Create Structure**.
The AquaLogic Service Bus project is updated or created.
6. Register the End Point
 - a) Right-click on the Project you used to start the Process Execution Engine.
 - b) Select **Register End Point**.
The **AquaLogic Service Bus Registration** window appears.
 - c) Select the Registration Configuration you want to use.

This can be the ALSB Process Registration External Resource you created earlier.

- d) Select Yes in the first column of the process you want to register.
- e) Click **Register**.

A message is written to the log window. The status of the process is changed to **up to date**. You can publish as many processes as necessary.

The process you registered is visible in the AquaLogic Service Bus Project Explorer under Business Services.

Process Property Reference

Process properties define the behavior of a process within AquaLogic BPM Studio.

The following properties can be defined for a Process:

Property	Description
Name	Specifies the name of the Process.
Id	Specifies the ID of the Process. This value is defined when the Process is created and cannot be modified.
Description	Provides a description of the Process. This is used when generating Project documentation
Variation	
Author	Specifies the author of the Process.
Generate Events	Defines how Auditing Events are generated for the Process. See Audit Events on page 99.

Process Web Service Reference

The properties listed in these topics are available when exposing an ALBPM Process as a Web Service.

Operations

Property	Description
Operation Name	Defines the name used to refer to the operation. This name will become the method name to execute this operation.
Operation Type	Specifies the operation type: <ul style="list-style-type: none"> • Process Creation if this operation is to create an instance. • Process Notification if this operation is to notify an instance waiting in a Notification Wait activity.
Activity	<ul style="list-style-type: none"> • if the operation is for Creation then the Begin activity is automatically selected. • if the operation is for Notification operations then select the Notification Wait activity in which the instance is waiting and to be notified using this operation. The Wait Activities receiving notifications be marked as Receiving External Events
Argument Set Name	Determines the argument set the operation uses to receive arguments. This allows you to define more than one operation of the same type, each one receiving different arguments. If it is a Notification operation then you can define to use a Correlation.
Argument / Type	Displays the argument variables and types defined in the Argument Set Name.

Activities

The following topics provide general information about Activities and describe how to create and use them.

What Is an Activity?

Activities define a manual or automated task that conforms one step within a process design. Adding a new activity allows you to create a new step and assign it to a role in a process.

A manual activity requires end user intervention whereas an automatic activity can be automatically completed by the Engine. An activity can include one or more tasks.

The following table describes different categories of Activities. For detailed information on each Activity see [Activity Types](#) on page 34:

Category	Description	Activities
Process Initiation/Termination Activities	Function as a beginning and end point for the process. The activities are automatically generated and define the scope of the process.	Begin, End
Human Interaction Activities	Allows user interaction with process.	Interactive, Grab, Decision
System Interaction Activities	Handle automatic interactions with business systems.	Automatic
Organizational Interaction Activities	Allow communication with other areas and processes of an organization.	Process Creation, Termination Wait, Process Notification, Notification Wait, Dynamic Process Call
Process Control Activities	Control process flow or generate copies of a process instance to allow flow through multiple paths simultaneously.	Split , Split N, Join, Conditional
Global Activities	Handle global requirements that are not associated with a specific process instance.	Global Creation, Global Automatic
Miscellaneous Activities	Provide other functionality with a process	Connectors, Measurement Marks

Activity Naming Conventions

It is recommended that you name your activity with a verb followed by a noun specifying the Activity's role within the Process. Providing descriptive names for your Activities allows your process to be self-documenting. For example, Create Order, Ship Product, Check Credit are all useful Activity names.



Note: After an Activity name has been defined, it cannot be changed. However, you can change the Activity label which is displayed to end users.

Activity Types

The following sections describe the different types of Activities available in Aqua Logic BPM Studio.

Automatic Activity

What is an Automatic Activity?

Automatic activities do not require any direct end user interaction. Applications and components interfaced with Automatic activities should not require any end user intervention.

Applications and components typically run on a remote server and perform work behind the scenes. For example, database maintenance, e-mail notification and so on.

Automatic activities are used in the process design where work can be performed without human intervention. Some typical uses are as follows:

- Database updates.
- Running batch programs.
- Sending e-mail notifications.
- Sending e-mail confirmation to customers.

Using the Automatic Activity

The following table outlines some of the considerations when using the Automatic Activity within ALBPM Studio.

WorkSpace	Automatic activities do not appear in WorkSpace because the Automatic activity does not require any end user intervention. Any BP-Method in Automatic activities is automatically executed.
Roles	Automatic activities can appear either in automatic roles or in the user defined role types. However, if the Automatic activity is in a user-defined role, it will not appear in WorkSpace.
Variables	The Automatic activity can access predefined, local or instance variables.
Pre Conditions	The Automatic activity is activated by an incoming instance from another activity in the process.
Post Conditions	After the BP-Method tasks in the Automatic activity have been completed, the instance flows to the next activity in the process according to transition rules.
Tasks	The task has to be defined by the developer. The automatic activity can have only one task to be completed, that is, the Main Task . See Tasks.
Business Process Methods	<p>Automatic activities can have only one BP-Method task. The BP-Method is automatically performed.</p> <p>Components invoked from Automatic activity BP-Method tasks are sent incoming parameters (set by BP-Method's variables) and may return outgoing parameters (which set BP- Method's variables.) Components accessed from Automatic activities should not require any end user intervention. If user interface is required, use an Interactive activity instead.</p> <p>You can use a display Method statement in the BP-Method of an Automatic activity for testing purposes. Instead of being displayed to an end user, the display statement writes to the Engine log.</p>

Automatic Activity Example

The examples in these topics show the types of tasks you can perform within an Automatic Activity.

Sending an Email

In the following BP-Method, the Send E-mail activity sends e-mail notifications to the system administrator's e-mail inbox depending on the value in the serverDown variable. The whole BP-Method is automatically executed.

```
if serverDown == true then
  send Mail
    using recipient = "sysAdmin@bea.com",
       from = "sysAdmin@bea.com",
       message = "Venus server is down. Please check!"
end
```

Updating a Database

In the following example, a database is updated with new employee's information. This BP-Method is in the Automatic activity because it does not require any end user intervention to update the database.

```
INSERT INTO employee (fname, lname, salary, title)
VALUES (firstName, lastName, salary, jobTitle);
```

Rolling Back an Automatic Activity

If an Automatic Activity fails the Process Execution Engine can be configured to retry the Activity.

The Engine will try to execute it as many times as defined in the Engine properties Retry times and Retry interval. If the execution is still not successful, then an exception is thrown.

If an action within the Automatic activity has not completed and needs to be reverted, you must manually define the rollback in an Exception Handling Flow for that Automatic activity. The best way to implement the rollback is to define an Exception Flow for each possible known exception. Therefore the Automatic activity will have multiple exception transitions, one for each known exception that might occur. To contain all other not expected exceptions, design the Exception Flow for the Others exception.

Automatic Activity Property Reference

Advanced Properties

Property	Description
Concurrent Executions	Allows you to control the number of instances that may flow through an activity at the same time.
Maximum Number of Instances	Determines the maximum number of instances that can flow through the activity at one time. Is only available if Unlimited concurrent executions is unchecked.
Generated Events	Defines how Auditing Events are generated for the Activity. See Audit Events on page 99.

Considerations

As you design your process, think about which activities may take longer to process instances due to component connections or resource allocations. You may want to slow down some of the fastest activities by limiting the number of instances that can be concurrently processed.

Begin Activity

What is a Begin Activity?

The Begin activity provides an entry point into every process. The Begin Activity creates a Process Instance. Instance variables are set for each instance as it flows through the Begin activity.

When a new process is created, Studio automatically creates a Begin activity. There can be only one Begin activity in a process. Instances can enter a Begin activity with arguments initialized by any of the following:

- A Global Creation activity in the process
- A Global Automatic activity in the process
- A Subflow activity in another process.
- A Process Creation activity in another process.
- An external application or web page (through a servlet) that uses the Process and ProcessInstance components to initiate an instance in a process, using PAPI or WAPI.




Note: There is always an implied path from any Global Creation Activity to the Begin Activity. This implied path is not displayed.

Using the Begin Activity

The following table outlines some of the considerations when using the Begin Activity within ALBPM Studio.


WorkSpace	The Begin Activity is the entry point into a process. It does not appear in WorkSpace.
Roles	The Begin activity automatically appears when you select File and New Process from the Studio menu options. The Begin activity can be moved around on the Process Designer workspace. The Begin activity can be either in an automatic or an interactive role.
Variables	The mapping between instance variables and process arguments is specified at the Begin activity. The Instance variables will be automatically completed with the received arguments, as designed in the Argument Mapping.
Pre Conditions	Argument variables come into the Begin activity from an external component or application, a Global Creation activity, a Global activity, a Global Automatic activity executing ProcessInstance component or from an activity type that starts a subprocess. These arguments respond to the Argument Mapping defined in the Begin Activity.
Post Conditions	An instance is created within the Begin activity. Instance variables are initialized from the incoming arguments, as mapped. The instance flows to the next activity according to transition rules.
Transitions	Incoming transitions are not allowed on Begin Activities. Begin activities must contain one or more out-going transitions. Begin activities cannot transition to a Grab Activity. If the Begin activity has multiple sets of arguments, then Message based transitions are available

Business Process Methods	<p>In order to pass arguments between processes, the Argument Mapping function is available.</p> <p>Advance scripting is available but because of compatibility with previous version. It is not recommended to add code either in the Begin or End PBL. In case you need to execute some coding, you can include that in an automatic activity immediately after the activity.</p> <p> Note: If any of the instance variables used in the argument mapping are modified in the script, the values passed in the mapping are lost and the new ones become valid.</p>
--------------------------	--

Begin Activity Property Reference

General Properties

The following table lists the general properties.

Property	Description
Unlimited concurrent process instances	Allows an unlimited number of concurrent processes to enter the Begin Activity. Unchecking this option allows you specify the number of concurrent processes allowed.
Action performed when limit is reached	<p>Specifies how concurrent executions are handled when the defined limit has been reached.</p> <p> Note: In a J2EE clustered environment, this property applies to the whole cluster.</p>

Advanced Properties

Property	Description
Generates Events	Defines how Auditing Events are generated for the Activity. See Audit Events on page 99.

Conditional Activity

What is a Conditional Activity?

The Conditional Activity helps you centralize different paths of the process into one activity and re-distribute the instances based on conditional transitions.

Using the Conditional Activity

The following table outlines some of the considerations when using the Conditional Activity within ALBPM Studio.

WorkSpace	Conditional activities do not appear in WorkSpace because the Conditional activity does not require any end user intervention.
Roles	Conditional activities can appear in either automatic roles or the user defined role types. However, if the Conditional activity is in a user-defined role, it will not appear in WorkSpace.

Variables	No variables are available.
Pre Conditions	The Conditional activity is activated by an incoming instance from another activity in the process.
Post Conditions	The instance flows to the next activity in the process according to transition rules.
Transitions	One or more incoming and outgoing transitions are required. Only Unconditional and Conditional transitions are available.
Tasks	No tasks are available.
Business Process Methods	No tasks are available.

Connectors

When two Activities are not close enough to each other to be displayed at the same time in a Process editor, you can add connectors to create a shortcut to an Activity. Connectors eliminate the need to draw a transition line across a large process design to connect two activities.

Decision Activity

What is a Decision Activity?

A Decision Activity is a type of Interactive Activity that presents the end users with a list of business values (taken from the Process Instance) and prompts them to make a decision (i.e. what to do next) while suggesting the right answer based on past experience.

The Decision Activity is ideal in situations where a user needs to make a decision to route a process instance to one of multiple possible out-going transitions. By implementing the Decision Activity within a process model, you can assist your users in making a decision based on specific process variables.

For example, a Car insurance policy could be classified as low, medium or high risk depending on input variables like insured age, geographic area. This classification could be done with a Decision Activity, and over time the system will learn and suggest the most probable answer for each new policy.

The Process Execution Engine records end-user decisions and does statistical analysis in order to come up with suggestions for future decisions. This algorithm is based on *Support Vector Machines (SVMs)* methods.

Recorded decisions are evaluated and presented to end-users as a set of probability percentages. This allows users to make decisions based on the actions of previous users.



Note: In order for the Engine to provide a meaningful recommendation, the data for previous decisions must be consistent. Wide fluctuations in previous responses will cause the Engine's recommendations to be inconsistent.




Note: The Process Engine analyses the recorded decisions at periodic intervals as part of the Engine's **Disposer** service (by default, every two days). End users will not get any suggested probabilities until this service runs.

Decision Activities use standard component `Fuego.Bis.DecisionProblem` behind the scenes in order to record end-user decisions and present the suggested probabilities. For advanced use cases this component may be used directly from PBL, which allows you to feed the statistical algorithm and leverage the Process Engine analysis from any type of activity or BPM Object.

Using the Decision Activity

The following table outlines some of the considerations when using the Decision Activity within ALBPM Studio:

WorkSpace	The Decision activity is visible in WorkSpace, but only to the participant(s) assigned to the role where the Decision activity exists.
Roles	<p>Decision activities must reside in user-defined roles.</p> <p> Note: If you add a new Decision activity and drop it in an automatic role, the BPM system will allow you to select an existing user-defined role or generate a new one (the Role dialog box populates.) The column-role is added to the design and the new Decision activity will automatically appear in the selected role.</p>
Variables	Decision activities can access instance, and predefined variables.
Pre-conditions	<p>An Incoming instance from another activity in the process with the necessary instance variables set.</p> <p>Decision activities can also be used as an exception handler.</p>
Post-conditions	Instance moves to the next activity in the process via one of the outgoing transition lines.
Transitions	One or more incoming and outgoing transitions are required.
Tasks	<p>The default behavior for Decision Activities consists in having one single Implementation or task.</p> <p>This greatly simplifies the understanding of Decision Activities.</p> <p>Furthermore, this simplifies the operation of WorkSpace, since executing an instance is ALWAYS executing its implementation.</p> <p>However, you can have Optional (or support) tasks that the participant may need to execute in some cases in order to help him do the required work (that is, executing the main task.)</p> <p>But if you look at optional tasks in this way, they make sense ONLY if you allow the user to execute them while executing the main task.</p> <p>To do so, these optional tasks must be read-only, meaning that they must not modify the instance data in order to avoid inconsistencies.</p> <p>The tasks have to be defined by the developer. The Decision activity has then a main task. Moreover, optional tasks can be added.</p>
Business Process Methods	<p>Decision activities can have one or more BP-Methods defined as tasks. Applications invoked from Decision activities are sent incoming parameters (set by BP-Method variables) and return outgoing parameters (which set BP-Method variables).</p> <p>Note:</p> <p>Human participants are unpredictable. If an end user begins a task in WorkSpace but does not complete it, a thread for that task remains open in the Engine. Too many open threads can cause poor server performance.</p>

To ensure that end users do not lock threads, use relay to in your BP-Method to immediately close threads and relay end user's response to another BP-Method task when the end user completes a task. See Controlling threads with relay to for further information.

Decision Activity Property Reference

Defines Decision Activity properties.

General Properties

The following table lists the general properties.

Property	Description
Suspendable	Suspends any due transitions from this activity. Due transitions will not expire and the process instance will only continue to the next Activity based on user input.
User Selects Transition	Allows the activity to have more than one unconditional transition. The user must select which Activity the process instance follows.
Auto Complete	Causes the process instance to automatically move to the next activity after all mandatory tasks are finished. If Auto Complete is not enabled, the user must explicitly route the process instance to the next activity.
Abortable	Allows the user to abort the process instance from this activity. The process instance proceeds directly to the End Activity in an aborted state.
Assignable	Allows the user to assign the process instance to another user.

Advanced Properties

The following table lists the advanced properties.

Property	Description
Unlimited concurrent executions	Allows an unlimited number of concurrent executions to occur in the Decision Activity. Unchecking this option allows you specify the number of concurrent executions allowed.
Action performed when limit is reached	Specifies how concurrent executions are handled when the defined limit has been reached.
Generate Events	Defines how Auditing Events are generated for the Activity. See Audit Events on page 99.

End Activity

What is an End Activity?

The End activity is always the last activity in a process, and provides an exit point from a process. When you create a new process, the BPM system automatically creates an End activity. There can be only one End activity in a process.

The End activity transforms instance variables into arguments (See Argument Mapping) that can be then passed to another process or to an external application. Upon completion, the End activity can return flow to any of the following:

- A Subflow activity that called the process.
- When a process was started by another process using a Process Creation activity, the called process' End activity can return instances to the calling process' Termination Wait activity.

- An external application.

Using the End Activity

The following table outlines some of the considerations when using the End Activity within ALBPM Studio.

WorkSpace	The End activity is an end point of the process. It will not appear in the WorkSpace.
Roles	The End activity automatically appears when you add a new process. It can be moved around on the process design workspace, but it must reside in an Automatic role lane.
Variables	<p>The End activity can define instance variables to determine the argument mapping. See Argument mapping.</p> <p>The End activity transforms instance variables into outgoing arguments.</p>
Pre Conditions	Instances come into the End activity from the process.
Post Conditions	When an instance completes the End activity, it is completed within the process. If the process was called by a Subflow or Process Creation activity, the instance variables are set to outgoing arguments, as defined in the Argument Mapping, and are passed back to the calling process from the End activity.
Transitions	One or more incoming transitions are required. End activities can only transition to a Grab activity (outgoing transition). Instances can be grabbed from the End activity unless the instance status is aborted or terminated.
Tasks	No tasks are available.
Business Process Methods	<p>To pass arguments between processes, the Argument Mapping function is available.</p> <p>Arguments are passed to the calling process to return information to it</p> <p>Advance scripting is available but because of compatibility with previous version.</p> <p>It is not recommended to add code either in the Begin or End PBLs.</p> <p>In case you need to execute some coding, you can include that in an automatic activity immediately before the End activity.</p> <p>The script is run first and then the argument mapping is set.</p>

End Property Reference

Defines End Activity properties.

Advanced Properties

The following table lists the advanced properties.

Property	Description
Generates Events	Defines how Auditing Events are generated for the Activity. See Audit Events on page 99 .

Global Activity

What is a Global Activity?

Global activities are primarily used to allow end users to run applications or database queries only when needed.

These applications are not an integral part of the process, but they contain information that can be accessed on an "as needed" basis. As a result of this, Global activities can be assigned to any user-defined role. Global activities never have transition lines going to or coming from them. They do not have interaction with instances either.

Global activities are useful to easily run lookup queries on databases, to quickly send e-mails or to invoke any applications that will help the end user as he or she interacts with the process.

You can also implement certain predefined functions using a Global activity, such as:

- Process image: shows the instance within the process graphic.
- Display instance: you can custom the instance panel.
- Workload: shows the process workload.
- Dashboard: shows business activity monitoring based on stored information.

Using the Global Activity

The following table outlines some of the considerations when using the Global Activity within ALBPM Studio.

WorkSpace	<p>The Global activity is visible in WorkSpace, but only to users assigned to the role where the Global activity exists. If the Global activity does not have access to the instance, it can be executed from the WorkSpace Application view. If the Global activity has access to the instance, it can be executed from the WorkSpace next to the instance.</p> <p>The following Global activities can be executed from the Optional tasks toolbar:</p> <ul style="list-style-type: none"> • All Global activities that do not have instance access, for example the Workload global activity • Global activities that have instance access but defined as Read-Only. • Global activities defined as Process image. <p>When you execute a Main task you have enabled the optional toolbar. Check Workload and SendMail are Global activities with no instance access.</p>
Roles	Global activities reside in user-defined roles.
Variables	Global activities can access predefined, local and argument variables from the BP-Method Editor.
Pre Conditions	No preconditions are required for the Global activity. It is an "as needed" activity that can interface with an instance. or be an independent activity with no relation to an instance.

Post Conditions	There are no post-conditions for the Global activity.
Transitions	There are no transitions to or from the Global activity.
Tasks	<p>The task has to be defined by the developer. The global activity can have only one task to be completed, that is, the Main Task.</p> <p>The Global activity can have instance access or no interaction with an instance. Based on this property, the task can have different type of implementation.</p>
Business Process Methods	There are no methods for this Activity.

Global Activity Property Reference

General Properties

The following table lists the general properties.

Property	Description
Has access to instance variables	the Global activity can apply to an instance or not. Based on this property, the activity can implement different types of tasks.
Customize instance information	if the activity has instance access, you can customize the instance information when you select to work with it. It indicates that the instance display information will not be the default one but built with a Component, Method, Screenflow or the instance's variables.

Global Automatic Activity

What is a Global Automatic Activity?

Global Automatic activities do not have any direct end user interaction. The applications/components invoked by the Global Automatic activity typically run on a remote server.

Global Automatic activities are useful for processing batch reports or downloading batch files at scheduled times. Global Automatic activities can also be used as event listeners in the process. They can be programmed to listen to a port or to a specific event, such as an end user mouse click or a broken connection to a remote component. And then, based on such event, they perform some type of action.

Global Automatic activities do not appear in the WorkSpace tasks list. These activity types can invoke a component or application that creates instances. But the instances are automatically created without user intervention.

Instances flowing through the process do not have any interaction with the Global Automatic activity. However, the Business Process Method in the Global Automatic activity may be run because of some action caused by an instance (Executes when an event occurs) or it may create an instance in the process.

Global Automatic activities:

- Do not have any relationship with an instance.
- Automatically execute the Business Process Method without receiving an instance, even though they can interact with an instance in the BP-Method. These kinds of BP-Methods do not work directly over an instance but, for example, they can send notifications to instances.
- Cannot be manually launched.

Using the Global Automatic Activity

The following table outlines some of the considerations when using the Global Automatic Activity within ALBPM Studio.

WorkSpace	The Global Automatic activity is not visible in WorkSpace because the activity does not require any end user intervention.
Roles	Global Automatic activities can reside in a user-defined or an automatic role lane.
Variables	The Global Automatic BP-Method can access predefined and local variables.
Pre Conditions	<ul style="list-style-type: none"> • Polling by Interval: waits for a specified interval before launching the BP-Method. • Executes when an event occurs (event listener): waits for a specified event before launching the Listening BP-Method. • Automatic Scheduled: waits for a scheduled date and time before launching the BP-Method. • Automatic JMS Listener: This type is applicable for J2EE Enterprise Editions. Listens to JMS message before launching a BP-Method.
Post Conditions	<p>Post-conditions vary depending upon the type of Global Automatic activity and the BP-Method that is launched.</p> <ul style="list-style-type: none"> • Polling by Interval: BP-Method runs after an interval. Once the Global Automatic task finishes, the new interval of time begins to count. Any action required by the BP-Method, such as instance creation, is completed. • Executes when an event occurs (event listener): BP-Method runs when a predefined event occurs. Any actions required by the BP-Method, such as instance creation, are completed. • Automatic Scheduled: BP-Method runs according to a time schedule. Any actions required by the BP-Method, such as instance creation, are completed. • Automatic JMS Listener: This type is applicable for J2EE Enterprise Editions. BP-Method runs when a predefined event occurs. This BP-Method has as argument a Fuego.Msg.JmsMessage that is a wrapper of the javax.jms.Message. Any actions required by the BP-Method, such as instance creation, are completed.
Transitions	There are no transitions to or from the Global activity.
Tasks	No tasks can be generated. A BP-method will be automatically created with the same name as the activity. Or, if the Global Automatic activity works when an event occurs , two BP-Methods will be generated. See below for further information.

Business Process Methods	<p>Global activities can contain only one BP-Method if the activity is defined as "Polling by interval" or "Automatic Schedule". This BP-Method is automatically created with the same name as the activity.</p> <p>If the activity "Executes when an event occurs", then two BP-Methods are automatically generated. One of them has the same name as the activity and the other ends with "_Listening". The last BP-Method is the one that will be launched each time the event occurs. The first BP-Method will only execute once (at the Engine start or at the deploying time), which is used for the listener component to get initiated.</p> <p>Any application can be invoked from a Global activity, but the information should not impact on the movement of instances through your process.</p>
--------------------------	--

Global Creation Activity
What is a Global Creation Activity?

The Global Creation activity is one way to create new instances in a process.

When the Global Creation activity executes, an instance begins creation in the process. The Begin activity finishes the instance creation.

Any user-defined role in a process can contain Global Creation activities. This means that an end user can run the activity by selecting it in WorkSpace.

The Global Creation activity has an implied transition to the Begin activity in the process. The implied transition will not appear on the Process Designer workspace.

Using the Global Creation Activity

The following table outlines some of the considerations when using the Global Creation Activity within ALBPM Studio.

WorkSpace	<p>The Global Creation activity is visible in WorkSpace and is used to initiate an instance in the process.</p> <p>All Global Creation activities appear in the Optional task toolbar as well.</p> <p>When you execute a Main task you have enabled the optional toolbar.</p>
Roles	<p>The Global Creation activity must reside in a user-defined role lane.</p>
Variables	<p>Global Creation activities can access the arguments (defined in the corresponding Argument Mapping of the associated Mapping name), the predefined variables that do not require an instance (such as a process, a participant, etc.) and local variables.</p>
Pre Conditions	<p>There are no preconditions for the Global Creation activity.</p>
Post Conditions	<p>After execution, an instance flows to the Begin activity in order to finish creation.</p>

Transitions	There are no transitions to or from the Global Creation activity. There is an implied (but not drawn) transition from the Global Creation to the Begin activity.
Tasks	A task will be automatically created with the same name as that of the activity. The task can be implemented as a Method or as a Screenflow.
Business Process Methods	You should set the arguments that the Begin activity expects to receive. Based on the chosen Argument set name within the Global Creation properties, the BP-Method completes the arguments that will be passed to the Begin activity as defined in the Argument Mapping in order to be converted into instance variables.

Global Creation Activity Examples

Creating an Instance

Use Global Creation as a means of initiating an instance in a process.

Create a Global Creation activity and assign it to a user-defined role (for example, Customer). Global Creation activity can appear in the process in any number of roles, but the created instance is always submitted to the Begin activity to complete its creation.

Within the Global Creation properties, in the **General** tab, the **Argument set name** defines the argument variables required by the Begin activity.

In the example below, custName and custNumber are defined as argument variables in the Begin activity within the Argument Mapping used by the Global Creation activity. Here, the Global Creation activity prompts the end user to enter a customer name and number, which will be sent to the Begin activity once "OK" has been selected. After the end user clicks the OK button, an instance begins creation. The instance completes creation in the Begin activity. If the Argument Mapping is set for both arguments, the received information is kept in instance variables.

```
//Prompts the end user to enter the customer name and
// number.
// The defined arguments names have to be referenced
// with "arg."
//
input "Customer Name: " arg.custName,
"Customer Number: " arg.custNumber
using title = "Enter Customer Information"
```

The Begin activity argument mapping will transform the received arguments into instances variables.

Creating Multiple Instances

Within a Global creation, you can create multiple instances by using the **create ProcessInstance** statement.

```
i = 1

input "Quantity of instances: " op

while i < op do
  create ProcessInstance
  i = i + 1
```

end



Note: In Studio, the maximum number of instances to be created at the same time is 5000. This maximum number can be simultaneously executed by the Engine. Therefore, it can not create more than such number.

Global Creation Property Reference

General Properties

The following table lists the general properties.

Property	Description
Auto Complete	Routes instances automatically through the Begin Activity to the next corresponding Activity.
Argument Set Name	Defines the argument of the selected target Activity.

Grab Activity

What is a Grab Activity?



Grab activities give processes the flexibility to deal with slowdown conditions and to redistribute instances as appropriate to alleviate such conditions. Grab activities are most commonly used in supervisory roles within the process.

This enables supervisors to monitor instance flow. They can use the Grab activity to easily move instances from one activity's queue to another activity's queue.

Using the Grab Activity

The following table outlines some of the considerations when using the Grab Activity within ALBPM Studio.

WorkSpace	All Grab activity types are visible in WorkSpace to the user(s) assigned to the role.
Variables	All Grab activity types can access argument, instance, local and predefined variables from the BP-Method Editor.
Pre Conditions	Grab activities require incoming grabbed instances from other activities in the process. Defined Grab activities can only receive instances from activities to which they are attached by transition. Grab From All and Grab From All/To All activities can grab instances from any activity in the process.
Post Conditions	Defined Grab: Once marked complete, instances are sent to the next activity in the process via one of the outgoing transition lines. Grab From All : Once marked complete, instances are sent to the next activity in the process via one of the outgoing transition lines. Grab From All/To All: Once marked complete, instances are manually sent to the appropriate activity in the process by the end user.

Transitions	<p> Note: Instances that have been grabbed to one Grab activity and remain in this activity cannot be grabbed from another Grab activity.</p> <p>Defined Grab: One or more incoming transitions are required. An outgoing transition is not necessarily required. There is an implied back transition that takes instances back to the activity from which they were grabbed. This is accomplished by clicking the Ungrab button in WorkSpace.</p> <p>If a Defined Grab activity has transitions to or from activities inside of a Split-Join circuit, this same Grab activity cannot have transitions to or from activities outside the Split/Join circuit. Similarly, a Grab activity with transitions to or from activities outside of a Split/Join circuit cannot have transitions to or from activities inside a Split/Join circuit.</p> <p>Grab From All: No incoming transitions are required. The Grab From All activity can grab instances from any activity in the process. An outgoing transition is not necessarily required. There is an implied transition that takes instances back to the activity from which they were grabbed.</p> <p>Grab From All/To All: Grab from All to All can take instances from any activity and can send them (re-deploy) to any activity.</p> <p> Note: If you do not want to see the Grab transitions, disable the Show Grab Transitions property from the View menu, Transitions option.</p>
Tasks	<p>Tasks have to be defined by the developer. The grab activity has a main task. Moreover, optional tasks can be added.</p>
Business Process Methods	<p>Grab activities only have access to another activity's instances. Grab activities cannot access the BP-Method for the activity it is grabbing. However, you can assign the same BP-Method to the Grab activity that is assigned to the activity that you are grabbing instances from. For example, if you have an Interactive activity called Review Order that has one BP-Method called enterCustomerInfo, you can apply the same BP-Method to the Grab activity.</p> <p>You can add tasks to the Grab activity to process the BP-Method according to business rules. You can also ungrab the instance and allow it to go back to the activity from which it was grabbed in order to continue processing.</p>

Interactive Activity

What is an Interactive Activity?

Interactive Activities allow you to add user interaction to a process model.


Within the BSO (Business Service Orchestration), it facilitates the interaction of human participants.

It manages multiple Methods to automatically invoke components that require end user interaction. Its Method can be as complicated as needed in order to accomplish a task, and each Interactive activity can contain multiple tasks. Tasks

are a list of functions that can be performed while an instance is in the activity. Each task may or may not be required or repeatable. See Tasks for further information.

Using the Interactive Activity

The following table outlines some of the considerations when using the Interactive Activity within ALBPM Studio.

WorkSpace	<p>The Interactive activity is visible in WorkSpace, but only to the participant(s) assigned to the role where the Interactive activity exists.</p>
Roles	<p>Interactive activities must reside in user-defined roles.</p> <p> Note: If you add a new interactive activity and drop it in an automatic role, the BPM system will allow you to select an existing user-defined role or generate a new one (the Role dialog box populates.) The column-role is added to the design and the new interactive activity will automatically appear in the selected role.</p>
Variables	<p>Interactive activities can access argument, instance, local and predefined variables from the Method.</p>
Pre Conditions	<p>An Incoming instance from another activity in the process with the necessary instance variables set.</p> <p>Interactive activities can also be used as an exception handler. In this case, the Interactive activity receives an incoming instance from an activity in which the BP-Method has thrown a user-defined exception. See Exception overview for further information.</p>
Post Conditions	<p>Instance moves to the next activity in the process via one of the outgoing transition lines.</p>
Transitions	<p>One or more incoming and outgoing transitions are required.</p>
Tasks	<p>The default behavior for Interactive Activities consists in having one single Implementation or task.</p> <p>This greatly simplifies the understanding of Interactive Activities.</p> <p>Furthermore, this simplifies the operation of WorkSpace, since executing an instance is ALWAYS executing its implementation.</p> <p>However, you can have Optional (or support) tasks that the participant may need to execute in some cases in order to help him do the required work (that is, executing the main task.)</p> <p>But if you look at optional tasks in this way, they make sense ONLY if you allow the user to execute them while executing the main task.</p> <p>To do so, these optional tasks must be read-only, meaning that they must not modify the instance data in order to avoid inconsistencies.</p>

Business Process Methods	<p>The tasks have to be defined by the developer. The interactive activity has then a main task. Moreover, optional tasks can be added. See Tasks.</p> <p>Interactive activities can have one or more BP-Methods defined as tasks. Applications invoked from Interactive activities are sent incoming parameters (set by BP-Method variables) and return outgoing parameters (which set BP-Method variables).</p>
--------------------------	---

Measurement Mark Activity

What is a Measurement Mark?

Measurement marks are checkpoints in the process to measure time or business variables.

When the Engine routes the instance through a transition with a Measurement Mark, it performs all the checkpoints associated with the transition including the list of business variables which values the user wants to persist.

The information generated can be retrieved in the Audit Trail in the WorkSpace or used for the Process Data Mart.

There are different types of measurement marks:

- Snapshot Start: indicates the start of a measurement. Time begins to run. As well, business variables are persisted.
- Snapshot Stop: indicates the end of the measurement. Elapsed time is marked. As well, business variables are persisted. All Stop Measurement marks are referred to a Start one.
- Snapshot Start & Stop: persists business variables values.

Using the Measurement Mark Activity

The following table outlines some of the considerations when using the Measurement Mark Activity within ALBPM Studio.

WorkSpace	Measurement activities do not appear in WorkSpace. The information that generates can be retrieved in the Audit Trail.
Roles	Measurement activities can appear either in automatic roles or in the user defined role types. However, if the Measurement activity is in a user-defined role, it will not appear in WorkSpace.
Variables	
Pre Conditions	
Post Conditions	
Transitions	<p>Measurement marks are associated to one transition.</p> <p>When the Engine routes the instance through that transition it performs all the checkpoints associated with the transition.</p>
Tasks	
Business Process Methods	
Audit Trail	<p>Measurement marks are registered in the Audit Trail.</p> <p>They show the values of the defined business variables.</p> <p>The Stop marks show also the elapsed time since the Start mark.</p>

Notification Wait Activity

What is a Notification Wait Activity?

The Notification Wait Activity pauses a process until it receives a notification from another Activity or an external program.

The Notification Wait Activity waits for a response from one of the following:

- A Process Notification activity in an alternate process based on a Child/Parent Relationship or a Parent/Child Relationship.
- An activity using the **send** method from the **Notification** standard component.
- An external program using the Process Application Program Interface (PAPI). For further information, see **External Notification** on this page.

Instances wait in a Notification Wait activity for a Process Notification activity or an external event to send a message to the Notification Wait activity before work flow continues. The exception to this rule is that due transitions can be added to a Notification Wait activity. This means that when the time for the due transition expires, the instance flows through the due transition path instead of continuing to wait for the message or event.

Any notification for an instance that has not reached the notification wait activity yet will wait there until the target instance arrives.

Using the Notification Wait Activity

The following table outlines some of the considerations when using the Notification Wait Activity within ALBPM Studio.

WorkSpace	The Notification Wait activity is not visible in WorkSpace.
Roles	Notification Wait activities can reside in user-defined and automatic role lanes.
Variables	<p>The Notification Wait activity can define instance variables to determine the Argument Mapping.</p> <p>The different argument mappings that can be defined are the incoming information from the Process Notification activity in another process or a notification from an external application using PAPI to communicate with the Notification Wait activity.</p> <p>The chosen argument set (see Message Based transition.) can be the one that will decide the next activity for the notified instance.</p>
Pre Conditions	<p>An instance will wait at the Notification Wait activity until a subprocess' Process Notification activity sends a message back to the waiting instance or the Notification Wait activity's due transition logic expires. The subprocess' Process Notification activity should set the required arguments defined in the Argument Mapping as expected by the Notification Wait activity.</p> <p>A Notification Wait activity can also wait for an external notification, which could be from either of the following:</p> <ul style="list-style-type: none">• Any activity in any process that uses the Notification component to send notification to the Notification Wait activity.

	<ul style="list-style-type: none"> An external program that uses PAPI to notify the Notification Wait activity.
Post Conditions	When an instance leaves a Notification Wait activity, it continues on through one of the transitions leaving the Notification Wait activity following transition rules. (see Message Based transition)
Transitions	<p>Notification Wait activities typically have one or more incoming transitions and one or more outgoing transitions. They may have an outgoing Due transition to allow instances to automatically transition out of the Notification Wait if it does not receive a notification within a specified time period.</p> <p>If the Notification Wait activity has multiple sets of arguments, the Message based transitions are available. See Message based Transition.</p> <p>If the Notification Wait allows interruptions, it will reside in its own flow and will not have an incoming transition.</p>
Tasks	
Business Process Methods	No scripting is available. To pass arguments between processes, the Argument Mapping function is available.

Parent/Child Relationship

In a Parent/Child relationship, the parent process contains a Process Creation activity that calls the Child process.

- The Process Notification activity always implicitly communicates with a Notification Wait activity in the calling process. It can only communicate back to the process that called it.

Child/parent relationship

This relationship is essentially the same as the Parent/child relationship with the exception that, by adding a new Notification Wait activity to the child process and a Process Notification activity to the Parent Process, the two processes can communicate with each other. This way, communication can be bidirectional between processes.



Note: As the **Notification Wait** will expect the notification from an instance in an activity in a child process, the **Notification Wait** activity should come after a **Process Creation** activity generating an instance in the child process.

External notification

In an external notification relationship, Notification Wait activities can wait for notification from the following:

- An external program connecting through PAPI. PAPI is the Application Program Interface that allows external services to communicate with BPM processes. For further information, read the PAPI javadoc document distributed with the product.
- An activity in a process that is not a subprocess (child process.) This activity does not need to be a Process Notification activity. Using the standard component **Notification** and its **send** method, you can send a notification to any process instance from any activity type.

In the external program scenario, a process includes a Notification Wait activity that has been created using the **External event** option on the Activity General Category.

In the **Waits for:** field, **External event** is selected to indicate that the activity is waiting for an event that is external to the process.



Note: A process that is not a subprocess (child) of the process which contains the Notification Wait activity is also considered an **External event**.

Allowing interruptions

A Notification Wait activity in your process indicates that all instances should wait in the activity for a specified time period only moving forward in the process if a notification is received or if due transition logic is enacted. This may not always be the best process design. What if you want to notify the process design only under certain conditions? For example, only in the event that customer has changed or cancelled an order? This kind of scenario does not require a Notification Wait directly in the main flow of the process design.

The **Allows interruptions** check box in the **Activity General Category** dialog box is used when you need to pull instances from a process flow at any time under certain conditions.



Note: If you are performing an external notification, you must indicate the instance Id you want to pull. If the notification is generated from a Process Notification, define the **peerInstanceID** with the Instance ID in the Argument Mapping.

Notification Wait activities that allow interruptions do not reside in the main process flow (between the Begin and End activities.) Instead, the activity resides in its own flow. The allows interruptions Notification Wait is shown with an **interruption** sign such as a small circle in its center in the Classic theme or a thunderbolt in the image (BPMN theme).

When a notification is received by the Notification Wait activity, it is receiving all information about the instance that should be notified. The Engine finds the instance that matches the information received in the notification. The instance is then pulled from the main flow and routed to the Notification Wait flow.



Note: Remember to return the instance routed to the Notification Wait flow back to the main flow. In the last activity within this parallel flow use the **BACK** or **SKIP** actions in the BP-Method

Deleting a Notification Wait activity

If the Notification Wait activity's argument mapping uses correlations then you **must not** delete the activity. When you re-deploy the new process version without the **Notification Wait** activity, you loose the correlation and there might be instances waiting in that activity to be notified using this correlation.

Notification Wait Property Reference

General Properties

The following table lists the general properties.

Property	Description
Process Creation Activity	select the Process Creation activity to which the Notification Wait activity will respond. It applies if the expected notification comes from a Child process.
Wait for (event type)	<ul style="list-style-type: none"> Parent process: The Notification Wait activity waits for some kind of notification from a Process Notification activity in a parent process ("Keep relation with child" property is enabled in the Process Creation to which the Notification Wait corresponds.) Child process: The Notification Wait activity waits for some kind of notification from a Process Notification activity in a child process. The Creation activity determines the Process Creation activity within the process. To achieve this scenario, the Notification Wait activity has to be located between the Process Creation activity and the Termination Wait activity -this last one, if applicable- as Notification is expected from

Property	Description
	<p>an instance generated in a child process (the process which the Process Creation calls) ("Keep relation with child" property is enabled.)</p> <ul style="list-style-type: none"> External event: The Notification Wait activity waits for notification from an external program using PAPI (for further information, see the PAPI Javadoc distributed with the product) or an activity in an external process using the send method from the Notification component (For further information, see the Notification Component documentation.)
Allows Interruptions	<p>Allows interruptions: the Notification Wait activity will behave as an exception catching activity with a remote control connected to the instance. An activity's BP-Method in the subprocess will send a notification with unique information to identify the correct instance to notify. This notification will be taken as an interruption and the instance will move from wherever it currently is and will go directly to the Notification Wait activity unless it has already reached the End activity. For further information, see Allowing interruptions on this page.</p> <p>If this check box is selected, the Notification Wait activity appears with an interruption sign such as a small circle in its center in the Classic theme or a thunderbolt in the image (BPMN theme -). This Activity can't be in the main flow.</p>
Process notification immediately	<p>Process notification immediately: speeds up the notification process. When this option is selected, the receiving activity's argument mapping will be run upon receipt of the notification. If it is disabled, the notification will be stored in a queue and its argument mapping will be processed in the order that it was added to the queue. This option is selected by default.</p> <p>Once the Notification has been processed, any other notification for the same instance is dismissed when the instance arrives at the end. It means that if two notifications for the same instance are sent, the first one to arrive will notify the instance and the second one will have no effect, unless the instance passes through this activity again.</p>

Advanced Properties

The following table lists the advanced properties.

Property	Description
Generate Events	Defines how Auditing Events are generated for the Activity. See Audit Events on page 99.

Process Creation

What is a Process Creation Activity?

Process Creation activities are used to invoke another process asynchronously.

Process Creation activities are used for the following reasons:

- To speed completion of tasks by simultaneously running more than one process.
- To make a complex business process more easily understood by using a Process Creation activity to abstractly represent the underlying process.
- To enable reuse - called processes can easily be reused by many calling processes.
- To enable Business-to-Business (B2B) communication between processes. As a process calls another process through a Process Creation activity, the called process can be run in another company behind a firewall.
- To Balance load. You can design subprocesses to distribute their execution among different servers in order to improve the performance and balance resources load.

Though a Process Creation activity is similar to a Subflow activity in that they both call a subprocess, the Process Creation activity is slightly different from a Subflow activity in the following ways:

- It invokes a called subprocess asynchronously, which means that the calling process does not have to wait for the called process to finish before moving forward.
- It is responsible for invoking the called subprocess and the called sub process has no requirement to return to the calling process. If this is desired, a Termination Wait activity is used to capture the return.

Using the Process Creation Activity

The following table outlines some of the considerations when using the Process Creation Activity within ALBPM Studio.

WorkSpace	The Process Creation activity is not visible in WorkSpace.
Roles	Process Creation activities can reside in an abstract and in automatic role lanes.
Variables	Process Creation activity can define instance variables to determine the argument mapping.
Pre Conditions	Incoming instance from another activity in the process. The Process Creation activity's Argument Mapping can set the called subprocess' incoming arguments, if applicable, with the local instance variables. A new instance is created in the called subprocess.
Post Conditions	As the called subprocess is invoked and the 'child' instance is successfully created, the instance moves to the next activity in the calling process according to transition rules.
Transitions	Process Creation activities have one or more incoming transition(s) and one or more outgoing transition(s).
Tasks	No tasks are available.
Business Process Methods	No scripting is available. To pass arguments between processes, the Argument Mapping function is available.

Process Notification Activity

What is a Process Notification Activity?

Process Notification and Notification Wait activities work together to allow communication between processes. Process Instances pause at a Notification Wait activity until they receive notification from a corresponding Process Notification activity.

The Process Notification activity informs the instance waiting in a Notification Wait activity when the related instance has arrived at the activity. This notification releases the instance from the Notification Wait activity to continue its flow in the process.

The instances in both processes can either be related (sent by a Process Creation activity with the **Keep relation with child process** property selected) or not. If the instances are not related, some tracking means must be used (for example, a relational database) in order to match the instance ID of the instance you want to notify.

There are three scenarios where you can use the Process Notification and Notification Wait combination:

- Parent/Child relationship.
- Child/Parent relationship.
- External relationship.


See [Notification Wait Activity](#) on page 52.



Note: In order to notify an instance, it must exist within the process. Not necessarily standing in a Notification Wait activity at the time of the notification. If you plan to notify an instance that might have not yet been created, design the **send notification** to be executed with the corresponding delay. If not an exception is raised

Using the Process Notifications Activity

The following table outlines some of the considerations when using the Process Notification Activity within ALBPM Studio.

WorkSpace	The Process Notification activity is not visible in WorkSpace.
Roles	Process Notification activities can reside in user-defined and automatic role lanes. However, if they are in user-defined roles, they will not appear in WorkSpace.
Variables	<p>The Process Creation activity can define instance variables to determine the Argument Mapping.</p> <p> Note: Arguments are created in the Notification Wait activity.</p>
Pre Conditions	There must be at least one incoming transition to the Process Notification activity.
Post Conditions	When an instance reaches a subprocess' Process Notification activity, it immediately sends a notification message to the corresponding Notification Wait activity. The message contains the arguments the Notification Wait activity is expecting (defined in the Argument Mapping). The subprocess' instance then continues on through one of the transitions leaving the Process Notification activity.

Transitions	The notification will wait for the success of the delivery before going through the right transition to the next activity.
Tasks	There can be one or more incoming transition(s) and one or more outgoing transition(s).
Business Process Methods	No scripting is available. To pass arguments between processes, the Argument Mapping function is available.

Split Activity

What is a Split Activity?

A Split activity allows an instance to travel over several parallel paths at the same time.

There are two ways to accomplish several paths flow:

- Copies of the instance are created for the concurrent processing of the instance as it flows through the different paths.
- The original instance is the one that flows through the different paths.

The number of copies that the Split generates is the number of outgoing unconditional transitions plus any conditional transitions that evaluate to **true** for the Split activity.

Split activities must have a corresponding Join activity in order to complete the circuit and resume the process flow.


Activities within the Split / Join circuit cannot have any transitions to or from activities outside it. An exception to this rule is when you have a Grab activity to handle overrun conditions within the Split / Join circuit. However, in this case, grabbed instances can only be sent back to the activity from which they were grabbed or to the End activity and never to another activity outside the Split / Join circuit.

When an instance reaches a Split activity, the original instance immediately flows to the corresponding Join activity. Copies of the original instance travel through the activities inside the Split / Join circuit.

Property Reference

Advanced Properties

The following table lists the general properties.

Property	Description
Generate Copies	<p>Copies of the original instance are created as well as the instances variables are generated at Split time. Then, each copy flows independently until they arrive at the Join activity.</p> <p>If you need to share the instance variables between the different copies within the Split-Join circuit, then do not select Generate Copies.</p> <p>If copies are not generated, then the original instance is the one that will hold the instance variables.</p> <p>If any of the activities within the Split-Join circuit modify the instance variables, the change will be immediately available for all the parallel instances that are flowing.</p> <p> Note: Not generating copies becomes useful when all the outgoing paths are not modifying instance variables but using the existing information (for example, to update other systems). As you do not need to consolidate data, there is</p>

Property	Description
	no need to generate additional code in the SPLIT and JOIN nodes. Keep in mind that under this scenario if you do modify the instance variables, the Engine will lock them to update any information.
Generate Events	Defines how Auditing Events are generated for the Activity. See Audit Events on page 99.

Split N Activity

What is a Split N Activity?

The Split-n is used to copy an instance n times for processing purposes.

The easiest way to visualize how the Split-n activity operates is to picture a process that solicits bids from external vendors. The company wants to get multiple bids from different vendors and uses the BP-Method to select the lowest bid that meets the company's specifications.

BP-Method statements in the Split-n activity create the individual copy instances and set their respective instance variables. As an instance flows into a Split-n activity, the original instance automatically flows to the corresponding Join activity, while copies of the original instance are created. The original instance stays in the Join activity until all copy instances arrive. However, there are four exceptions to this rule:

- If the **number of copies to wait to release** property is set, the original instance leaves the Join activity after that number of copies have arrived to this activity.
- If there is a due transition leaving the Join activity, the original instance must follow the logic in the due transition.
- If there is a deadline for the entire process, the original instance will be rerouted to the Instance Expiration exception if such deadline expires.
- If the **action** variable is set to **release** in the Method of the Join activity, **action = release** releases the original instance from the Join activity.

The copy instances automatically inherit all the attributes of the original instance as they leave the Split-n activity. If end users add or change attachments of the copy instances, the attachments are automatically associated to the parent instance once they have reached the Join activity.



Note: You should be careful with the values of instance variables that may be changed within the Split-n / Join circuit since the original instance will take the value of the variable from the last copy instance to reach the Join activity, unless the Method in the Join activity dictates otherwise.

Activities within the Split-n / Join circuit cannot have any transitions to or from activities outside it. An exception to this rule is when you use a Grab activity that resides outside the Split-n circuit to handle overrun conditions within the Split-n / Join circuit.



Note: Grabbed instances can only be sent back to the activity from which they were grabbed or to the End activity. They can never be sent to another activity outside the Split-n / Join Circuit.

Using the Split N Activity

The following table outlines some of the considerations when using the Split N Activity within ALBPM Studio.

WorkSpace	The Split-n and Join activities are not visible in WorkSpace.
Roles	Split-n and Join activities reside in automatic roles. They can also reside in user-defined roles. However, no activity will appear in WorkSpace.

Variables

Split-n and Join activities can access instance, local and predefined variables from the BP-Method Editor.

Copies have to be created manually and not graphically as it occurs in the Split activity where each outgoing transition will produce a copy creation.

To create copies, you have to generate them in the **Split-N** BP-Method with the following code:

```
i = 0
while i < numOfCopies
do
    // create a copy of the process
instance
    copy= clone(this)

    // Get ready for next loop
    i = i + 1
end
```

Join activities can also access instance copies by using **copy**, as shown below:

```
// Set the original instance variable with
the
// copy instance variable
bidTotal = copy.bidTotal
```

Pre Conditions

Split-n activities require an incoming instance from another activity in the process.

Join activities rejoin each instance copy reaching the Join activity (if the original instance is still there).

Post Conditions

Split-n: When an instance reaches a Split-n activity, the original instance is automatically sent directly to the corresponding Join activity. While still in the Split-n activity, instance copies are created and each copy flows across the path in the Split-n/Join activity circuit. The number of generated copies will depend on the number of copies generated using the variable "**copy**".

Join: The original instance can leave the Join activity due to one of these three reasons:

- If the **number of copies to wait to release** property is set, the original instance leaves the Join activity after that number of copies have arrived to this activity. If it is not set, then,
- After all instance copies have reached the Join activity, the instance moves to the activity following the Join activity according to transition rules.
- When a copy reaches the Join activity and the Join's Method sets the predefined variable **action** to **RELEASE** (action = RELEASE).

Transitions

- When the original instance expires either because of a due transition or the process' Instance Expiration exception handling has occurred due to a missed process deadline.

Split-n activities require at least one or more incoming transition(s). Only one outgoing transition is allowed.

Join activities must have only one incoming transition in a Split-n / Join circuit. One or more outgoing transition(s) are required.

Tasks

No tasks can be generated. A BP-method will be automatically created with the same name as that of the activity.

The Join Activity associated script will be automatically generated after you check the design. At that moment the Split-Join circuit is consolidated.

Business Process Methods

The Split-n / Join activity circuit creates copies of the original instance based on BP-Method logic. The Split-n activity creates the copies as shown in the following BP-Method:

```
i = 0
while i < numOfSuppliers
do
    // create a copy of the process
instance for
    // this supplier's quote
    copy= clone(this)
    copy.supplier = "Supplier" + String(i)

    copy.supplierNum = i

    // Get ready for next loop
    i = i + 1
end
```



Note: If the instance has associated a Separated Instance variable, its value will not be automatically copied to the copies separated instance variable. Within the Split-N BP-Method, you have to assign it manually (e.g., **copy.separatedvariable = separatedvariable**).

The Join activity acts as a marshaling point where each copied instance is used to update variables in the original instance. For example, the Join activity's BP-Method will contain lines like the following:


```
// Take the copy instance variables and use
them to set
// original instance variables. Use
'copy.' .

supplierName = copy.supplierName
costQuote = copy.supplierQuote
```

Split N Property Reference

Advanced Properties

The following table lists the general properties.

Property	Description
Maximum Number of Simultaneous Copies	<p>Determines the number of copies that can flow simultaneously within the circuit. At runtime any number of copies above this maximum will be queued and later released when one of the already flowing copies reaches the Join activity or is aborted.</p> <p>For example, if you set the maximum number of copies to 10, and then the PBL generates 100, the last 90 will be queued until a copy arrives at the Join activity or is aborted. When this happens, one of the 90 copies will be released. A use case on when to define limits is if your process needs to upload 100 files and has only 5 connections available. Therefore, each file to upload is a copy (100 instance copies are generated) but the number of simultaneous uploads is limited to the number of connections, so the Maximum number of simultaneous copies is set to 5.</p> <p> Note: Any Interactive activity added within the circuit can not be Suspendable.</p>
Generate Events	<p>Defines how Auditing Events are generated for the Activity. See Audit Events on page 99.</p>

Subflow Activity

What is a Subflow?

The Subflow Activity is used to call a subprocess, which is a different process that can exist internally to your organization or in a separate organization.

Subflows allow you to do the following::

- Make a complex process more easily understood using the Subprocess activity to abstractly represent the underlying process.
- Enable reuse - Subprocess activities (and therefore the underlying processes they represent) can be easily reused by many calling processes.
- Enable Business-to-Business (B2B) communication between processes. When a process calls another process through a Subprocess activity, the called process can be run in another company behind their firewall.
- Balance load, you can design subprocesses to distribute their execution among different servers to improve the performance and balance resources load.

Using the Subflow Activity

The following table outlines some of the considerations when using the Subflow Activity within a Process.

WorkSpace	The Subprocess activity is not visible in WorkSpace as any other automatic activity.
Roles	Subprocess activities can reside in user-defined and automatic role lanes
Variables	Instance variables can be used to fill in the arguments, as constant values or expressions can be used

Pre Conditions	Subprocess activities expect an incoming instance from another activity in the process. The Subprocess activity's 'create part' sets the called subprocess' incoming argument variables through the argument mapping. A new instance is created in the called subprocess
Post Conditions	<p>The Subprocess activity's 'termination part' sets the returning argument into instance variables through the argument mapping.</p> <p>When the child instance finishes at its end activity, a notification is received by the subprocess activity and the parent instance variables are set based on the Argument Mapping. The parent instance moves to the next activity according to transition rules.</p>
Transitions	Subprocess activities have one or more incoming transitions and one or more outgoing transitions
Tasks	No Tasks are available.
Business Process Methods	No scripting is available. To pass arguments between processes, the Argument Mapping function is available

Subflow Property Reference

Defines the properties for the Subflow Activity.

Related Process Properties

The following table lists the Related Process properties.

Property	Description
Dynamic Process Invocation	
Related Process	
Argument Set Name	

Advanced Properties

The following table lists the Advanced properties.

Property	Description
Attachments can be visible to related process	
Process Notification Immediately	
Generate Events	Defines how Auditing Events are generated for the Activity. See Audit Events on page 99.

Termination Wait Activity

What is a Termination Wait Activity?

The Termination Wait activity gives an optional synchronization point in a calling process for a called subprocess only if the "keep relation with child" property is enabled in the Process Creation activity. It is always used in combination with the Process Creation activity.

The combination of having Process Creation and Termination Wait activities in a process is very similar to that of using a Subflow activity. The advantage to the combined Process Creation and Termination Wait activities is that several activities can occur between the two of them while the subprocess is running. Termination Wait activities are optional and only need to be used if you wish to halt processing an instance until the called subprocess completes.

Using the Termination Wait Activity

The following table outlines some of the considerations when using the Termination Wait Activity within ALBPM Studio.

WorkSpace	The Termination Wait activity is not visible in WorkSpace.
Roles	Termination Wait activities can reside in user-defined and automatic role lanes. However, if the Termination Wait activity is in a user-defined role, it will not appear in WorkSpace.
Variables	The Termination Wait activity can define instance variables to determine the argument mapping received from the End activity of the called process.
Pre Conditions	<p>Somewhere in the process previous to a Termination Wait activity, there must be a corresponding Process Creation with keep relation with child enabled.</p> <p>The Termination Wait activity has defined which Process Creation activity it is related to.</p> <p>Termination Wait activities expect an incoming instance from another activity in the process. Once reached, the instance will wait there until the subprocess called from the Process Creation activity reaches its End activity.</p>
Post Conditions	When an instance reaches the End activity in the called subprocess, it is returned to the Termination Wait activity in the calling process. Instance variables are updated based on the defined Argument Mapping. The instance then continues on through one of the transitions leaving the Termination Wait activity.
Transitions	Termination Wait activities have one or more incoming transition(s) and one or more outgoing transition(s).
Tasks	
Business Process Methods	No scripting is available. To pass arguments between processes, the argument mapping function is available.

Property Reference

Defines the properties for the Termination Wait Activity.

General Properties

The following table lists the general properties.

Property	Description
Process Creation Activity	<p>Selects the Process Creation activity which the Termination activity corresponds to.</p> <p>The subprocess associated to the Process Creation determines the arguments returned (found in the subprocess' End Argument Mapping.) Based on the Argument Mapping defined in the</p>

Property	Description
	Termination Wait activity, the received arguments set the instance variables defined and become available within the process.

Advanced Properties

The following table lists the Advanced properties.

Property	Description
Generate Events	Defines how Auditing Events are generated for the Activity. See Audit Events on page 99.
Process notification immediately	Speeds up the notification process. When this option is selected, the receiving activity's argument mapping will be run upon receipt of the notification. If it is disabled, the notification will be stored in a queue and its argument mapping will be processed in the order that it was added to the queue. This option is selected by default.

Adding an Activity

You add an activity in the process designer. Follow these steps to add any activity except for the three global activities (interactive, automatic, and creation).

To add an activity:

1. Click on the activity icon in the design editor toolbar (shown below) and insert the automatic activity in the first unlabeled (automatic) swimlane. Note that as you move the activity around, the transition between the Begin and End activities is highlighted in purple. Place the activity right on the transition line.



Figure 1: Activity icons on the Design Editor toolbar

The **Activity** dialog box appears.

2. Enter the name of the new activity in the **Name** field, and click **OK**.
The activity is added to the process.

You have added an activity to your process, but have not set the activity to do anything. Each type of activity supports a different set of tasks, and configuration steps vary. See the section about the activity you have added for further information.

Adding a Global Activity

Global activities are not a part of the process flow. They are called *global* activities because they do not run from within an instance. Rather, they are executed outside of, and independent from, any existing instances.

To add a global activity:

1. In the process design editor, click on the **Global Creation** (🔴), **Global Automatic** (🔵), or **Global Interactive** (🔴) icon. When you do this, do *not* hold the mouse button.
The mouse cursor will go into insertion mode in the process design editor.
2. Insert the activity in an appropriate swimlane. Remember you must place the interactive globals, like any interactive activity, in a role swimlane. The global automatic can go anywhere.
The **Activity** dialog box appears.

- 3. Enter the name of the activity in the **Name** field. Note that the Activity ID, above the name field, is automatically completed using the name you type in, but ignoring spaces. This is the name that you can use to identify the activity in PBL code.
- 4. Click **OK**.
The new global activity is created.

Editing Activity Properties

Each activity has a number of of properties and settings. You access these through the context menu.

To edit the properties of an activity:

- 1. Right-click on the activity you want to edit.
A context menu appears.
- 2. For basic activity properties, click Properties (📄). For other activity settings, see note below.
The **Properties** dialog box appears. On the left side the Category list is displayed. Standard categories are:

Category	Properties
Activity Id	<ul style="list-style-type: none">Activity ID (read only)NameDescription
General	The properties available in this category, when available, depend on the activity.
Image	You can change the icon that represents the activity. Click on Custom Image to do this. Click on Reset Image to use the default ALBPM symbol for that activity,
Advanced	The properties available in this category, when available, depend on the activity.

- 3. Set or enter each property in each category as desired and click **OK**.

Other activity settings vary depending upon the activity. Many activities have a Main Task and interactive activities may also have optional tasks. You access these tasks from the context menu that appears when you right click an activity.

For begin and end activities, you can also access argument mapping from the context menu.

General Activity Property Reference

Provides detailed information on properties shared by all Activities.

Activity ID Properties

The following table lists the general properties.

Property	Description
Activity ID	Specifies the ID used internally by ALBPM Studio and the Process Execution Engine to refer to the Activity. When you create a new Activity, the Activity ID and Name are the same. After an Activity is created, you cannot change its ID.
Name	Defines a label that describes the name of the Activity. This label is used within process diagrams and WorkSpace.
Description	Provides a general description of the Activity.

Image Properties

Property	Description
Invert Color Palette When Image is Selected	When using a Custom image, enable this option to get if you want the activity icon to be highlighted when selected.
Custom Image	Allows you to specify a custom image for this Activity.
Reset Image	Resets the image to the default icon based on your current theme.

Activity Groups

Provides information on creating and using Activity Groups.

What is an Activity Group?

An Activity Group is a compound activity. It is composed of a set of activities that may include other Activity Groups.

Activity Groups are primarily used to provide exception and compensation handlers to a group of Activities. They are also commonly used to handle timeouts for a group of Activities.

A Group can be formed by any activity (also called leaf activity) with the exception of Global, Global Creation, Global Automatic, Begin and End activities. Also keep in mind that a group cannot contain only a split or a join activity.

Using Activity Groups

Groups are useful in the following scenarios:

- You need to define a due interval within which a group of activities must be completed.
- You need to manage a specific exception that can occur in any activity within a group. (Instead of handling the exception in each activity, you can define the group and handle the exception from within it.)
- You need to reverse (compensate) a certain situation that involves more than one activity to be rolled back.
- You need to manage a group of activities as a unique transaction (Atomic transaction.)
- You need to clean up the design. Groups help you better visualize the process design since you can group a set of activities and collapse the group in order to see only one activity that represents a group of activities.

Rolling-back Work in an Activity Group

If you want to rollback the actions in a group, you should call an exception (which can be a localized exception or the process-wide exception) and then use the compensation activity.

Activity Groups and Transitions

Transitions to and from Activity Groups behave differently than Transitions to and from individual Activities.

Due Transitions

A due transition can be specified for 'leaf' activities as well as for 'group' activities.

A group can have a due transition that applies to the instance in any activity within the group. In such case, due time will begin running as soon as the instance arrives at the first activity within the group (entry point).

To define which calendar rule to apply, the engine looks for the organizational unit where the process of the instance being executed has been deployed. If no calendar rule is defined for the organizational unit, the engine keeps looking in the upper levels of the organizational hierarchy until it finds a calendar rule to apply. If no calendar rule is found, it is assumed that all days are working days. If Enterprise is installed, the Engine will also take into account the calendar rule set for the organizational unit where the process is deployed and the activity role where the instance is running. The calendar rule set at role level is first evaluated by Engine and overrides rules defined for the organizational unit, if any are defined. Refer to Calendar Rules for further information.

Group due transitions override any single activity's (within the group) due transition. If a group has a due transition and it exists inside another group that has a due transition, the instance is ruled by the outermost group's due transition if both transitions expire at the same time. For example, between an activity due transition and the group that contains the activity, due transition, the instance will be routed according to the group due transition

When an activity task is being executed over an instance, the task timeout is set to the shortest due interval of:

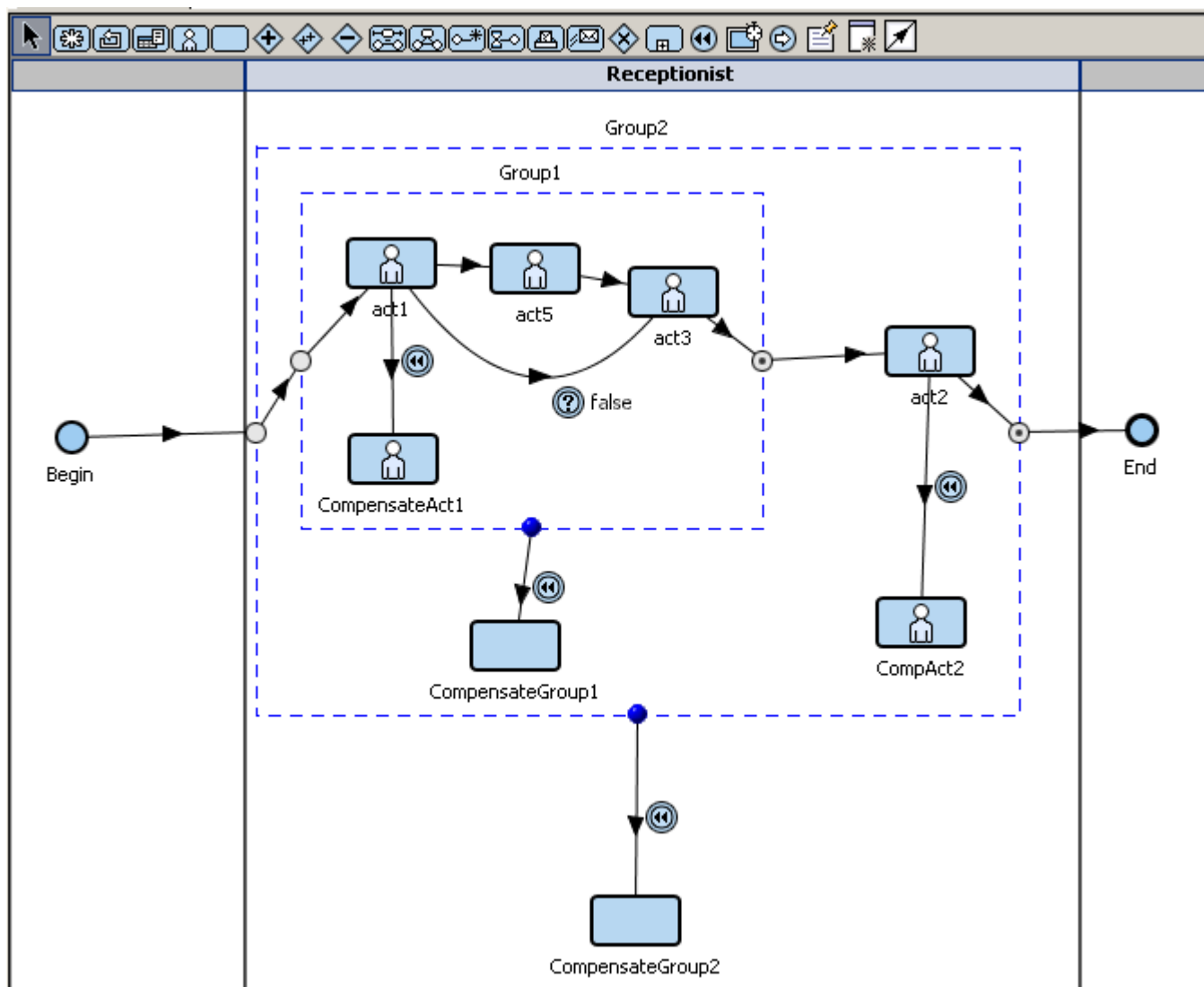
- The task timeout interval (defined as an engine property),
- BP-Method timeout interval that is currently being executed (i.e., the BP-Method that is being executed within the list of BP-Methods that might be connected with relay-to),
- The activity due transition interval,
- Each nesting group due transition (if there are multiple nested groups, all due transition are evaluated),
- The process deadline.

Compensate Transitions

Each leaf activity can have its own compensate flow. In addition, the group can also have its own compensate flow. The group's compensate transition can send the instance to a leaf activity or to a group activity that will perform any necessary action to compensate the situation. This group represents the compensate flow.

Within the group's compensate flow, you might not reference a specific inner compensation (compensation defined in some inner activity). Therefore, the engine will call all the compensations defined in the group in the inverted order in which they are executed. If it has reference to an inner compensation, the engine will call only this compensation.

In the following figure, Group 1 is an inner group of Group2. Moreover, Group1 has activities act1, act3 and act5 within it.



- If act1 needs to be compensated, the CompensateAct1 activity will be executed.
- If act3 needs to be compensated, as there is no compensate transition going out of this activity, no compensation is performed.
- If no compensate transition is defined for a Group, then all inner activities compensations are performed. They are performed in reverse order of the process flow. In the example, if there were no compensate transition for Group2, inner compensations are executed: CompAct2 and then CompensateGroup1 in this order.

A group can also be defined by grouping activities inside the compensate flow or exception flow. In this case, no exit point for the group will exist. See [Compensate Transition](#) on page 86.

Exception Transitions

Groups can call and manage their own exception flow. See [Exception Transition](#) on page 85 for more information.

Activity Groups and Grab Activities

An instance can be grabbed from any activity, regardless of whether the activity is grouped or not.


While grabbed, the instance will remain as grabbed in the grab activity, as in the following scenarios:

- A Grab activity's task is executed over the instance, or
- The instance is ungrabbed, or

- The instance is sent to an activity different from the source activity, the source activity group's property will rule the operation (i.e., due transition expiration, exception handling, compensation handling).

The source activity's due time is removed when the instance is grabbed. On the other hand, the process instance deadline is kept active. Moreover, the due time for groups nesting the source activity are also kept active (in the event that the activity is within a group that is actually within another group).

When a grabbed instance is ungrabbed, from that moment on, the instance can be processed as usual in the source activity (activity from which it was grabbed).

 **Note:** Once an instance has been grabbed, it can be sent to an activity different from the source activity but only within the same Activity Group.

Creating an Activity Group

The following procedures outline the process for creating an Activity Group.

1. Determine which Activities you want to include in the Activity Group.
2. Select the Group icon in the toolbar.
3. Select all the activities/groups within the design that should be moved to the group.
4. Right-click and select Create group with selection.
Transitions between the activities will remain and two new ones will be generated: a unique incoming transition to the group and a unique outgoing transition from the group.
The **Group** window appears.
5. Provide a name and optional description for the Activity Group.
6. Edit the Activity Group properties as necessary.
See [Activity Group Properties](#) on page 70 for more information.
7. Click **OK**.

The Activity Group is created and appears as dotted lines around the selected Activities.

Activity Group Properties

General Properties


The following table lists the general properties.

Property	Description
Is Atomic	<p>A Group can be flagged as Atomic in order to be executed as a single transaction. An atomic group is defined as a group of automatic activities all executed in one transaction. For a group to become atomic, all its Sub-Groups must be flagged as atomic. An atomic group cannot include Interactive activities or external notifications.</p> <p>Activities inside an atomic group can belong to different roles.</p> <p>An atomic group can contain ONLY:</p> <ul style="list-style-type: none">• Automatic activities.• Split/Join activities.• SplitN/Join activities.• Notification/Wait activities (only synchronization between copies.)

Property	Description
	<p>If any other activity (such as Interactive) is dropped into a group, you will be warned about it and the action will be automatically undone.</p> <p>An atomic group can contain other atomic groups. However, it cannot contain non-atomic groups.</p> <p>If a non-atomic group is dropped into an atomic group, you will be warned about it and the action will be undone.</p> <p>Atomic groups cannot handle exceptions or compensations within the group. Exceptions that occur inside the atomic group have to be handled by the group exception flow or any outer group.</p>

Advanced Properties

The following table lists the general properties.

Property	Description
Generate Events	<p>Defines how Auditing Events are generated for the Activity Group. See Audit Events on page 99.</p> <p> Note: If the group does not generate events, it will not be listed in the Audit Trail, but the activities within it, if they do generate events, will appear</p>

Tasks

The following topics provide information about using Tasks within ALBPM Studio.

What is a Task?

A task consists of one or more actions that need to be executed in order to achieve an activity's goal. Each type of activity can contain only one task, multiple tasks or no tasks at all.

There is a Main task and, additionally, some Optional tasks can be added. Activities that require human intervention can have optional tasks, such as the Interactive activity and the Grab activity. The Main task is automatically generated and the optional tasks have to be added and defined by the developer. The associated BP-Methods will also be generated. The first task in the list is the Main task, and the order of all Optional tasks is the order in which they appear on the list within the dialog box where they are defined.

Some tasks are mandatory, which means that they should be processed before the instance can be sent to the next activity in the business service. Additionally, a task can be defined as repeatable, which means that this task can be performed as many times as necessary until the instance moves to the next activity.

Tasks appear in a list in WorkSpace. A WorkSpace user selects the task to run in the order that he or she chooses.

Implementation Type

Each Task contains an Implementation Type which defines how the Task is implemented and how it behaves within a business process. You can define the implementation type within a Task's properties. Not all Implementation Types are available within each task.

Using Tasks Within Activities

Activity	Number of Tasks	Automatically generated	Description
Begin	None	None	.
End	None	None	.
Global Creation	1	Yes	.
Global Automatic	1	Yes	.
Global	1	If it is a Method	This can be a Method and therefore it is created with the same name as the activity.
Interactive	Multiple: Main and optional tasks.	No	The developer should define as many tasks as required in the activity.
Automatic	Main one	No	The developer should define as many tasks as required in the activity. No repeatable or mandatory properties will appear for this type of activity.
Split	1	Yes	.
Split N	1	Yes	.
Join	1	Yes	.
Grab	Multiple: Main and optional tasks.	No	The developer should define as many tasks as required in the activity.
Subflow	None	None	.
Process Creation	None	None	.
Termination Wait	None	None	.
Notification Wait	None	None	.
Process Notification	None	None	.

Implementation Types

The topics in the section describe the different Implementation Types within an ALBPM Task. Different Activities can contain different Implementation Types.

Method

The topics in this section provide information on using a Methods within an ALBPM Task.

What is a Method?

The Method Implementation Type allows you to define a PBL Method to perform the Task for an Activity.

A Method is a high-level scripting language used to define the business rules and the logic of activity types and certain transitions within a process. Each activity type initiates a very specific action and the BP-Method provides the script for this action. BP-Method includes statements that integrate variables, expressions, operators and components.

Using the Method Editor

The Method Editor is a freeform text editor that allows you to copy, cut and paste partial or entire BP-Methods objects, including statements, expressions, arguments, variables, operators and components

Method Timeout

Timeouts apply to BP-Method. Timeouts have a default value (5 minutes) that can be redefined using the predefined variable timeout. As well, no matter the defined timeout for each of them, they are all limited to the maximum specified in the Engine property Maximum BP-Methods timeout.

See [Method Timeout](#) on page 73 for more information.

Method Execution Results

When Methods are executed, several results may occur. The following information provides a complete list of possible instance behavior according to the result of the BP-Method. The predefined variable action is used to indicate the Method result.

The Engine acts according to this variable and saves or undoes (commits or rollbacks) the changes.

The action variable is an enumeration that accepts the following valid values:

Calling Components from Methods

A number of standard components (Library Components) are available at the time of installation for you to use in your process design. You may also add your own Java, SQL, EJB, JNDI, Web Service and BPM Objects.

The syntax for calling a component is the same, irrespectively of the type of component you are using. In BPM skin, it will be as follows:

Syntax: [method name] [component name]

Example:

For example, a method name is calculateTotal and the component name is Pricing .

calculateTotal Pricing

Method Timeout

A timeout defines the amount of time a Task will wait to complete an action before processing continues.

Default Value

By default the BP-Method has the timeout set to 5 minutes. You can change this value using the timeout Predefined Variable.

Extending the Timeout Value

This method is not recommended for all situations. Instead, it should be used only occasionally. During the BP-Method execution, several Engine resources are locked, therefore if you extend the timeout for each BP-Method, the risk of having all the resources locked increases and it may produce a bottleneck.

The syntax is as follows:

```
// timeout is an interval
// Increasing it to 20
minutes timeout = '20m'
```



Note: If you set the BP-Method timeout to be greater than the Engine property 'Maximum BP-Methods timeout', the BP-Method will fail at runtime and a Engine log message is generated.

If you set the timeout predefined variable to null, then the Engine property (Maximum BP-Methods timeout) value applies.

Timeout Limit

The Administrator can limit the timeout to a maximum for all processes deployed in a specific engine using the engine property Maximum BP-Methods timeout. Maximum BP-Methods timeout is a tool for the Administrator to ensure that

the Engine resources are enough to serve all deployed processes. If a BP-Method timeout is greater than the 'Maximum BP-Methods timeout' property, such BP-Method will fail.

The Maximum BP-Methods timeout can be set from Studio Engine Properties as well as from Process Administrator Engine Properties.

Integrated Components Within a BP-Method

When you use Components within a BP-Method, you can set a timeout as one of the component attributes. This timeout applies to the component execution time.

If the BP-Method runs one or more component, although they can have individual timeouts, the BP-Method timeout rules the full processing.

For example, the maximum BP-Method timeout is set to 3 minutes. You define a BP-Method that runs 2 components (Component A and Component B), and both components have the timeout attribute set to 2 minutes.

Component A begins running and finishes in 1 minute, 45 seconds. Although Component B has an individual timeout set to 2 minutes, its execution will not last more than 1 minute, 15 seconds as passed that time the BP-Method execution is aborted.

Components

The following topics describe the Component implementation type.

What is a Component?

Provides a general description of the Component Method Implementation Type.

The Component Implementation Type allows you to specify a component method as the Task for an Activity.

Component Task Timeout

A timeout defines the amount of time a Task will wait to complete an action before processing continues.

Default Timeout

The timeout is set in the Activity task execution timeout that works the same way as the BP-Methods timeout but applicable to the tasks implemented through components (not through a BP-Method).

By default the Activity task execution timeout property has the timeout set to 5 minutes. You can change this value if required (minutes:seconds).

Procedures

The following topics describe how to create and use Procedures within ALBPM Studio.

What is a Procedure?

Procedures provide a graphic definition of component methods that do not have interaction with end users. Procedure use a graphical syntax similar to ALBPM Studio processes.

Procedures contain a set of activities that is executed by a single participant. It also has a reduced set of activities that can be used. No roles are allowed because a procedure is limited to automatic activities.

Procedures are designed to be used in any part of a process. A procedure cannot be used outside the project that it belongs to, but it can be reused among Processes in the same Project.

As it has an automatic behavior, a procedure does not have roles and it only includes automatic activities. That is, activities of the following types: Begin, End, Process Creation with no Termination Wait activity, Process Notification, Automatic, and Split-Join). It also includes Groups and Compensate Transitions.

Automatic Activities within the procedure can have:

- Process Business Language (PBL) Methods
- Component calls (Runs on server components only.)
- Procedure calls.

When to Use Procedures

Procedures should be used in order to reuse part of a process. They are the right way to share process behavior between more than one process or inside the same process (calling the same procedure several times) instead of using IPC.

A procedure is a set of automatic activities that does not define a Process, but it defines some automatic instance behavior. For example:

- Having a set of automatic activities that perform several notifications to different third-party applications and this behavior is repeated by most of the processes within the project.
- A set of automatic activities that should be used several times within a Process. Therefore, adding several Procedure calls to the same Procedure would clarify the design.
- If your BP-Method has a large number of lines, you should consider the possibility of moving the BP-Method to be implemented as a procedure and break down the method into several activities in a graphical design.
- To graphically show a sequence of components methods to improve its comprehension.
- To hide certain details of the implementation and push some logic to another level of the design.

Procedures within a Process Instance

Once the process instance reaches an activity that executes a procedure, it remains there until the procedure finishes. The process instance is never in the procedure. The process instance actually remains in the process therefore, whatever you 'apply' in the procedure does not affect the process instance.

The procedure instance is a separate instance and the operations performed over the instance within the procedure will not apply to the process instance. For example, neither adding notes nor attachments will apply to the process instance. If you need to use any of the process instance variables data, they need to be passed as arguments.

Rolling Back Procedures

Procedures are defined as atomic. The rollback is automatically done by the Engine.

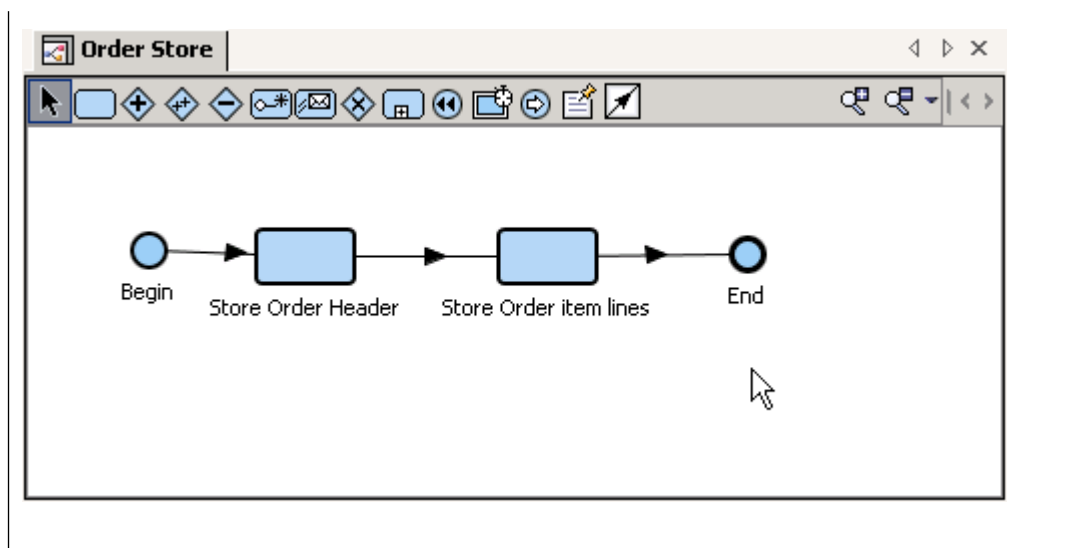
Operations guaranteed to be rolled back are:

- Transactions performed using External Resources that handle transactions such as Data Base transactions or EJB. In a BPM J2EE Engine the rollback is guaranteed by the two-face commit. In a BPM Engine Standalone, the commit is an optimist commit. This means that each external resource receives an independent commit but it is expected that neither of them will fail, but if one does, then the transaction is not fully rolled back. For example, if you have two databases with different transactions, and the second one fails, the commit performed to the first one is not rolled back.
- Update to Instance information.
- Update to variables information.

Exceptions cannot be treated inside a Procedure, they are thrown to the immediate outer group (or parent group).

Example: Storing a Purchase Order

The Order is composed of a header and line items. The first procedure activity can be the storage of the header and the second one can be the storage of each individual line item. You can implement the loading of the order in the same way-a procedure activity will load the header and another activity will load the order line items.



Creating a Procedure

The following steps describe how to create a new Procedure:

1. Right-click on the Processes resource in the **Project Navigator**.
2. Select **New Procedure**.
3. Enter a Name for the new Procedure and an optional description.
4. Click Ok.

The new Procedure opens in an editor window showing the default Begin and End Activities.

5. Add required activities to the procedure.

Procedures are available under the Process resource in the Project Navigator.

Screenflows

The following topics describe how to create and use Screenflows in ALBPM Studio.

What is a Screenflow?

A screenflow is a user interaction flow. Screenflows are similar to processes in that they are designed graphically, have a start and an end activities, support conditional expressions, and have their own instance variables.

Screenflows differ from business processes in that the entire sequence of a screenflow is executed by a single participant, so the first thing you will notice in a screenflow design editor is that there are no swimlanes.

Screenflows are called from an interactive activity. The state of a screenflow is not persisted in the process instance till the screenflow is complete and returns to the calling activity. If a participant starts to execute a three page screenflow and inputs information into the first two pages and then logs out, it is as if he had input nothing.

You can call a screenflow are called from any interactive activity of any process in the project. Use them to implement interactive tasks.

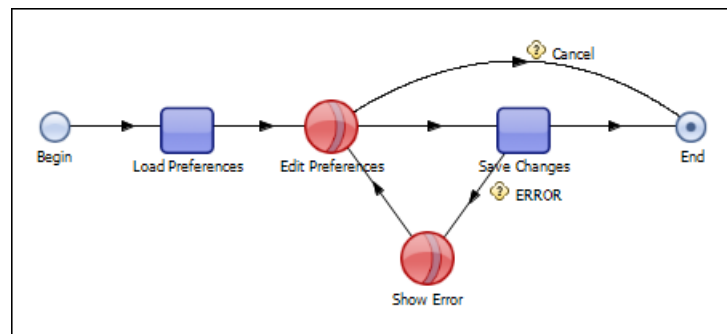


Figure 2: Simple Screenflow example. Note that there are no swimlanes.

Screenflows should be thought of as components with a process-like interface. Since screenflows are not actual business processes, they are built from a much smaller set of components. Only the following activities are supported:

- Begin
- End
- Interactive Component Call
- Automatic
- Subscreenflow

These activities have similar behavior as those in the process, but they are much simpler. In addition, screenflows support three types of transitions:

- Unconditional
- Conditional
- Exception
- Due

You should see interactive activities as a point in the process where instances get into an inbox for something to be done. The execution of this activity task will involve the invocation of interactive components. This sequence of interactive component invocations should be mapped to the screenflow's activities. Later on, this screenflow should be mapped as the implementation of the interactive activity.

When to Use Screenflows

You should use screenflows in most interactive tasks whenever possible. They have several advantages over BPM Objects with hand written interactions:

- Virtually no code is necessary to connect the different presentations.
- The flow between the screens is explicit and graphically shown.

Differences between Screenflows and Sub-processes

The most important differences are the following:

- Screenflows are executed by a single participant to complete a process task.
- They are designed to be used as a single task within a process.

Model-View-Controller Analogy

If you are familiar with Model-View-Controller (MVC) architecture, you can think of a screenflow as the controller, while the BPM Object is the model, and presentations are the view. Just as with an MVC pattern, the screenflow, acting as the controller, determines which page the user will see, and is responsible for initializing it to whatever values are required.

Screenflow logic also handles user responses. For instance, note that in the example screenflow shown above, a conditional transition is taken when the user clicks on the Cancel button in the Edit Preferences Interactive Component Call. There

are some characteristics, which do not fit exactly with the Model-View-Controller model. For example, presentations are a part of the BPM Object definition, while an ASPX or JSP page is an independent file.

Screenflow Timeout

A timeout defines the amount of time a task will wait to complete an action before processing continues.

If you deploy a process that has a screenflow, you need to consider that some timeout settings are available in the Process Administrator. You can set the BP-Method timeout and the Interactive component timeout

In a screenflow you can combine different types of activities and timeouts apply for each activity or group of them. In the following example the BP-Method timeout is set to 60 seconds and the Interactive component timeout is set to 720 minutes.

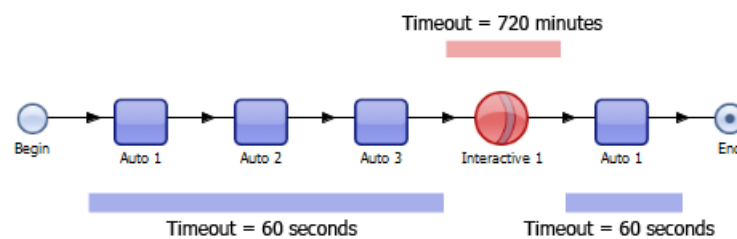


Figure 3: Screenflow Timeout Example

Once the screenflow begins to execute, the first 3 automatic activities (Auto 1, Auto 2, and Auto3), will have a *total* timeout of 60 seconds. When an interactive activity is reached, the Maximum BP-Methods timeout is reset.

Then the Interactive Component Call activity, Interactive 1, begins to execute. The user has 720 minutes maximum to complete the task.

To complete the screenflow, the Auto 4 activity executes and has 60 seconds to complete its execution.

External Tasks

The following topics describe how to access external systems within an Activity Task.

What Is an External Task?

External tasks are tasks that are completely implemented outside the BPM system.

External tasks are tasks that are completely implemented outside the BPM system. The External task implementation allows you to implement the interactive activity (more precisely the GUI presented to the user) outside the BPM system.

It is a framework to allow adding an external UI interface when executing an interactive task without using any of the BPM system's presentation capabilities (BPM Object Presentations, Screenflows, etc). These tasks are accessible from PAPI/PAPIWS by calling the methods. For further information see the PAPI's JavaDoc documentation.

For example, this is useful if you need to integrate the Engine with a .NET client.

```
prepareExternalActivity( ... )
commitExternalActivity( ... )
```

The framework functions as follows:

1. A page displays all instances (you can get this using PAPI or PAPIWS).
2. You select the instance for execution (this instance is waiting in an interactive activity).
3. To start the execution, the prepare method should be executed. This method can retrieve some information from the instance as well as leveraging retrieval of other information using the BPM system's integration capabilities. Upon successful execution of the prepare method, the instance will remain locked for that participant to prevent other participant from starting the execution for that instance.
4. You can render your own UI (ASP.NET, etc).
5. When this is submitted, the commit method should be executed. This method receives information collected from the external UI and most likely synchronize this data with instance variables. You can also perform transactions against your backend systems using the BPM system's integration capabilities. Upon successful execution of this method, the lock over the instance will be released and the task will be marked as completed.

When you select the External implementation you have to specify two methods:

- The `prepareExternalActivity()` method can be used to get any values needed by the presentation before actually displaying it.

This method extracts and processes information from the instance variables or BPM Objects, and makes it accessible through its output arguments

- The `commitExternalActivity()` method should be invoked once the user finished with the GUI and processing can continue in the BPM system.

This method completes the task (and the instance if necessary), and maps its input arguments to instance variables.

Both methods can declare input/output arguments that will be passed along whenever they are invoked. As well both methods need to be invoked through PAPI or through PAPI-WebServices.

For example:

The instance arrives to the activity and waits there until someone sends the `prepareExternalActivity()` method to it. It executes the method and answers back a set of arguments (valid values lists or predefines/default values to be used by the GUI). After the user finishes with the GUI (which could be implemented anyway you want), the `commitExternalActivity()` method is invoked and the values entered by the user are passed along so that the BPM system can use them (set them into a BPM Object for example). Then the instance moves forward in the process (unless you finish the `commitExternalActivity()` method with an Action different than OK or RELEASE).

- Configuration: you can optionally define an URL. When the task is executed from Workspace, Workspace redirects the execution to the URL.

Transitions





What Is a Transition?

A transition is the bridge between two activities.



Transitions use directional arrows that display the direction of the flow. An instance flows through a process by following the logic that applies to a transition.

Transition Types

AquaLogic BPM provides many types of transitions. The most common transitions are *unconditional*, *conditional*, and *due*. The *exception* transition is a little bit more advanced, but is also used frequently, while the *business rule* transition is new:

Transition	Description
Unconditional	Instances flow through the transition unconditionally.
 Business Rule	Instances flow through the transition if the specified dynamic business rule evaluates to <code>true</code> .
 Conditional	Instances flow through the transition if certain conditions are met.
 Due	Instances flow through the transition according to time conditions.
 Exception	Instances flow through the transition if an exception occurs.

The *compensate*, *message-based*, and *precedence* transitions are used less frequently. If you are just beginning to use ALBPM, you do not need to be familiar with these yet:

Transition	Description
 Compensate	Instances flow through the transition if compensation processing is required. The actions performed reverse (or undo) any work done in the previous activity in the event that BP-method failure occurs.
 Message-based	Instances flow through the transition if the activity that manages different argument mappings receives a message.
Precedence	Only available in a Split-Join circuit. Copies within a Split-Join circuit can have a synchronization or a precedence. The precedence is represented by a dotted transition.

Which transition is used?

All activities at least have an outgoing unconditional transition so there is always a way to continue the process. However, in most real-world processes several activities also have outgoing *conditional* transitions. In this case, the conditional transitions are evaluated first, and the unconditional transition is taken only if the conditional transitions all evaluate to *false*. In programming terms, the unconditional transition is like an *else* clause in an if-then-else construct.

Business rule transitions are evaluated before conditional transitions, so if a business rule transition and a conditional transition both evaluate to **true**, the business transition is used.

Due transitions act separately. They "pull" the instance from the activity as soon as a time condition is met. In this case, all other outgoing transitions are ignored.

Transition Types

Describes each type of Transition.

Unconditional Transition

Provides detailed information on the Unconditional Transition.

When your process requires unrestricted workflow between two activities, you should add an Unconditional Transition. This type of transition indicates that no conditions exist to prevent instances from moving to the next activity. Therefore, the transition occurs unconditionally.

After you create the transition, a line with a directional arrow connects the two activities on the design workspace. No icon is displayed next to the transition.



Note: If you do not want to see the Unconditional Transitions, disable the Show Unconditional Transitions property from the View menu, Transitions option.

Rules

The following rules apply to unconditional transitions:

- Each activity must have at least one **outgoing unconditional transition**. Exceptions to this rule are the following:
 - if the activity has a **due transition**. The **Split and Split-N activities** are the only ones that **must** have an unconditional transition, as an exception to this rule.
 - Global Creation, Global, Global Automatic and End activities have no outgoing transitions.
- Each activity must have only one outgoing unconditional transition. Exceptions to this rule are:
 - Split activities can have more than one outgoing unconditional transition.
 - Interactive activities may have more than one outgoing unconditional transition if the **User selects transition** check box is selected on the Interactive's **Activity Property** dialog box.
- Activities cannot have an unconditional and a conditional transition to the same destination activity.

Business Rule Transitions

Business rule transitions are a special kind of conditional transition which evaluates a dynamic business rule instead of an expression. Business rule transitions are evaluated before conditional transitions.

Conditional Transition

Provides detailed information on the Conditional Transition.

When your process requires restricted workflow between two activities, you should add a **Conditional transition**. A Conditional transition indicates that workflow will only occur if specified conditions are met. The special conditions are added by using a BP-Method in the **Condition** field in the **Transition Properties** dialog box.

For example, in an Order Management process, a conditional transition directs instances from the **Review Order** activity to the **Special Care** activity if the order status is equal to "Expedite" or "Alert". The BP-Method for this condition is as follows:

```
orderStatus in ["Expedite", "Alert"]
```

After you create the transition, a line with a directional arrow connects the two activities on the design workspace. The icon next to the transition indicates that it is a **Conditional transition**.

As well, the transition's **Name**, **Description** or **Condition** may appear next to it depending on the chosen preference.



Note: If you do not want to see the **Conditional transitions**, disable the **Show Conditional Transition** property from the **View** menu, **Transitions** option.

Rules

The rules for using conditional transitions are as follows:

- Conditional transitions are available for most activities, with the exception of End and Split-N. (Global, Global Creation and Global Automatic do not require transitions.)
- Activities cannot have an unconditional and a conditional transition to the same destination activity at the same time.

Defining the Condition

The **Condition** is defined in the Transition Properties dialog box by typing a BP-Method in the **Condition** field. By default the transition's **name** is used as the expected **result** of the condition.

For example if the transition represents the normal flow then you can **name** it as **OK** and the condition is automatically built as **result == "OK"**. This is the default condition and is valid while the condition is empty.

If you manually define a condition then the default condition is no longer valid.


Instance variables can be used in the condition as well.

More than one conditional transition might be required. Multiple conditional transitions flowing out of an activity are ranked in order to determine the evaluation precedence. The precedence is assigned in ascending order according to the creation order given when the process is first designed in Studio. Nevertheless, the conditional transitions' order can be changed by using the **Conditional transitions order** properties window that Studio displays in the activity shortcut menu (right-click on the activity) when more than one conditional transition has been defined.



Note: it is highly recommended to **name** the condition to represent its condition. Furthermore, use the predefined variable **result** to set the result of it.

- The syntax is automatically checked. If everything is correct, a **blue flag** appears on the upper-right corner of the **Conditional** field. If something is incorrect, a **red flag** appears. Drag the mouse over the red line below the red flag and the error displays. In addition, the error is shown at the bottom of the dialog box if you click on the statement that has the problem.
- Click **Ok** to close the Transition Properties dialog box. When the dialog box closes, the last check is performed. If something was not corrected, the error message will display.


 **Note:** It is not recommended to use a variable type ANY because, in order to compare it, you will have to cast it before. If this is not done, the comparison might fail.

See Process Business Language Programming Guide for further information.

Due Transition

Provides detailed information on the Due Transition.


A Due Transition is used when a process requires an instance to move to the next activity within a specified time period. Due transitions are used to implement deadline processing.

 **Note:** Activities can use only one due transition to link to another activity.

After you create the transition, a line with a directional arrow connects the two activities on the design workspace.

The icon next to the transition indicates that it is an **Interval expression or Interval constant due transition**. The due condition is displayed as well.

The icon next to the transition indicates that it is a **Scheduled based due transition**.

 **Note:** If you do not want to see the Due Transitions disable the **Show Due Transitions** property from the **View** menu, **Transitions** option.

Defining the Due Condition

The Due Condition can be defined as Schedule based, Interval expression, or Interval constant.

In any of these cases, you can decide to use Calendar Rules. So, if the calculated due date is not a working day, the applicable due date is moved to a valid date.

Schedule based

If the next scheduled due time falls on a non-working day, you have the ability to select whether the due time is passed to the **Next working day, same time** or **Next scheduled based time**.

For example, if you set the due time to happen once a week, every Monday at 11:00 AM and one Monday is a holiday, then if you have selected:

Next working day, same time the due time is set to Tuesday at 11:00 AM, or


Next scheduled based time the due time is set to next Monday at 11:00 AM.

The Scheduled option indicates that the due transition is set to run on a specific date at a specific time.

The **DAILY** choice allows you to select a time you want the due time to apply each day. For example, every day at 3:00 AM.

The **WEEKLY** choice allows you to select a "**day of the week**" and a time that the due time applies every week. For example, every **Monday** at "**10:18 A.M.**"

The **MONTHLY** choice allows you to select the month, the week of the month (First-Second/Fourth-Last)/the day of the week (Sun-Sat) or the day of the month and the day (1-31) and the time to which the due time applies. For example, you can choose the third (3rd) Thursday of every month at 3:45 P.M.

 **Note:** Notice that the due is moved based on the day and not on the time as explained on the dialog box

Interval Expression

The Interval Expression option allows the due time to begin the moment the instance arrives at the activity plus an interval time. The interval time to add to the arriving time can be the result of an expression.

If the calendar rules apply and the calculated due time is on a non-working day, the due time is postponed to the first working time.

- In the Due Interval field, type a time interval, making sure that you surround the time interval with a single quote. (See below for time syntax examples.) For example, if you require a time interval of five minutes, you need to type '5m'. This means that the instance has five minutes (5m) to flow through the Due transition from the source activity to the next activity. To implement this example, you can also define the Due transition as an Interval constant (see below for more information).

If you require a more complex condition, add a BP-Method in the Due Interval field to force an instance through the process in a specified amount of time under certain conditions. In the following example, the BP-Method indicates that an order greater than \$5000 can sit in the Account Manager's queue for only 2 minutes ('2m'). If the order is less than \$5000, the order can remain in the queue for 5 hours ('5h').

```
(OrderAmount > 5000) ? '2m' : '5h'
```

- The syntax is automatically checked. If everything is correct, a green flag appears on the top right corner of the Conditional field. If something is wrong, a red flag appears. The error is shown at the bottom of the dialog box if you click on the statement that contains the problem.



Note: Time will start running immediately after the instance reaches the source activity of the due transition.

Time syntax

Due interval logic to indicate an interval of time is added to the due transition properties. The syntax is:

- d for day
- h for hour
- m for minute
- s for seconds

Examples of valid intervals to enter in Due transition logic include:

- '3d ' for three days
- '1h' for one hour
- '4m' for four minutes
- '2m30s' for two minutes and 30 seconds.

See METHODS REFERENCE GUIDE for further information on time and interval syntax.

Interval Constant

The Interval Constant option allows the due time to begin the moment the instance arrives at the activity plus a defined time (number of months/days/hours/minutes/seconds).

If calendar rules apply and the calculated due time is on a non-working day, the due time is postponed to the first working time.

For example, you decide that the due time is always the instance arrival time plus 7 days and 12 hours.

No expressions are available. If so, define the due time as an Interval expression.



Note: Time will start running immediately after the instance reaches the source activity of the due transition.

Priorities for the Due times

A due transition can be specified for leaf activities as well as for group activities. Therefore, an instance can be affected by multiple due times as follows:

- the activity due transition interval,
- each nesting group due transition interval,
- the process instance deadline.

When an instance arrives at an activity, these three options are considered in order to determine the shortest due time and which due transition is first priority.

You should consider that a group's due transition time begins when the instance arrives at the first activity within that group.

The due time assigned to an instance, as it arrives at an activity, is the shortest time of all the three possible due time options listed above.

For example, as shown in following figure:

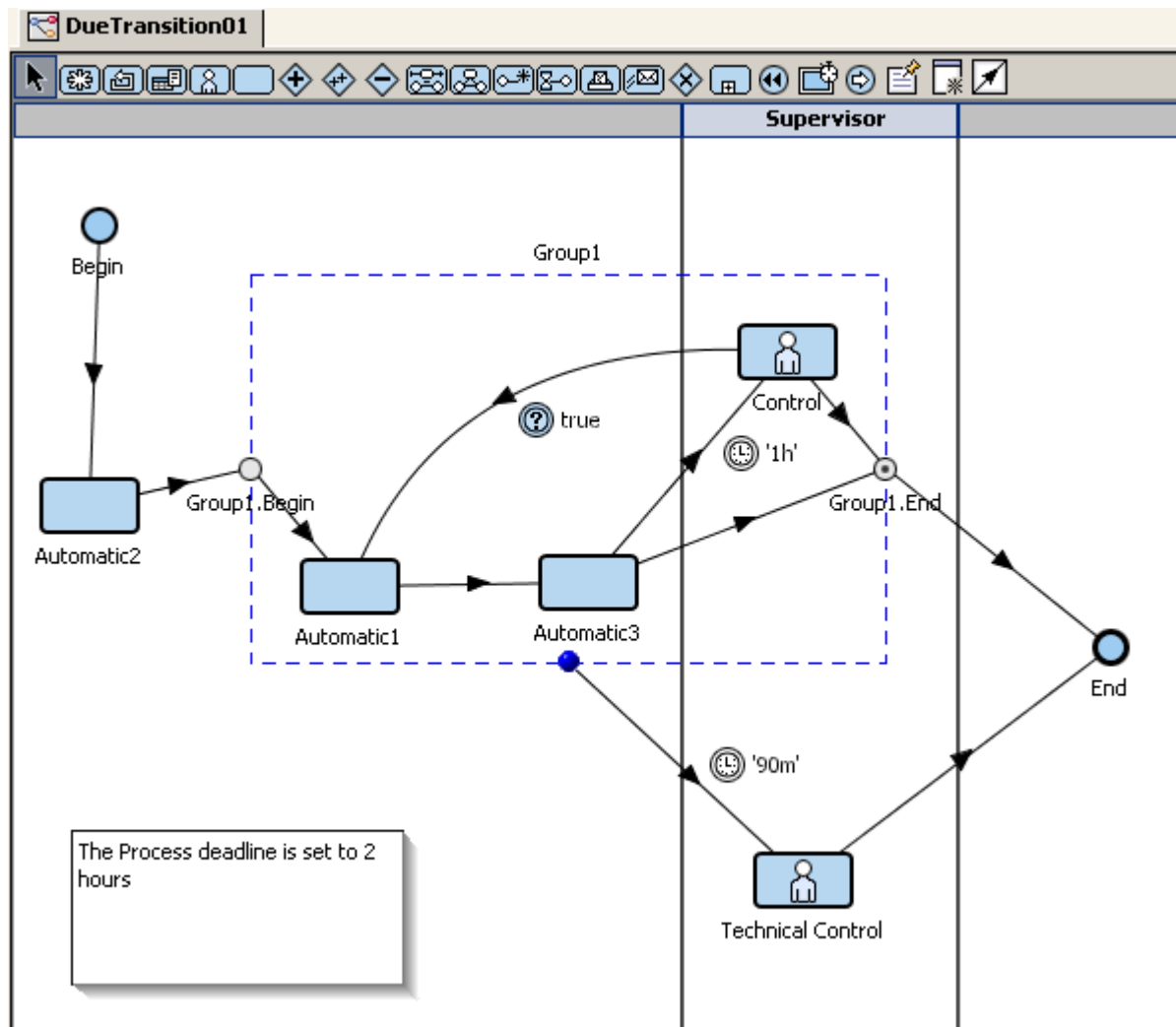


Figure 4: Due Transition Example

when the instance arrives at the activity **Automatic3**, as this activity has a due transition, the due time is the shortest time between:

- 1 hour: defined in the Automatic3 due transition.
- The remaining time of 90 minutes as defined for the Group1 due transition. This means that if the instance has remained for 50 minutes in the Automatic1 activity, the remaining time for the group is 40 minutes. In consequence, the instance will probably flow through the Group1 due transition to the Technical Control activity instead of flowing to the Control activity (1 hour).
- The remaining time of 2 hours defined as the process due deadline. If the instance has been in the Automatic2 Activity for 100 minutes, the remaining time for the process is 20 minutes. Thus, the instance will probably flow to the End Activity instead of flowing to the Technical Control activity (90 minutes) or to the Control activity (1 hour).

If more than one due transition expires at the exact same time, the instance is sent through the outermost due transition (of those transitions which time interval expired at that moment).

For example, the instance is within an activity that has a due transition that expires at noon. Additionally, this activity belongs to a group that has also a due transition that expires at noon. At noon the instance will flow through the group due transition.

When a due interval is calculated for an activity (leaf or group), the calendar rules are taken into account as regards the role where the instance is or whether the group spreads over many roles.

Task Timeout

If a task is running, the time it has to complete depends on the task timeout, defined with the predefined variable timeout or 5 minutes by default. But the due time explained above also applies. Therefore, if there is a task running for a specific instance and any of the due time at activity, group or process level expires, although the task might have some remaining time, the task will finish and the instance will flow through the corresponding due transition.

Moreover, if the task timeout was set with an interval greater than specified as the Engine property "Maximum task timeout", the task will fail until the process or maximum task timeout is fixed.

Continuing with the example image above, the Automatic1 activity has a task that has set a timeout to 30 minutes. And the instance has been in the Automatic2 activity for 100 minutes, the remaining time for the process is 20 minutes. So, although the task theoretically has 30 minutes to execute, after 20 minutes the process due deadline expires, the task is terminated and the instance flows to the End activity.

Due Time Calculation Failure

If a due time calculation fails, the transaction of the activity from which the instance came from, fails.

For example, if you send an instance from an Interactive Activity to an Automatic Activity that has a due transition but its calculation fails, the previous **Interactive** activity execution fails.

Exception Transition

Provides detailed information on the Exception Transition.

An Exception Transition is used to submit the instance to an Exception Handler flow when the activity or group of activities fail or throw an exception.



Note: Activities or groups can use only one exception transition to link to the exception flow. The exception flow can have as many activities as required to fix the exception condition.

After you create the transition, a line with a directional arrow connects the two activities on the design workspace.

The source of the transition is the activity or group where the exception occurred, and the target is the first activity in the exception handler flow. As the exception handler flow is independent from the main process flow, it cannot have transitions back to the main process flow.

Exception holder variable: you can create and define an instance variable as the **Exception holder variable**. If you do not define one, Studio creates it automatically.

Once the exception occurs it is stored in this variable and it is available within the exception flow that is handling the exception. It can be used by the developer to debug or analyze the exception in depth.

You can create only one variable and re-use it in all the exception transitions or you can associate different instance variables to each exception flow. Normally the variable type matches the exception type.

If there is no variable defined (backward compatibility), a default exceptionHandler instance variable (Any type) is created when you check the design.

At runtime if the variable receives an unmatching type, then a warning is logged.

The variable content is available only within the Exception flow to where the instance is routed through the exception transition. Therefore the exception contained in the variable is not propagated to other processes.

Compensate Transition

Provides detailed information on the Compensate Transition.

A Compensate Transition is used when an activity or group of activities require that the actions performed by the BP-methods should be reversed. Reversal is needed in case of total or partial BP-method failure, which could be caused by any number of things such as a call to an external system that fails, equipment failure, a database call with bad data and so on.



Note: Activities or groups can use only one compensate transition to link to the compensate flow. The compensate flow can have as many activities as required. However, no other compensate transition between them applies.

After you create the transition, a line with a directional arrow connects the two activities on the design workspace.

The source of the transition is the activity to be compensated and the target is the first activity in the compensation handler flow. Since the compensation handler flow is independent from the main process flow, it cannot have transitions back into the main process.

For further information, see [Compensate Handling](#) and [Compensate Activity](#).

Message Based Transitions

Provides detailed information on the Message Based Transition

Message Based Transitions are available for the Begin and Notification Wait activity types. A Message Based Transition can be added by using the source activity argument mapping sets.



Begin and Notification Wait activities can receive different sets of arguments. Each set has a name defined as the Argument set name. Within each set, the arguments can be mapped to instance variables or predefined variables. For any of these mappings, a new outgoing transition can be added to the activity. The transition type is called Message Based Transition. Basically, you define the transition that the instance will flow through based on the received message.

There cannot be more than one outgoing message based transition for the same Argument set name.

Adding a Transition

You add transitions in the process design editor. This task shows you how to add any type of transition, including unconditional transitions.

To add any transition:

1. Click on the **Add Transition** icon ().
2. Click on the activity the transition will originate *from*.
3. Click on the activity the transition will flow *to*.
An unconditional transition is added to the process diagram.
4. To change this transition into another type of transition, right-click on it and click **Properties** ().
The **Transition** dialog box appears.
5. Enter a name for the transition in the **Name** field.
6. Click on the **Properties** tab.
The **Properties** page appears.
7. Select a transition type from the drop-down list in the **Type** section.
A section showing properties corresponding to the transition you selected type will appear.
8. Specify the data or options as required, and click OK.
The transition is changed.

Adding a Business Rule Transition

You can add business rule transitions directly from the activity context menu in the process design editor.

To add a business rule transition:

1. In the process design editor, right-click on the activity *from* which the business rule transition will flow, and click **Add business rule transition** (💰).
2. Click on the activity the transition will flow *to*. Note that as you do move the mouse, the transition line is shown. The **Transition** dialog box appears.
3. In the **Name** field, enter a name for the business rule transition.
4. Switch to the **Properties** page of the **Transition** dialog box.
5. In the **Select Business Rule** section, select a business rule from the drop-down list.



Note: If you have not yet defined any business rules in the project, click **New** and give the business rule a name. After completing this task, you must edit the business rule following the steps in [Editing a Business Rule in Studio](#).

6. Click **OK**.

The business rule transition is added to the process diagram.

Adding a Conditional Transition

You can add conditional transitions directly from the activity context menu.

To add a conditional transition:

1. In the process design editor, right-click on the activity *from* which the transition will flow, and click **Add conditional transition** (?).
2. Click on the activity the transition will flow *to*. Note that as you move the mouse, the transition line is shown. The **Transition** dialog box appears.
3. In the **Name** field, enter a name for the new transition.
4. Switch to the **Properties** page of the **Transition** dialog box.
5. Enter a conditional expression using PBL syntax. This expression should evaluate to a boolean (a value which is either `true` or `false`).

The condition can use instance variables. To see the instance variables which are available, click on the **Instance Variables** icon.

The following illustrates valid and invalid conditional expressions, where `total` is an instance variable of type `Decimal`:

Valid Expression	Invalid Expression
<code>true</code>	<code>"true"</code>
<code>total > 2500</code>	<code>2500</code>
<code>total > 0</code>	<code>total</code>

6. Click **OK**.

The conditional transition is added to the process diagram.

Adding a Due Transition

You can add due transitions directly from the activity context menu in the process design editor.

To add a due transition:

1. In the process design editor, right-click on the activity *from* which the transition will flow, and click **Add due transition** (📅).
2. Click on the activity the transition will flow *to*. Note that as you move the mouse, the transition line is shown. The **Transition** dialog box appears.
3. In the **Name** field, enter a name for the new transition.

- 4. Switch to the **Properties** page of the **Transition** dialog box.
- 5. Select an expression type, which can be *Schedule Based*, an *Interval Expression*, or an *Interval Constant*.
- 6. Enter the data required and click **OK**.

The due transition is added to the process diagram.

Variables

What is a Variable?

Variables are placeholders in memory for values in your process. Each variable has a name, description, type and value. ALBPM used different categories of variables based on the scope and context where they are used.

The following table describes the categories of variables in an ALBPM Project:

Variable	Description
Instance Variables	Store variable information across a process and are passed from one activity or task to another.
Project Variables	Store information across a project. When a project is published, Project Variables become available externally. If a Project Variable is defined as a Business Variable, it can be used to store information with the Process Data Mart.
Argument Variables	Function as an interface to instance variables so that they may be passed to or from external components or between processes.
Predefined Variables	Provide predefined information to set or get information from a process instance. Each activity encompasses a specific action. Therefore, some activities accept more kinds of variables than other activities.
Local Variables	Store information within a PBL Method. Local Variables are only available within BP-Method task where the variable is created.

Order of Precedence within a PBL Script

The order of precedence for Variables is:

- 1. Local
- 2. Argument
- 3. Instance

When accessing variables within a Method, you should be aware of the order of precedence among different types of variables. This is particularly important if you have variables of different types that have identical names.

In situations where you have overlapping variables names of different types or when you want to explicitly state the scope of a variable, you can use the following keywords:

Keyword	Description
this.	Explicitly specifies an instance variable.
arg.	Explicitly specifies an argument variable.

Variable Types

Provides a list of available Variable Types in ALBPM Studio.



Instance Variables

Instance Variables are user-defined variables whose scope is contained within a process. Instance Variables contain information that flow through the process from the Begin to the End activities. All variables are stored independently for each process instance.

Most Activities within a Process can access and modify the value of an Instance Variable, with the exception of Global Creation and Global Automatic activities. Global Interactive activities can access instance variables only when the **Has Instance Access** property of the activity is checked.

Instance Variable Categories

Instance variables have a property called *category*. This property determines how the Process Engine stores the value of the variable in the database, but does not affect the logic of the process. This property takes one of the following values:

Category	Description
Normal	<p>By default instance variables are defined as <i>normal</i>, and only those that have some special characteristics need to be categorized in a different way. For each process instance, the process execution engine stores all normal instance variables together by serializing their values into a single BLOB in the process engine database. After the execution of each process activity, the process engine updates this BLOB in the database.</p> <p>To keep resources under control, the process engine limits the size of this BLOB for each process instance. The Maximum Instance Size property of the Engine defines this limit (in kilobytes), and is configurable from the Process Administrator application. The default value is 16 KB.</p>
Separated	<p>The process execution engine stores separated variables in a separate table in the database.</p> <p>The Separated storage category is intended for variables which must hold large amounts of data (10 KB or more) and are used (and modified) rarely in the process.</p> <p>A common use case for separated variables is when the input to the process is a potentially big XML document, which must be parsed only once and is not modified throughout the process. If the variable holding the XML is defined as Normal, every time the process instance flows from activity to activity the XML is serialized to the database along with the other Normal variables (even if the XML was not modified). By defining the variable as Separated, the Process Engine updates Normal variables independently, updating the XML only if it was modified.</p> <p> Important: In Split-N activities, separated instance variables values are not automatically copied to the separated instance variables of the copies. They have to be copied manually in the Split-N PBL method.</p> <p> Note: The size of a BLOB for Separated variables is only limited by the underlying DBMS used by the Process Execution Engine. Mind though that storing big data elements as instance variables (Normal or Separated) is not recommended as it has a negative impact on Engine performance.</p>

Default Values

Instance Variables are initialized based on their type according to the following table:

Type	Default Value
Numeric (int, real, decimal)	0

Type	Default Value
Boolean	FALSE
All other types	Null

Accessing Instance Variables

All PBL methods of the process can refer to any instance variable by the variable's name. You may also use the explicit `this.` prefix to avoid naming conflicts with variables defined at another scope (i.e.: argument or local variables).


Project Variables

Project variables are instance variables defined with project scope. Unlike regular (process) instance variables, project variables are stored in their own columns in the instance table of the process execution engine's database.

Because they stored in their own table column, project instance variables have special advantages and limitations in comparison to process instance variables:

- You define a project variable once for a given project. The characteristics of a project variable, such as name, type, or length, will be the same in any process.
- You can view project variable values in the Workspace **Work List** panel. Each project variable will show up as an available column, and the user will be able to select it as a presentation column.
- You can perform a search or to sort instances based on the value of a project variable.
- Fewer data types are available for project variables, as described in the Project Variable Data Types section below.

Project variables are defined during the development phase. After you have defined a project variable, you should avoid changing its type or name.

 **Note:** Each project variable adds some overhead, so you should avoid using too many project variables. It is recommended that you use fewer than 10, but the maximum limit is 256 project variables in a project.

Defining Project Variables as a Business Indicator

You can define a Project Variable as a Business Indicator. A Business Indicator is used primarily to generate Business Activity Monitoring (BAM) and Business Activity Data Mart information.

Project Variable Data Types

Project variables support fewer data types than regular instance variables. For example, neither BPM Objects nor binary values can be stored in project variables. A project variable can have one of the following data types:

Data Type	Remarks
Bool	
Decimal	Up to 255 digits
Int	
Real	
String	Up to 255 characters
Time	

Argument Variables

Provides general information on Argument Variables.

Argument variables act as an interface. You can transform instance variables with argument variables so that they may be passed to or from external components or between processes. Arguments variables can be defined as:

- In
- Out
- In/Out

For each activity, the valid options depend on how arguments can be treated in each case. Therefore, arguments can be in each activity as follows:

- Interactive, Grab, Automatic, Global, Global Creation: **In/Out**
- Subflow: **Out** for the create BP-method, **In** for the wait BP-method.
- Process creation, End, Process Notification: **Out**
- Termination wait, Begin, Join, Notification Wait: **In**
- Split, Split-N, Conditional: N/A

Argument variables can be considered as External or as Internal based on what they were defined for.

External

The first type of arguments are those defined in the activities (Begin, End and Notification Wait) that can deal with external components of the process. These external components could be an HTML form, a Java program, another process, Web Services, and so on. "External" Argument variables are the interface to a process. They are responsible for passing variables from a process to an external component and receiving variables from external components into the process. Arguments can be grouped in argument sets.

Two BPM processes that need to communicate with each other use external argument variables to pass the variables between the processes. One process, the parent process, passes variables through a Subflow activity to another process (a subprocess or child process). The child process's Begin activity receives these argument variables and maps them to instance variables. When the child process has finished processing the instance, it passes the instance variables to the End activity, which then maps the instance variables back to argument variables to pass back to the Subflow activity in the parent process.

Internal

The second type of argument variable is internal to a process. It is available for BP-Methods used within the process (for example, a BP-Methods that receives arguments and can be invoked from another BP-Method assigned to an activity task). In either case, argument variables appear as follows in the Method Editor:

```
arg.myArgument
```

The following sets an argument variable equal to an instance variable as follows:

```
arg.myArgument = myInstance
```

The following sets an instance variable equal to an argument variable as follows:

```
myInstance = arg.myArgument
```



Note: The usage of the **arg** keyword is optional. However, it is necessary when you need to distinguish a local or instance variable that has the same name as an argument variable.

For example, suppose that there is a method that contains a local variable called **orderNo**. The method also receives an argument called **orderNo**, and the process has an instance variable called **orderNo** as well. In order to distinguish the usage of each variable you will need to qualify it with the corresponding keyword:

To refer to the instance variable you must type:

```
"this.orderNo"
```

To refer to the argument variable you must type:

```
"arg.orderNo"
```

Local variables have no qualification.

Copy Argument

There is a predefined argument called `copy` that is only available when writing a Join activity BP-Method. `copy` represents the instance-copy that arrived at the Join activity.

In the corresponding Split of the Split-Join circuit, if copies are created, they can be processed in the Join activity upon arrival. The way to reference copies is by using the `copy` argument keyword.

`copy` is used to access the values of the variables of the copies of an instance in the Join activity.

```
// Set the original instance variable with
// the copy instance variable.
orderTotal = copy.orderTotal
```

Predefined Variables

Provides a listing and description of Predefined Variables.

Predefined variables are global to all parts of the process and, just as their name implies, they have already been defined. Most of the predefined variables deal with instances as they relate to activities and are used to keep track of an instance and its status as it flows through a process. Some predefined variables are modifiable and others are not. The following table lists the predefined variables included in Studio.

Predefined Variable Reference

The following table lists the Predefined Variables:

Predefined Variable	Permitted Values	Type	Description
action	Modifiable : OK ; FAIL ; RELEASE ; CANCEL ; REPEAT ; ABORT ; BACK ; SKIP ; NONE	Fuego.Lib.Action	Marks the action to be performed on a process instance as a result of previous BP-Method statements. See Action Variable on page 95
activity	Read only	Fuego.Lib.Activity	Returns an object that is the current activity.
activity.deadline	Read only	Time	Set automatically if there is a Due transition going from the current activity (receptionTime + the due transition time interval.)
activity.source	Read only	Fuego.Lib.Activity	The activity from which the current instance came into the current activity. Null in the case of Global and Begin activities.
attachments	Read only	Fuego.Lib.Attachment[]	Array of the instance attachments.
children	Read only	String[ordered Object]	Array of the Ids of the instances that are children of this instance, ordered by the list of activities that created them.
creation. participant	Read only.	Participant	Participant that created the process instance.

Predefined Variable	Permitted Values	Type	Description
creation.time	Read only	Time	Creation time of the instance.
currentException	Read only	String	Name of the exception, if the instance is within an exception flow, which was given to it by the throw statement .
deadline	Modifiable.	Time	The instance will expire if it is not completed before the specified time.
description	Modifiable: Any string that does not contain special characters	String	Name of the instance that appears in WorkSpace. By default, it is ProcessName+ id.number.
id.id	Read only	String	An instance's identifier, which uniquely identifies an instance. It includes the deployed process name, organization, organizational unit and the instance Id (including thread id).
id.number	Read only	Int	A number that uniquely identifies an instance within an Engine.
id.copy	Read Only	.	The current instance thread number.
notes	Read only	Int	An array of all notes added to an instance.
organization	Read Only	String	Name of organization where the process is running.
organizationalUnit	Read Only	String	Name of the process' organizational unit where the process is running, such as Marketing or Finance.
parent.id	Read only	String	The Id of the parent instance of the current instance if there is one; NULL otherwise. In the case of a Procedure, it contains the id of the calling process
parent.copy	Read only	Int	The parent instance thread number.
parent.number	Read only	Int	A number that uniquely identifies the parent instance within an Engine.
participant	Read Only. Any participant	Fuego.Lib.Participant	The human participant stored in directory services that is currently processing the instance.

Predefined Variable	Permitted Values	Type	Description
participant.locale	Read Only	Fuego.Util.Locale	The language, country and variant, if applicable, the participant has set.
participant.next	Modifiable: Any participant	Fuego.Lib.Participant	The participant to which the instance will be sent next. Note: If the instance is grabbed, the participant.next is cleared. If the instance is sent BACK or SKIP from an Exception flow or from an Interruption, the only way to reset the participant.next is using the Unselect method from the Participant component. See Participant component documentation for further information.
participant.sticky	Modifiable: Bool	Fuego.Lib.Participant	If set to true, the participant.next is set as the preferred participant. Each time the instance moves to an activity and the participant.next is enabled to work with it (belongs to the role where the activity is), the instance is submitted to the participant.next work queue. In the Split and SplitN activities, all generated copies maintain the participant.sticky value from the original instance. In the SplitN method you can change its value. Note: if the participant.next is changed during the process, be sure that participant.sticky is reset unless the new participant.next is considered the preferred participant.
priority	Modifiable: 1 Lowest ; 2 Low ; 3 Normal ; 4 High ; 5 Highest	Int	Priority of the instance.
process	Read Only. Any process	Fuego.Lib.Process	The process the instance belongs to.
process.id	Read Only	String	The Identifier of the deployed process containing the deployed process' name, its organization and organizational unit.
process.idNumber	Read Only	Int	A number that identifies the process inside an Engine
process.name	Read only	String	The name of the process of the instance.

Predefined Variable	Permitted Values	Type	Description
receptionTime	Read only	Time	The time that the current instance was received at the current activity.
result	Modifiable: Any string	String	It contains the result that you set in the BP-Method. For example, you can set a result and in the outgoing conditional transition of the activity ask for that value. See below for more information.
status	Read Only. RUNNING, EXCEPTION, SUSPENDED, GRABBED, COMPLETED, ABORTED, ACTIVITY_COMPLETED	ProcessInstanceState	Current status of the process instance.
timeout	Modifiable: String indicating an interval, enclosed in single quotes: '5m'	Interval	The length of time that a BP-Method or an external component has to complete before the engine cancels its execution. By default this variable is set to 5 minutes.
totalCopies	Read only	Int	Number of threads or copies of an instance.



Action Variable

When a BP-Method is executed, multiple types of output can result from the execution. BP-Method execution status is indicated by the value of the predefined Action Variable.

Depending on the value of **action**, the engine responds accordingly and saves or undoes (commit or rollback) the changes invoked when the BP-Method is executed. The following table lists the valid values for the Action Variable:

action =	Description	Result
OK NONE	Indicates that BP-Method execution was successful. This is the default value.	BP-Method changes to "completed" status. If the activity is marked auto-complete on its Activity Properties dialog box, the instance flows to the next activity according to transition rules. Otherwise, the instance waits in WorkSpace for further processing in the activity or for the user to click the Send icon.
FAIL	Indicates that the BP-Method has failed its execution. The BP-Method must be executed again, if it is so required.	The error is logged in the Process Administrator log (in the Enterprise Edition) or it can be visualized at the bottom of Studio, in the Log tab. If a rollback BP-Method is included, it is executed. If the rollback BP-Method fails, the Engine will retry the original BP-Method until it succeeds or invokes the maximum number of retry times and is routed to an exception activity. The maximum number of retry times is set in

action =	Description	Result
		<p>the Process Administrator or in the Engine Properties in Studio. See Engine Properties.</p> <p> Note: A component exception is treated like Action=FAIL by the Engine.</p>
CANCEL	BP-Method execution is aborted.	<p>The instance is rolled back to the point before BP-Method execution. No trace of the BP-Method failure or execution appears in the Audit Trail.</p> <p>Can be used in activities of Interactive, Grab or Global Creation type.</p> <p>If the activity type is any other (i.e., Automatic, etc.), this value is ignored, as if it had never been set.</p> <p>If the activity type is Join, the original instance will be released and all the active copies will be aborted.</p>
REPEAT	Indicates that the BP-Method execution is ignored.	<p>The instance is committed. However, the task's status remains in a pending state. BP-Method should be executed again.</p> <p>If the activity type is Interactive or Grab REPEAT indicates that, although the task was successfully executed, it will remain pending. Therefore, the participant will be able to execute it again. Note that if the task is Mandatory, the participant will have to execute it repeatedly until the task appears as completed.</p> <p>If the activity type is any other (i.e., Global, Automatic, etc.), this value is ignored, as if it had never been set.</p>
RELEASE	Ends the BP-Method execution.	<p>The instance is released to the next activity without processing any of the BP-Methods in the current activity, even if they are marked mandatory.</p> <p>If the activity type is Interactive, Grab or Automatic the instance is released to the next activity without processing any of the BP-Methods in the current activity, even if they are marked Mandatory.</p> <p>If the activity type is any other (i.e., Global, etc.), this value is ignored, as if it had never been set.</p>
ABORT	Ends BP-Method execution and aborts the entire instance.	<p>The instance is not processed and is sent directly to the End activity. Warning: Instances that are aborted cannot be recovered.</p>

action =	Description	Result
BACK	Ends BP-Method execution and sends the instance back to the activity where the exception occurred.	<p>Used in activities of Interactive, Grab or Automatic type.</p> <p>Used in activities of Join type. The original instance will be aborted and therefore, all copies will be aborted.</p> <p>Aborts the instance. The instance is sent to the End activity and marked as aborted.</p> <p>Used in an exception handling activity to send an instance back to the activity where the exception occurred. If the exception can be corrected, fix it in the exception handling activity before sending it back.</p> <p>Used in activities of Interactive, Automatic and Grab type, when they are in an exception handling flow.</p> <p>If the activity type is any other (i.e., Global, etc.) or the activity is not an exceptions handler, this value is ignored, as if it had never been set.</p> <p>Within procedures or atomic groups, the instance goes back to the Begin activity of the procedure or the group.</p> <p> Note: The transaction is committed. Therefore, any change performed in the BP-method is persisted.</p>
SKIP	Ends BP-Method execution and sends the instance back to the activity where the exception occurred and skips it.	<p>Used in an exception handling activity to send an instance back to an activity in the process. The instance goes to the point of BP-Method failure and is released to the next activity without re-performing the BP-Method that caused the failure, even if the BP-Method task is marked mandatory on the Implementation dialog box.</p> <p>Used in activities of Interactive or Automatic type, when they are exceptions handlers.</p> <p>If the activity type is any other (i.e., Grab, Global, etc.) or the activity was not an exceptions handler, this value will be ignored as if it had never been set.</p> <p>Within procedures and atomic groups, the instance goes to the next activity that follows within the procedure or group.</p> <p> Note: The transaction is committed. Therefore, any change performed in the BP-method is persisted.</p>

Action Variable Example

The following BP-Method example shows how you can manually set the **action** variable if a Boolean expression evaluates to **true**.

```
if selectedButton == "Yes" then
  action = OK

elseif selectedButton == "Abort" then
  action = ABORT

else
  action = BACK
```

Result Variable

The **result** variable contains the result that you set in the BP-Method. For example, you can set a result and in the outgoing conditional transition of the activity ask for that value. Further in the process this value will no longer be available. The **result** variable is persistent between calls to the BP-Methods and is reset when the instance has already flown through the corresponding transition to another activity.

As well if the **action** variable is not used in the BP-Method, the predefined variable **result** values are mapped to the variable **action**. For example, if the BP-Method has the line **result = "fail"**, this is equivalent to **action = Action.FAIL**.

If the BP-Method uses the **action** variable, then the **result** can have any value, either predefined or not. For example:

```
result = "release"; action = Action.FAIL';
//in this case the BP-Method fails
result = "abort"; action = RELEASE';
// in this case the BP-Method does not fail
```

The Action Variable is reset at every BP-Method invocation to the default value while the **result** variable is reset after the instance flows into the next activity.



Note: The predefined variable **result** is used to maintain compatibility to previous version.

Timeout Variable

The **timeout** variable defines the time that a BP-Method has to complete its execution. If this time is exceeded, the Engine cancels the BP-Method execution and assumes that it has failed.

There is a Maximum Timeout defined for all BP-Methods of all processes within the engine. This value is defined in the Process Administrator or Studio Engine Preferences.

The timeout set within a BP-Method (timeout variable) cannot exceed the Maximum timeout set in the Process Administrator. If you set a greater value, a runtime exception is thrown and the BP-Method execution is aborted.

See Timeout for further information.

Suspend Action Example

Find the project **ActionsCase01.fpr** in the installation directory, in the studio/samples directory to study an example about handling the SUSPEND action and assigning the participant who suspended the instance.

To test this example, use the participant: **test**.

Local Variables

Provides a description of Local Variables.

Local Variables store information that is used only inside a single process method. The scope of a Local Variable is within method itself and cannot be accessed from outside. Once the method has been executed, the information stored in a local variable is lost.

Creating Project and Instance Variables

Outlines procedures for creating and editing Variables.

Before beginning this task, ensure that the Variables view is visible.

Instance and Project Variables are created and edited within the Variable view. You can also use the Variables view to map incoming and outgoing Argument Variables.

1. Click the + icon in either the Project or Instance Variable area, depending on which type of variable you are creating. The Variable properties window appears.
2. Enter the Name and Label of the Variable.
See [Variables Overview](#) on page 263 for more information on Variable Types.
3. Select the Variable type.
4. If you are creating a String or Decimal Variable, specify the variable size.
5. Click **OK**.

The new Variable appears in the Variables view.

Audit Events

Audit Events allow you to keep track of the events that occur while an instance is flowing through the process. An event is registered each time the instance enters or exits an activity (simple activity, group, process). The Process Execution Engine generates one event per action.

Audit Events Overview

When Audit Events are Generated

You can define which process activities will generate auditing events.

You set whether an activity generates events in design time, and you can set this for each activity, for activity groups, or for the whole process. You can also set whether the process engine generates events or not at run time.

Design Time

Design time event generation options are set in Studio. At design time, the following options are available for each *activity*:

Setting	Description
<i>Default</i> (default setting)	Indicates that the activity will record events according to the group or process default, as described below.
<i>Generate Events</i>	The activity will generate events, regardless of the group or process default.
<i>Do not Generate Events</i>	The activity will not generate events, regardless of the process default.

Also at design time, the following options are available for each *group*:

Setting	Description
<i>Default</i> (default setting)	Indicates that the group's activities will record events according to the process setting, as described below.

Setting	Description
<i>Generate Events</i>	The default for the group's activities will be to generate events, regardless of the process default.
<i>Do not Generate Events</i>	The default for the group's activities will be not to generate events, regardless of the process default.

The following options are available for each *process*. These settings will be used by activities or groups set to *Default*. If a group or activity is not set to *Default*, it will ignore the process setting.

Setting	Description
<i>Generate Events for Interactive Activities</i> (default setting)	Each activity set to the <i>Default</i> option will generate events if it is interactive, and will not generate events if it is automatic.
<i>Generate Events for all Activities</i>	Each activity or group set to <i>Default</i> will generate events.
<i>Do not Generate Events</i>	Each activity or group set to <i>Default</i> will not generate events.

Run Time

You set run time event generation in Process Administrator, for each process engine. You can set each process engine to one of the following event recording modes:

Setting	Description
<i>Depends on Process</i> (default setting)	Indicates that the process engine will follow the settings of each process. That is, it will follow the design time settings as described in the section above.
<i>Never</i>	No events are recorded, except instance begin and end activities. Design time settings are ignored.
<i>Always</i>	All activities generate events, regardless of process, group, or activity settings. Design time settings are ignored.

Remarks

If all settings are left at their defaults, a process will generate events for interactive activities and not for automatic activities. Begin and End activities are always generated.

This is a reasonable default because activities that require human interaction have the most variable execution times. However, you may want to measure some automatic activities, for example those that invoke external systems that for whatever reason have variable execution times.

Each event generated has a slight performance cost. This cost is not important for interactive activities since these activities spend most of their time waiting for participants to execute them. However, it may have significant impact on automatic activities that are performed frequently.

Which Audit Events are Generated

The following auditing events are generated:

- All the activities generate the same events (IN, OUT, EXECUTE, SELECT, UNSELECT, among others.)
- The Begin activity has no Activity IN event as the instance is created in it.
- The End activity has no Activity OUT event as the instance terminates there.
- The Join activity generates events only if the Split associated activity generates events.
- When an instance is created, a CREATION event is generated instead of an Enter event. This event is always automatically generated if the Engine stores events. All original instances (copy 0) have the CREATION event.
- When an instance is finished, an END event is generated. This event is always automatically generated if the Engine stores events. All terminated original instances (copy 0) have the END event.

- Interactive activities have additional events that occur between the Enter and End events.

If you have any Generates events check box selected, the Audit Trail in WorkSpace is enabled. The Audit Trail displays all events that have occurred for an instance.

Configuring Audit Event Generation

Configuring Auditing Events for a Process

Auditing events generation can be configured at a process level. This configuration defines the default auditing behavior for all the activities in the process.

To configure auditing events for a process:

1. Right-click on the process.
2. Select **Properties**.
3. Choose one of the following options:

Option	Description
Generate Events for Interactive Activities	Enables auditing events generation only for interactive activities.
Generate Events for all Activities	Enables auditing events generation for all the activities in the process.
Do not generate events	Disables auditing events for this Process.

Configuring Auditing Events for an Activity

Auditing events generation can be enabled independently for each activity.

To configure auditing events for an activity:

1. Right-click on the activity.
2. Select **Properties**.
3. Select **Advanced**
4. Choose one of the following options:

Option	Description
Default	Uses the configuration of the process to which the activity belongs to.
Generate Events	Enables auditing events generation for this activity.
Do not generate events	Disables auditing events for this activity.

Modifying the Generation of Audit Records for an Activity Group

To modify the configuration for generating audit records for an activity group, use ALBPM Studio and follow these steps:

1. In the navigator pane, expand the project and the process in which the activity group you want to modify resides.
2. Right-click inside the dotted line of the activity group.
A pop-up menu appears.
3. From the pop-up menu, select **Properties**.
The Activity dialog box appears.
4. In the Activity window, in the Category pane, select **Advanced**.
The Advanced pane appears on the right of the window.
5. In the Generate Events section, specify whether and at what level to generate events for the activity.
You have these options:

- Default--Indicates that the group's activities will record events according to the default process setting.
- Generate Events--The default for the group's activities will be to generate events, regardless of the process default.
- Do not Generate Events--The default for the group's activities will be not to generate events, regardless of the process default.

6. After you have made the changes, click **OK**.

Configuring the Generation of Audit Records for an Activity Group

An activity group is a compound activity. It is a set of activities that may include other activity groups. During the design time, you can configure audit record generation for an activity group when you create the group.

To configure audit record generation for an activity group, use ALBPM Studio as follows:

1. In the navigator pane, expand the project and the process in which the activity you want to modify resides.
2. After you decide the activities you want to group:
 - a) Click and hold the mouse button on the background of the process editor.
 - b) Drag the mouse to select the activities you have decided to group. The activities are surrounded by a dotted line.
 - c) Right-click inside the dotted line. A pop-up menu appears.
 - d) Select **Create Group with Selection**. The Activity dialog box appears.
3. In the Activity dialog box, in the Category pane, select **Advanced**. The Advanced pane appears on the right of the dialog box.
4. In the Generate Events section, specify whether and at what level to generate events for the activity. You have these options:
 - Default--Enables audit record generation at the activity level depending on the group to which the activity belongs, if any, or the process event generation definition
 - Generate Events--Enables audit record generation for all events at the activity level only, and not at the process level or within an engine. If no audit records are generated at the group or process level, they will be generated at the activity level.
 - Do not Generate Events
5. Once you have made your selection, click **OK**.

Configuring the Generation of Audit Records for a Process

To configure the generation of audit records for a process:

1. In the navigator pane, expand the project for which you want to create a new process. The navigator pane displays the resources for that project.
2. Right-click **Processes**, then select **New**, then **Process**. The Process dialog box appears. It provides three options for configuring audit record generation:
 - Generate Events for Interactive Activities--Each activity set to the default option will generate events if it is interactive, and will not generate events if it is automatic.
 - Generate Events for all Activities--Each activity or group set to the default option will generate events.
 - Do not Generate Events-- Each activity or group set to the default option will not generate events.
3. Select the audit level, and then click **OK**.

Organization

Organization Overview





The processes within a project always operate in the context of an *organization*. This means that an organization must be defined for each project.




In a project, the organization can represent all or part of a real organization, can include all or parts of other organizations, such as contractors or affiliates, and can also include roles that belong to no organization, such as customers. In AquaLogic® BPM, an organization is defined by the following elements:

- Organizational Units
- Roles
- Groups
- Participants
- Holidays
- Calendars
- Business Parameters

An organization is defined for each project. Therefore, it is not necessary to include elements of the organization which will have no function in the processes of a particular project. If you are not sure you will require a given element, leave it out. It can be always be added later, even on a running Process Engine.

Organization elements are accessed from Studio's Project Navigator or from the left pane in Process Administrator. A description of each type of element follows:

Element		Description
Organizational Units		Organizational Units are used to represent departments or divisions within the organization. Organizational Units can be defined hierarchically so that, for example, you can represent divisions within an organization, departments within a division, areas within a department, and so on. You can assign Participants, Calendars, and Business Parameters to an Organizational Unit. You can also deploy processes under an organizational unit.
Roles		Roles are used to represent functions performed by people related to the organization. Roles are assigned to participants or groups, and these assignments define the permissions the participants have when executing AquaLogic BPM tasks through WorkSpace.
Groups		Groups are collections of roles. In this way, it is possible assign multiple roles to participants in a single step. Groups may also contain other groups.
Participants		Participants are the actual people who participate in the organization, usually as end users of the BPM implementation.

Element		Description
Holidays		Holidays Define the organization's non-working days. These rules inform the Process Execution Engine that there is an exception to the normal calendar rules on certain days of the year.
Calendars		Calendars define the organization's work week and work schedule. Calendar rules can be assigned to organizational units.
Business Parameters		Business Parameters are used to maintain constant values defined either for the entire organization, or at the Organizational Unit level. These parameters are visible to all instances and all processes across the Organization. Although business parameters may be changed every once in a while, they are not meant to be used as variables. Rather, they provide a way of storing long-lived values, such as a sales tax rate, without having to hard-code them into PBL methods.

Working With Organization Elements

When the embedded Process Engine is started from Studio (select **Run ► Start Engine** from the menu), all the information about the organization is copied to an isolated environment where the engine executes processes.

Changes introduced while the Engine is running will not be updated to the runtime environment until the Process Engine is stopped and started, or until a **Refresh Engine Data** operation is performed. Depending on how runtime engine properties are set, the **Refresh Engine Data** operation might be performed automatically after introducing changes to the organization structure.

See Engine Properties for further information on how the runtime environment is updated with the latest changes. Some changes might require users currently logged in to WorkSpace first log out before having the changes available in their WorkSpace sessions. Refer to **Refreshing the Embedded Execution Engine Data** for further details on this topic.




Note: In this documentation, the parts of the organization are referred to as organizational elements or objects.

Importing an Organization

You can import an organization separately from a project. This way, you do not need to setup your organization with every new project.

To import an organization

1. In the Project Navigator, expand your project so you can see **Organization** ().
2. Right-click on **Organization** and chose **Import Data**.
The **Open** dialog box will be displayed.
3. Choose an organization data file, and click **Open**.

The organization data will be imported.

Exporting an Organization

You can export an organization separately from a project. The organization data will then be available for other projects.

To export an organization

1. In the **Project Navigator**, expand your project so you can see **Organization** (🏠).
2. Right-click on **Organization** and chose **Export** (📄).
The **Save As** dialog box will be displayed.
3. An organization data file name will appear by default, with an XDML extension. You can choose a different name or path, but you should keep the file type the same. Click **Save**.

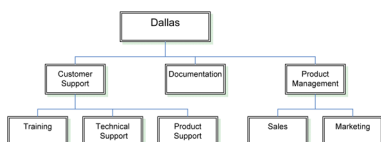
The organization data will be exported to the file you specified.

Organizational Units

About Organizational Units

Organizational units are typically departments or divisions within an organization. Organizational units can be organized in a hierarchy.

For example:



In this hierarchy, Dallas is a single top-level organizational unit which contains the Customer Support, Documentation, and Product Management organizational units, while Customer Support contains Training, Technical Support, and Product Support organizational units.

Once the organizational units have been defined, participants may be assigned to one of the organizational units in the hierarchy. Processes can be deployed for one of the organizational units defined so that only participants in that organizational unit and in lower levels within the hierarchy are able to perform tasks in a process.

Every organizational unit might have a different calendar rule associated to it. This allows the Process Execution Engine to take into account time zones and working schedules set for the organizational unit where processes are deployed and to calculate deadlines accordingly.

Studio allows you to define the organizational hierarchy and the properties of each organizational unit. Remember that all the changes introduced to the organizational structure require a **Refresh Engine Data** operation if they are to be made available to processes on a currently running Process Engine.

Creating an Organizational Unit

You can add an Organizational Unit from the **Project Navigator**.

To add an organizational unit:

1. Expand **Organization** (🏠) in the **Project Navigator**.
Organization will expand to show **Organizational Units**, **Roles**, **Groups**, **Participants**, **Holidays**, **Calendars**, and **Business Parameters**.
2. If possible, expand **Organizational Units** (🏠).
Any existing organizational units will be listed. If you cannot expand, there are no organizational units present.
3. Right-click on either **Organizational Units** or on an existing organizational unit, and select **New** from the context menu.
The **Name** dialog box will appear.
4. Input the name of the organizational unit and click **OK** to add it to the node where you obtained the context menu.
the new organizational unit is either created under the root organization node in the tree (if you right-clicked on **Organizational Units**) or under the organizational unit you right clicked on. In this way, hierarchical organizations can be defined. An editor for organizational unit data will open.

5. In the organizational unit editor, you can add a description in the **Description** text box, and you can select a calendar. These are optional fields and you can edit them later if you wish.



Note: The calendar rule set to an organizational unit *does not* affect the time zone used to display information to participants belonging to that organizational unit. The time zone taken into account to display dates in WorkSpace is the one set in the WorkSpace **Settings** dialog box. Settings, including time zones, can be unique for each user.

Roles

About Roles

A role in the organization is a title or job function which is associated to a set of activities performed by participants of the organization.

Examples of roles include Accounts Manager, Sales Clerk, or Customer. Roles are similar to job titles, but are more flexible because a participant can be assigned to several roles, and some roles, such as Customer, may not be jobs at all.

Roles and Activities

Every interactive activity is defined under a role. This is done by placing the activity within a *swim lane* with the name of the role. Swim lanes with no role name are only used for automatic activities which require no user interaction, and are not assigned to a role.

Roles and Participants

Participants are assigned one or more roles. This is how the process can determine which participants can execute a given activity.

Parametric Roles

A role can be defined as *parametric*. A parametric role includes a parameter which can adopt one of a set of values defined with the role.

For example, the role could be called "sales support" and the parameter could define a set of regions, such as East, West, and South. Even though there is only one role from a functional point of view, participants are assigned based on the location parameter.

Parametric roles require an instance because an the parameter to be used is defined as an instance variable. This means that global activities cannot be assigned to parametric roles.



Restriction: Global activities cannot be assigned to parametric roles.

Creating a Role

You can add a regular or parametric role to the organization from the **Project Navigator**.

To create a role:

1. In the **Project Navigator**, expand the project where you want to create a role.
2. Expand **Organization** (📁).
3. Right-click **Roles** (📁), then select **New** from the context menu. The **Name** dialog box is displayed.
4. Enter a name for the new role, then click **Ok**. The new role is created and an editor opens for the role.
5. In the editor, you can enter a label for this role in the **Label** field. The default value for the field is the name of the role, so changing it is optional.
6. In the editor, you can also enter a description for this role in the **Description** text box. This is optional.

7. If you want the new role to be parametric, click the **Parametric** checkbox. In the **Values** pane, add values as required with the **Add** button. You can remove unwanted values by selecting them and clicking on the **Remove** button.
A parametric role must have at least one defined value. Otherwise will let you save the role, but reports an error.

8. Save the role.

The new role has been created.

Groups

About Groups

Groups are collections of roles. In this way, it is possible assign multiple roles to participants in a single step. Groups may also contain other groups.

Unlike an organizational unit, which can belong to only one parent organizational unit, a group may be included in many other groups. Groups are therefore not organized in a hierarchical structure. However, if a group is included in another group, then it cannot have as a member that group. That is, so long as group A includes group B, group B cannot include group A.

Creating a Group

You can add a group from the **Project Navigator**.

To add a group:

1. Expand **Organization** in the **Project Navigator**.
2. If possible, expand **Groups** (📁).
Any existing groups will be listed. If you cannot expand, no groups exist.
3. Right-click on **Groups**, and select **New** from the context menu.
The **Group** dialog box will appear.
4. Input the name of the group in the **Name** field and click **OK** to add it.
The new group is created. An editor for the group will open.
5. In the group editor, you can add a description in the **Description** text box. This is optional.
6. Save the group.

The new group has been created.

Participants

About Participants

Participants defined in the organization are all the people enabled to track and perform tasks of business processes designed and developed with Studio.

A participant might belong to an organizational unit. If so, he can only perform tasks on processes deployed in that organizational unit or any organizational units that are below it.


You can assign a set of roles to a participant. A participant who logs in to Workspace can perform all the tasks defined for the roles assigned to him.

You can create, edit, and delete participants from the **Project Navigator**. Participants are usually created within Studio for process design and testing purposes. When the process is implemented into production, actual participants will normally be imported from an existing company directory or will be defined within Process Administrator.

Creating a Participant

You can add a participant from the **Project Navigator**.

To add a participant:

1. Expand **Organization** in the **Project Navigator**.
2. If possible, expand **Participants** ().
Any existing participants will be listed. If you cannot expand, there are no participants.
3. Right-click on **Participants**, and select **New** from the context menu.
The **Participant** dialog box will appear.
4. Input the name of the participant in the **Name** field and click **OK**.
The new participant is created. An editor for the participant opens.
5. Optionally complete the **First Name**, **Last Name**, and **Display Name** fields.
6. If the participant belongs to an organizational unit, select it from the **Organizational Unit** drop-down list.
7. Optionally complete the **E-mail address** field.
8. If you will use this participant in simulations, enter values in the **Efficiency** and **Cost per hour** fields.
9. Optionally set the Locale and Time Zone drop-down lists to values appropriate for the participant.
10. Add the groups the participant belongs to by clicking **Add** in the **Groups** pane, and selecting the desired group(s) from the **Groups** dialog box.
11. Add the roles the participant carries out by clicking **Add** in the **Roles** pane, and selecting the desired role(s) from the **Roles** dialog box.
12. Save the participant. If you close the editor without saving, the participant will still exist, but will not have any of the settings entered in steps 5 through 11.

Holiday Rules

About Holiday Rules


Holiday rules are collections of holidays that can be applied to calendar rules.

Multiple holiday rules can be created as needed for different [calendar rules](#). Holiday rules affect the available work days for participants and the scheduling of activity deadlines.

Creating a Holiday Rule

You can add a holiday rule from the **Project Navigator**.

To add a holiday rule:

1. Expand **Organization** in the **Project Navigator**.
2. If possible, expand **Holiday Rules** ().
Any existing holiday rules will be listed. If you cannot expand, no holiday rules exist.
3. Right-click on **Holiday Rules** , and select **New** from the context menu.
The **Holiday Rule** dialog box will appear.
4. Input the name of the holiday rule in the **Name** field and click **OK** to add it.
The new holiday rule is created, but it contains no holidays. An editor for the holiday rule will open.
5. In the holiday rule editor, you can add holidays by clicking **Add** in the **Roles** pane, and adding the desired holiday(s) from the **Holiday Rule** dialog box.
6. When you are done adding holidays, save the holiday rule.

The new holiday rule has been created.



Note: If you close the editor without saving, the holiday rule will still exist, but will no have any of the holidays added in step 5.

Calendar Rules

About Calendar Rules


Calendar rules define the work hours, time zone, and holiday rule assignment for organizational units.

Multiple calendar rules can be created as needed for different organizational units (such as day shift, night shift, east coast, west coast, etc.). Calendar rules determine the available work days for participants and the scheduling of activity deadlines.

Creating a Calendar Rule

You can add a calendar rule from the **Project Navigator**.

To add a calendar rule:

1. Expand **Organization** in the **Project Navigator**.
2. If possible, expand **Calendar Rules** ().
Any existing calendar rules will be listed. If you cannot expand, no calendar rules exist.
3. Right-click on **Calendar Rules** , and select **New** from the context menu.
The **Calendar Rule** dialog box will appear.
4. Input the name of the calendar rule in the **Name** field and click **OK** to add it.
The new calendar rule is created, with default calendar values. An editor for the calendar rule will open.
5. In the calendar rule editor, mark the checkbox for each day of the week which is a work day in this calendar rule.
6. For each workday, set the first work period of the day, between the Starting Time and the Finish Time on the left side. A second work period can be specified by setting the Starting Time and Finish time on the right side. You control whether the second work period is enabled by setting the checkbox adjoining it.
7. Save the calendar rule.

The new calendar rule has been created.



Note: If you close the editor without saving, the calendar rule will still exist, but with the default calendar values.

Business Parameters

About Business Parameters

Business parameters are used to store long-lived information defined at the organization level.

Information suitable for storage as a business parameter includes company address and phone data, tax rates used in calculations within the process, or infrequently changed economic values such as the prime lending rate. Business parameters are visible from any process within a project and should generally be thought of as constants, though they can be changed.



Tip: Business parameters should be used for infrequently changed values which you do not want to include in the actual code. For example, company address data, the prime lending rate, or a sales tax rate are all good uses for business parameters.

It is strongly recommended *not* to use Business Parameters for values which will change very frequently (once a day or more). For those cases consider other options.

If you do need to change a Business Parameter from PBL code, you can change it at runtime using the component Business Parameter in the Lib category. See the Studio. component documentation.

Notes:

- If you change a Business parameter from a method you must be aware that the new value is not immediately available for all instances. Even more, if this value is changed from a BP-Method, the result may not always be the expected one and not available at the same time across the all participants.
- If the business parameter is used in a due transition expression of an activity, the business parameter value that applies is the one defined at the time the instance enters the activity. For example, let's say the business parameter "MAXTIME" is used in the due transition expression of the activity "Reply to Customer". When the instance "Request Customer 1" arrives, the due time is calculated using the value that the MAXTIME has at that moment. If another instance (in any process) changes the value of MAXTIME or you manually change it in the Process Administrator, the new value does not apply for the due time of the instance "Request Customer 1" for the activity "Reply to customer". It will apply for all instances that arrive to that activity *after* the business parameter was changed.




Caution: If you change the Business Parameter at runtime, and you then stop and restart the Studio Process Engine, all business parameters are restored from the project definition. However, the *Enterprise* Process Engine *does* maintain Business Parameter values through a start/stop cycle, because in a production environment Business Parameter changes are assumed to be permanent.

Creating a Business Parameter

You can add a business parameter from the **Project Navigator**

To add a business parameter:

1. Expand **Organization** in the **Project Navigator**.
2. If possible, expand **Business Parameters** ().
Any existing business parameters will be listed. If you cannot expand, no business parameters are defined.
3. Right-click on **Business Parameters**, and select **New** from the context menu.
The **Business Parameter** dialog box will appear.
4. Input the name of the business parameter in the **Name** field and click **OK** to add it. Business parameter names are must be all upper case and the first character cannot be a number. Underscores are allowed.
The new business parameter is created. An editor for it will open.
5. In the business parameter editor, select the data type of the parameter in the **Type** drop-down list. Data type choices are *Bool*, *Int*, *Real*, *Time*, *Decimal*, and *String*.
6. Enter a value for the whole organization in the **Organization Value** field. You can override this value for individual organizational units in the next step.
7. You can add values by organizational unit by clicking **Add** in the **Organizational Units** pane, and adding the desired organizational unit from the **Organizational Units** dialog box. For each organizational unit, specify a value in the **Values** column of the Organizational Units table.
8. Save the business parameter.

The new business parameter has been defined.

Deleting an Organizational Element

You can delete organizational elements in the Project Navigator. You should take care when deleting an organizational element, as it might be referenced by other organizational elements or by processes.

To delete an organizational element:

1. Expand **Organization** in the **Project Navigator**
Organization will expand to show **Organizational Units**, **Roles**, **Groups**, **Participants**, **Holidays**, **Calendars**, and **Business Parameters**
2. Expand the element category where the element to be deleted is to be found. For example, expand **Organizational Units** to be able to delete an organizational unit.
Any existing elements will be listed. If you cannot expand an element category, there are no elements under that category which can be deleted.

- Right-click on the element you want to delete and select **Delete** (✖) from the context menu. Alternatively, select the element to be deleted and press the Delete key.
Often an element will be referenced, that is used, by other elements or objects. For example, a role may have activities assigned to it. If this is not the case, The organizational element will be deleted with no further confirmation. If the element is referenced, the following prompt will appear:

This object is being referenced by others. Delete anyway?

In this case, proceed to step 4.

- Determine if the dependency is important to your project. If it isn't, press **OK**. If it is or if you are not sure, press **Cancel**.

The element will be deleted or not according to your choice.



Note: You cannot undo an Organizational element deletion.

When deleting an organizational element, other elements or objects in the project may be affected, as follows:

Element	Description
Organizational Units	<ul style="list-style-type: none"> When deleting an organizational unit to which participants belong, the participants are automatically moved to the first level in the organization. After deleting the organizational unit, the deploy preferences for any processes currently deployed to the unit are changed to be deployed to the root organization instead. When deleting an organizational unit to which other organizational units belong, the dependent organizational units are also deleted.
Roles	<ul style="list-style-type: none"> If a role which participants are assigned to is deleted, the participants will not be assigned to any role.
Groups	<ul style="list-style-type: none"> Groups may include other groups. If you delete a group which is referenced by another group, you may unwittingly remove roles from the referencing group. For this reason, you will be asked to confirm the deletion of a group referenced by others.
Holidays	
Calendars	
Business Parameters	

Editing an Organizational Element

The properties of any organizational element can be edited at any time. The name of the organizational element cannot be changed, however.

To edit an organizational element:

- Expand **Organization** (🏢) in the **Project Navigator**
You will see the top level of the Organization tree.
- Navigate to the organizational element you wish to edit.
- Double-Click on the existing organizational element, or right-click on it and select **Open** (🔗) from the context menu.
The editor for the organizational element will open.
- In the editor, change the properties of the organizational element to the desired values, and save it.

Dynamic Business Rules

In ALBPM you can define dynamic business rules. A dynamic business rule is a set of one or more conditions evaluated against project variables or business parameters.

You define dynamic business rules in the business rule editor, which has two modes: A simple GUI mode and an advanced mode where you define the rule by writing PBL code. You define each business rule with a unique name.

Dynamic business rules are defined for the project. Once you define a dynamic business rule, you can use it in any process from a [Business Rule Transitions](#) on page 81, or from code in any PBL method. You can design your project so that WorkSpace participants can edit business rules at run time.

Business Rule Transitions

In ALBPM, conditional transitions can be used to control the flow of an instance within a process. However, standard conditional transitions are defined at design time and cannot be edited in run time.

Business rules can also be used to control process flow when used by business rule transitions. Business rule transitions are similar to conditional transitions except that they evaluate a business rule instead of a conditional expression.

Access From PBL Code

The Rules Editor and rule evaluation use the standard component `Fuego.Rules.Rule`. For advanced use cases this component may be used directly from PBL code. This allows you to add or evaluate rules from any type of activity or BPM Object.

Business Rules at Run Time

Dynamic business rules are considered dynamic because they can be modified at run time. Dynamic business rules can be edited by any participant who is a member of a role which has been enabled to edit business rules.

When to use Dynamic Business Rules

Dynamic business rules are suitable for specific situations.

You can use dynamic business rules when you want to do any of the following:

- You want at least some participants to be able to change a business rule at run-time, from the WorkSpace, within a deployed process.
- You want to define a simple business rule without using code, or you want to allow Business Analysts to do so. Dynamic business rules can be edited with a simple Business Rules Editor, or, for advanced capability, by writing code.
- You want to share a single business rule across processes or activities in a project. Dynamic business rules are *named*, and they are defined at the project level.
- You want to audit the use of a business rule. Every time a business rule is evaluated, this evaluation (and its result) is recorded in the Audit Trail.

You should not use dynamic business rules if any of the following are true:

- You need the rule to operate on instance variables. Business rules cannot access instance variables. They can only evaluate project variables and business parameters.
- You don't have a specific reason to use dynamic business rules. They require more system resources because they are audited, require project variables, and are evaluated at run time.

Using Dynamic Business Rules

Business rules often need to change depending on your business context and requirements. The Business Rules Editor allows you to easily view and change the business rules for each of your ALBPM Projects.

Business Rule Editing Modes

The Business Rules Editor has two modes: *simple editor* and *advanced editor*. The simple editor is a UI which lets you define one or more conditions to be checked. You can determine if you want the rule to match all the conditions or any of them, and you must always compare a variable or business parameter to a constant.

The advanced editor allows you to enter PBL code and can therefore be used to implement more complex conditions.

Editing Rules from the WorkSpace

A web version of the Business Rules Editor can also be used from the ALBPM WorkSpace web application. To have access to the editor, a participant must have a role where a global activity has been defined with an *Edit Business Rules* implementation type.

This activity will be visible in the **Applications** panel of the WorkSpace page.

Versioning

Because dynamic business rules can be edited at run time, they are versioned. This allows the user to revert to an earlier version if a newly edited one does not work as expected.

Version Numbers

Every time a business rule is edited, the business rules version is incremented. Business rules are not versioned individually. Rather, there is a single version number for the whole set of rules in the project. Hence, if you have two business rules A and B, and you edit rule B three times and rule A twice, your version numbers may look like those shown below:

Version number	Rule Edited
1	A
2	B
3	B
4	A
5	B

In this way, the rule A will be available in versions 1 and 4, while rule B will be available in versions 2, 3, and 5. WorkSpace users are able to select any available version of a given rule in the WorkSpace business rules editor.

Rule Compatibility Checking

At run time, every time a rule needs to be evaluated, the first (most recent) compatible version of it is fetched from the Directory Database. A compatible version of a business rule is a version which accesses project variables and business parameters that exist in the project and have the type expected by the rule.

If no compatible rule is found, the rule that was originally published with the current project version is used. In other words, none of the versions of the rule edited at run time are used. This is a fail-safe mechanism so that a new version of the process will be able to run even when previously edited (and no longer compatible) rules are present.

Auditing

Every use of a business rule can be audited. You can control whether or not the evaluation of the rule will be visible in the audit trail.

Dynamic business rules can be evaluated from a business rule transition or from code. The following table describes how to handle each case:

Rule Evaluated In	To Enable Auditing
Business Rule Transition	In the activity where the business rule transition originates (the <i>from</i> activity), the Generate Events option must be set.
PBL Code	In the activity which causes the execution of the PBL code, the Generate Events option must be set.




For further information, see [Audit Events](#) on page 99.

Defining a Business Rule

The following task outlines the procedures for creating and editing Business Rules.

To create a business rule, your project must have either one or more project variables, or one or more business parameters.




To define a business rule:

1. Right-click on the project where you want to define Business Rules.
2. Select **Business Rules** ().
The Business Rules editor appears.
3. Click the **Add** icon ().
The **New Business Rule** dialog box appears.
4. Enter the name of your new business rule, and click **OK**.
The new Business Rule appears in the table.
5. To edit the business rule, click on the Open icon () , or double-click on the business rule name.
The business rule editor page for that rule appears.
6. By default, the business rule editor page is in simple editor mode. You can also edit in advanced editor mode. Go to the corresponding task listed below for instructions in the mode you will use.
7. To close the business rule editor and all open business rules, click the **X** on the top tab. To close only the page with the business rule you have just saved, click the **X** on the bottom tab.

Simple Editor Mode

Use the simple editor mode to define simple business rules without having to write any PBL code.

To edit a business rule:

1. To add a condition, select one of the available project variables or business parameters from the **Add Condition** drop-down list.
2. Click the **Add** button ().
A new condition line is added.
3. Select a comparison operator from the first drop-down list.
Different operators are available as a function of the data type of the variable or parameter you are comparing. Operators include *Equals*, *Not Equals*, *Greater Than*, *Less Than*, and so on.
4. Enter the value that will be compared to in the field on the right. Special options may be available depending on the data type of the value being compared. For example, a time value field will be accompanied by a calendar tool to help you pick a date.
5. Repeat steps 1 through 4 to add more conditions.
6. If you wish to remove a condition, click the **Remove** button () next to it.
7. Once you have set all the conditions in your business rule, save your changes by clicking the Save button () , or by clicking **File ► Save** from the main menu.

Advanced Editor Mode

Use the advanced editor mode to write complex business rules. This mode requires PBL coding, so to use it you should be familiar with PBL syntax.



Note: If you write a complex business rule in the advanced editor mode, you will no longer be able to edit this rule in the simple editor mode.

To use the advanced editor:

1. In the simple editor, click **Switch to Advanced Editor**.
The advanced editor opens. This is a PBL code editor. If you have already defined any conditions, these will be shown in code form.
2. In the editor, enter any PBL code you require to implement the business rule. You must exit your code with a `return` statement which returns a boolean value (`true` or `false`).
For more information about PBL, consult the [Process Business Language \(PBL\)](#) on page 205.
3. If you want to switch back to the simple editor mode, right-click anywhere on the advanced editor and click **Switch to Simple Editor**.
Recall that you will not be able to do this if you've edited the rule in such a way that it can no longer be represented in the simple editor.
4. Save and close when you are done.


Letting Participants Edit Business Rules

You can enable participants to edit business rules at run time using the WorkSpace web application. To do this, you add a global interactive activity with an "Edit Business Rules" implementation type.

To complete this task, at least one role must be defined.

When you provide access to the business rules editor by adding a global interactive activity to a role, you give every participant in that role the ability to edit the business rules selected in that activity.

To add a Global Interactive of type "Edit Business Rules":

1. Insert a global interactive activity () in the role you wish to enable to edit business rules.
The **Activity** dialog box appears.
2. Enter a name for the activity in the **Name** field, and click **OK**.
3. Right-click on the activity you have just added, and click **Main Task**.
The **Main Task** dialog box appears.
4. In the Implementation Type section, select *Edit Business Rules* from the drop-down list.
A list of available business rules appears.
5. To enable WorkSpace editing of a business rule, click on the right side column so it says **Yes**. Alternatively, you can click **All** to enable all business rules, or **None** to disable them. You must enable at least one business rule.
6. When you have enabled the desired business rules for this activity, click **OK**.

The global interactive activity is now set to edit business rules, and will appear in the **Applications** pane of any WorkSpace participant in the role where you added the activity.

Simulations

The following topics describe how to use simulations by creating process and project simulation models.

What is a Simulation?

After you have designed a business process model, ALBPM Studio allows you to run process simulations to determine the efficiency of the process. You can also use simulations to test the effects of changes on your process design.

Simulations allow you to mimic the execution of a process to identify any potential problems in its design. Within a project, you can simulate the execution of multiple processes simultaneously using different simulation models.

A process simulation does not execute each individual task within a process. The code within an activities task is not executed, variables are not assigned values, and external resources are not updated. However, you can mimic the behavior of activities within your process by configuring different attributes of the activity within a simulation model, including the following:

- duration
- resources
- costs
- transitions

Simulation Models

To execute a process simulation, you must define simulation models. Simulation models allow you to specify behavior of each element of your process. There are two types of simulation models in ALBPM Studio:


Simulation Model	Description
Process Simulation Models	Allow you to define the behavior for an individual process. See Process Simulation Model on page 116.
Project Simulation Models	Allow you to define the behavior for an entire project. A project simulation model is composed of a group of process simulation models. Within a project simulation models you can choose which process simulation models to run. See Project Simulation Models on page 118.

You can configure multiple simulations models for a project and its processes. This allows you to mimic different combinations of resources, etc.

Round-trip Simulation


Round-tip Simulation provides a way to create process simulation models based on real-world data. When creating a process simulation model using round-trip simulation, data is imported from the Process Execution Engine Database. You can use data imported from the following environments:

Process Execution Engine Database	Uses
Studio	Using data from Studio's process execution engine, you can quickly create a process simulation model.
Enterprise	Using data from the Enterprise process execution engine allows you to create process simulation model based on 'real-world' data. You can use this data to create a base-line simulation that mimics the performance of your processes. This base-line can be compared to simulations of revised processes to determine how performance can be improved.

 **Note:** After creating a process simulation model using round-trip simulation, you must manually enable it in your Project Simulation Model. See [Creating a Project Simulation Model](#) on page 120.

Process Simulation Model

Process simulation models allow you to define how a process behaves as part of a Project Simulation Model. You can define multiple process simulation models for each process. This allows you to create different simulations based on different combinations of resource allocations and activity behavior.

 **Note:** Grab, Global, and End activities are not simulated.

Process Information Tab

The Process Information Tab allows you to configure in:


Option	Description
Enable amount of current instances	Allows you to define the number of instances that can exist within the simulation at one time. The process simulation will run until the duration is completed or the maximum number of instances is reached.

The following table lists the Distribution Types used when creating process instances:

Option	Description
Constant	Generates simulated process instances regularly as defined by the period property.
Uniform	Generates simulated process instances regularly, taking into account the variation specified in the delta property.
Exponential	Creates simulated process instances using an average frequency of instances within a specific interval.
Normal	Generates simulated processes according to a Gauss Bell distribution based on a mean and standard deviation.
Real	Creates simulated process instances based on specific time-based criteria. This type of distribution is primarily used with round-trip simulations. You can specify the interval criteria used to categorized the distribution. You can also specify the mean and standard deviation.

Activities Tab

The Activities tab allows you to define the simulated behavior of each activity in your process. The following table describes the different tabs that are used to define an Activities behavior. The exact tabs displayed depend on the Activity type.

Tab	Description
Duration	<p>Defines how long the instance will remain in the Activity during simulation. Since simulations do not execute tasks, this interval should be an estimate of how long an activity's task takes to complete.</p> <p>You can specify a distribution type. See the Process Information Tab documentation above.</p>
Resources	<p>Specifies the simulated resource allocation for each activity. You can select from the following options:</p> <ul style="list-style-type: none"> Use Organization resources: Causes the simulation model to use organizational resources defined in the Project Simulation model. You can also define a Participant Selection Policy which is based on minimum cost, maximum efficiency, or can be generated randomly. Cost and efficiency values are defined in the Project Simulation Model for each participant. Use Fixed Resources: Defines the number of participants assigned to the Activity. This option is used when costs and efficiency parameters are not used in the simulation. <p> Note: When using this option, the Project Resources tab of the Project Simulation Model is not used.</p>
Cost	<p>Defines the costs of a resource within this Activity.</p> <p>The Activity Cost Type defines how Activity costs are calculated. The following values can be specified:</p> <ul style="list-style-type: none"> Fixed Base Cost Fixed Base Cost + Resource Costs

Tab	Description
	Fixed Base Costs applies each time a process instance is run. The Resource Cost is calculated using the defined cost per hour and the time it takes this resource to execute the process instance.
Queue Info	<p>Allows you to define how process instances are queued before executing an Activity.</p> <p>Queue Warning Size defines the size of the incoming queue.</p> <p>The Activity Queue Policy allows you to define the order instances are executed. The following values can be specified:</p> <ul style="list-style-type: none"> • F.I.F.O. • L.I.F.O. • Random • Instance Priority
Transitions	<p>Allow you to specify the probability that a process instance will follow the Activity's outgoing transitions. In addition to defining a probability for each outgoing transition, you can also add a probability for an aborted instance.</p>
Related Processes	<p>Allows you to define which simulation model of the related process is run during simulation. This tab is only available for Subflow Activities.</p> <p>At runtime, the simulation is executed based on the configuration of the subprocess' simulation model. If the checkbox is disabled, the Subflow Activity is treated as a simple activity. No subprocess is executed.</p>

A process simulation does not execute the actual code of each activity within the process. However, by configuring parameters within the Process and Project Simulation Models, you can mimic the behavior of your process.

Project Simulation Models

A Project simulation model defines the behavior of the simulation for the entire project.

Project simulation models allow you to customize the following parameters to see how they influence the performance of your project:

- Start time and duration of the simulation
- Determine which process simulation models you want to include in the project simulation
- Allows you to define the participant resources you want to include in the simulation
- Allows you to define the priority distribution of instances within the simulation

Within a project, you can define multiple project simulations. Defining different project simulations models allows you to test different combinations of resources and priorities. The following parameters can be configured for a project simulation model.

Parameter	Description
Start Time	Defines the start time for the simulation. This time is used only for logging. It is not used for scheduling purposes.
Duration	Defines the period the simulation will run. This interval is specified in months, days, hours, minutes, and seconds.
Use Calendar Rule	Determines if calendar rules are used in simulation. Checking this box allows the simulation to account for calendar rules when determining participant allocations.

Project Simulation Models also contain the following tabs which allow you to further define your simulation.




Project Tab

The Project Tab contains a table that lists all of the processes within the current project. For each process, you can select which process simulation model you want to use for each process. Also, you must specify which processes to include in the simulation.

For all of the included processes, instances are generated when the simulation is run.

Resource Tab

The Resource Tab allows you to define the resources used within the process simulation. All process included in the simulation will share these resources. The cost of each resource is defined per hour.

Icon	Description
	Loads resources from the organization into the project simulation model.
	Adds a new resource to the project simulation model.
	Deletes the currently selected resource from the project simulation model.

Priority Tab

The Priority Tab allows you to specify the probability for priority distribution of an instance. This priority determines the way an instance flows withing a process. The sum of all priority distributions must equal 100.

Simulation View

The simulation view allows you to run and view simulations.

After defining Project and Process simulation models, you can run a Project simulation from the Simulation View.

The Simulation View toolbar allows you to perform the following actions:


Toolbar Element	Description
Play Icon	Starts the simulation. If the processes included in your simulation are not open in an editor window, Studio opens them.
Stop Icon	Stops the simulation.  Note: If you stop a simulation, you must restart it from the beginning.
Pause Icon	Pauses the simulation. You can press the start icon to resume the simulation.
Run to End Icon	Runs the simulation in the background with no animation. This allows you to run the simulation faster.
Simulation Speed	Determines how fast simulated process instances are created. In normal speed, instances are created at rate of one per second.

Chart Tab

The Chart Tab provides allows you to configure various aspects of a simulation.

Toolbar Element	Description
Display Type	
Activities	
Resources	
Categories	

Toolbar Element	Description
Indicators	

Log Tab

The Log Tab displays a log of all the actions performed during the simulation.

Tasks

Creating a Process Simulation Model

The following steps describe how to create a Process Simulation Model.

1. In the Project Navigator View, expand the Project where you want to create the Process Simulation Mode.
2. Expand **Processes**.
3. Right-click on the Process.
4. Select **New Process Simulation Model**.
5. Enter a name for you new Process Simulation Model.
6. Click **OK**.

The Process Simulation Model appears in the editor window. It also appears as a Resource in the Project Navigator View.

You can define the behavior of your Process Simulation Model.

Creating a Project Simulation Model

The following steps describe how to create a Project Simulation Model.

You should define any Process Simulation Models that you want to include as part of your new Project Simulation Model. See [Creating a Process Simulation Model](#) on page 120.

1. In the Project Navigator View, right-click **Simulations**.
2. Select **New Simulation**.
3. Enter a name for your Project Simulation Model.
4. Click **OK**.
5. The Project Simulation Model appears in the editor window. Each Process Simulation Model that you have defined appears in the table.

The Project Simulation Model also appears as a Resource under Simulations in the Project Navigator View.

6. Using the drop-down menu, select **Yes** in the **Include in Simulation** for each Process Simulation Model that you want to include in your Project Simulation.

Running a Simulation

The steps in this task outline how to run a Process Simulation based on Process and Project Simulation Models.

Ensure that you have created Process Simulations Models and at least one Project Simulation Model. See [Creating a Process Simulation Model](#) on page 120 and [Creating a Project Simulation Model](#) on page 120 .

1. In the Project Navigator View, expand **Simulations** .
2. Select the Project Simulation Model that you want to run.
3. Ensure that the Simulation View is visible.

See [Simulation View](#) on page 16 .



Note: It may take several seconds for the Simulation View to load.

4. Click the Start icon.

The simulation animation plays in the Process Editor window. Simulation data is updated in the Simulation View.

Running a Round-trip Simulation in Studio

This topic outlines the procedures for running a round-trip simulation using within Studio. For testing purposes, you can create a round-trip simulation model using WorkSpace to generate data based on real world situation. This data is then imported from the Embedded Process Execution Engine into Studio.

Before running the following procedures, ensure that you have created a project and at least one process that you want to simulate.

1. Start the Embedded Process Execution Engine within Studio.
2. Run WorkSpace
3. Use your deployed process to generate simulation data.

You should create and use enough instances of your process to create meaningful test data. There is no minimum threshold of data. The exact amount of data required depends on the process you are simulating.



Note: Before continuing to the next step, you must wait several minutes for the embedded process engine database to update.

4. Right-click on your Project.
5. Select **Extract Simulation**.
 - a) Provide a name for the new Process Simulation Model
 - b) Select the Process you where you are creating the simulation model.
 - c) Select the distribution criteria for this simulation.
6. Click **OK**.

Sharing Files with Version Control

The following topics describe how to share ALBPM Projects and Resource with your version control system.

What is Version Control?

Version control systems allow you to share and control resources within your organization. sALBPM Studio allows you to easily integrate Project resources within your version control system.

ALBPM Studio uses the standard Eclipse version control system. Eclipse provides plugins for many common version control systems. ALBPM includes the CVS plugin by default.

Sharing Files with Version Control

The following procedures show you how to share a resource using version control in ALBPM.

1. In the Project Navigator, right-click on the Resource you want to share.
2. Select **Team ► Share Project**

The Share Project with CVS Repository wizard appears. This wizard allows you to define connectivity information for your version control system.

3. Follow the procedures defined in the Team CVS Tutorial in the *WorkBench User Guide*.

Extracting Files from Version Control

The following procedures show you have to access ALBPM Resources that are stored in a version control system.

1. In the Project Navigator, select **File ► Import**.
2. Expand **CVS**.
3. Select **Projects from CVS**.
4. Click **Next**.
5. Follow the procedures defined in the Team CVS Tutorial in the *WorkBench User Guide*.

Localizing a Project

The following sections show you how to add languages to a Project and how to add localized labels to a Process.

Localization Overview

ALBPM Studio allows you to localize elements of your business process. Most elements of a process that are visible within WorkSpace can be localized.

Supported Languages

When localizing a Project, you can add as many of the supported languages as necessary. The following languages are supported:

- Spanish
- Chinese (Traditional)
- Chinese (Simplified)
- Korean
- Japanese
- German
- French
- Italian
- Dutch
- Portuguese

English is the default language for a new ALBPM Project and is included automatically.

Project Elements that Can Be Localized

After adding a languages to your project, you can add localized labels to the following aspects of your Project:

- Project Variables
- Business Rules
- Process Names
- Activity Names
- Input Tasks
- Decision Tasks
- Role Names
- View Names
- Presentations

Adding a Language to a Project

Before you can localize different elements of your process, you must add a language to your process.

To add a language to your process:

1. Right-click on the Project where you want to add a language.

2. Select **Project Preferences**.
3. Click **Languages**
4. Click **Add**.
5. Select the language you want to add.
See [Localization Overview](#) on page 122 for a list of available languages.
6. Click **Ok**.

After adding a language to your Project you can localize individual elements of your process.

Localizing a Process Name

The following procedures show you how to localize a Project name. This localized name is used within ALBPM WorkSpace.

Ensure you have added a language to your process before localizing Activity names. See [Adding a Language to a Project](#) on page 122 for more information.

1. Right-click on the Project whose name you want to localize.
2. Select **Properties**.
3. Click the icon.
4. Enter the localized label next to the appropriate language.
5. Click **Ok**.
6. Click **Ok**.

Localizing an Activity within a Process

The procedures outlined in this topic show you how to localize Activity names within your process. These localized names are used within ALBPM WorkSpace.

Ensure you have added a language to your process before localizing Activity names. See [Adding a Language to a Project](#) on page 122 for more information.

1. Open the Process containing the Activity you want to localize.
2. Right-click on the Activity.
3. Select **Properties**.
4. Select **Activity ID**.
5. Click the icon.
6. Enter the localized label next to the appropriate language.
7. Click **Ok**.
8. Click **Ok**.

Correlations Overview

In ALBPM, a correlation is a map that relates one or more values to a process instance. A correlation allows you to access an instance in a process without knowing the instance ID. Instead, you can define a correlation that uses values with business meaning, such as an invoice number or a product code.

Consider a situation where you have a process to manage purchases and your suppliers can access the system through some kind of interface. You start the process by sending a purchase order to a supplier, creating an instance in the purchase process. This process will expect the seller to return an acknowledgment for the order, stating if the order is accepted or not.

For the process to continue, the acknowledgment must be routed to the same instance that originated the purchase order. However your supplier's system does not have the instance ID. So then the question is: How can the acknowledgment be routed to the correct instance? You could send the instance ID to your supplier, but this number has no meaning to the supplier, and the supplier may not even have a way of storing it. In fact, the instance ID has no meaning anywhere outside the process. Instead, you define a *correlation set* that correlates a *business token* to a given instance.

A business token is simply a value or set of values that has business meaning and is also unique to each instance. For example, a purchase order number could be used as a business token, but the date or amount of the purchase order cannot be, because neither are necessarily unique.

Correlation Sets

You define correlation sets at design time. A correlation set has a name and one or more *correlation set properties*. The properties are references to the selected arguments of the Argument set.

Correlation Sets

You can have more than one correlation relationship. You may want to map the purchase order described above to more than one system. You may have a supplier and a separate shipping service, for example. You may also have more than one process in your project that can use the same business token.

To resolve these cases, ALBPM allows you to define *correlation sets*. A correlation set is composed of one or more *correlation properties* that uniquely identify the instance.

Defining a Correlation Set

To define and use correlation set, you must first create the set, then add properties to it, and then map these properties to process arguments.

Before you define a correlation, you should know the name and data type for each of the properties you will add to the correlation. The following steps assume that you have a process design open in the editor.

To define a correlation set:



1. [Creating a Correlation Set](#) on page 124.
2. [Adding Correlation Properties](#) on page 125.

Creating a Correlation Set

You create a correlation set from the **Argument Mapping** dialog box. You can define argument mappings in the Begin, Subflow, Process Creation, Termination Wait, Process Notification, and Notification Wait activities.

Before you define a correlation, you should know the name and data type for each of the properties you will add to the correlation. The following steps assume that you have a process design open in the editor.

To define a correlation set:

1. In the process design editor, right-click on the Begin activity and click **Argument Mapping**.
The **Argument Mapping** dialog box will appear.
2. Click on the **Correlations** Icon ().
The **Correlations** dialog box appears.
3. Click the **Add** button () in the **Available Correlations** section.
The **New Correlation Set** dialog box appears.
4. Specify the **Correlation Set Name**, and click OK.
The new correlation set is created, and added to the **Available Correlations** list.

Adding Correlation Properties

Once you have defi

To perform this task, you need to have created at least one Correlation Set. The task assumes that the **Correlations** dialog box is open.

To add a correlation property:

1. In the **Available Correlations** list of the **Correlations** dialog box, right-click on the name of the Correlation set you want to add a property to, and click **Add Property**.
The **Property** dialog box appears.
2. In the **Property** dialog box, enter a name for the property.
3. Select a [Correlation Property Data Types](#) on page 125 from the **Type** drop-down list.
4. Click **OK**.

Correlation Property Data Types

Correlation Properties can be defined with one of several data types, as described in this section. Complex data types, such as objects or arrays, are not allowed.

A correlation property can be one of the following data types:

- Bool
- Int
- Real
- Time
- Decimal
- String

Using BPEL

This section describes ALBPM Studio support for BPEL processes. It includes topics on how to create and use BPEL processes within AquaLogic BPM.

ALBPM BPEL Support

The Business Process Execution Language (BPEL) is an industry standard language for defining business process models. AquaLogic BPM Studio provides full support BPEL 1.1.

ALBPM Studio allows you to do the following:

- Import Existing BPEL processes and their corresponding WSDL files into an ALBPM Studio Project.
- Create new BPEL process models within ALBPM Studio.
- Consume BPEL processes from a native ALBPM process.
- Publish and deploy BPEL processes in the Process Execution Engine.

See <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html> for complete information on the BPEL standard.

BPEL Process Modeling

AquaLogic BPM Studio provides a BPEL editor Eclipse plugin for creating and editing BPEL processes. See [Using the BPEL Process Editor](#) on page 129 for more information.

BPEL Process Execution

The AquaLogic BPM Process Execution Engine can run BPEL processes natively without using extensions. This allows the ALBPM Engine to execute BPEL 1.1 compliant processes created with tools from other vendors.

See [BPEL Processes Execution](#) on page 127.

Enabling BPEL Process Design and Editing

Support BPEL process modeling is not enabled by default. The following procedures show you how to enable BPEL modeling.



Note: After enabling BPEL support, it is available for all projects within Studio.

To enable BPEL Process Modeling:

1. Start ALBPM Studio using the Developer profile.
2. Open the Studio preferences editor.
3. Select **BPEL**
4. Select enable **BPEL Support**.
5. Click **Ok**.

You must restart ALBPM Studio for these changes to take effect.

BPEL Processes Execution

When you start the process execution engine within Studio, the engine automatically loads all of the BPEL processes within the project. For each BPEL process, the engine exposes the Web Service interface.

At runtime, the engine compiles the BPEL process. However, if there are any compilation errors, the engine continues to start. You must check for any validation errors within the BPEL editor before starting the engine.



Note: You must save any changes to your process before running the engine.

Viewing BPEL Processes as Web Services

After starting the Process Execution Engine in Studio, you can view your BPEL processes by accessing the following URL: <http://localhost:9000>.

This page displays all of projects currently deployed on the engine. When accessing the Process Execution Engine within Studio, only the processes in the current project are displayed. Clicking on the project on this page displays a list of all the deployed Web Services, including those of your BPEL processes.



Note: If you do not see the Web Services of your BPEL process on this page, this indicates an error. Ensure that there are no validation errors in your BPEL process. If an error still occurs, check the Process Execution Engine error log.

Creating a New BPEL Process

The following procedures show you how to create a new BPEL Process within ALBPM Studio.

Ensure that you have created the process where you want to create the new BPEL process.



Note: You can only edit BPEL processes from the Developer profile.

1. Enable BPEL support.

See [Enabling BPEL Process Design and Editing](#) on page 126 for more information.

2. In the Project Navigator, select the Process where you want to create the new BPEL process.
3. Select **File ► New ► New BPEL Process**
4. Provide the following information:

Option	Description
Name	The name of your new BPEL process.
Namespace	The namespace used by your new BPEL process.
Template	The template you want to use to generate the BPEL process.



Note: If you choose the Empty BPEL Process template, no WSDL is generated for your process.

5. Click **Next**.
6. Expand the folder of the project where you want to create the new BPEL Process.
7. Select **processes**



Note: You must select the processes folder. If create the process in another folder, it is not recognized by ALBPM Studio.

8. Click **Finish**.

The Wizard creates the new BPEL process and generates the corresponding WSDL file. The BPEL process is visible in the Project Navigator and Navigator views. The WSDL file is visible in the Navigator view only.

You can edit the BPEL process using the BPEL editor provided in ALBPM Studio. See [Using the BPEL Process Editor](#) on page 129 for information.

Importing a BPEL Process

The following procedures show you how to import an existing BPEL into ALBPM Studio. When importing a BPEL Process you must also import its corresponding WSDL file.

1. Ensure that you have enabled BPEL support.
See [Enabling BPEL Process Design and Editing](#) on page 126 for more information.
2. Follow the procedures outlined in [Importing Designs](#) on page 31 to import your BPEL process.
3. Edit the XML namespace definition.

To use the editor provided with ALBPM Studio, you must manually edit the namespace of your BPEL Process.

- a) Ensure that the Navigator View is visible.
- b) Expand the directory of the Project where you imported the BPEL process.
- c) Expand the processes directory.
- d) Right-click on your BPEL process.
- e) Select **Open With ► XML Editor**
- f) Ensure that the BPEL namespace declaration points to this URL:
`http://schemas.xmlsoap.org/ws/2004/03/business-process/`

The namespace declaration is the `xmlns:NAME` attribute of the top-level `<NAME:process>` element (where NAME may be any valid xml namespace).

For example, if you have defined your XML namespace as:

```
<bpel:process
xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
  xmlns:ns1="test"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" exitOnStandardFault="yes"
  name="test1" suppressJoinFailure="yes"
targetNamespace="http://test1">
```

You must replace the `xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"` with `http://schemas.xmlsoap.org/ws/2004/03/business-process/`

- g) Save your BPEL process.
4. Import the WSDL File
 - a) Ensure that the Navigator view is visible.
 - b) Expand the folder of the Process where you want to import the BPEL process.
 - c) Right-click on the process folder, then select **Import**.
 - d) Expand **General**
 - e) Select **File System**.
 - f) Click **Next**.
 - g) Browse to the directory containing the WSDL file for the BPEL process you imported above.
 - h) In the left-hand pane, click the checkbox next to the folder name.
 - i) In the right-hand pane, select the WSDL file and any other files you need to import.
 - j) Click **Finish**.

Opening a BPEL Process

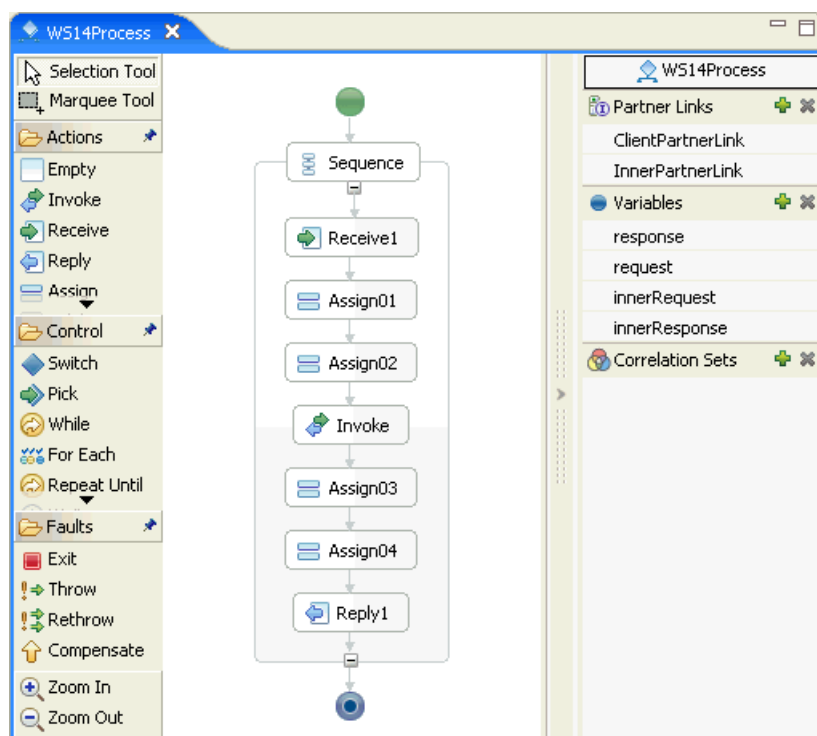
The following procedures show you how to open an existing BPEL Process in ALBPM Studio.

1. Ensure that the Navigator View is visible.
See [Showing Views](#) on page 19 for more information.
2. Expand the Project folder containing your BPEL Process
3. Expand **processes**.
4. Double-click on your process.

The BPEL process model appears in the BPEL editor window.

Using the BPEL Process Editor


ALBPM Studio provides a plugin-based BPEL process designer. This plugin is part of the Eclipse BPEL project. The BPEL editor plugin provides a full featured BPEL process editor.



See <http://www.eclipse.org/bpel/users.php> for tutorials and more information on using the Eclipse BPEL Designer.

Views Used by the BPEL Process Designer

In addition to the main editor window, the BPEL Process Designer also uses the following Eclipse Views:

View	Description
Navigator View	Provides a tree-structure view of the entire ALBPM Studio Project. BPEL processes and their corresponding .wsdl files are stored in the processes subfolder.
Outline View	Provides a tree-structure view of the components with a BPEL process.
Properties View	Provides information about the properties of the selected component within the BPEL process editor.  Note: This Properties View is different from the ALBPM Properties View.

See [What is a View?](#) on page 12 for more information on using Views in ALBPM Studio.

BPEL Example Project

ALBPM Studio provides an example BPEL project that demonstrates how to integrate BPEL processes within a native ALBPM process.

This project is available in `BEA_HOME/studio/samples`. See [ALBPM Examples](#) on page 10 for a list of other example projects.

Exception Handling

Exception Handling in ALBPM

ALBPM provides multiple ways of handling exceptions that occur outside of the normal flow of a program. The specific way used depends on where the exception occurs and what causes it.

Within ALBPM exceptions can be classified according to the following distinctions:

- System versus Business Exceptions
- Code-level versus Process-level Exceptions

System Versus Business Exceptions

This distinction defines the nature of the exception.

A system exception occurs when there is a problem with one of the components used by a process. These components can include databases, network connections, etc. System exceptions are included in the catalog as part of the standard Java exceptions.

A business exception is designed as part of a process business process, but is something that occurs outside of the normal flow of a process. This allows you to create cleaner processes where the main flow follows the normal use cases. System exceptions are defined as BPM objects within the catalog.

Another major difference between system and business exceptions is that business exceptions do not roll back activity transactions. This is because business exceptions are considered as a normal part of the process design rather than an error.

Code-level Versus Process-level Exceptions

This distinction defines where an exception is handled. All exceptions originate at the code level. However, they can be handled at either the code or process level depending on the requirements of your process design.

Code-level exception handling occurs within the scope of a PBL program. Code-level exceptions handling allows you to write code that directly accounts for the exception within the PBL task where it occurred.

Process-level exception occurs as part of the process design. When an exception occurs that is not explicitly handled within the code, it is propagated up to the process level. Process-level exceptions are handled within in exception flows.

Cataloging Exceptions

All exceptions are stored in the Catalog. System exceptions are stored as standard Java exceptions. Business Exceptions are stored within a user-defined BPM Objects.

System Exceptions

System exceptions are often caused due to temporary errors such as connectivity failure or timeouts. These are often temporary errors that can be resolved by re-running the method.

System errors can be handled at either the code or process level. System exceptions always start at the code level. If the exception cannot be resolved at the code level, it should be sent up to the process level for handling. The process should be to anticipate any system exceptions that occur and cannot be handled at the code level.

All system exceptions have a corresponding java exception that is included as part of the catalog.

Engine Exceptions

Time Out and Execution Aborted exceptions are generated by the Process Execution Engine and cannot be caught within a PBL program or as part of an exception handling flow. These exceptions are generated during the execution of Interactive and Automatic activities.

Business Exceptions

Business exceptions are exceptions that occur outside of the main flow of your process. Unlike system exceptions, business exceptions are implemented as part of your business process model. However, they are handled as exceptions because they fall outside of the main flow. A credit score check, for example, could be handled as a business exception.

Business exceptions allow you to create cleaner processes and allow you to define exceptional situations of the process as true exceptions.

Business Exceptions and the Catalog

Business Exceptions must be defined as BPM Objects within the Catalog. They behave like other BPM Objects and can contain methods, attributes, and presentations. They can be used anywhere within a process.

Throwing a Business Exception

Within a PBL program, you can manually throw a Business Exception using the throw keyword as in the following example:

```
if creditScore > 700 then
  create scoreTooLow ex = ScoreTooLow()
  ex: value = scr
  throw ex
```

Code-level Exception Handling

Code-level Exception handling allows you to write code to deal with problems that occur within the scope of a PBL task.

PBL uses the following block structure to handle exceptions:

```
do
on Exception
on Exit
end
```

The do-end block, though not required, allows you to clearly define the scope of an exception within your code. This is particularly important if you need to nest exceptions within multiple do/end blocks.

The on Exception statement allows you to define and execute the exception code.

The on Exit statement allows you to perform any clean up that is required as a result of the exception. This can include cleaning up temporary systems files or database table created within your PBL code.

Exception Handling Example.

The following code example demonstrates the syntax for using the on Exception and on Exit constructs.

```
on Objects.TooCloseToDueDate do
  sendAlert project
    using mailSubject = project.name + "is getting too close to due date",
      mailMessage = "The project " + project.name + "is getting too close to due
date."
end

on Objects.Overdue do
  sendAlert project
    using mailSubject = project.name + "is overdue",
      mailMessage = "The project " + project.name + "is overdue."
end

on exit do
  sendAlert project
    using mailSubject = "Work on project " + project.name + " was started.",
      mailMessage = "Work on project " + project.name + " was started on " +
project.startDate
      + ". The project leader for is: " + project.projectLeaderId
end
```

This example uses the default PBL programming style. Other programming styles use different keywords for exception handling, but the underlying concept is the same. If you are using the Java programming style, this is implemented within a try-catch block.

Propagating Exceptions to the Process Level

In many cases, you may not want to catch code exceptions. Within your process it may make more sense to let them propagate up to process-level exceptions. Any exceptions you choose to explicitly handle within a PBL program must be able to be resolved within the code. If an exception cannot be completely resolved it should propagate up as a process-level exception.

Handling Exceptions Directly in a Method

You can also handle exceptions directly in a method. See [Compound Statement](#) on page 284 for more information. can also be directly included in an onException. The do/end block is implicit/ contains the do/end block.

This also allows you to narrow the scope of exception catching.

Process-level Exception Handling

Process-level exceptions occur when a code/PBL exception occurs and is not handled. The exception is then propagated to the process level.


When a process-level exception occurs, it prevents the normal flow of the instance from continuing. Process-level exceptions frequently occur within an Automatic Activity.

Typical Exception Handling Flow

The following list demonstrates how exception handling occurs within a typical process:

1. The Process Execution Engine begins executing the process instance.
2. An exception occurs within an Activity at the code level.
3. If exception handling code is available, then that code is executed.

The Activity completes successfully, transactions are committed, and the Process Execution Engine continues with the next Activity in the instance.

4. If no exception handling code is available, then the Process Execution Engine cannot continue. The exception is propagated to the Process level
 5. At the process level, the following options are possible:
 - If no exception flow has been defined, the instance continues directly to the End Activity and the STATUS predefined variable is set to ABORTEDSTATE.
-  **Note:** When designing your business process, you should always create at least one default exception handling flow.
- If an exception flow has been defined, the flow of the process instance continues through the process flow. If you have defined different exception transitions, the flow is routed through the appropriate transition.
6. Process flow continues through each Activity in the Exception flow.
 7. In the Final Activity of the Exception Flow, the Process Execution Engine evaluates the ACTION predefined variable to determine where to continue the flow of the instance.
 8. Based on the value of the ACTION predefined variable, instance flow is returned to the main process flow.

The Project Catalog

Any non-trivial BPM project will usually need to communicate with outside resources, store complex data about each instance, or interact with the user through a Web-based interface. This is done by using standard or user-defined BPM Objects.

BPM Objects are organized by *module*. Each module can contain BPM Objects, other modules, or both.

Standard Modules








The following standard modules are included in every new project:

- Fuego
- Java
- Plumtree

You cannot add your own BPM Objects or sub-modules to the standard modules.

User-Defined Modules

You can add your own modules to the project catalog. You can also add modules within modules, organizing them in the manner of a directory structure. Within a module, you can create the following components:

Component	Description
 Module	Another module nested within the existing module.
 BPM Object	A user-defined component that contains <i>attributes</i> , <i>methods</i> , and <i>presentations</i> .
 BAM Dashboard	A Business Activity Monitor presentation
 Enumeration	A data type with a fixed set of possible values.
 Business Exception	An exception triggered by a foreseeable business condition.
 PUnit Suite	Unit test suite for processes.
 CUnit Suite	Unit test suite for individual BPM Objects.

You can also catalogue external components into your module. External components are components which contain the information that Studio needs in order to communicate with the software application, service, API, database, or other software resource your project needs to exchange information with.

A catalogued external component will act as the "glue" between studio and a given external resource. It provides an interface to that resource that can be used from PBL code. The following external component types are available:

- .NET Component
- AquaLogic Service Bus
- BPM Object
- COM
- CORBA Service
- EJB
- Enumeration
- JMX
- JNDI
- JPD

- Java
- SAP
- SQL
- SQL Query
- Web Service
- XML Schema

BPM Objects

What is a BPM Object?

A BPM Object is a user-defined component that contains *attributes*, *methods*, and *presentations*.

A BPM Object can be used to encapsulate any type of information that the process requires. For example, It can be to input and store information that would be persisted in another type of data container, such as a database or an XML file.

A BPM Object is composed of:

BPM Object Element	Description
Attributes	<p>Attributes are data elements (like variables), used to store data that define and describe the BPM Object.</p> <p>Attributes can be defined as virtual, in which case they do not actually contain any data, but are instead implemented as a pair of methods for reading and writing. Virtual attributes are accessed like regular attributes.</p>
Methods	<p>Methods are like functions or subroutines associated to the object. Youwrite methods in PBL, and can use them to access or set BPM Object data indirectly. For example, you may want to obtain the sum of several numeric attributes. In this case you can read each attribute from the BPM Object and add them, or you can add a method to the BPM Object, called for example <code>attributeSum</code>, which will return the summed value.</p> <p>The resulting code is easier to maintain. For example, if you add a new attribute to the BPM Object which must also be summed, you can edit the <code>attributeSum</code> method to include the new attribute. By doing it this way, you avoid the need to track every piece of code which requires the sum of the attributes of the object.</p>
Groups	<p>Groups are objects made up of one or more related attributes and stored in an array. Groups are designed to be used wherever you require a list of items and each item has several attributes. For example, in an invoice there can be several items, and each item has a description, a quantity, and a price.</p> <p>If you are a Java programmer, you can think of a group as being analogous to an inner class.</p>
Presentations	<p>BPM Object Presentations are essentially forms which either show or allow input of BPM Object properties. A presentation can show all or some of the properties of a BPM Object</p> <p>Every BPM Object may contain one or more presentations. Each of the fields on the presentation are tied to one of the attributes. Presentations provide a simple way for end users to view or to input the attributes of the BPM Object.</p> <p>When a BPM Object is created, it is only a data container or non-presentable BPM Object until a presentation is added to it; then it becomes presentable.</p> <p>See User Interface Overview<i>Most ALBPM processes include one or more interactive activities, meaning activities which are carried out by people who are participants in the process. For people to interact with an activity, you will need to build a user interface.</i> for more information about how BPM Object data is presented to end users.</p>

Example BPM Object

The following figure shows the structure of an expense report BPM Object:

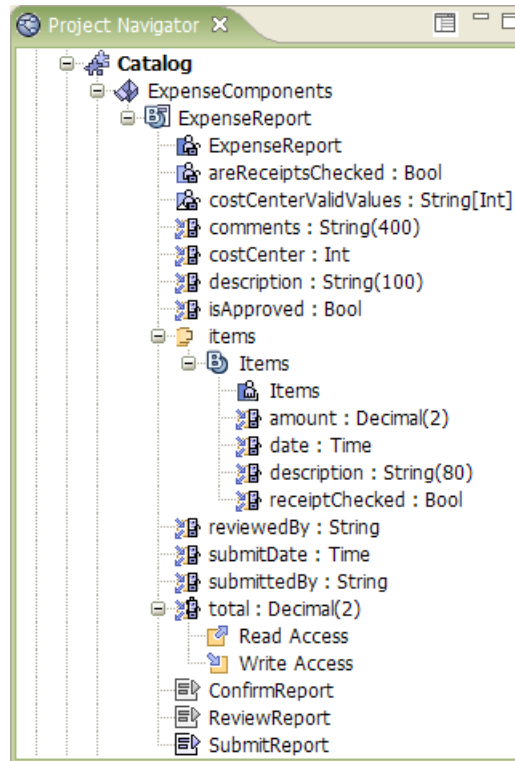


Figure 5: Expense Report BPM Object

The BPM Object above shows methods, attributes, groups, and presentations. Most of these were added by the developer, while a few of the methods were created automatically based on the properties set when creating the different attributes. An example of an automatically generated method is the `costCenterValidValues` method. This method is created by setting a list of valid values to the `costCenter` attribute when defining it. This method returns an array containing all the values allowed for that attribute, which is used by presentations to display a drop-down list with the possible values of an attribute.

Benefits of Using BPM Objects

BPM Objects provide a number of benefits:

- Once built, BPM Objects can be reused across multiple processes, projects, and Engines.
- The time invested in process development and maintenance is reduced because of BPM Object reusability and the wizards provided by Studio.
- BPM Object methods are coded in Process Business Language, which is simple but powerful.
- BPM Object method bodies can be completely changed, and as long as their interface is not changed, the calls in your process can remain the same. This allows you to change the behavior of a BPM Object across your processes and projects by modifying just a few lines of code. This reduces the time cost invested in maintenance, reducing the whole project time cycle, but gives you flexibility when changes are required
- External components implemented in different technologies can be integrated by wrapping them in a single BPM Object. The functionality provided by multiple technologies and applications can be combined in order to create a completely new business service, which is not limited to a specific technology. A single BPM Object can be used to exchange information with all the supported external components.
- BPM Objects can be used to customize the information display to different users based on their role, authorization, and/or training.
- Coding time and effort can be saved by using a BPM Object and coding with PBL rather than C++ or Java.

What is an Attribute?

Attributes are the data elements, such as numbers, strings, or date values, that describe the state and contents of a BPM Object.

Read and Write Access

If an attribute has no read access, this means that its value is not accessible. On the contrary, if an attribute has no write access, its value cannot be changed.

All real attributes have both read and write access set by default. When an attribute is added to a BPM Object, it is created as a real attribute by default. When access methods are added to real type attributes, they override this default. When access methods are removed, they reset to the default. These changes never modify the read/write access of the attribute.

An attribute can be redefined as real or virtual by selecting or deselecting the Virtual check box available for the attribute in the BPM Object editor panel.

You must define at least one access method for a virtual attribute. If no access method is defined, the BPM Object compilation fails.

Read access has a return type identical to the attribute's return type. Write access receives an argument called `value`, which is of the same type and has no return type.

When to Use a Real or Virtual Attribute

You must use a real attribute when you need to store a data element. You should use virtual attributes to expose values which can be calculated from existing real attributes. It may not always be obvious which values should be calculated and which should be directly stored. In the temperature example above we could have stored the Fahrenheit temperature while calculating the Celsius temperature

As a general rule, you should store what you consider to be the most natural value or the value you expect to use the most. The main thing is to avoid storing redundant information, since you risk ending up with divergent values for the same thing. Continuing with our example, if the object stores the temperature in Fahrenheit and also in Celsius, it will be possible to change only one of the values. The BPM Object would then contain two divergent values.

Attribute Data Types

Available data types to define an attribute are:

Type	Description
Bool	A boolean (True or False) value
Int	Integer number
Decimal	Decimal number value with defined precision
Real	Real (floating point) number
String	Text string
Time	A date, time, or date-time value
Interval	A time interval
Binary	Binary container, for example, an image, a file, etc.
Any	Can hold any data type. Similar to a Visual Basic variant.

When defining the attribute type by clicking the browse button next to the Type field, you can select any cataloged component in the Project Catalog as the defined type for the attribute you are declaring. This gives you the ability to typify an attribute within a BPM Object with any component defined in the Project Catalog, even another BPM Object. If it the attribute is typed after a different BPM Object, it is referred to as an Inner BPM Object. Find an example in the Inner BPM Object section below.

Required Fields When Defining an Attribute

The following table shows the required fields for defining an attribute:


Field	Description	Attribute type
Type	Domain type of data that the attribute contains.	Real & Virtual attributes.
Not null	The attribute may be assigned, or not, a null value.	Attribute must have write access.
Primary Key	Is the attribute part of the primary key of the BPM Object?	Attribute must have read access.
Time Precision	Only for Time attributes, you can choose the following: Timestamp : The entire date and time, Date only : The date, Time only : The time.	Date attributes.
Absolute	The value of a time attribute which is set as absolute is stored in GMT-0.	
Maximum length	For String. The Maximum length field defines the maximum number of characters allowed for the attribute.	Real & Virtual attributes.
Default Value	For Int, Decimal, Real, String, Time and Interval types. Initial value that the attribute contains. A default value for a Bool type attribute is defined by selecting the True or False radio button.	Attribute must have write access.
Require Expression	The Require Expression field allows you to enter a Boolean expression to be checked as a precondition to the assignment of a value to the attribute. It can be built either by a simple, one line expression that evaluates an expression or by calling a method. This method must return a boolean type.	Attribute must have write access.
Check Expression	Check Expression defines integrity validation of the attribute within the BPM Object instance. It can be built either by a simple, one line expression that evaluates an expression or by calling a method. This method must return a boolean type.	Any attribute.
Valid Values	List of valid values that an attribute can be assigned. It can be built either by a hard-coded list of valid values that the attribute can contain or by a call to a method that returns a list (array) of valid values. This method must return an array of the same domain type as the attribute's type. The implementation allows you to show the actual value or its description. For further details, see section Valid Values on this page.	Any attribute

What is a Presentation?



A presentation is an interface where the participants of the process can input or review data when a process instance reaches their activity.


Creating a BPM Object

You define BPM Objects in several steps. The first step is to create the BPM Object, where you will specify its basic properties. Afterwards, you can add attributes, groups, methods, and presentations.


 **Note:** If you are familiar with object-oriented programming, you can think of creating a BPM Object as defining a class.

To create a BPM object:

1. In the **Project Navigator**, within your project, expand **Catalog** ().
You will see a list of catalog modules. If you are working on a new project, these will be the standard modules *Fuego*, *Java*, and *Plumtree*.
2. You define BPM Objects in your own modules. To add your own module, right-click on Catalog and click **New ► Module** ().

 **Note:** You can define several BPM Objects in one module, so you do not need to execute this step if you already have defined a module.



The **Module** dialog box appears.

3. Enter a name such as `MyModule` in the **Module Name** field, and click OK.
The module you specified is added to the catalog.
4. Right-click on the module icon and click **New ► BPM Object** ().
The **BPM Object** dialog box appears.
5. Enter a name for the new object, such as `MyObject`, in the **Name** field. Click OK.
The BPM object is added to the module.
6. Expand your module and then expand your new BPM object.
You will see the contents of the new object. It contains one *method*, with the same name as the BPM Object. This is the *constructor* method, which means that it will execute whenever an object of this type is created (or "constructed"). If you need to include code that will initialize something in the BPM object, you can add it to this method.

Defining an Attribute

You define an attribute in Studio from the Project Navigator.

To define a BPM Object attribute:

1. In the Project Navigator, right-click on the BPM Object (), and click **New ► Attribute** ().
The **Attribute** dialog box appears.
2. Enter the attribute name in the **Name** field.
3. Select the data type from the **Type** drop-down list, and click **OK**.
The attribute editor for the new attribute opens.
4. For some data types, you can specify length or precision, as shown below:

Data Type	Property	Possible values
String	Maximum Length	1 to 10,000
Decimal	Decimal Digits	0 to 100
Time	Time Precision	Date Only, Time Only, or Timestamp

5. In the **Storage Constraints** section of the editor, you can check one or more of the following:

Storage Constraint	Description
Virtual	If set, the attribute will not hold data. Instead, you must define a Read Access method, a Write Access method, or both. See Virtual Attributes When an attribute is defined as virtual, it means that it stores no data value, and is defined by its read and write access methods. .

Storage Constraint	Description
Primary Key	If set, the attribute will be used as an identifier to determine if two BPM Objects are the same. You would typically set this for an attribute containing a unique value, such as an ID number.
Not Null	If set, a null value is not allowed, and you will need to set a default value in the Default Value section.

- In the **Default Value** section, you can set an initial value the attribute will be given when the BPM Object is created. If you did not set the **Not Null** option, you may also set the default value to Null.
- In the **Required Expression** field, you can enter a PBL expression that will be evaluated when the attribute field is changed in a presentation.

For example, you can make sure the user enters a value greater than zero with the following expression:

```
attributeValue > 0
```

- Save (💾) the changes and close the editor window by clicking on the **X** in the *bottom* tab. If you close from the top tab, you will close the BPM Object.

Valid Values

BPM object attributes can be configured with a set of predefined valid values. When you do this, the attribute will be presented to the user as a drop-down list, rather than a field. Thus, the user will only be able to select from the choices provided.

There are three possible **Valid Values** settings for a BPM object attribute:

Valid Values Setting	Description	Presented As
All	Any value within the bounds of the data type is accepted. This is the default setting.	Text field
Static List	One of a list of values is accepted. The list is fixed at design time.	Drop-down list
Dynamic Method	One of a list of values is accepted. The list is dynamically built by a method in run-time.	Drop-down list

The dynamic method is more flexible. For example, you can pull the information from a database. The static list is easy to configure without writing any code.

Value Descriptions

You will often want end-users to choose among a set of values they can easily recognize, while in the background you need an identifier to return to your system. A typical situation is with a database, where you might want to obtain a record ID that has no meaning to the user. In other cases, you may need an expense account number or an abbreviated code of some kind, such as a country code, while you want the user to be able to choose from a more descriptive list, such as actual country names.

This can be easily done either with the static list or dynamic method options by choosing to use value *descriptions*.

Setting a Static Valid Values List

A static valid values list is set at design time. You can use a static valid values list when you know that the list is unlikely to change very often, and when it will contain relatively few options.

The main advantage of a static valid values list is easy to configure, and does not require writing any code. If you want the list of valid values to change frequently, you are better off using a dynamic list obtained from a database or other data source.

To define a list of valid values:

- In the **Valid Values** section of the attribute editor, select **Static List**. and check the **Edit Value Descriptions** option.

The Values table will appear.

2. You can choose to add value descriptions by checking the **Edit Value Descriptions** checkbox.
If you set this option, a *Description* column will appear in the table.
3. To add an entry to the table, click on the Add icon (+) and enter a numeric value in the *Value* column and, if you've set value descriptions, also add a descriptive text in the *Description* column.
To delete an entry in the table, select the entry and click on the Remove icon (-).
4. Once you are done adding entries to the list, make sure the **Default Value** field is set to a value (not a description) that exists on the list.



Note: When an attribute is configured with a valid values list, the **Default** field must be set to one of the valid values. If it isn't, an error condition is indicated.

5. Save your changes and close the editor for this attribute.

Defining a Valid Values Method

When a list of valid values will change frequently, you will usually choose to load it dynamically. To do so, you must write a PBL method which will return the list or choose an existing method or array attribute.

The advantages of using a PBL method to populate a list of valid values are that you can use data from any source, and that you read this data when it is needed, so it is always current. You can also write a method that generates values with an algorithm and does not obtain data from an external source.

You can load a list of valid values dynamically using:

- A method that obtains the values from an external source, such as a database or a Web service, or that generates the values in some other way.
- An array attribute of the same data type as the attribute the valid values are for. For instance, an `Int []` array can hold valid values for an `Int` attribute.

To define a dynamic method valid values list:

1. In the **Valid Values** section of the attribute editor, select **Dynamic Method**.
The **Method Descriptions** option and the **Method or attribute used to load valid values** panel appear. In the panel, the drop-down list contains currently available methods which return an array of the data type required, as well as array attributes also of the required data type.
2. If you want the valid values list to contain value-description pairs, click on the **Method Descriptions** option.
If you check this option, an associative array is expected. If not, an indexed array is expected. Therefore, the list of matching methods and attributes will depend on your choice.
3. You can select an existing method or attribute from the drop-down list. If you need to create a new method, go to step 4. Otherwise, select the method or attribute and save your changes.
4. To create a new method click **New**.
The **Method** dialog box appears.
5. Enter the name of the new method in the **Method Name** field, and click **OK**.
A PBL method editor opens. In the **Properties** window, note that Studio set the return type to match that required by the attribute.
6. If your method will access a database or other catalog component with external dependencies, set *Server Side Method* to *Yes* in the **Properties** window.



Note: In ALBPM, a server side PBL method executes in the process execution engine. If a method is not server side, it executes where it is called, and may execute in the Workspace server. For the purposes of this setting, the Workspace considered to be the "client". PBL methods are never executed at the browser.

7. Enter your method into the editor. Your method must contain a return statement at the end which specifies the variable to be returned. See the examples below.
8. Save your changes and close the PBL editor and the attribute editor.

Examples

The following PBL method loads a list from SQL table `suppliers` into the valid values list of an integer attribute. This table was previously introspected into the project catalog as an SQL component. In this table, `supplierId` is an `Int`, and `supplierName` is a `String`.

In this case the valid values list has descriptions, so an associative array in the form `String[Int]` must be returned:

```
listValues as String[Int]

for each record in
    SELECT supplierId, supplierName
    FROM suppliers
do
    listValues[record.supplierId] = record.supplierName
end

return listValues
```

The following loads the first 11 numbers of the Fibonacci sequence into a valid values list with no description, so it returns an indexed array:

```
fNumber as Int[]

fNumber[0] = 1
fNumber[1] = 1

for n in 2..10 do
    fNumber[n] = fNumber[n - 1] + fNumber[n - 2]
end

return fNumber
```

Creating a Presentation

In Studio, you can build a presentation quickly by using the Presentation Wizard.


To create a presentation, you should already have defined a BPM Object with one or more attributes.

To create a presentation:

1. In the Project Navigator, right-click on the BPM Object (B) you will create the presentation for, and click **New ► Presentation** (P).
The **Presentation Wizard** appears.
2. Enter the name of the new presentation in the **Presentation** field.
3. You want your presentation to be based on the BPM Object. To select this, check the **From Template** checkbox. Do not check the **BPM Preferences** checkbox.
4. Click **Next**.
The **Presentation Referenced Attributes** page of the **Presentation Wizard** appears.
5. In this dialog box you select the attributes which will be shown as fields in the presentation. Select the attributes that you want the user of this presentation to see or edit.

The fields will be placed in the presentation as they are ordered in the attribute list. You control this order either by adding each attribute in the desired sequence, or by selecting an attribute in the list and using the **Up** and **Down** buttons to place it in the proper position.

6. Click **Finish**.
The new presentation is created, and a presentation editor is opened with it.
7. Make sure the **Properties** window is open in Studio. It should be on the right side. If not, open it by clicking **Window ► Show View ► Properties** from the menu.

8. By default, each attribute is added to the presentation as an input field. If you don't want the user to enter or edit an attribute value, select it in the presentation editor and go to step 9. Otherwise continue to step 10.
The properties for the text field appear in the **Properties** window. The *Name* property of the text field should be `items_text2`.
9. Set the *Editable* property to *No* simply by clicking on the property.
10. Save your changes.
11. You can preview what the presentation will look like. To do so, click on the HTML Preview () icon in the presentation editor.

An HTML preview of the presentation is displayed in your browser.
12. Close the presentation editor.

BPM Object Catalog

What is the BPM Object Catalog?

The Project Catalog enables you to catalog the components and organize and group them into modules by some logical criteria. Therefore, a catalog contains the following elements:

Modules	Containers that create logically grouped components.
Components	Business Objects that are available to Process Methods.

Each project has its own catalog of components, so all the processes inside the project are checked and compiled using this catalog. Every time a process is published, the jar file of the BPM Object catalog is also published to the Directory Service. For further information on this topic, see *Implementing Business Objects using BPM Objects*.

General Features of the BPM Object Catalog

The BPM Object Catalog provides the following functionality:

- Supports the ability to efficiently store and manage up to 100,000 components.
- Supports unlimited nesting levels of components.
- Achieves the above requirements with minimum memory utilization and maximum performance.
- Provides simple wizards to simplify configuration and introspection of external resources Java, EJB, SQL, CORBA, etc.

The project catalog of components is stored in your local hard drive. The files are located in the `componentCatalog` directory that corresponds to the project.

The Project Catalog includes a set of default components. Go to the Help menu and clicking the Component Index option you get the list of all the standard components, the category or module to which it belongs to, and a brief description about it. If you double click on the component, a new tab in the main panel opens for that component. To learn about them please refer to *Implementing Business Objects Using Library Components*.

BPM Object Catalog Files

Project folders contain a folder called `componentCatalog`. This folder contains the structure of your catalog, which is a directory tree that corresponds to the catalog modules, and an `xcdl` (XML Component Definition Language) file for each defined module and component.

When the project is published and deployed, the configurations defined when cataloging the components are deployed in FDI, thus automatically creating a pair. At the same time, the BPM

Object class is attached to the project. For the java components. For further information, see Publish & Deploy.

Modules

Modules are used to logically group components. Grouping enables you to quickly find a specific component.

Modules can be nested in an unlimited quantity of levels provided that:

- Introspectors (Java, SQL, etc) are able to return a tree as the result of the introspection instead of a list of components.
- Groups for BPM Objects are components defined inside the BPM Object and they can be used from outside the group.
- PM Objects can implement the Inner classes because nested levels are allowed.
- It is possible to organize components from default cataloging sub-modules.

AquaLogic BPM is installed with two default modules, java and Fuego. These modules contain standard components, other modules, components, and Java implementations that run many functions in AquaLogic BPM. Some of these standard components are also useful when you are designing your business processes. See Implementing Business Objects using Library Components for further information.

Components

Components are the elements that work with each other to create a system. Generally, any system that exposes its Application Program Interface (API) can be introspected into the Project Catalog, as long as it exposes the API through one of the following technologies:

- Automation
- Java
- XML
- Web Services
- Enterprise JavaBeans (EJB)
- Java Naming Directory Interface (JNDI)
- SQL
- CORBA

Once a component has been added to the catalog, it can be integrated into the project processes, thus enabling you to utilize existing applications already in use in your company structure. For example, you can introspect components of an existing database into the Project Catalog and then use them in the process method of a Global activity to give users the ability to query or modify information contained in the database.

There are two types of components utilized by AquaLogic BPM: Library Components (Java components) and user cataloged components. Library components and Java components are already defined and included with Studio. Components cataloged by the user are stored in user created modules in the Project Catalog.

BPM Objects can be used as containers to hold several unrelated components. In this way, you can create your own specialized business service by combining several different types of components into one BPM Object.

Documenting the Catalog

Studio provides the ability to create, modify, and delete documentation for each component. Documentation consists of component comments (defined by the designer at component creation), methods, attributes, etc. (as with JavaDoc.)

For detailed information, please refer to Documenting a Project.

External Components

.NET Component

What is a .NET Component

Provides a general description of the .NET Catalog Component.

Microsoft® .NET enables a software integration through the use Web services. Microsoft Visual Studio® .NET and the Microsoft .NET Framework allow developers to develop XML Web services and integrate them easily with other applications.

Integrating .NET Assemblies

Assemblies are the building blocks of .NET Framework applications. An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. An assembly provides the common language runtime with the information it needs to be aware of type implementations. To the runtime, a type does not exist outside the context of an assembly.

If you plan on calling an external application that exposes itself using .NET Assemblies, from your process or BPM Object methods, make sure you read the .NET documentation provided with that application (or contact the software vendor for further information).

Using the .NET Bridge

.NET Bridge is a Windows application that acts as a 'bridge' between BPM applications and .NET Assemblies. AquaLogic BPM supplies this application to provide all the necessary services to introspect and use .NET components.

The .NET bridge runs as a standalone process. Is itself a .NET application, so it runs in the CLI (the CLI is managed automatically by Windows).

All components called by the bridge, share the same CLI than the bridge itself since they are called using System.Reflection APIs (it follows that they share the same process).

Cataloging .NET Assemblies

Before using .NET Assemblies from processes or BPM Objects, they should be cataloged into the project's catalog. When you catalog a .NET assembly, you are gathering all the necessary information that Studio needs in order to call and execute it at runtime. This information is also used by Studio to detect potential errors at compile time, which dramatically reduces the time spent in fixing errors in a process.

Cataloging a .NET Component

Microsoft .NET Service

Provides detailed information for configuring a Microsoft .NET Service.

General Properties

This section defines the general properties for this External Resources:

Property	Description
Name	Defines the name of this external resource.
Type	Specifies the external resource type.

Microsoft .NET Service Properties

The following properties must be defined for a Microsoft .NET Service:

Property	Description
Host	Defines the location of the .NET Bridge host
Port	Defines the port used by the .NET Bridge host

AquaLogic Service Bus

What is AquaLogic Service Bus?

AquaLogic Service Bus is a SOA

BEA AquaLogic Service Bus (ALSB) is an enterprise service bus designed for connecting, mediating, and managing interactions between heterogeneous services. ALSB can handle Web services, Java, .Net, messaging services, and legacy end points. BEA AquaLogic Service Bus is designed to handle the deployment, management, and governance tasks required to implement SOA at any scale.

AquaLogic Service Bus is stateless and policy-driven. It enables you to establish loose coupling between service clients and business services while maintaining a centralized point of security control and monitoring.

Cataloging an AquaLogic Service Bus Component

Before performing the procedures in this task, ensure that you have installed and configured AquaLogic Service Bus and that it is currently running.

Also ensure that you have created a module where you want to catalog the ALSB component. See [Creating a Custom Catalog Component](#).

To catalog a AquaLogic Service Bus Component:

1. Right-click on the module where you want to catalog the ALSB component.
2. Select **Catalog Component** ► **AquaLogic Service Bus**
3. Select one of the following options:


Option	Description
Use existing configuration	Select this option if you have already configured an External Resource for the ALSB Management Host Component.
Create a new configuration	Select this option if you need to configure a new External Resource for the ALSB Management Host Component. See Creating an External Resource on page 183.

4. Click **Next**.
AquaLogic BPM Studio connects to the AquaLogic Service Bus. This may take several minutes.
5. Select the ALSB Project you want to catalog.
AquaLogic Service Bus resources are organized into projects. You must select a Project that has at least one Proxy Service defined.
6. Select the Proxy Service you wish to use.
7. Provide a Module name.
This is the name that will appear in the Project Navigator. The default Module name is the name of the Proxy Service you select.
8. ALBPM Studio generates a catalog component for each of the Proxy Service elements.
9. Click **Finish**.

The introspected AquaLogic Service Bus module appears in the Project Navigator.

AquaLogic Service Bus Example

The following example shows how to access AquaLogic Service Bus objects from within a PBL program.

 **Note:** You must add AquaLogic Service Bus objects to the BPM Object Catalog before they can be used within Studio.

Accessing ALSB Objects

The following PBL program shows how to access an AquaLogic Service Bus object after it has been cataloged. This example is based on the default example that is provided in ALSB.

```
service as ServiceBus.LoanGateway1.MyService =
    ServiceBus.LoanGateway1.MyService();
request as ServiceBus.LoanGateway1.LoanStruct =
    ServiceBus.LoanGateway1.LoanStruct();

request.amount = 12345
request.name = 'John Smith'

//display request.name
processLoanApp service
    using loanRequest = request
    returning result2 = result

display result2.name
display result2.notes
```

AquaLogic Service Bus

Provides detailed information for configuring an AquaLogic Service Bus as an External Resource.

General Properties

This section defines the general External Resource properties:

Property	Description
Name	Defines the name of the external resource.
Type	Specifies the type of external resource.
Supported Types	Specifies the type of AquaLogic Service Bus connection.

Management Host Properties

The following properties must be configured for a Management Host:

Property	Description
Host	
Port	
User	
Password	

Proxy Service Properties

The following properties must be configured for a Proxy Service:

Property	Description
Host	
Port	
User	
Password	

Process Deployment Properties

The following properties must be configured for a Process Deployment:

Property	Description
Management Configuration	
Project Name	
WSDL Folder	
Business Services Folder	
WS-Security Account	
Transport	
Host	
Port	

COM Component

COM Component

COM components are software programs that use the Microsoft Component Object Model (COM).

COM is a set of standards that specify several constraints that each component should follow in order to comply with the standard, such as which types are valid for a method's argument, or how to manage memory.

COM objects can be created in several different ways and with different tools, such as Visual Basic or C++. Several applications expose functionality as COM objects - this includes most of Microsoft's applications (Office, Internet Explorer, etc.) and many third-party applications.

Integrating COM Components

If you plan on calling an external application that exposes itself using COM from your process or BPM Object methods, make sure you read the COM documentation provided with that application. For example, if you wish to call a Microsoft product, such as Microsoft Excel, you will need to understand Excel's object model, which is documented in the Microsoft Developer Network Website.

Using COM Components

Before using COM components from processes or BPM Objects, you must catalogue the component into the project's catalog. When you catalog a component, you are gathering all the necessary information that Studio needs in order to call and execute it at runtime. Studio also uses this information to detect potential errors at compile time.

See [Cataloging a COM Component](#) on page 151.

BPM COM Bridge

BPM COM Bridge is a Windows application that acts as a 'bridge' between AquaLogic BPM applications and COM. AquaLogic BPM supplies this application to provide all the necessary services to introspect and use COM components. It is in charge of asking COM invocations, type conversions and marshaling of arguments across the net.

BPM COM Bridge is packaged as a separate application to allow you to:

- Run your processes in a robust environment by effectively isolating the Engine from misbehaving components.
- Run the Engine in a different host environment from the one used to run COM components (e.g., run your Engine in a Solaris box and your COM components in one or several Windows servers or even use one COM Bridge with several servers).

Because COM components do not necessarily reside on the same machine as the Engine, COM Bridge should be installed on the machine where the component resides. However, this is not absolutely necessary for all components because DCOM components, if properly configured, can be located in a different machine from the one that is running COM Bridge.

See [Installing COM Bridge](#) for information on installing COM Bridge.

There are no special considerations when the process that uses COM Components is deployed in a J2EE/UNIX environment. A BPM COM Bridge has to be running on a Windows box accessible through the LAN and it will work as any other service (like a data base, for example).

COM Bridge

ALBPM COM Bridge allows you to catalogue and use COM components in ALBPM processes.

The COM Bridge acts as a connector between COM Objects and Studio. There are two versions of AquaLogic BPM COM Bridge:

combridge.exe	This program is installed with Studio and runs automatically when required. It is intended for the development stage of a project.
combsvc.exe	Is the service version of BPM COM Bridge. It is intended for production environments, where no GUI interaction is required with the COM components or applications that are being automated. After installation, it will start whenever the machine starts.

Running the COM Bridge

Standalone Version

Command	Description
combridge -install	Installs the COM Bridge to start on every login.
combridge -remove	Uninstalls the COM Bridge.
combridge -debug	Disables asynchronous logging.
combridge -stop	Stops another COM Bridge running.
combridge -log filename	Sets the log file name (default is %TEMP%\COMBridge.log).
combridge - ?	Displays COM Bridge usage dialog
combridge [option] [port]	The port is the TCP port where COM Bridge is listening to incoming calls. Defaults to port 4042.

Service Version

Command	Description
combsvc -install	Installs the COM Bridge as a service.

Command	Description
combsvc -remove	Uninstalls the COM Bridge as a service.
combsvc -debug [params]	Run the COM Bridge as a console application for debugging.
combsvc -start	Starts the COM Bridge service.
combsvc -stop	Stops the COM Bridge service.
combsvc - ?	Displays COM Bridge service commands.

Cataloging a COM Component

Before performing the procedures in this task, you should ensure that the COM Bridge is installed and running on your system. See [Installing COM Bridge](#) for more information.

You should also ensure that you have created a module where you want to catalog the COM component. See [Creating a Custom Catalog Component](#).

To catalog a COM Component:

1. Right-click on the module where you want to catalog the COM component.

2. Select **Catalog Component ► COM**

3. Select one of the following options:

Option	Description
Use existing configuration	Select this option if you have already configured an External Resource for the COM Component.
Create a new configuration	Select this option if you need to configure a new External Resource for the COM Component. See Creating an External Resource on page 183.

4. Click **Next**.

AquaLogic BPM Studio creates a list of COM Type Libraries that can be cataloged. This may take several minutes.

5. Select the COM Type Libraries you want to introspect.

6. Click **Next**.

The libraries are analyzed for dependencies and introspected.

7. Click **Finish**.

The libraries you chose to catalog appear in the Project Navigator.

MS Word Example

To run this example you have to catalog the Microsoft Word Object Library under a module name ComCatalogation

The PBL methods included in this example are part of the WordExample project under the sample directory of your Studio installation. The input.doc Word document used in the second example is under the WordExample project directory, move it to 'C:\tmp' to make the process work or modify the line in the PBL to the real location of the file in your installation.

Creating a New Word File

The PBL below, shows how to create and fill with text a Word file. This PBL is the one corresponding to the activity 'CreateWordDocument' of the 'Word Document Creation Example' process.

```
filename = "C:\\\\WordTest" + id.number + ".doc"

Application.visible = false

// get ready to use Word
worddocs = Application.documents

// create a new Word document
worddoc = add(worddocs)

// find something specific in the document.
// in this case there isn't anything,
// but the rangevar variable is still needed
// for the next command which inserts the text.
rangevar = range(worddoc)

// tell Word where you want to put the sentence.
// I say after rangevar, but as you know from above,
// that isn't very specific.
insertAfter rangevar
    using text = sentence + "\\nFuego is the
        greatest software ever!"

// save as the filename you want.
// We built the name somewhat unique in a variable
saveAs worddoc
    using fileName = filename

// close the newly saved document
close worddoc

// quit
quit Application
```

Setting Values in an Existing Word File

The PBL below, shows how to complete form fields in a Word file used as template, and saved as a new file. This PBL is the one corresponding to the activity 'Input Word Document Example' of the analog process.

```
// Initialize variables
custName = customerName
worddocs = wordappl.documents
Application.visible = false

// Open Word file
open worddocs using
    fileName = "C:\\\\tmp\\\\input.doc" returning worddoc

// Initialize the form fields into their object
wordformfields = worddoc.formFields

// Select the form field you want to work on
item wordformfields using
    index = "Text2" returning wordformfield

// Set the value of the form field you selected
wordformfield.result = custName

// Save as a different Word file
saveAs worddoc using
    fileName = "C:\\\\tmp\\\\result.doc"
```



```
// Quit Word
quit wordappl
```

Microsoft COM Service

Provides detailed information for configuring a Microsoft COM Service as an External Resource.

COM components are software programs that use Microsoft Component Object Model (COM) technology. COMBridge is a Windows application that acts as a "bridge" between BPM applications and COM. It provides all the necessary services to introspect and use COM components.

General Properties

This section defines the general properties for this External Resources:

Property	Description
Name	Defines the name of this external resource.
Type	Specifies the external resource type.

Microsoft COM Properties

The following properties must be defined for a Microsoft COM Service:

Property	Description
Host	Indicates the host where the COM Bridge is installed
Port	Indicates the port where the COM Bridge is installed. The default is 4060.

CORBA

What is CORBA?

Studio allows you to catalog CORBA objects that reside in an Interface Repository. Once cataloged, you can manipulate the components of the CORBA object in your Method tasks in a process design.

Supported CORBA Objects

Any CORBA type can be cataloged through Studio into the Project Catalog. However, the following are the only types supported at run time:

- Interfaces with attributes and operations
- Structs
- Unions
- Sequences
- Enumerations
- Arrays
- Aliases

CORBA Module Implementation

The current implementation has several considerations to be kept in mind. The main constraints include POAs: the CORBA module uses Portable Object Adapters (POAs) instead of Basic Object Adapters (BOAs.) BOAs are not supported.

Method Considerations

The CORBA services provided by Studio support the following objects at runtime:

- Interfaces
- Structs
- Unions
- Sequences
- Enumerations
- Arrays
- Aliases

The Boxed Values object is not supported.

Primitive Types

The table below shows the primitive types that are implemented and how they are mapped to BP-Method types:

CORBA type	Method type
Boolean	Bool
char	String
wide char	String
string	String
wide string	String
octet	Int
short	Int
unsigned short	Int
long	Int
unsigned long	Int
long long	Int
unsigned long long	Int
float	Real
double	Real

Out arguments

The OUT argument modifier can be used for every primitive type listed in the table above. However, for **Struct** and **Union** objects there is a limitation, they have to be instantiated **before** being passed as an output argument. Hence, for structures and unions, an example Method is the following:

```
structTest = CorbaTestsTestStruct()  
  
structOutOp paramTest returning  
  
structTest = aTestStruct
```

and

```
testUnion = CorbaTestsTestUnion()
```

```

testUnion.aBoolean = false

paramTest = CorbaTestsParamTest("ParamTest")

unionOutOp paramTest returning

testUnion = aTestUnion

```

Summarizing, just as arguments should be instantiated with IN and INOUT arguments, arguments should also be instantiated with OUT arguments.

Union implementation

Union objects are implemented by using helpers that permit marshaling and un-marshaling to and from the server side. Before a union is marshaled, **any attribute of the union should be set**. In other words,

```

testUnion = CorbaTestsTestUnion()

// the attribute is set before the union is used as
// an argument
testUnion.aBoolean = false

paramTest = CorbaTestsParamTest("ParamTest")

unionOutOp paramTest returning

testUnion = aTestUnion

```

This is **required** for all operations involving the use of unions. Furthermore, since unions also have default attributes, they can be used as any other attribute. For example:

```

union ShortUnion switch(short)
{
    case0: string aString;
    case1: wstring aWString;
    default: short defaultShort;
};

```

Then, the default attribute is **defaultShort** and can be used as follows:

```

shortUnion = CorbaTestsShortUnion()

shortUnion.defaultShort = 1

unionTest = CorbaTestsUnionTest("ParamTest")

shortUnionTest unionTest using

aShortUnion = shortUnion

```

If no attribute is set and the union is used in a remote invocation, the execution framework will raise an exception indicating that any attribute should be set before the union is marshaled to the server side.

Struct implementation

The structure implementation is similar to the Union implementation. All of the structure members should be initialized before use. This means that no attribute (member) can be Null when it is used with remote invocations. Should this happen, the server will respond with a marshal error.

CORBA Sequence Examples

Sequences are bound in Methods. If the sequence length exceeds its bound, an exception is thrown. For example, in the following Method script the sequence has 11 members but can only have 10 members.

```
seqTest    = Module1.CorbaTests.SeqTest("SeqTest")
stringSeq = [ "1", "2", "3", "4", "5", "6", "7", "8",
              "9", "10", "11" ]

boundStringSeqOp seqTest using
aBoundStringSeq = stringSeq returning stringSeq

display stringSeq
```

When this example is run, it displays the following exception on the screen:

```
Exception:
Sequence size is incorrect, it is 11 and should have been 10.

line=5
column=1
```

Sequences of primitive types

Any of the primitive types shown in Primitive types can be used to create a sequence. The following example shows how to create a **string** sequence.

```
seqTest = Module1.CorbaTests.SeqTest("SeqTest")

stringSeq = ["one", "two", "three"]

stringSeqOp seqTest using aStringSeq = stringSeq
returning stringSeq

display stringSeq
```

Sequences of Structs

The Method script below shows how structure sequences can be used in the Method.

```
ts = 'now' // timestamp

seqTest = Module1.CorbaTests.SeqTest("SeqTest")

// create the first struct
structOne = Module1.CorbaTests.SeqTest.TestStruct()
structOne.longMember = 10
structOne.stringMember = String(ts)

// create the second struct
structTwo = Module1.CorbaTests.SeqTest.TestStruct()
structTwo.longMember = 20
structTwo.stringMember = String(ts)

// the array of structures
structSeq = [structOne, structTwo]

testStructSeqOp seqTest using

aTestStructSeq = structSeq
```

Sequence of Unions

Using sequences of unions is the same as using sequences of structs, as shown in this example:

```
ts = 'now' // timestamp

seqTest = Module1.CorbaTests.SeqTest("SeqTest")

unionOne = Module1.CorbaTests.SeqTest.TestUnion()
unionOne.longMember = 10

unionTwo = Module1.CorbaTests.SeqTest.TestUnion()
unionTwo.stringMember = String(ts)

unionSeq = [unionOne, unionTwo]

testUnionSeqOp seqTest using

aTestUnionSeq = unionSeq
```

Sequences as INOUT/OUT parameters

Sequences can also be passed as OUT or INOUT arguments in operation invocations. Note that when using sequences of structures or unions as OUT arguments, the restriction that these objects must be instantiated first before being used is still applicable, shown as follows:

```
seqTest = Module1.CorbaTests.SeqTest("SeqTest")

// union is instantiated
testUnion = Module1.CorbaTests.SeqTest.TestUnion()
testUnion.stringMember = "aString"

testUnionSeqOutOp seqTest returning

// union will be received in the sequence
testUnionSeq = aTestUnionSeq

unionOne = testUnionSeq[0]
unionTwo = testUnionSeq[1]
```

CORBA Enumeration Examples

CORBA enumerations can be used in many situations. The following examples show how to use the enumeration that appears in the Project Catalog after the IDL is cataloged.

Using enums in Operation Invocations

The following example shows how enumerations can be used in operation invocations or as the discriminator of a union:

```
module CorbaTests
{
  interface EnumTest
  {
    enum Slot { s1, s2, s3 };

    union UnionTest switch(Slot)
    {

      case s1: string  stringMember;
      case s2: long    longMember;
      default: boolean booleanMember;

    };
  };
}
```

```

void enumOp(in Slot aSlot);

void unionOp(in UnionTest aUnionTest);

Slot retEnumOp(in Slot aSlot);

void outEnumOp(out Slot aSlot);

void inoutEnumOp(inout Slot aSlot);
};
};

```

Using enums as IN Arguments

```

s1 = Module1.CorbaTests.EnumTest.Slot.s1

enumTest = Module1.CorbaTests.EnumTest("EnumTest")

enumOp enumTest using aSlot = s1

```

Using enums as OUT Arguments

```

enumTest = Module1.CorbaTests.EnumTest("EnumTest")

outEnumOp enumTest returning s3 = aSlot

```

Using enums as IN/OUT Arguments

```

enumTest = Module1.CorbaTests.EnumTest("EnumTest")

inoutEnumOp enumTest using aSlot = s2 returning s4 = aSlot

```

enum as the Return Type

```

s2 = Module1.CorbaTests.EnumTest.Slot.s2

enumTest = Module1.CorbaTests.EnumTest("EnumTest")

retEnumOp enumTest using aSlot = s2 returning s1

```

enum as the Discriminator of a Union

```

enumTest = Module1.CorbaTests.EnumTest("EnumTest")

unionTest = Module1.CorbaTests.EnumTest.UnionTest()

unionTest.longMember = 10

unionOp enumTest using aUnionTest = unionTest

```

```

unionTest.stringMember = "aString"

unionOp enumTest using aUnionTest = unionTest

unionTest.booleanMember = true

unionOp enumTest using aUnionTest = unionTest

```

CORBA Array Examples

The following examples illustrate different ways to use arrays with CORBA objects.

Unidimensional Arrays

Unidimensional arrays are supported with CORBA in BP-Methods. An example of arrays is shown in the following IDL.

```

interface ArrayTest
{

typedef string  StringArray[];
typedef long    LongArray[];
typedef boolean BooleanArray[];

struct TestStruct
{

long    longMember;
string  stringMember;

};

union TestUnion switch(boolean)
{

case TRUE: long longMember;
case FALSE: string stringMember;

};

typedef TestStruct TestStructArray[];
typedef TestUnion  TestUnionArray[];

};

```

arrays as IN arguments

```

arrayTest = Module1.CorbaTests.ArrayTest("ArrayTest")

stringArray = [ "one", "two", "three",
                "four", "five" ]

stringArrayInOp arrayTest
    using aStringArray = stringArray
    returning stringArray

display stringArray

```

Arrays as OUT Arguments

```
arrayTest = Module1.CorbaTests.ArrayTest("ArrayTest")

stringArray = [ "one", "two", "three",
                "four", "five"]

stringArrayOutOp arrayTest
    returning stringArray = aStringArray

display stringArray
```

Arrays as INOUT Arguments

```
arrayTest = Module1.CorbaTests.ArrayTest("ArrayTest")

stringArray = [ "one", "two", "three",
                "four", "five"]

stringArrayInoutOp arrayTest
    using aStringArray = stringArray
    returning stringArray = aStringArray
```

Cataloging a CORBA Component

Before performing the procedures in this task, you should ensure that you have created a module where you want to catalog the COM component. See [Creating a Custom Catalog Component](#).

1. Right-click on the module where you want to catalog the CORBA component.
2. Select **Catalog Component ► CORBA Service**
3. Select one of the following options:

Option	Description
Use existing configuration	Select this option if you have already configured an External Resource for the COM Component.
Create a new configuration	Select this option if you need to configure a new External Resource for the COM Component. See Creating an External Resource on page 183.

4. Set the Naming Service Values.
5. Set the Interface Repository IOR.
6. Select the source of the IDL Definitions.
7. Select the Components.
8. Click **Next**.

CORBA Service

Provides information on configuring a CORBA Service as an External Resource.

Studio allows you to catalog CORBA objects that reside in an Interface Repository. Once cataloged, you can manipulate the components of the CORBA object in your Method tasks in a process design. To catalog a CORBA object, you need a configuration of CORBA type. Note that creating a configuration allows you to reuse it each time you need to add new components or to reintrospect existing ones

General Properties

This section defines the general SQL Database properties:

Property	Description
Name	Defines the name of the external resource.
Type	Specifies the type of external resource.
Supported Types	

CORBA Service Properties

The following properties must be configured for a CORBA Service under the **Basic Settings** tab:

Property	Description
Use Application Server Orb	Determines whether BEA Systems should lookup the default ORB when running on an application server. If unchecked, the default ORB is used.

The following properties must be configured for a CORBA Service under the **Naming Service** tab:

Property	Description
Read IOR from URL	if you have the IOR exported to some service (for example, a web server) and a URL exists that can be used to fetch it
Use This IOR	Allows you to explicitly define an IOR. (recommended)
Resolve Initial Reference	to request the ORB to get the reference of the service trying to resolve its reference. Note that this option is not recommended since it has interoperability problems when using different ORB
Do Not Use Naming Service	select this option if you do not need this service. However, remember that objects used in methods are found using the Naming Service. Hence, you will need to use the IOR for any object to be used in a method - passing it as a parameter in its constructor

The following properties must be configured for a CORBA Service under the **Interface Repository** tab:

Property	Description
Read IOR from URL	if you have the IOR exported to some service (for example, a web server) and a URL exists that can be used to fetch i
Use This IOR	Allows you to explicitly define an IOR (recommended)
Resolve Initial Reference	to request the ORB to get the reference of the service trying to resolve its reference. Note that this option is not recommended since it has interop problems when using different ORB
Use Interface Repository	this service can be used when an ORB does not have an implementation of the Interface Repository service. Note that this service must be launched separately
Host	
Port	

JNDI

What is JNDI?

Java Naming and Directory Interface (JNDI) components act as an interface to your processes. One interface can communicate with directory services to retrieve, delete, and add objects.

Using JNDI Components

After adding the JNDI component to the catalog, you can use it in the Business Process Methods in Studio.

To call the component in a process design:

- Open Studio.
- Open an existing process or create a new process.
- Add an activity to the process.
- Open the BP-method Editor by double-clicking on the activity.



Note: If the BP-method Editor does not open when you double-click an activity, you can change your preferences. Choose File and then Preferences from the menu options. Click the Activity tab. In the On double click show: field select sources from the drop-down menu. Click the Ok button. Next time you double-click an activity, the Editor appears.

- Click the Component Browser folder at the bottom of the BP-method Editor window. Your new module should appear on the list.
- Drill down from your new module to see your JNDI component. Drill down from your component to see all the methods you chose to include.
- Complete your Method with the JNDI component usage, methods, variables, as needed.
- Save and check your method.

Cataloging a JNDI Component

Before performing the procedures in this task, you should ensure that your directory service is configured and running.

You should also ensure that you have created a module where you want to catalog the JNDI component. See [Creating a Custom Catalog Component](#).

To catalog a JNDI Component:

1. Right-click on the module where you want to catalog the JNDI component.
2. Select **Catalog Component ► JNDI**
3. Select one of the following options:

Option	Description
Use existing configuration	Select this option if you have already configured an External Resource for the COM Component.
Create a new configuration	Select this option if you need to configure a new External Resource for the COM Component. See Creating an External Resource on page 183.

4. Click **Next**.
The list of JNDI methods appears.
5. Select the JNDI methods you want to introspect.
6. Click **Next**.
The libraries are analyzed for dependencies and introspected.
7. Click **Finish**.

The libraries you chose to catalog appear in the Project Navigator.

JNDI Examples

Searching Within the Directory Service

This example uses a common where statement to search for activities whose activity name is 'End'. The second example compares directory service entries to two conditions set in the method. The participant must be named 'Robert' AND must belong to the 'Documentation' Organizational Unit to be considered a match.

```
//common where

for each ad in activitydefinition
  where activityname = "End"
  do
    display "This process has an End activity: " + ad.cn
  end

//Operand equivalence and
for each hp in humanparticipant
  where hp.ou = "Documentation" and hp.cn = "Robert"
  do
    display "The participant: " + hp.cn
  end
```

Search in the Directory Service by dn

A search by dn cannot be done using the for each statement, as it is like the primary key of an entry in the Directory Service. To search by dn you must use the lookup statement, as shown below.

```
data = "c=Argentina"
result = lookup(country, dn : data)

if (result) then
  delete country
end
```

Store a new Item in the Directory Service

This example shows how to store a new item in the directory.

```
o1 = OrganizationalUnit()
o1.ou = "test21"
o1.description = "foo1"
o1.dn = "ou=test21"
o1.postalCode = "1001"
o1.store()
```



Note: When you run JNDI component calls in the Method Debugger, a set of false information is returned. Actual information stored in directory services is not available until runtime, when the process is published and deployed.

SAP

What is SAP BAPI?

Business Application Programming Interfaces (BAPIs) are standard SAP interfaces that enable software vendors to integrate their software into the mySAP Business Suite. BAPIs are technically implemented using RFC (Remote Function Call) enabled function modules inside SAP systems. BAPIs are defined in the Business Object Repository (BOR) as methods of SAP business objects that perform specific business tasks. They allow integration at business level, not technical level. This makes it much easier to find suitable BAPIs compared to non-BAPI based function modules.

Integrating SAP with AquaLogic BPM

AquaLogic BPM uses the SAP Java Connector (JCo) to use BAPIs to access SAP. The SAP Java Connector (JCo) is a toolkit that allows Java applications to communicate with SAP systems. JCo is a high-performance encapsulation of the RFC Library that supports all features of RFC. It combines an easy to use API with unprecedented flexibility and performance. It can be used to implement BAPI based integrations. You also need additional files to be able to use this integration:

If you are using the Studio in a Windows environment:

- librfc32.dll, it has to be located under the system32 directory, and
- sapjcorfc.dll, it has to be located under the ext directory of the Studio installation.

If you are using the Studio in a Unix environment:

- Copy the file librfccm.so to your system file and add the folder that contains it to environment variable LD_LIBRARY_PATH,
- libsapjcorfc.so, it has to be located under the ext directory of the Studio installation.

SAP BAPI Objects, Tables, and Structures

After cataloging SAP BAPI, the following are available:

SAP Object	Description
BAPI Objects	An object is created for each of the introspected BAPIs
SAP Structures	Represent SAP structures which are a set of attributes.
SAP Tables	Represent an SAP table where each row has a set of attributes

BAPI Objects

Each BAPI Object contains the following:

Imports	Input arguments for the BAPI.
Exports	Output arguments for the BAPI
Tables	Input/Output arguments for the BAPI.
Call Method	Used to invoke BAPI.

Tables

Use the `currentRow` attribute to access the fields in the current row and the `rows` attribute to get an iterator to access all the rows in a for each statement.

Cataloging SAP BAPI

Before performing the procedures in this task, ensure you have installed the SAP JCo Library in the SAP Introspector Plugin. See your SAP documentation for more information.

You should also ensure that you have created a module where you want to catalog the COM component. See [Creating a Custom Catalog Component](#).

To catalog an SAP BAPI Component:

1. Right-click on the module where you want to catalog the SAP component.
2. Select **Catalog Component ► SAP**
3. Select one of the following options:

Option	Description
Use existing configuration	Select this option if you have already configured an External Resource for the COM Component.
Create a new configuration	Select this option if you need to configure a new External Resource for the COM Component. See Creating an External Resource on page 183.

4. Click **Next**.
5. Browse to the location of your SAP JCo file (sapjco.jar).

The first time you add a BAPI with Studio, the SAP JCo is required (sapjco.jar file). Browse the file system to the location where the jar file is and click Next. The application needs to be restarted.

6. Click **Next**.

SAP Example

The following is a template example that uses the introspected BAPI
BAPI_ALM_ORDERHEAD_GET_LIST

Test is the module's name given when introspecting the BAPI.

```
imports = B.Test.BapiAlmOrderheadGetList.Imports()
tables = B.Test.BapiAlmOrderheadGetList.Tables()

//Completing a Table
// currentRow is used to refer to the row to complete
tables.itRanges = B.Test.BapiAlmOrderListtheadRanges()
tables.itRanges.currentRow.fieldName =
    "OPTIONS_FOR_DOC_TYPE"
tables.itRanges.currentRow.sign = "I"
tables.itRanges.currentRow.option = "EQ"
tables.itRanges.currentRow.lowValue = "PM01"

//Adds the row and moves to the next row to complete
appendRow tables.itRanges

tables.itRanges.currentRow.fieldName =
    "OPTIONS_FOR_PLANPLANT"
tables.itRanges.currentRow.sign = "I"
tables.itRanges.currentRow.option = "EQ"
tables.itRanges.currentRow.lowValue = "5300"

appendRow tables.itRanges

tables.itRanges.currentRow.fieldName =
    "OPTIONS_FOR_COMP_CODE"
tables.itRanges.currentRow.sign = "I"
tables.itRanges.currentRow.option = "EQ"
```

```

tables.itRanges.currentRow.lowValue = "5560"

appendRow tables.itRanges

tables.itRanges.currentRow.fieldName =
    "SHOW_COMPLETED_DOCUMENTS"
tables.itRanges.currentRow.sign = "I"
tables.itRanges.currentRow.option = "EQ"
tables.itRanges.currentRow.lowValue = "X"

appendRow tables.itRanges

tables.itRanges.currentRow.fieldName =
    "SHOW_DOCUMENTS_IN_PROCESS"
tables.itRanges.currentRow.sign = "I"
tables.itRanges.currentRow.option = "EQ"
tables.itRanges.currentRow.lowValue = ""

appendRow tables.itRanges

tables.itRanges.currentRow.fieldName =
    "SHOW_OPEN_DOCUMENTS"
tables.itRanges.currentRow.sign = "I"
tables.itRanges.currentRow.option = "EQ"
tables.itRanges.currentRow.lowValue = ""

appendRow tables.itRanges

tables.itRanges.currentRow.fieldName =
    "OPTIONS_FOR_CHANGE_DATE"
tables.itRanges.currentRow.sign = "I"
tables.itRanges.currentRow.option = "GE"
tables.itRanges.currentRow.lowValue = "20060710"

// "Exports" is defined to receive
// the returned result from the BAPI
Exports as B.Test.BapiAlmOrderheadGetList.Exports

// Invoking the BAPI using the "call" method
call B.Test.BapiAlmOrderheadGetList
    using imports,
        tables
    returning tables = tables,
        Exports = Exports

// Table iteration to display the rows
for each row in tables.itRanges.rows do
    display row
end

// Working with a table
if size(tables.etResult) > 0 then
    display tables.etResult
end

```

SAP Service

Provides detailed information for configuring an SAP Service.

General Properties

This section defines the general properties for this External Resources:

Property	Description
Name	Defines the name of this external resource.
Type	Specifies the external resource type.
Supported Types	

SAP Properties

The following properties must be for an SAP Service:

Property	Description
Service Name	
Client	
User Id	
Password	
Hostname	
System Number	
Language	
Pool Size	

SQL Database

What is a SQL Component?

The Studio Project Catalog can introspect SQL tables, which can then be called from Method scripts to write business rules in your process design or BPM Object methods. In Studio, when using SQL components, the operation is in auto-commit mode. This means that the SQL operation is committed although the Method is not successful.

The Store Method

The store method determines automatically if an update or an insert has to be performed. It attempts to perform the update and no record is updated then executes the insert. It is depending on the component status that executes the insert or update first.

For example, if the SQL component was loaded from the database executing the load method, then it executes the update first, and the insert later. But, if the component was instantiated from a BP_Method and the fields were manually set, the store executes the insert first and then the update.

If a store is performed on the client side, it will fail, as it cannot be execute on the client side.

During an update, all the component fields are updated. There is not dynamic update for the changed fields to avoid affect the statements cache.

SQL Components as Instance Variables

You can use SQL components cataloged in your Project catalog as instance variables. This is very useful when you want to persist data attached to a process instance. You do not have to store each database value as separate instance variables.

The configuration used to access the database is the one used when the sql component was cataloged.

Automatic Actions

The **load** action is automatically performed, so you do not have to worry about retrieving the information. The **load** is performed in a lazy way.

Manual Actions

You have to manually do the **store** and **remove** of the sql component.

- **store**--you should call the store method if you are adding a new item to the database or if you have modified it during processing. **store** is "intelligent" and will execute an **insert** or **update** depending on the situation.
- **remove** if you need to delete the item from the database. It is not possible to automatically determine this situation.

SQL as instance variable and accessDatabase

When a process instance variable is "persisted" (when the task is completed) all their fields are stored into the database. However, SQL objects are a bit more intelligent and decide which fields have to be stored depending on the value of the "accessDatabase" attribute.

If "accessDatabase" is set to true, it assumes that the object is already stored in a database and therefore it only stores its primary key. And if it is false the whole object is stored.

However, these rules do not apply for Screenflow because their variables are not persisted between tasks. Everything works as you are running only one task.

Benefits

Using a SQL component as an instance variable is a practice that helps you optimize the server database as the only data stored regarding the SQL Component is its primary key. This reduces the storage usage as you do not need to store the data that you are interested in attaching to the process instance in different instance variables. For example, if you want to store the customer's id, its credit limit and discount percentage, you will have to define three instance variables. Instead, if you store the sql component, the only data stored would be the customer id which is the primary key of the identity customer.

Using the SQL component as an instance variable brings the additional benefit of having the data always updated. As you do not keep the information separately from the database, the information may be changed by other users and next time you access the data, it will be updated. If you store information, such as the credit limit of a customer in an instance variable and it is changed, you may be processing with outdated information and you must write extra code to check these changes. Using the SQL component as an instance variable, you always have the latest updated information.

Another important benefit is that the load is automatically performed. More than that, it is done in a **lazy** way. This means that the data is not retrieved until it is required. The **load** is performed the first time an attribute of the component is referenced.

Configuring SQL Components with no Database Access

SQL introspected components have an extra predefined bool field named accessDatabase. You can visualize it by expanding the Fuego.Sql.SqlObject entry in the structure panel of the SQL component.

The accessDatabase default value is set to true. If the value is set to false, the object is considered as a simple object structure. In this case, any setting value or getting value actions performed on the object do not have effect on the database, but on the object structure.

The code shown below does not interact with the database:

```
c = CUSTOMER()

c.accessDatabase = false
// or c = CUSTOMER(accessDatabase: false)

c.custtype = "GLOBAL"
```


.....

load

When the primary key of an SQL component is filled and another attribute that is not primary key is accessed, an automatic load is performed internally. If you want to disable it from working like that, set the `accessDatabase` attribute to `false`.

store

If the `accessDatabase` field is set to `true`, when the sql is serialized as a process instance, only the primary key is stored, but, all the sql component fields are store if the `accessDatabase` is set to `false`".



Note: When the object is serialized as a `ProcessInstance` variable having the `accessDatabase` field set to `false` not only the primary is store but also all the object fields are. See more information in [SQL Components as instance variables](#).

accessDatabase Attribute and Methods Behavior

The SQL methods `load`, `store` and `remove`, force access to the database, by setting the default value to the `accessDatabase` field, that is `true`.

Cataloging a SQL Component

Before performing the procedures in this task, you should ensure that your database is installed and running on your system.

You should also ensure that you have created a module where you want to catalog the SQL component. See [Creating a Custom Catalog Component](#).

To catalog a SQL Component:

1. Right-click on the module where you want to catalog the SQL component.
2. Select **Catalog Component ► SQL**
3. Select one of the following options:

Option	Description
Use existing configuration	Select this option if you have already configured an External Resource for the SQL Component.
Create a new configuration	Select this option if you need to configure a new External Resource for the SQL Component. See Creating an External Resource on page 183.

4. Click **Next**.
AquaLogic BPM Studio creates a list of SQL schema that can be cataloged. This may take several minutes.
5. Select the SQL components you want to introspect.
6. Click **Next**.
The components are analyzed for dependencies and introspected.
7. Click **Finish**.

The components you chose to catalog appear in the Project Navigator.

SQL Method Examples

The following are some basic database Method examples:

Query

```

for each { variable } in { table }

where { condition }
do
{ body }
end

```

Insert

```

INSERT INTO { table }({ column1, column2, ... })
VALUES({ val1, val2, ... });

```

Delete

```

DELETE FROM { table }
WHERE { condition };

```

Update

```

UPDATE { table }
SET { column1 = value1, column2 = value2, ... }
WHERE { condition };

```

Exists

```

select Name, Skill from EMP where EXISTS
    select * from EMP where EMP.Name = empName
group by Name
having COUNT(Skill) > 1);

```

Primary Key

If the Table has a primary key, the following methods are available:

- `load()` : Loads the record if the primary key is set. Returns a boolean indicating if the record was found.
- `store()` : Inserts or updates the record in the database. The primary key must be set.
- `delete()` : Deletes the record in the database. The primary key must be set.

Table (SQL Query)**What is a Table Component?**

The BPM system provides out-of-the-box functionality for introspecting tables from relational database resources (tables, views and store procedures). At times, though, it may be necessary to operate with complex views of those tables or generic SQL queries. As, in some cases, join among different tables are performed to retrieved information localized in

different RDBMS tables. Studio now allows defining these complex queries as a single resource to prevent replicating these complex queries in different areas of the product. These SQL queries can also be customized to receive arguments and make them reusable within a Project.



Note: Only SELECT-type queries are allowed and that the generated components will provide read-only access to the results.

Query Categories

We further classify queries into two categories, named selfcontained and parametric queries. They differ only in that the latter allows the caller to pass parameters to the query at run-time.

Parametric queries, as the name suggests, allow the caller to specialize the query according to values that are determined at run-time, instead of when the query is designed and introspected. As an example, consider the a query, where we want to be able to specify the actual range for the amount column of the orders when the query is executed.

Cataloging a SQL Query

The following procedures show you how to catalog a table using a SQL Query.

Before performing these procedures, ensure that you have create a Module within the Catalog.

1. In the Project Navigator, right-click on the Module where you want to Catalog a SQL query.
2. Select **Catalog Component ► Table**
3. Select an existing External Resource configuration.

See [Creating an External Resource](#) on page 183 for information on creating a new External Resource.

4. Click **Next**.
5. Enter the SQL command to return data from your table.
6. Click **Next**.

The SQL query is executed. This may take several minutes depending on the size of your query. The results of your query are displayed in a table.

7. Click **Next**.

The component is cataloged.

8. Click **Finish**.

The SQL table appears as a resource with the Catalog.

Web Service

Web Service Catalog Component

XML Schema

XML Schema Catalog Component

XML schemas can be introspected into the component catalog. The XML schema component can be called from your process design to incorporate functionality for typed parsing and typed XML document generation.



Note: Schema redefinition is not supported. Only data types from an XML schema are supported. If you only have the XML document, but do not have a schema or you have a DTD file, you can create a schema by using one of many tools available on the Internet in order to create a schema.

Handling Large XML Schema

How should you read in a large and complex XML file?

Reading a very large and complex XML file into memory all at once is not efficient. It causes large amounts of memory to be consumed and will degrade the Engine's performance.



Note: Remember that if you don't have an XSD file, it can be easily generated using an XML editor (like XML Spy).

By having an XSD file that you can introspect, there is a much more efficient technique that does not require that the whole file be read into memory. The code shown below does not read the XML whole file into memory and only accesses the file when it needs to read in more object information from it.

After introspecting the XML , create a new BPM Object method:

```
// In production, do not hard code a local directory
// like this.
// It is only here in this example for clarity.

customer as XMLComp.Customer.Customer
customer =
  XMLComp.Customer.Customer( "file://c:/tmp/customer.xml" )

display customer.billAddress
display customer.custDisc
display customer.customerName
```

The above code expects a "customer.xml" XML file in the "c:\tmp" directory. Use the following XML text as this customer.xml file for this example: :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<Customer>
  <CustomerCode>NEW</CustomerCode>
  <CustomerName>John Smith</CustomerName>
  <CustDisc></CustDisc>
  <BillAddress>Madison 2939 NY</BillAddress>
  <ShipAddress>Madison 2939 NY</ShipAddress>
  <CodPay></CodPay>
  <CustType></CustType>
  <Mail>pp@com.com</Mail>
</Customer>
```

Cataloging XML Schema

Before performing the procedures in this task, ensure that you have created a module where you want to catalog the XML Schema component. See [Creating a Custom Catalog Component](#).

To catalog an XML Schema Component:

1. Right-click on the module where you want to catalog the COM component.
2. Select **Catalog Component ► XML Schema**
3. Enter the location of the you want to catalog.



Note: Only XML Schema Definitions (.xsd) can be cataloged.

4. Enter a Module Name
5. Click **Next**.
The XML schema is analyzed for dependencies. Any errors or warnings are listed.
6. Click **Finish**.

The XML schema appears in the Project Navigator.

XML Schema Examples

XML Schema Introspection Example

The following is an XML schema example for an XML document that may be used in a business process.

```
<?xml version="1.0" encoding="UTF-8" ?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:simpleType name="Address" type="xsd:string"/>
  <xsd:simpleType name="CustomerCode" type="xsd:string"/>
  <xsd:simpleType name="CustomerName" type="xsd:string"/>
  <xsd:simpleType name="CodPay" type="xsd:string"/>
  <xsd:simpleType name="CustDisc" type="xsd:string"/>
  <xsd:simpleType name="CustType" type="xsd:string"/>
  <xsd:simpleType name="Mail" type="xsd:string"/>

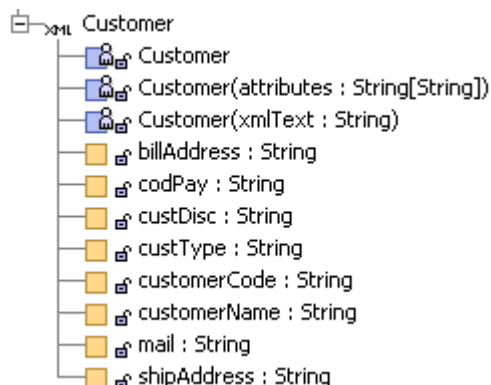
  <xsd:group name="shipAndBill">
    <xsd:sequence>
      <xsd:element name="ShipAddress" type="Address"/>
      <xsd:element name="BillAddress" type="Address"/>
    </xsd:sequence>
  </xsd:group>

  <xsd:complexType name="Customer">
    <xsd:sequence>
      <xsd:choice>
        <xsd:element name="CustomerCode"
          type="CustomerCode"/>
        <xsd:element name="CustomerName"
          type="CustomerName"/>
        <xsd:element name="CodPay" type="CodPay"/>
        <xsd:element name="CustDisc" type="CustDisc"/>
        <xsd:element name="CustType" type="CustType"/>
        <xsd:group ref="shipAndBill"/>
        <xsd:element name="Mail" type="Mail"/>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:element name="customer" type="Customer"/>

</xsd:schema>
```

After introspecting the XML schema detailed above, the Project catalog structure appears as shown in the following image:





Note: You can drag and drop the XML attributes and methods to the Method Editor in order to visualize the usage template.

Create an XML File Based on an Introspected Definition

```
customer as XMLComp.Customer.Customer
customer = XMLComp.Customer.Customer()

customerDOS as XMLComp.Customer.Customer
customerDOS = XMLComp.Customer.Customer()

customer.customerCode = "NEW"
customer.customerName = "John Smith"
customer.custDisc = null
customer.billAddress = "Madison 2939 NY"
customer.shipAddress = "Madison 2939 NY"
customer.codPay = null
customer.custType = null
customer.mail = "pp@com.com"

store customer
    using targetFile = "C:/tmp/customer.xml"
```

Loading an XML File Using the loadFromUrl

The loadFromUrl method can receive as parameter an URL, pointing the xml file, like file://home/sharedDocuments/test.xml or http://....

```
customer as XMLComp.Customer.Customer
customer = XMLComp.Customer.Customer()

customer.loadFromUrl("file://c:/tmp/customer.xml")
```

While **loadfromUrl()** method to parse XML documents is interesting from a coding perspective, it is recommended to avoid its use on a large XML file as it uses DOM and reads the whole file into memory.

Load an XML data file containing a customer using the load method

The **load** method can receive as parameter:

- an URL,
- a path location to the xml file OS dependant. This is, for example : **C:\\documents\\test.xml** or **//home//sharedDocuments//test.xml**, Windows and Unix.
- the xml file content.

Business Activity Monitoring (BAM)


The following topics describe the Business Activity Monitor and provide information on the BAM database, creating BAM Dashboards, and configuring BAM in ALBPM Studio.

What is BAM?

Business Activity Monitoring (BAM) allows you to store, analyze, and display statistics about your business process execution.

BAM provides information about process instance performance and process workload. This information can be used to present almost real-time business processes metrics. You can then use these to analyze and then improve or adapt business processes based on real-world conditions.

To store and present this information, BAM contains the following:

Database	BAM data is stored within a database. In ALBPM Studio, this information is stored internally as part of the embedded process execution engine database. In ALBPM Enterprise, you must configure an external database to function as the BAM database.
SQL Queries	You can write queries that access the information stored in the BAM database. These queries are contained within a BPM Object Method. When you create a BAM Dashboard using the wizard, the wizard automatically creates queries based on the type of Dashboard template you choose. You can customize these queries or create your own queries and dashboards to customize the way you present BAM.
BAM Dashboards	<p>BAM Dashboards allow you to display BAM information in a meaningful and useful way. BAM Dashboards also allow you to drill down from a general view of a process to more specific information such as an order or claim.</p> <p> Note: BAM Dashboards require the Flash Plugin.</p>

Enabling and Configuring BAM in Studio

BAM is included within ALBPM Studio. This allows you to test real-time dashboards using BAM data generated by the embedded Process Execution Engine.

By default, ALBPM Studio does not generate BAM data. The following procedures show you how to enable BAM and set other BAM properties.

1. Right-click on any project within Studio.
2. Select **Engine Preferences**.
3. Click BAM in the left-hand tree.
4. Click the checkbox next to **Enable BAM**.

You can also set other BAM properties on this page.

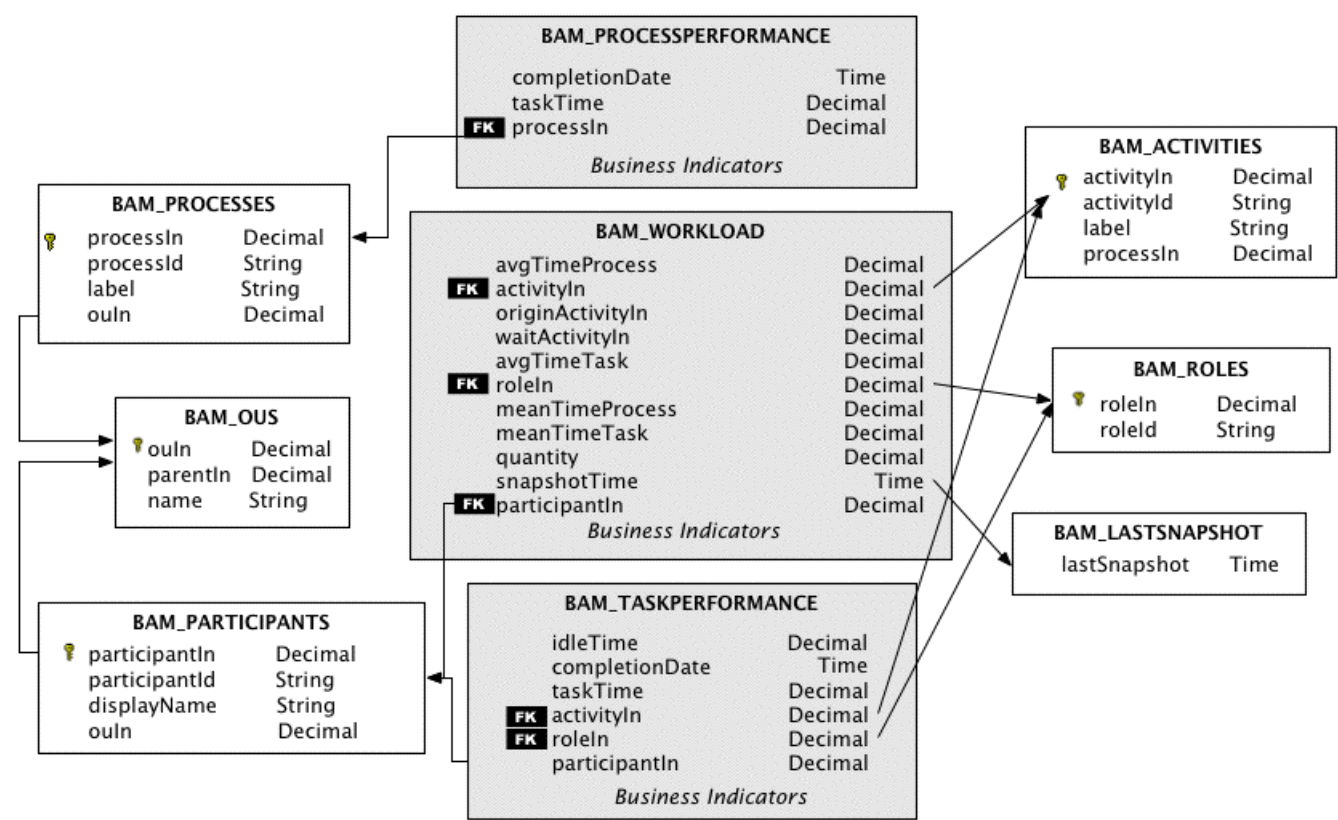
BAM Database

The BAM database is used to store information about your business processes.

The BAM database stores the following types of information about a process:

- 1. Workload
- 2. Task Performance
- 3. Process Performance

The following diagram shows the relationship between each of the BAM tables:



How BAM Database is Populated

BAM database is populated with the information generated by auditing events. Auditing events generation can be enabled for the whole process, for a subset of activities or for a particular activity. For more information on how to configure auditing events generation please see [Audit Events](#) on page 99.

Using Variables in BAM

When creating a Project variable, you can define it as a Business Indicator variable. This allows the variable to be stored in BAM the database.

When you add Business Indicator variable to your process, a column is added to the following BAM database tables: Workload, Task Performance and Process Performance. The name of this column is the Business Indicator name preceded by the prefix "V_".

If you define a business dimension, the workload table contains one row for each possible value of this business dimension present in the process. Each of this rows will show the quantity of instances that match that business dimension.

When you define a measurement business variable the sum of this variable's value for all in flight is stored into workload table. If business dimensions were defined as well, then this sum will be divided into as many rows as business dimension values present in flight instances.

Task performance table stores one row for each instance that completes an activity. Each of these rows contains the value of dimensions and measurements at the time the instance completed the activity.

In a similar way, process performance table stores one row for each instance that gets to the end activity. Each of these rows contains the value of dimensions and measurements at the time the instance completed the whole process.

Creating a Predefined BAM Dashboard

ALBPM Studio provides a quick method of creating BAM Dashboards using predefined templates.

To create a Predefined BAM Dashboard:

1. Ensure that you have enable BAM in Studio.
See [Enabling and Configuring BAM in Studio](#) on page 175 for more information.
2. Right-click on the Project where you want to create a BAM dashboard.
3. Select **Add BAM Predefined Dashboard**.
4. Select the type of template you want to use. You can select more than one template if necessary.
5. Click **OK**.
6. Select the Process whose BAM information you want to report.
7. Select the Role you want to allow to view the BAM information.

Anyone assigned to this role is able to view the BAM dashboard.

Studio creates BAM Dashboards for each of the template you selected. Dashboard Global activities for each dashboard are added to the process.

See [Viewing BAM Dashboards in Studio](#) on page 177 for information on viewing BAM Dashboards in Studio.

Viewing BAM Dashboards in Studio

After creating a BAM Dashboard, you can view it using ALBPM Studio's Process Execution Engine.

Ensure that you have a participant in the process who has been assigned the role where the BAM Dashboard reports are located.

1. Start the ALBPM Studio Process Execution Engine
2. Launch WorkSpace.
3. Login with the username of a participant who has been assigned the role used to view BAM information.
4. Click Applications to view the BAM Dashboard reports.

BAM Database Reference

The BAM database contains the following tables.

BAM_OU

Row Name	Value	Null Value
ouIn	DECIMAL(10)	NOT NULL
parentIn	DECIMAL(10)	NOT NULL
name	STRING(255)	NOT NULL

Primary Key: ouIn**BAM_ROLES**

Row Name	Value	NULL Value
roleIn	DECIMAL(10)	NOT NULL
roleID	DECIMAL(10)	NOT NULL

Primary Key: roleIn**BAM_Participants**

Row Name	Value	NULL Value
participantIn	DECIMAL(10)	NOT NULL
participantID	STRING(255)	NOT NULL
ouIn	DECIMAL(10)	NOT NULL
displayName	STRING(255)	

Primary Key: participantIn**Foreign Key:** ouin, referenceTable="OUs"**BAM_Processes**

Row Name	Value	NULL Value
ouIn	DECIMAL(10)	NOT NULL
processIn	DECIMAL(10)	NOT NULL
processId	STRING(255)	NOT NULL
label	STRING(255)	NOT NULL

Primary Key: processIn .**Foreign Key:** ouIn, referencedTable="OUs" .**BAM_Activities**

Row Name	Value	NULL Value
activityIn	DECIMAL(10)	NOT NULL
activityId	STRING(255)	NOT NULL
processIN	DECIMAL(10)	NOT NULL

Row Name	Value	NULL Value	Unit
			sup
			ata
			t M
			dra
			sup
			sub
			via
			gim
			MAB
			sit
			via
			s i
			elt
			vol
			r o
			sup
			nuc
			via
			tat
			cac
			elt
			si
			rat
			oni
			ura
			n i
			elt
			tor
			dr
quantity	DECIMAL(10)	NOT NULL	sup
			elt
			rom
			f o
			si
			n i
			elt
			via
			g w
			o t
			e b
			dep
avgTimeTask	DECIMAL(10)	NOT NULL	sup
			elt
			con
			nt
			n i
			, as
			tat
			n a
			si
			st w
			o t
			e b
			dep

Row Name	Value	NULL Value	Unit
avgTimeProcess	DECIMAL(10)	NOT NULL	seconds

Foreign Keys:

- activityIn, referencedTable="Activities"
- waitActivityIn, referencedTable="Activities"
- origActivityIn, referencedTable="Activities"
- roleIn, referencedTable="Roles"
- participantIn, referencedTable="Participants"

BAM_TASKPERFORMANCE

This table contains a record for each instance that is processed in the activityIn, roleIn, and participantIn.

Row Name	Value	NULL Value	Unit
activityIn	DECIMAL(10)	NOT NULL	
roleIn	DECIMAL(10)	NOT NULL	
participantIn	DECIMAL(10)	NOT NULL	
completionDate	DECIMAL(10)	NOT NULL	Specifies the date when the instance was processed in the activity and flew to the next activity. To maintain the coherence between the data, the completionDate is stored in GMT-0, as there might be different servers running with different hours.
taskTime	DECIMAL(10)	NOT NULL	Specifies the total processing time, in seconds, for the instance in the activity.

Foreign Keys:

- activityIn, referencedTable="Activities"
- roleIn, referencedTable="Roles"
- participantIn, referencedTable="Participants"

BAM_ProcessPerformance

Row Name	Value	NULL Value	Description
processIn	DECIMAL(10)	NOT NULL	Date when the instance reached the End activity of the process. To maintain the coherence between the data, the completionDate is stored as GMT-0 since there might be different servers running with different hours.
completionDate	TIMESTAMP	NOT NULL	
taskTime	DECIMAL(10)	NOT NULL	Stores the time, in seconds, required to process the instance.

Foreign Key: processIn, referencedTable="Processes".

BAM_LASTSNAPSHOT

This table stores a view of the BAM_WorkLoad table, including the time the BAM updater was last executed.

Row Name	Value	NULL Value
lastshapshot	TIMESTAMP	NOT NULL

External Resources

The following topics discuss how to use External Resources.

What is an External Resource?

Provides a general description of External Resources.

External Resources provide a common method for connecting to other resources in an enterprise including databases, Web Services, etc. External Resources are used to define connectivity and configuration information. It is useful to define these separately since connectivity and configuration information is different between systems. This allows you to easily deploy a project into a new environment because you only need to redefine the External Resource without having to edit application code.

See [External Resource Reference](#) on page 183 for a complete list of External Resources available in AquaLogic BPM.

Each project in AquaLogic BPM Studio contains an External Resources directory.

External Resources and the Catalog

Outlines the relationship between External Resources and the Catalog.

Creating an External Resource

Provides the basic procedures for defining an External Resource.

To define a new External Resource:

1. In the Project Navigator View, expand the project where you want to define a new External Resource.
2. Right-click **External Resources** and select **New External Resource**.
The **Edit External Resource** window appears.
3. Enter a name for the External Resource.
4. Select the type of External Resource you want to define.
5. Provide values for each of the External Resource properties.

For complete information on the properties for each type of External Resource see [External Resource Reference](#) on page 183.

6. Click **Ok**.

The new External Resource appears in the Project Navigator.

External Resource Reference

Provides comprehensive information about each External Resource type.

SQL Database

Provides detailed information for the SQL Database External Resource.

General Properties

This section defines the general properties for this External Resource:

Property	Description
Name	Defines the name of the external resource.
Type	Specifies the type of external resource.
Supported Types	Specifies the type of JDBC connection.

BEA DB2 Driver Properties

The BEA DB2 Driver supports the IBM DB2 versions 8.x and 9.x.

The following properties must be configured for a BEA DB2 driver under the **Basic** tab:

Property	Description
Host	Specifies the database server host.
Port	Specifies the port of the database host.
User	Defines user ID you want to use to connect to the database. This user must already exist in DB2 and have permissions to create the schema and tables used to store information.
Password	Specifies password for the user.
Database	Specifies the database you wish to connect to.
Schema	Specifies the database schema to use. (optional)
URL	Defines the URL for the database entry.

The following properties must be configured for a BEA DB2 driver under the **Properties** tab:

Property	Description

The following properties must be configured for a BEA DB2 driver under the **Runtime** tab:

Property	Description
Maximum Pool Size	
Maximum Connections Per User	
Connection Idle Time (minutes)	
Minimum Pool Size	
Maximum Opened Cursors	

BEA Informix Driver Properties

The BEA Informix database driver supports versions 9.2, 9.3, 9.4, and 10.

The following properties must be configured for a BEA Informix driver under the **Basic** tab:

Property	Description
Host	

Property	Description
Port	
User	
Password	
Database	
Server	
URL	

The following properties must be configured for a BEA Informix driver under the **Advanced** tab:

Property	Description
Root dbspace name	

The following properties must be configured for a BEA Informix driver under the **Properties** tab:

Property	Description

The following properties must be configured for a BEA DB2 Driver under the **Runtime** tab:

Property	Description
Maximum Pool Size	
Maximum Connections Per User	
Connection Idle Time (minutes)	
Minimum Pool Size	
Maximum Opened Cursors	

BEA SQL Server Driver Properties

The BEA SQL Server driver supports SQL Server version 2005.

The following properties must be configured for a BEA SQL Server driver under the **Basic** tab:

Property	Description
Host	
Port	
User	
Password	
Database	
URL	

The following properties must be configured for a BEA SQL Server driver under the **Properties** tab:

Property	Description

The following properties must be configured for a BEA SQL Server driver under the **Runtime** tab:

Property	Description
Maximum Pool Size	
Maximum Connections Per User	
Connection Idle Time (minutes)	
Minimum Pool Size	
Maximum Opened Cursors	

BEA Oracle Driver Properties

The BEA Oracle database driver supports versions 9.2 and 10.

The following properties must be configured for a BEA Oracle driver under the **Basic** tab:

Property	Description
Host	
Port	
User	
Password	
SID	
Schema (optional)	
URL	

The following properties must be configured for a BEA Oracle driver under the **Advanced** tab:

Property	Description
Tablespace	
Temporary Tablespace	
Profile	

The following properties must be configured for a BEA Informix driver under the **Properties** tab:

Property	Description

The following properties must be configured for a BEA DB2 Driver under the **Runtime** tab:

Property	Description
Maximum Pool Size	
Maximum Connections Per User	
Connection Idle Time (minutes)	
Minimum Pool Size	
Maximum Opened Cursors	

BEA Sybase Driver Properties

The BEA Sybase Driver supports Sybase versions 11.x, 12.x and 15.

The following properties must be configured for a BEA DB2 driver under the **Basic** tab:

Property	Description
Host	
Port	
User	
Password	
Database	
Device	
Allocation Size	
URL	

The following properties must be configured for a BEA DB2 driver under the **Properties** tab:

Property	Description

The following properties must be configured for a BEA DB2 driver under the **Runtime** tab:

Property	Description
Maximum Pool Size	
Maximum Connections Per User	
Connection Idle Time (minutes)	
Minimum Pool Size	
Maximum Opened Cursors	

BEA DB2 AS/400 JDBC Properties

The following properties must be configured for a BEA DB2 AS/400 JDBC driver under the **Basic** tab:

Property	Description
Host	
Port	
User	
Password	
Database	
Schema	
URL	

The following properties must be configured for a BEA DB2 driver under the **Properties** tab:

Property	Description

The following properties must be configured for a BEA DB2 driver under the **Runtime** tab:

Property	Description
Maximum Pool Size	
Maximum Connections Per User	
Connection Idle Time (minutes)	
Minimum Pool Size	
Maximum Opened Cursors	

BEA DB2 OS390 Properties

The BEA DB2 Driver supports the IBM DB2 versions 8.x and 9.x.

The following properties must be configured for a BEA DB2 driver under the **Basic** tab:

Property	Description
User	
Password	
Database	
Schema	
URL	

The following properties must be configured for a BEA DB2 OS390 driver under the **Properties** tab:

Property	Description

The following properties must be configured for a BEA DB2 OS390 driver under the **Runtime** tab:

Property	Description
Maximum Pool Size	
Maximum Connections Per User	
Connection Idle Time (minutes)	
Minimum Pool Size	
Maximum Opened Cursors	

Generic JDBC Version 1 Properties

The following properties must be configured for a Remote JDBC Connection under the **Basic** tab:

Property	Description
JDBC Driver	
URL	
User	
Password	

The following properties must be configured for a Remote JDBC Connection under the **Runtime** tab:

Property	Description
Maximum Pool Size	
Maximum Connections Per User	
Connection Idle Time (minutes)	
Minimum Pool Size	

Remote JDBC Properties

The following properties must be configured for a Remote JDBC Connection:

Property	Description
Database Type	
J2EE	
Lookup Name	

SAP Service

Provides detailed information for configuring an SAP Service.

General Properties

This section defines the general properties for this External Resources:

Property	Description
Name	Defines the name of this external resource.
Type	Specifies the external resource type.
Supported Types	

SAP Properties

The following properties must be for an SAP Service:

Property	Description
Service Name	
Client	
User Id	
Password	
Hostname	

Property	Description
System Number	
Language	
Pool Size	

Web Service

Provides detailed information for configuring a Web Service.

General Properties

This section defines the general Web Service properties:

Property	Description
Name	Defines the name of the external resource.
Type	Specifies the type of external resource.
Supported Types	Specifies the type of Web Service.

Producer Web Service Properties

The following properties must be configured for a Producer Web Service External Resource:

Property	Description
Authentication Type	Defines the authentication type of this web service. <ul style="list-style-type: none"> None - uses no authentication for the web service. Username Token Profile -

Consumer Web Service Properties

The following properties must be configured for a Consumer Web Service under the **Endpoint** tab:

Property	Description
Static Endpoint Binding	
UDDI Dynamic Endpoint Binding	
Transport Type	
Transport Configuration	
Use System Exceptions	

The following properties must be configured for a Consumer Web Service under the **Security** tab:

Property	Description
Send Username Token	
Send Nonce and Timestamp	
Username	
Password	
Confirm Password	

UDDI Registry

Provides detailed information for configuring a UDDI Registry.

General Properties

This section defines the general properties for this External Resources:

Property	Description
Name	Defines the name of this external resource.
Type	Specifies the external resource type.
Supported Types	

UDDI Properties

The following properties must be for a UDDI Registry:

Property	Description
HTTP Server Configuration	
Inquiry Server Path	
Publication Service Path	
Security Server Path	
User	
Password	

HTTP Server Configuration

Provides detailed information for configuring an HTTP Server as an External Resource.

General Properties

This section defines the general properties for this External Resources:

Property	Description
Name	Defines the name of this external resource.
Type	Specifies the external resource type.
Supported Types	

HTTP Server Properties

The following properties must be defined for an HTTP Server configuration:

Property	Description
Protocol	
Host	
Port	
Requires HTTP Basic Authentication	
User	

Property	Description
Password	

Microsoft .NET Service

Provides detailed information for configuring a Microsoft .NET Service.

General Properties

This section defines the general properties for this External Resources:

Property	Description
Name	Defines the name of this external resource.
Type	Specifies the external resource type.

Microsoft .NET Service Properties

The following properties must be defined for a Microsoft .NET Service:

Property	Description
Host	Defines the location of the .NET Bridge host
Port	Defines the port used by the .NET Bridge host

Mail Outgoing Service

Provides detailed information for configuring a Mail Outgoing Service.

General Properties

This section defines the general properties for this External Resource:

Property	Description
Name	Defines the name of this external resource.
Type	Specifies the external resource type.
Supported Types	

SMTP Properties

The following properties must be for an SMTP Mail Outgoing Service:

Property	Description
Server Host	
Server Port (optional)	
User	
Password	
Secure Connection	Defines the security protocol used. Valid values are: <ul style="list-style-type: none">No - No security protocol is used.TLS, if available

Property	Description
	<ul style="list-style-type: none"> • TLS • SSL - Uses the Secure Sockets Layer (default).

J2EE Application Server

Provides detailed information on configuring a J2EE Application Server as an External Resource.

To create Enterprise JavaBeans components, you must define a J2EE Application Server as an External Resource.

General Properties

The following table defines the general properties for this External Resource:

Property	Description
Name	Defines the name of the external resource.
Type	Specifies the type of external resource.
Supported Types	Specifies the type of J2EE server

Local J2EE Application Server Properties

This type of J2EE application server is used when the process is deployed in a J2EE environment, and the resources are located in the same Application Server

The following properties must be configured for a Remote JDBC Connection:

Property	Description
User Transaction Lookup Name	

Generic J2EE Application Server Properties

The following properties must be configured for a J2EE Application Server using the **Basic** tab:

Property	Description
Initial Context Factory	
URL	
Principal	
Credentials	

The following properties must be configured for a J2EE Application Server under the **Advance** tab:

Property	Description
User Transaction Lookup Name	

The following properties must be configured for a J2EE Application Server using the **Properties** tab:

Property	Description
Initial Context Factory	

The following properties must be configured for a J2EE Application Server using the **Runtime** tab:

Property	Description
Maximum Pool Size	
Maximum Connections Per User	
Connection Idle Time (minutes)	
Minimum Pool Size	

Enterprise JavaBean (EJB)

Provides detailed information for configuring an Enterprise JavaBean as an External Resource.

General Properties

This section defines the general properties for this External Resource:

Property	Description
Name	Defines the name of this external resource.
Type	Specifies the external resource type.
Supported Types	

EJB Properties

The following properties must be defined for an EJB:

Property	Description
J2EE	
Lookup Name	

Java Class Library

Provides detailed information for configuring a Java Class Library as an External Resource.

General Properties

This section defines the general properties for this External Resource:

Property	Description
Name	Defines the name of this external resource.
Type	Specifies the external resource type.
Supported Types	

Java Class Library Properties

The following properties must be defined for a Java Class Library:

Property	Description
property	
property	

AquaLogic Service Bus

Provides detailed information for configuring an AquaLogic Service Bus as an External Resource.

General Properties

This section defines the general External Resource properties:

Property	Description
Name	Defines the name of the external resource.
Type	Specifies the type of external resource.
Supported Types	Specifies the type of AquaLogic Service Bus connection.

Management Host Properties

The following properties must be configured for a Management Host:

Property	Description
Host	
Port	
User	
Password	

Proxy Service Properties

The following properties must be configured for a Proxy Service:

Property	Description
Host	
Port	
User	
Password	

Process Deployment Properties

The following properties must be configured for a Process Deployment:

Property	Description
Management Configuration	
Project Name	
WSDL Folder	
Business Services Folder	
WS-Security Account	
Transport	
Host	
Port	

Mail Incoming Service

Provides detailed information for configuring a Mail Incoming Service as an External Resource.

General Properties

This section defines the general properties for this External Resource:

Property	Description
Name	Defines the name of this external resource.
Type	Specifies the external resource type.
Supported Types	

IMAP and POP3 Properties

The following properties must be when using IMAP or POP3:

Property	Description
Service Host	
Server Port (optional)	
User	
Password	
Secure Connection	
Secure Authentication	

Microsoft COM Service

Provides detailed information for configuring a Microsoft COM Service as an External Resource.

COM components are software programs that use Microsoft Component Object Model (COM) technology. COMBridge is a Windows application that acts as a "bridge" between BPM applications and COM. It provides all the necessary services to introspect and use COM components.

General Properties

This section defines the general properties for this External Resources:

Property	Description
Name	Defines the name of this external resource.
Type	Specifies the external resource type.

Microsoft COM Properties

The following properties must be defined for a Microsoft COM Service:

Property	Description
Host	Indicates the host where the COM Bridge is installed
Port	Indicates the port where the COM Bridge is installed. The default is 4060.

JMX Service

This service is not working in the product.

CORBA Service

Provides information on configuring a CORBA Service as an External Resource.

Studio allows you to catalog CORBA objects that reside in an Interface Repository. Once cataloged, you can manipulate the components of the CORBA object in your Method tasks in a process design. To catalog a CORBA object, you need a configuration of CORBA type. Note that creating a configuration allows you to reuse it each time you need to add new components or to reintrospect existing ones

General Properties

This section defines the general SQL Database properties:

Property	Description
Name	Defines the name of the external resource.
Type	Specifies the type of external resource.
Supported Types	

CORBA Service Properties

The following properties must be configured for a CORBA Service under the **Basic Settings** tab:

Property	Description
Use Application Server Orb	Determines whether BEA Systems should lookup the default ORB when running on an application server. If unchecked, the default ORB is used.

The following properties must be configured for a CORBA Service under the **Naming Service** tab:

Property	Description
Read IOR from URL	if you have the IOR exported to some service (for example, a web server) and a URL exists that can be used to fetch it
Use This IOR	Allows you to explicitly define an IOR. (recommended)
Resolve Initial Reference	to request the ORB to get the reference of the service trying to resolve its reference. Note that this option is not recommended since it has interoperability problems when using different ORB
Do Not Use Naming Service	select this option if you do not need this service. However, remember that objects used in methods are found using the Naming Service. Hence, you will need to use the IOR for any object to be used in a method - passing it as a parameter in its constructor

The following properties must be configured for a CORBA Service under the **Interface Repository** tab:

Property	Description
Read IOR from URL	if you have the IOR exported to some service (for example, a web server) and a URL exists that can be used to fetch i
Use This IOR	Allows you to explicitly define an IOR (recommended)

Property	Description
Resolve Initial Reference	to request the ORB to get the reference of the service trying to resolve its reference. Note that this option is not recommended since it has interop problems when using different ORB
Use Interface Repository	this service can be used when an ORB does not have an implementation of the Interface Repository service. Note that this service must be launched separately
Host	
Port	

JMS Messaging Service

Provides detailed information for configuring a JMS Messaging Service as an External Resource.

General Properties

This section defines the general properties for this External Resources:

Property	Description
Name	Defines the name of this external resource.
Type	Specifies the external resource type.
Supported Types	

JMS Messaging Service Properties

The following properties must be configured for a JMS Messaging Service:

Property	Description
J2EE	
Destination Type	<ul style="list-style-type: none"> Queue Topic
Connection Factory Lookup	
JMS Listener Port (WebSphere only)s	

JNDI Directory Server

Provides detailed information on configuring a JNDI Directory Server as an External Resource.

General Properties

This section defines the general properties for this External Resource:

Property	Description
Name	Defines the name of the external resource.
Type	Specifies the type of external resource.
Supported Types	Specifies the type of JNDI Directory Server.

Active Directory Properties

The following properties must be configured for an Active Directory JNDI Directory Server using the **Basic** tab:

Property	Description
Initial Context Factory	Defines the name of the initial context factory you want to use.
URL	Defines the URL you want to use to connect to the directory service.
Principle	Defines the root distinguished name for the directory service.
Credentials	Specifies the password for the directory service.
Referrals	<ul style="list-style-type: none"> • follow: the entry will be looked for directly. • ignore: the entry is not looked for. • throw: you must catch and manage any exceptions.

The following properties must be configured for an Active Directory JNDI Directory Server using the **Properties** tab:

Property	Description
Properties	Define any name/value pair properties that need to be passed to the directory service.

The following properties must be configured for an Active Directory JNDI Directory Server using the **Runtime** tab:

Property	Description
Maximum Pool Size	
Maximum Connections Per User	
Connection Idle Time (minutes)	
Minimum Pool Size	

Sun One LDAP Properties

The following properties must be configured for a Sun One LDAP JNDI Directory Server using the **Basic** tab:

Property	Description
Initial Context Factory	Defines the name of the initial context factory you want to use, for example: com.sun.jndi.ldap.LdapCtxFactory.
URL	Defines the URL you want to use to connect to the directory service.
Principle	Defines the root distinguished name for the directory service.
Credentials	Specifies the password for the directory service.
Referrals	<ul style="list-style-type: none"> • follow: the entry will be looked for directly. • ignore: the entry is not looked for. • throw: you must catch and manage any exceptions.

The following properties must be configured for a Sun One LDAP JNDI Directory Server using the **Properties** tab:

Property	Description
Properties	Define any name/value pair properties that need to be passed to the directory service.

The following properties must be configured for a Sun One LDAP JNDI Directory Server using the **Runtime** tab:

Property	Description
Maximum Pool Size	
Maximum Connections Per User	
Connection Idle Time (minutes)	
Minimum Pool Size	

Java Process Definition (JPD)

Provides detailed information for configuring a Java Process Definition as an External Resource.

General Properties

This section defines the general properties for this External Resources:

Property	Description
Name	Defines the name of this external resource.
Type	Specifies the external resource type.
Supported Types	

Java Process Definition Properties

The following properties must be configured for a Java Process Definition:

Property	Description
HTTP Server Configuration	
Path	

Unit Tests

The following topics describe how to create and run CUnit and PUnit tests.

What is a Unit Test?

A unit test is a piece of code used to test pieces of code. ALBPM Provides a framework for testing individual BPM Components or an entire Process.

The ALBPM unit tests are based on the JUnit unit test framework. See <http://www.junit.org> for more information.

ALBPM provides two types of unit test suites:

Unit Test Type	Description
CUnit Test	Allows you to create unit tests for individual BPM Objects. New CUnit test suites are created with a default method. You can add other methods as necessary.
PUnit Test	Allows you to create a test framework for an entire Process. New PUnit test suites are created with a default, setUp, and tearDown method.

Within ALBPM, unit tests behave like other BPM Objects. They can contains Attributes, Groups, Presentations, and Methods.

Creating a Unit Test

The following procedure shows you how to create CUnit and PUnit test suites.

1. In the Project Navigator, right-click on the Module where you want to create a Unit Test.
2. Select **New ► PUnit Suite** or **New ► CUnit Suite**
3. Enter a name for your test suite.
4. Enter the destination module.
5. Click **Ok**.

The new test unit suite appears as a resource under your Module. You can now define your unit test within the Methods of your test suite.

Running a Unit Test

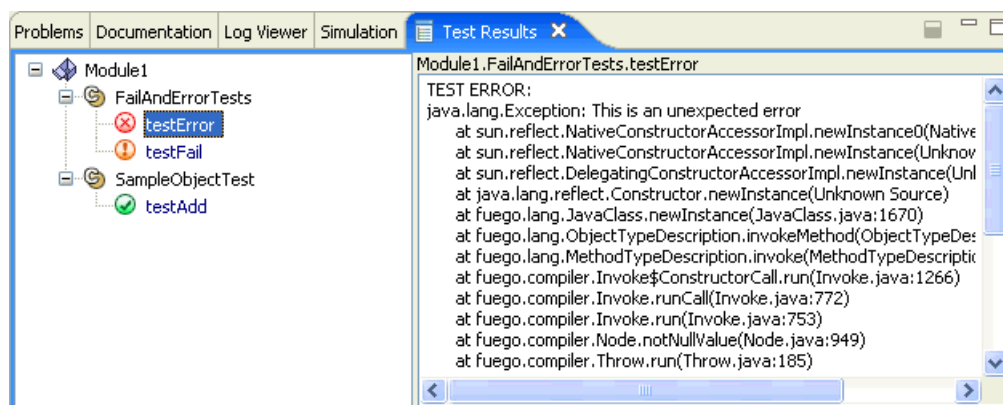
After you have created unit tests for your Components or Project, you can run the tests within ALBPM Studio. Unit test results are displayed in the Test Results View.

To run a unit test:

1. Right-click on the Module or unit test Component.
2. Select **Run Unit Test**.
The test results appear in the Test Results View.
3. Select the test in the left-hand pane.

Test Results View

The Test Results View displays results from PUnit and CUnit tests.



Process Execution Engine

Describes the Embedded Execution Engine.

What is the Process Execution Engine?

The Process Execution Engine is the ALBPM component that runs the business processes designed in Studio, and run from the user

When you run a process from Studio, you are actually running Studio's built-in process engine, which is automatically configured when Studio is installed.

The Process Execution Engine actively executes and manages the orchestration of business services according to the rules defined in a published process model. Orchestrations may be performed on services within the company or across the firewall in a B2B environment. Orchestrated processes can also be exposed as Web Services. The Engine can execute Web Services transparently across internal and external processes.

The process execution engine is responsible for:

- Accepting requests generated from WorkSpace
- Executing required tasks
- Maintaining the state of all instances flowing through the deployed processes
- Providing extensive version control that allows process models to be modified, published, and deployed with zero latency since multiple versions can be run simultaneously

The Process Execution Engine is an essential element of the BPM system. The process execution engine must also be running if you will:

- Access tasks defined in a process
- Publish, deploy, or activate a process

AquaLogic BPM Studio automatically creates and configures an embedded execution engine for each new project. This engine requires no deployment step, so you can design your processes and immediately put them into work after starting the Engine. However, some properties can be changed according to your business requirements.

The Engine preferences window allows you to set and modify a few properties related to the Embedded Execution Engine that can modify the execution behavior.

When the Embedded Execution is started, processes start to be executed. The **Engine Preferences** dialog box also allows you to view the deployment settings currently applied to that runtime environment.

Changes made to Engine preferences through this window while the engine is running will be automatically or manually applied to the runtime environment depending on the values set for the **Runtime** preferences.

Category	Description
Engine	Enables you to change some engine properties as well as to view the detailed configuration information that Studio automatically sets.
Runtime	Enables you to change runtime properties.
Deployment	Allows for changing deploy preferences information of all your processes.
Deployed Processes	Only available when the runtime environment is initiated. Displays the information of the currently deployed processes in the runtime environment.
Referrals	Provides the interface to define the information required to enable process to process communication.

Category	Description
BAM	Enables the BAM configuration

Starting the Process Execution Engine

This task outlines the procedures for starting the Process Execution Engine that is embedded within AquaLogic BPM Studio.


1. In the Project Navigator, select the Project you want to use to start the Process Execution Engine.
2. Click the green play icon in the Eclipse toolbar.
3. Optionally, select from the following options:

Option	Description
Delete Process Instances	Clears all previous process instances before starting the Engine.
Delete Log Files	Clears all log files before starting the Engine.
4. Click **Ok**.
The **Progress Information** window appears while the engine is starting. This may take several minutes.

When the Process Execution Engine starts, the green play icon changes to a red stop icon.

Setting Engine Preferences

Engine preferences allow you to customize the ALBPM Process Execution Engine behavior. Engine preferences are saved independently for each ALBPM project.

1. Ensure that the Project Navigator is visible.
See [Showing Views](#) on page 19.
2. Right-click on the ALBPM Project whose Engine preferences you want to edit.
3. Select **Engine Preferences** ().
The **Preferences** window appears.
4. Select a preference category from the list. The set of preferences for the selected category appears on the right.
5. Click **Ok**.

Process Business Language (PBL)

PBL Overview

Process Business Language is the programming language used within AquaLogic BPM projects where code is required to implement process features or to integrate with external resources.

PBL is simple, high-level language which treats components as objects. PBL can be used to define business rules and logic within Activities and certain types of Transitions. The PBL development environment is integrated within AquaLogic® BPM Studio.

This language is specifically designed to integrate systems and to clearly express business process logic. In addition, PBL supports the following features:

- Choice of syntax style, which can be native PBL, Java, or Visual Basic
- Integration with various back-end technologies including COM, CORBA, XML, SQL, Web Services, and Java
- A modern editor that supports syntax coloring, code completion, templates, and real-time error checking
- Component Libraries
- Regular Expressions

Methods compiled from PBL code are published together with the rest of a project for deployment, but a deployed project contains JVM byte code and not PBL source code. Therefore PBL code can only be created, edited, and tested using Studio.

PBL Methods

Introduces Business Process and Business Object methods

In Studio, a method can either be a *Business Process* or a *Business Object*. Both kinds support similar features, but they differ in:

- Their visibility
- The sets of available predefined variables they can access
- Their runtime environment

Business Process methods can only be accessed from the process to which they belong. They are usually the implementation of an Activity. They have several process-related, predefined variables and they always execute inside a process-controlled transaction.

Business Object methods can run on the server side or the client side. They can be defined as functions and inherit behavior from a superclass of the object that contains them. Typically, they are visible from the entire project.

Comments

Describes purpose and syntax of comments

Comments are text notes which can be read by humans but are ignored by the compiler. Including comments in your code helps make it easier for you and others to read it. Comments are especially useful to record the intent of a particular piece of code, since it may not be clear to others or, after a period of time, to the original programmer. That said, code

readability is not achieved exclusively by including comments. It is also enhanced by adhering to coding conventions and using explicit variable and object names.

Under PBL, comments can be single line or multi-line, and are delimited either with standard C++/Java syntax in the Java and PBL Styles, or with Visual Basic syntax in the Visual Basic style, as described below.



Tip: Always remember that when you write comments, you should explain the *why* and not the *how*. The how can be read from the code itself.

PBL and Java Style Comments

Single line comments are denoted with a double forward slash (//):

```
//This is a single line comment.
```

Multi-line comments are enclosed between a forward slash and an asterisk (/*) and an asterisk and a forward slash (*//):

```
/* This is a multi-line comment. It can span multiple lines
of code and can be as long as you want it to be.
You do not have to worry about line breaks, although
you may add them if you want to. */
```

Visual Basic Style Comments

Visual Basic style uses a single apostrophe (') for single line comments:

```
' This is a comment in the Visual Basic style.
```

Expressions

Expressions are operations in algebraic format that yield a value when evaluated.

An expression consists of *operators* and *operands*. Operators are special symbols commonly used in expressions, denoting the operations to be performed with the operands they are adjacent to. Operands can be variables or function invocations which return a value that can be operated on by the relevant operators

Expressions must operate on compatible variable types. Type checking is performed at compile time to guarantee that no runtime errors occur due to an invalid mix of types. Some expression examples follow.

Expressions with numerical values:

```
//Variable c is assigned the sum of a and b.
c = a + b
```

```
// myVariable is assigned the product of 12 by the sum of yourVariable and ourVariable.
myVariable = 12 * (yourVariable + ourVariable)
```

Expressions with string values:

```
employeeName = firstName + " " + lastName
```

Precedence

Notice the parentheses in the expression example above. Parentheses play a role in *precedence*. Precedence is the order in which operations take place in a mathematical equation. This is sometimes called *order of operations*. Any operation inside parentheses is evaluated first, and then followed by other operations.

If you evaluate the example:

```
myVariable = 12 * (yourVariable + ourVariable)
```

using 10 for yourVariable and 5 for ourVariable, the order of the operation is the following:

1. yourVariable is added to myVariable resulting in 15.
2. 15 is multiplied by 12 resulting in 180.
3. myVariable is assigned the value 180.

For further information on the different operators, please see Operators.

Conditional expressions

Conditional expressions assign a value to a variable depending on the result of an expression. The format of a conditional expression is as follows:

```
<Boolean expression> ? <expression> : <expression>
```

If the Boolean expression evaluates to true, the value to the left of the colon is assigned. If it evaluates to false, the value to the right of the colon is assigned. Now, look the next example.

```
myResult = (show = 1) ? "on" : "off"
```

In this example, the variable `myResult` is assigned the value "on" if the value of the variable `show` is equal to 1. Otherwise, "off" is assigned to `myResult`.

Programming Styles

Describes PBL, Java, and Visual Basic programming styles

Studio supports different programming styles to reduce the time needed to learn how to program business process methods. Each style mimics a well-known programming language as precisely as possible and adds the features that are required to write your business rules effectively.

ALBPM Studio supports the following programming styles:

- PBL: The native Process Business Language (PBL) syntax
- Java
- Visual Basic

All available styles are functionally identical except where specifically noted. In other words, the Java and Visual Basic styles are *not* Java or Visual Basic. They are actually PBL formatted with Java or Visual Basic syntax as a programming aid to people familiar with these languages.

PBL Programming Style

This is the native and recommended style. Also, most of the programming examples in this documentation are in the native PBL style. The following example shows some of the characteristics of the PBL programming style:

```
firstName as String
lastName as String
selectedButton as String

// Ask the person's name
input "First Name:" : firstName, "Last Name:" : lastName
    using title = "Enter Your Name", buttons = ["Done", "Cancel"]
    returning selectedButton = selection

// Check the button pressed
if selectedButton = "Done" then
    display "Hello " + firstName + "!"
else
    display "Hello!"
end
```

Java Programming Style

This style emulates Java syntax and adds several features to match PBL expressions. These added features include:

- Output arguments
- Input and display statements
- Variable auto-initialization

- Transformations

The following example shows some of the characteristics of the Java programming style:

```
String firstName;
String lastName;
String selectedButton;

// Ask the person's name
input("First Name:" firstName,
      "Last Name:" lastName, title : "Enter Your Name", buttons : { "Done", "Cancel" },
out selection : selectedButton);

// Check the button pressed
if (selectedButton == "Done")
{
    display("Hello " + firstName + "!");
}
else
{
    display("Hello!");
}
```

Visual Basic Programming Style

This style emulates Microsoft Visual Basic syntax. However, unlike Visual Basic, the Visual Basic style is case sensitive. This programming style also has several additional features including:

- Input and display statements.
- Variable auto-initialization.
- Transformations.

The following example illustrates the Visual Basic programming style:

```
Dim firstName As String
Dim lastName As String
Dim selectedButton As String

' Ask the person's name
Input "First Name:" : firstName,
      "Last Name:" : lastName, title := "Enter Your Name", buttons := { "Done", "Cancel"
}, Out selection := selectedButton

' Check the button pressed
If selectedButton = "Done" Then
    Display "Hello " & firstName & "!"
Else
    Display "Hello!"
End If
```

Data Types Overview

Introduces PBL data types

PBL supports a number of data types, in four categories:

- [Numbers Overview](#) on page 209
- [String Overview](#) on page 219
- [Time and Interval Overview](#) on page 226
- [Booleans](#)

Often it is necessary to convert data types to other data types. For instance, a number can be converted to a string for display, or an input string may have to be converted to a date. For more details, see [Type Conversion](#) on page 209.

Type Conversion

Explains how to convert between data types

Conversion between variable types can be accomplished by "forcing" a type on a variable of another type. There are two syntaxes to make the conversion: *functional syntax* and the *conversion operator*.

Functional Syntax

The value to be converted is passed as an argument to the type name. Any variable type can be converted into a String. The following examples show you how to force a type on a variable:

```
someNum as Int = 23
someString as String

someString = String(someNum)
someString = String('now')
```

Any variable type can be converted from a String. However, this operation can fail if the format of the String is not valid for the type to which you are converting. For example, to convert to a Time data type, certain formats must be followed, as described in [Time and Interval Overview](#) on page 226.

```
localTime as Time
localTime = Time("2002/01/20 17:39:23")
```

The following example creates an Int from a String:

```
intNumber as Int

intNumber = Int("0001920")
```

Conversion Operator

Conversions can also be used by using the conversion operator to. The conversion operator is especially important when dealing with Transformations.

```
localTime as Time
localTime = "2002/01/20 17:39:23" to Time
```

Numbers Overview

Describes numeric data types

Studio supports the following number data types:

Data Type	PBL Declaration
Integers on page 210	Int
Reals on page 210	Real
Decimals on page 211	Decimal

Integers (type Int) are generally used for counting and where whole numbers are suitable for the job. Decimals (type Decimal) can be defined with a fixed decimal point and are particularly suited to store currency values. Reals (type Real) have a floating decimal point and can therefore adopt a very large range of values, but shouldn't be used for currency values due to rounding effects.

Integers

Describes integer data types

Integers are whole numbers with no fractional part. In PBL all integer types are signed. Integers are suitable for storing unit quantities and are also used within program code as counters in loops or to handle various system values such as colors, character codes, and so on.

In PBL, integers constants can be specified in one of three bases:

Type	Example	Digits Allowed
octal	0567	0, 1, 2, 3, 4, 5, 6, 7
decimal	579	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
hexadecimal	0x5AF3	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Octal (base 8) numbers are prefixed by a 0 (zero), and are rarely used. Hexadecimal (base 16) numbers are prefixed by '0x' and may be required for some types of values. However, in most situations with PBL, only decimal (base 10) integers will be used. These should not be confused with the Decimal data type, described below.

Integer variables have a fixed range of values that depends on their size in bits, or *precision*, as follows:


Precision	Maximum Value	Minimum Value
64	9,223,372,036,854,775,807	-9,223,372,036,854,775,808
32	2,147,483,647	-2,147,483,648
16	32,767	-32,768
8	127	-128

Precision is specified in parenthesis after the data type, as follows:

```
n As Int(<precision>)
```

For example, if you wish to declare a 32-bit Int variable, you would use:

```
n As Int(32)
```

 **Note:** Only the four precision values shown in the table are defined for integers.

Reals

Describes Real numbers

Real numbers are implemented as floating point numbers according to the IEEE 754 specification, which is the specification used by Java. All constants of type Real must include a period and they may have an exponential part denoted by 'e' or 'E'. They may also be denoted by the suffix 'f', 'F', 'd' or 'D'. The suffix is optional if the exponent is included.

Reals provide a fast and compact way of handling a very large range of values. However, they introduce rounding errors and are therefore not suitable when every digit, and particularly trailing digits, must be exact. This is a requirement when operating on monetary amounts, which should be stored as *Decimals* on page 211.

The following are all valid real constants:

- 2.0f
- 2.0E20

- 5.324d

Variables of type Real can be declared in one of two precisions, measured in bits of storage:

Precision	Maximum Value	Minimum Value
64	$1.7976931348623157 \times 10^{308}$	4.9×10^{-324}
32	3.4028235×10^{38}	1.4×10^{-45}

The precision is specified in parenthesis after the data type, as follows:

```
x As Real(<precision>)
```

For example, if you wish to declare a 64-bit real, you would use:

```
x As Real(64)
```



Note: Only the two precision values shown in the table are supported.

Decimals

Describes Decimal numbers

Decimal numbers are arbitrary precision numbers which do not use an exponent multiplier as reals may. Unlike reals, decimals must include every single digit. This eliminates the rounding problems of Reals but increases storage requirements for large values. As a general rule, decimals are used for currency amounts.

Each decimal value has a *precision* and a *scale*. The scale is the number digits to the right of the decimal separator. The precision is the total number of significant digits.

When you declare a decimal number you can do the following:

- Fix neither the precision nor the scale
- Fix only the scale
- Fix both scale and precision

The following example shows various ways of declaring decimal values:

```
unspecified as Decimal
fixedScale as Decimal(2)
fixedBoth as Decimal(5, 2)
```

In this example, `unspecified` is of arbitrary precision and scale, `fixedScale` can be any number with only two digits to the right of the decimal digit (e.g. 600000.25), and `fixedBoth` can hold only numbers with up to two digits to the right of the decimal point for a total of 5 digits.

Decimal constants are a sequence of decimal digits followed by a period (.), followed by another sequence of decimal digits. Note that unlike Real constants, the number of digits to the right of the decimal point is significant and specifies the scale of the constant. The scale affects how a number is displayed as in the following example:

```
display 12.3456780 to Decimal(2)
```

This example displays 12.35. Note that it *rounds*, and does not truncate, the original value.

Decimal Arithmetic

It is very important to bear in mind the rules that apply to decimal arithmetic when dealing with variables with different decimal precisions. The are the following:

- If you want a variable to handle a determined precision, you must declare it: `Decimal(precision)`.

- In an addition or a subtraction, the result takes the highest precision of the two operands.
- In a multiplication, the precision of the result is the sum of the precisions of the two operands.
- In a division, the result has the precision of the left operand (the *dividend*).
- In an assignment:
 - it takes the precision resulting from the operation if the left operand has no precision defined or if its precision is higher.
 - it takes the left operand precision when it is lower than the precision resulting from the operation.
- Every time the precision is reduced, the resulting value is *rounded*. For example, 0.5 is rounded to 1 and not to 0.

Assignment

```
dec4 = 10.20
dec2 = dec4
dec1 = dec4
dec3 = dec4

display 'dec1: ' + dec1 + ', dec2: ' + dec2 + ', dec3: ' + dec3 + ', dec4: ' + dec4
```

This example displays the following:

```
dec1:10.2000, dec2: 10.20, dec3: 10, dec4: 10.2
```

Addition

```
dec1 = 10.20
dec2 = 1.04
dec3 = 100.003

res1 = dec1 + dec2
res2 = dec1 - dec2
res3 = dec1 + dec2 + dec3
res4 = dec3 - dec2

display 'res1: ' + res1 + ', res2: ' + res2 + ', res3: ' + res3 + ', res4: ' + res4
```

This example outputs the following:

```
res1:11.2400, res2:9.1600, res3: 111.2400, res4:98.96
```

Multiplication

```
dec1 = 10.20
dec2 = 1.04
dec3 = 100.003
res1 = dec1 * dec2
res2 = dec2 * dec1
res3 = dec3 * dec1
res4 = dec1 * dec1
display 'res1: ' + res1 + ', res2: ' + res2 + ', res3: ' + res3 + ', res4: ' + res4
```

This example outputs the following:

```
res1: 10.608000, res2:10.608000, res3: 1020.0000, res4: 104.04000000
```

Division

```
dec1 = 10.20
dec2 = 1.04
dec3 = 100.003
```

```

res1 = dec1 / dec2
res2 = dec2 / dec1
res3 = dec3 / dec1
res4 = dec1 / dec1
display 'res1: ' + res1 + ', res2: ' + res2 +
', res3: ' + res3 + ', res4: ' +
res4

```

This example outputs the following:

```
res1: 9.8077, res2: 0.10, res3: 10, res4: 1.0000
```

Real and Decimal Numbers

The choice between Real or Decimal numbers is important. Both can hold large numbers with a certain amount of precision, but there are fundamental differences between the two types of numbers:

- Real numbers are designed for speed in calculations where accuracy is not so important and where a close value is good enough.
- Decimal numbers are designed to provide a bound to the error you are willing to accept, sacrificing some performance if it is needed to guarantee the accuracy.

These differences become especially important when dealing with money. As a rule, whenever you are handling numbers that represent money, use Decimal numbers.

Enumerations

Enumerations are sets of related integer constants, where each value has a name. Studio has built-in support for enumerations (sequential and non-sequential). These are some properties of enumerations:

- Type safe: The compiler checks that the values belong to the enumeration.
- User-Friendly: Understanding a piece of code that uses them is easier. Enumerations have a name that indicates what they represent.
- Improved Performance: Comparison of enumeration values is reduced to a comparison between integers.

Using Enumerations

Enumerations can be used with a qualified name:

```
action = Action.SKI
```

or without qualification, when the type of the enumeration can be inferred from the context:

```
action = SKIP
```

To check the value of an Enumeration, you can use the is operator:

```

if action is CANCEL then
//Do something
end

```

or if you prefer, the multi-path conditional statement:

```

case action
when SKIP then
// Process SKIP
when CANCEL then
// Process CANCEL
else

```

```
// Handle all other values
end
```

Creating Enumerations

For further information on creating enumerations, refer to: Enumerations

Number Functions Reference

abs

Returns the absolute value of a numeric value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.

Arguments	Number numeric argument whose absolute value is to be determined
Returns	Numeric value of the same datatype as the input argument

The following special cases apply to this function:

- If the argument is positive, zero, or negative zero, the result is positive zero.
- If the argument is infinite, the result is positive infinity.
- If the argument is not a number (NaN), the result is also not a number.

acos

Returns the arc cosine of an angle, in the range of 0.0 through pi.

Arguments	Number Numeric argument whose arc cosine is to be determined. This value can range from -1 to 1.
Returns	Real - the arc tangent of the argument.

The following special cases apply to this function:

- If the argument is NaN or its absolute value is greater than 1, the result is NaN.

asin

Returns the arc sine of an angle, in the range of -pi/2 through pi/2.

Arguments	Number Numeric argument whose arc sine is to be determined. This value can range from -1 to 1.
Returns	Real - the arc sine of the argument.

The following special cases apply to this function:

- If the argument is NaN or its absolute value is greater than 1, the result is NaN.

- If the argument is zero, the result is a zero with the same sign as the argument.

atan

Returns the arc tangent of an angle, in the range of $-\pi/2$ through $\pi/2$.

Arguments	Number Numeric argument whose arc sine is to be determined.
Returns	Real - the arc tangent of the argument.

The following special cases apply to this function:

- If the argument is NaN, the result is NaN.
- If the argument is zero, the result is a zero with the same sign as the argument.

ceil

Returns the smallest (closest to negative infinity) real value which is not less than the argument and is equal to a mathematical integer.

Arguments	Real
Returns	Real - the smallest (closest to negative infinity) floating-point value that is not less than the argument and is equal to a mathematical integer.

The following special cases apply to this function:

- If the argument value is already equal to a mathematical integer, the result is the same as the argument.
- If the argument is positive zero or negative zero, the result is the same as the argument.
- If the argument value is less than zero but greater than -1.0, the result is negative zero.



Note: The value of `ceil(<Real>)` is exactly the value of `-floor(-<Real>)`.

cos

Returns the trigonometric cosine of an angle.

Arguments	Real - an angle, in radians.
Returns	Real - the cosine of the argument.

The following special cases apply to this function:

- If the argument is NaN or an infinity, the result is NaN.

exp

Returns Euler's number e raised to the power of a real value.

Arguments	Real - the exponent to raise e to.
Returns	Real - the value e^a , where e is the base of the natural logarithms.

The following special cases apply to this function:

- If the argument is NaN, the result is NaN.

- If the argument is positive infinity, the result is positive infinity.
- If the argument is negative infinity, the result is positive zero.

floor

Returns the largest (closest to positive infinity) real value that is not greater than the argument and is equal to a mathematical integer.

Syntax	<code>Real = floor (Real)</code>
Arguments	Real - a value.
Returns	Real - the largest (closest to positive infinity) floating-point value that is not greater than the argument and is equal to a mathematical integer.

The following special cases apply to this function:

- If the argument value is already equal to a mathematical integer, the result is the same as the argument.
- If the argument is NaN, an infinity, positive zero, or negative zero, the result is the same as the argument.

log

Returns the natural logarithm (base e) of a numeric value.

Arguments	Number - a number greater than 0.
Returns	Real - the value ln argument, the natural logarithm of argument.

The following special cases apply to this function:

- If the argument is NaN or less than zero, the result is NaN.
- If the argument is positive infinity, the result is positive infinity.
- If the argument is positive zero or negative zero, the result is negative infinity.

max(numA, numB)

Returns the greater of two numeric values. That is, the result is the argument closer to positive infinity. If the arguments have the same value, the result is that same value. If either value is NaN, the result is NaN. Unlike the numerical comparison operators, this method considers negative zero to be strictly smaller than positive zero. If one argument is positive zero and the other is negative zero, the result is positive zero (see the References in the footnotes).

Arguments	numA	numeric value A
	numB	numeric value B
Returns	Numeric value - the larger of the two arguments.	

min(numA, numB)

Returns the smaller of two numeric values. That is, the result is the argument closer to the value of Real.MIN_VALUE . If the arguments have the same value, the result is that same value.

Arguments	numA	numeric value A
	numB	

	numeric value B
Returns	Numeric value - the smaller of the two arguments.

pow

Returns the value of the first argument raised to the power of the second argument.

Arguments	Real - the base. Real - the exponent.
Returns	Real

The following special cases apply to this function:

- If the second argument is positive or negative zero, the result is 1.0.
- If the second argument is 1.0, the result is the same as the first argument.
- If the second argument is NaN, the result is NaN.
- If the first argument is NaN and the second argument is nonzero, the result is NaN.
- If
 - the absolute value of the first argument is greater than 1 and the second argument is positive infinity, or
 - the absolute value of the first argument is less than 1 and the second argument is negative infinity, the result is positive infinity.
- If
 - the absolute value of the first argument is greater than 1 and the second argument is negative infinity, or
 - the absolute value of the first argument is less than 1 and the second argument is positive infinity, the result is positive zero.
- If the absolute value of the first argument equals 1 and the second argument is infinite, the result is NaN.
- If
 - the first argument is positive zero and the second argument is greater than zero, or
 - the first argument is positive infinity and the second argument is less than zero, the result is positive zero.
- If
 - the first argument is positive zero and the second argument is less than zero, or
 - the first argument is positive infinity and the second argument is greater than zero, the result is positive infinity.
- If
 - the first argument is negative zero and the second argument is greater than zero but not a finite odd integer, or
 - the first argument is negative infinity and the second argument is less than zero but not a finite odd integer, the result is positive zero.
- If
 - the first argument is negative zero and the second argument is a positive finite odd integer, or
 - the first argument is negative infinity and the second argument is a negative finite odd integer, the result is negative zero.
- If
 - the first argument is negative zero and the second argument is less than zero but not a finite odd integer, or
 - the first argument is negative infinity and the second argument is greater than zero but not a finite odd integer, the result is positive infinity.

- If
 - the first argument is negative zero and the second argument is a negative finite odd integer, or
 - the first argument is negative infinity and the second argument is a positive finite odd integer, the result is negative infinity.
- If the first argument is finite and less than zero
 - if the second argument is a finite even integer, the result is equal to the result of raising the absolute value of the first argument to the power of the second argument.
 - if the second argument is a finite odd integer, the result is equal to the negative of the result of raising the absolute value of the first argument to the power of the second argument
 - if the second argument is finite and not an integer, the result is NaN.
- If both arguments are integers, the result is exactly equal to the mathematical result of raising the first argument to the power of the second argument if that result can, in fact, be represented exactly as a real value.

In the foregoing descriptions, a floating-point value is considered to be an integer if and only if it is finite and a fixed point of the method `ceil` or, equivalently, a fixed point of the method `floor`. A value is a fixed point of a one-argument method if and only if the result of applying the method to the value is equal to the value.

round

Returns the closest int to the argument. The result is rounded to a real by adding 1/2, taking the floor of the result and casting the result to type int. That is, round is equivalent to `floor(num + 0.5)`.

Arguments	Number numeric value to be rounded
Returns	Real or Decimal value of the argument rounded to the nearest whole number. If num is Real or Int, the function returns a Real. If num is Decimal, a Decimal is returned

The following special cases apply to this function:

- If the argument is NaN, the result is 0.
- If the argument is negative infinity or any value less than or equal to the value of `Real.MIN_VALUE` , the result is equal to the value of `Real.MIN_VALUE` .
- If the argument is positive infinity or any value greater than or equal to the value of `Real.MAX_VALUE` , the result is equal to the value of `Real.MAX_VALUE` .

sin

Returns the trigonometric sine of an angle.

Arguments	Real - an angle, in radians.
Returns	Real - the sine of the argument.

The following special cases apply to this function:

- If the argument is NaN or an infinity, the result is NaN.
- If the argument is zero, the result is a zero with the same sign as the argument.

sqrt

Returns the correctly rounded positive square root of a Real value.

Arguments	Real - a value.
-----------	-----------------

Returns	Real - the positive square root of argument. If the argument is NaN or less than zero, the result is NaN.
---------	---

The following special cases apply to this function:

- If the argument is NaN or less than zero, the result is NaN.
- If the argument is positive infinity, the result is positive infinity.
- If the argument is positive zero or negative zero, the result is the same as the argument.
- Otherwise, the result is the real value closest to the true mathematical square root of the argument value.

tan

Returns the trigonometric tangent of an angle

Arguments	Real - an angle, in radians.
Returns	Real - the tangent of the argument.

The following special cases apply to this function:

- If the argument is NaN or an infinity, the result is NaN.
- If the argument is zero, the result is a zero with the same sign as the argument.

String Overview

A string is zero or more characters put together. A character is anything that you can type, such as a letter, a digit, a symbol, or a space. Strings are used to hold any kind of text information.

String literals

A string literal consists of zero or more characters enclosed in double quotes. Each character may be represented by an escape sequence. The following are examples of string literals:

```
// empty string
display ""

// a string containing 16 characters
display "This is a string."

// a string containing one double quote
display "\""
```

In PBL style, consecutive strings are automatically merged:

```
display "This is a string "
       "made from separate strings."
```

The above code displays:

This is a string made from separate strings.

Escape sequences

Character and string escape sequences are used to denote some special characters as well as the single quote, double quote, and backslash characters in string literals. The following table lists the most common characters:

Escape code	Unicode	Description
\t	0009	Horizontal tab (HT)

Escape code	Unicode	Description
\n	000a	Newline or line feed (LF)
\f	000c	Form feed (FF)
\r	000d	Carriage return (CR)
\ '	0027	Single quote
\ "	0022	Double quote
\\	005c	Backslash

If you know the Unicode code, any character can be specified. You must prefix the four-digit hexadecimal code of the desired character with "\u". For example, the double quote would be expressed as "\u0022".

String concatenation

Strings can be concatenated at runtime with the string concatenation operator '+'. You can use any data type to build a string, so long as at least one of the elements is a string:

```
total as Int
total = 200
display "Total is: " + total + " units"
```

Regular Expressions

Advanced string pattern matching and manipulation can be done with *regular expressions*. For further information, please see [Regular Expression Overview](#) on page 291.

String Functions

substring

Returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string.

Arguments

- String: the string on which the function operates.
- Int first: the beginning index, inclusive.

Returns

- String: the specified substring.

Example

```
text as String
text = "Hello World"
display substring(text, first : 5)
```

The previous example displays "World!".

substring

Returns a new string that is a substring of this string. The substring begins at the specified first index and extends to the character at index last - 1. Thus, the length of the substring is last - first.

Arguments

- String: the string on which the function operates.
- Int first: the beginning index, inclusive.

- Int last: the ending index, exclusive.

Returns

- String: the specified substring.

Example

```
text as String
text = "Hello World!"
display substring(text, first : 5, last : 11)
```

The previous example displays "World".

fields

Given a source string and delimiter character, it returns an array of strings containing substrings of the original string delimited by the specified character. The delimiter is not included in the result.

Arguments

- String: the string on which the function operates.
- String delim: Character that delimits field.

Returns

- String[]: an array of strings containing the fields delimited by **delim**.

Example

```
text as String
text = "Hello World!"
display fields(text, delim : " ")
```

The example above displays ["Hello", "World!"].

length

Returns the number of characters in the string.

Arguments

- String: the string on which the function operates.

Returns

- Int: Number of characters in the string.

Example

```
text as String
text = "Hello World!"
display length(text)
```

The previous example displays 12.

replace

Returns a new string with all the occurrences of **from** in the original string replaced by **to**.

This function has a variation that accepts a regular expression for matching the pattern to be replaced. See Regular Expressions for details.

Arguments

- String: the string on which the function operates.

- String from: the string to find.
- String to: the replacement text.

Returns

- String: a new string with the replacements.

Example

```
text as String
text = "Hello World!"
display replace(text, from : "World", @to : "Mary")
```

The previous example displays "Hello Mary!".

charAt

This function returns the character contained in the specified index position.

Arguments

- String: the string on which the function operates.
- Int position: zero-based index of a character inside the string.

Returns

- String(1): the character at the specified index.

Example

```
text as String
text = "Hello World!"
display charAt(text, position : 6)
```

The previous example displays "W".

indexOf

Searches inside a string for another string and returns the index where the first occurrence happens.

This function has a variation that accepts a regular expression for matching. See Regular Expressions for details.

Arguments

- String: the string on which the function operates.
- String text: the text to find.

Returns

- Int: the index of the occurrence or -1 if not found.

Example

```
text as String
text = "Hello World!"
display indexOf(text, text : "Wor")
```

The previous example displays 6.

lastIndexOf

Searches inside a string for another string and returns the index where the last occurrence happens.

This function has a variation that accepts a regular expression for matching. See Regular Expressions for details.

Arguments

- String: the string on which the function operates.
- String text: the text to find.

Returns

- Int: the index of the occurrence or -1 if not found.

Example

```
text as String
text = "Hello World!"
display lastIndexOf(text, text : "o")
```

The previous example displays 7.

split

Splits a string using a regular expression. The delimiters are not included. See Regular Expressions for details.

Example

```
text as String= "One Two Three"
display split(text, '/\w+ \w+/m')
```

The previous example produces this output ["", "Three"].

count

This function counts the number of times that a character is found in a string.

Arguments

- String: the string on which the function operates.
- String(1) ch: character to find.

Returns

- Int: the number of occurrences of the specified character in the string.

Example

```
date as String = "10/12/2004"
if count (date, ch: "/" ) = 2 then
  date = replace(date, "/", "-")
  display date
else
  display "Bad Date Format"
end
```

The previous example displays "10-12-2004".

toUpperCase

Returns a new string with all the characters in uppercase.

Arguments

- String: the string on which the function operates.

Returns

- String: a new string with all the characters in uppercase.

Example

```
text as String
text = "Hello World!"
display toUpperCase(text)
```

The previous example displays "HELLO WORLD!".

toLowerCase

Returns a new string with all the characters in lowercase.

Arguments

- String: the string on which the function operates.

Returns

- String: a new string with all the characters in lowercase.

Example

```
text as String
text = "Hello World!"
display toLowerCase(text)
```

The previous example displays "hello world!".

trim

Returns a new string with all the whitespace removed from the beginning and the end of the string.

Arguments

- String: the string on which the function operates.

Returns

- String: a new string with all the leading and trailing whitespace removed.

Example

```
option as String = " Yes  "
if toLowerCase(trim (option))= "yes" then
  display "The option is correct"
else
  display "The option is wrong".
end
```

isMatch

Checks whether a string matches a regular expression. See [Regular Expression Overview](#) on page 291 for details.

Example

```
text as String= "One Two Three"
display isMatch(text, '/\w+ Two \w+/g')
```

The previous example displays true.

contains

This function returns true if a substring in a text matches the specified regular expression. See Regular Expressions for details.

Example

```
text as String= "One Two Three Four Five"
display contains(text, '/\w+ Tw/g')
```

The previous example displays true.

chars

Returns an array of String(1) containing all the characters in the string.

Arguments

- String: the string on which the function operates.

Returns

- String(1[]): the characters in the string.

Example

```
text as String= "fuego"
characters as String(1)[]
characters = chars(text)
for each i in characters
do
    display "Char is " + i
end
```

The previous example displays:

```
"Char is f"
"Char is u"
"Char is e"
"Char is g"
"Char is o"
```

pad

Returns a new string completed with spaces until the specified length is reached.

Arguments

- String: the string on which the function operates.
- Int len: the desired length of the string. If you pass a negative number (e.g.: -1), it is ignored and returns an empty string.

Returns

- String: a new string of the specified length.

Example

```
text as String
text = "Hello World!"
display pad(text, len : 20)
```

strip

The `strip` function returns a new string truncated to a specified length. If the string is shorter than the length specified, it is left as it is.

Arguments

- String: the string on which the function operates.
- Int len: the desired length of the string. If you pass a negative number (e.g.: -1), it is ignored and returns an empty string.

Returns

- String: a new string of the specified length.

Example

```
text as String
text = "Hello World!"
display strip(text, len : 5)
```

The previous example displays "Hello".

How to convert a String to a Time

The following will convert the string in `strDate` into a Time value to be set in `realDate`.

```
strDate = "Tue Feb 22 15:26:02 ART 2005"
strPattern = "EEE MMM dd HH:mm:ss z yyyy"

simpleDateFormat = Java.Text.SimpleDateFormat(strPattern)
realDate = parse(simpleDateFormat, strDate)

display realDate
```

The value of `realDate` becomes **2005-02-22 15:26:02-03**

`realDate` is a variable of type Time. It contains the same date that was expressed as a string in `strDate`.

String Attributes

Empty

This attribute returns true if the string is empty (its length is zero).

Returns

- Boolean: true if its length is zero. False otherwise.

Example

```
text as String
text = "Hello World!"
display text.empty
```

Time and Interval Overview

Studio supports two built-in types for time management:

- Time - represents a specific point in time.
- Interval - represents the difference between two moments in time.

Times are stored as the number of microseconds since Jan 1, 1970 (also known as UNIX epoch).

Intervals are stored as months, days, and microseconds.

Times

The string representation of a time is ISO 8601 compliant. When converting a string into a time, Studio supports a somewhat relaxed subset of ISO 8601; even dates without separators are accepted. The accepted string formats match those of time literals.

Time Literals

Time literals are specified between single quotes. The following is a list of valid time literals:

```
'23:30'
'23:30:23'
'23:30:23.001023'
'23:30:23.001023Z'
'23:30:23.001023-05'
'23:30:23.001023-3:30'
'1995-02-03'
'1995-02-03 23:30'
'1995-02-03 23:30:23'
'1995-02-03 23:30:23.001023'
'1995-02-03 23:30:23.001023Z'
'1995-02-03 23:30:23.001023-05'
'1995-02-03 23:30:23.001023-3:30'
'1995-02-03T23:30'
'1995-02-03T23:30:23'
'1995-02-03T23:30:23.001023'
'1995-02-03T23:30:23.001023Z'
'1995-02-03T23:30:23.001023-05'
'1995-02-03T23:30:23.001023-3:30'
'19950203T'
'19950203T233023.001023-330'
```

Time Zones

Following ISO 8601, time zones are specified as offsets from UTC. Named time zones are not supported because there is no international standard for time zone abbreviations. If no time zone is specified in a time literal, the default time zone for the current locale is used.

Note that:

- Interactive methods run in the locale of the current user.
- Automatic methods run in the Process Execution Engine's locale.

When a time is presented to a user, the format associated to the user's locale is used. For custom time formatting, the format function can be used. For further information, please see Time Functions.

Intervals

Intervals have two primary parts:

- Two calendar dependent components (months and days)
- A calendar independent component (hours, minutes, seconds and microseconds)

The calendar dependent component exists, so arithmetic between time and interval is consistent. When using a Gregorian calendar (the most common calendar in use), you cannot assume that a month equals to 30 days. In fact, you cannot even assume that a day lasts 24 hours.

The Gregorian calendar inserts two corrections:

- The leap year - Every four years a day is added to February, unless the year is a multiple of 100 and not a multiple of 400 (which is why the year 2000 had 366 days, instead of 365 like 1900).

- The leap second - Sometimes a second is added or removed from the last minute of certain days to cope with the accumulated error caused by the Earth's change of speed.


So, for example, if you want to obtain a time two months from now, you can do:

```
display 'now' + '2M'
```

Interval Literals

Interval literals are enclosed by single-quotes ('). They are formed by a sequence of fields, where each field is a number plus a unit suffix. The following table lists all valid suffixes:

Unit Suffix	Description
Y	Years
M	Months
d or D	Days
h or H	Hours
m	Minutes
s or S	Seconds
x	Microseconds

 **Note:** A 'T' in the constant forces the interpretation of 'M' to be equal to 'm', e.g.: '2MT2M' equals '2M2m'.

All magnitudes can contain a '.' to express a fractional part, although the fractional part is dropped for days or months.

The following example shows some valid Interval constants:

```
display '2MT2.5M'  
display '1Y1M3h2m1.500s'  
display '1.5h'
```

Arithmetic

As mentioned before, time and intervals have some arithmetic rules.

The following table lists the behavior of addition and subtraction with time and interval:

Operations	Result Type
Time - Time	Interval
Time + Interval (or Interval + Time)	Time
Time - Interval	Time
Interval + Interval	Interval
Interval - Interval	Interval

For further information on Time and Interval, please refer to the following:

- Time Attributes
- Interval Attributes
- Time and Interval Functions

Time Attributes

A Time object contains several attributes to manipulate the different components of time, such as days and hours. The following table lists all Time's attributes and provides some examples:

Attribute	Description
AD	Field that indicates the calendar era indicating the common era (Anno Domini.).

Example:

```
time as Time
/*this will display 1 because
the current era is AD*/
display time + "\n\n AD = " + time.AD
```

Attribute	Description
am	Boolean value which indicates whether the Time's period is between midnight to just before noon.

Example:

```
time as Time
time = '2004-12-25 02:45:00-03'
//this will display true
display time + "\n\n am = " + time.am
time = '2004-12-25 20:45:00-03'
display time + "\n\n am = " + time.am
```

Attribute	Description
ampm	Field that indicates the period of the day. If the period is am, it will return 0; otherwise, it will return 1.

Example:

```
time as Time
time = '2004-12-25 02:45:00-03'
//this will display 0
display time + "\n\n ampm = " + time.ampm
time = '2004-12-25 20:45:00-03'
//this will display 1
display time + "\n\n ampm = " + time.ampm
```

Attribute	Description
BC	Field that indicates the calendar era indicating the period before the common era (before Christ).

Example:

```
time as Time
/*this will display 0 because
the current era is not BC*/
display time + "\n\n BC = " + time.BC
```

Attribute	Description
date	String value containing the date formatted with the default mask.

Example:

```
time as Time
time = '2004-12-25 20:45:00-03'
//this will display Dec 25, 2004
display time + "\n\n" + time.date
```

Attribute	Description
day	Int value representing the day component of the time.

Example:

```
time as Time
time = '2004-12-25 20:45:00-03'
//it will display 25
display time + "\n\n day: " + time.day
```

Attribute	Description
dayOfMonth	Int value representing the day part of the time.

Example:

```
time as Time
time = '2004-12-25 20:45:00-03'
//this will display 25
display time + "\n\n day of month: " +
time.day
```

Attribute	Description
dayOfWeek	Day of the week. Returns an integer from 1 to 7, where Sunday is 1.

Example:

```
time as Time
time = '2004-12-25 20:45:00-03'
display time + "\n\n day of the week: " +
time.dayOfWeek
```

Attribute	Description
days	Returns the number of days elapsed since 00:00 January 1, 1970 GMT (UNIX epoch).

Example:

```
time as Time
display "days since EPOCH: " + time.days
```

Attribute	Description
EPOCH	1969-12-31 00:00:00-00. Base time from which milliseconds to calculate dates are counted.

Example:

```
//1969-12-31 00:00:00-00
display "EPOCH: " + Time.EPOCH
```

Attribute	Description
firstDayOfMonth	Time value corresponding to the first day of the month.

Example:

```
time as Time
//firstDayOfMonth is a Time object
display "first day of month: " +
time.firstDayOfMonth
```

Attribute	Description
hour	Int value representing the hour component of the date in the format h.

Example:

```
time as Time
time = '2004-12-25 20:45:00-03'
display time + "\n\n hour: " + time.hour
```

Attribute	Description
hourOfDay	Int value representing the hour component of the date in the format hh.

Example:

```
time as Time
time = '2004-12-25 20:45:00-03'
display time + "\n\n hour of the day: " +
time.hour
```

Attribute	Description
hours	Returns the number of hours elapsed since January 1, 1970 GMT.

Example:

```
time as Time
display "hours passed since EPOCH: "+
time.hours
```

Attribute	Description
lastDayOfMonth	Time value corresponding to the last day of the month.

Example:

```
time as Time
//lastDayOfMonth is a Time object
display "last day of month: " +
time.lastDayOfMonth
```

Attribute	Description
locale	This attribute is used to change the current locale. It is a write only attribute.

Example:

```
time as Time
display "date fomatted with default locale: " +
time.formatDate
Time.locale = Java.Util.Locale.GERMAN
```

```
display "date with German locale: " +  
time.formatDate
```

Attribute	Description
maxvalue	The maximum value a Time object can have.

Example:

```
display "Time object's maximum value: "  
+ Time.maxvalue
```

Attribute	Description
microSecond	Int value representing the microseconds component of the date.

Example:

```
time as Time  
display time + "\n\n microseconds in time: " +  
time.microSecond
```

Attribute	Description
microSeconds	Returns the number of microseconds elapsed since January 1, 1970 GMT.

Example:

```
time as Time  
display "microseconds since EPOCH: " +  
time.microDeconds
```

Attribute	Description
milliSeconds	Returns the number of milliseconds elapsed since January 1, 1970 GMT.

Example:

```
time as Time  
display "milliseconds since EPOCH: " +  
time.milliSeconds
```

Attribute	Description
minute	Int value representing the minutes component of the date.

Example:

```
time as Time  
time = '2004-12-25 20:45:00-03'  
display time + "\n\n minutes: " + time.minute
```

Attribute	Description
minutes	Returns the number of minutes elapsed since January 1, 1970 GMT.

Example:

```
time as Time  
display "minutes since EPOCH: " + time.minutes
```

Attribute	Description
minvalue	The minimum value a Time object can have.

Example:

```
time as Time
display "Time object's minimum value: "
+ time.minvalue
```

Attribute	Description
month	Int value representing the month component of the date.

Example:

```
time as Time
time = '2004-12-25 20:45:00-03'
display time + "\n\n month: " +
time.month
```

Attribute	Description
second	Int value representing the seconds component of the date.

Example:

```
time as Time
time = '2004-12-25 20:45:10-03'
display time + "\n\n seconds: " + time.second
```

Attribute	Description
seconds	Returns the number of seconds elapsed since January 1, 1970 GMT.

Example:

```
time as Time
display "seconds since EPOCH: " + time.seconds
```

Attribute	Description
timeOnlyHour	Int value representing the hours component of the time, without calendar corrections.

Example:

```
time as Time
display time + "\n\n" +
"hours: " +
time.timeOnlyHour
```

Attribute	Description
timeOnlyMicroSecond	Int value representing the microseconds component of the time, without calendar corrections.

Example:

```
time as Time
display time + "\n\n" +
"microseconds: "+
time.timeOnlyMicroSecond
```

Attribute	Description
timeOnlyMinute	Int value representing the minutes component of the time, without calendar corrections.

Example:

```
time as Time
display time + "\n\n"+
"minutes: " +
time.timeOnlyMinute
```

Attribute	Description
timeOnlySecond	Int value representing the seconds component of the time, without calendar corrections.

Example:

```
time as Time
display time + "\n\n"+
"seconds: " +
time.timeOnlySecond
```

Attribute	Description
time	String value containing the time of the day formatted with the default mask.

Example:

```
time as Time
display "time formatted with default mask:" +
time.date
```

Attribute	Description
timeZone	Time according to the locale.

Example:

```
time as Time
Time.timeZone = TimeZone.getTimeZone( "GMT-3" )
display "GMT-3: " + time
Time.timeZone = TimeZone.getTimeZone( "GMT-8" )
display "GMT-8: " + time
```

Attribute	Description
weekOfMonth	Int value indicating the week number within the current month, starting from 1.

Example:

```
time as Time
time = '2004-01-01 20:45:00-03'
display time + "\n\n week of month: " +
time.weekOfMonth
time = '2004-01-07 20:45:00-03'
display time + "\n\n week of month: " +
time.weekOfMonth
```

Attribute	Description
weekOfYear	Int value indicating the week number within the current year, starting from 1.

Example:

```
time as Time
time = '2004-01-07 20:45:00-03'
display time + "\n\n week of year: " +
time.weekOfYear
time = '2004-02-07 20:45:00-03'
```

```
display time + "\n\n week of year: " +
time.weekOfYear
```

Time Functions

addDays

Adds a specified number of days to a Time object.

Arguments

- Time - the Time object to which days will be added.
- Int i - the number of days to be added to the time object.

Returns

The Time object resulting from adding the specified number of days to the given time.

Example

The following example adds 15 days to the current time and displays the result.

```
display "time in 15 days will be: " +
    addDays('now', i : 15)
```

addHours

Adds a specified number of hours to a Time object.

Arguments

- Time - the Time object to which hours will be added.
- Int i - the number of hours to be added to the Time object.

Returns

The Time object resulting from adding the specified number of hours to the given time.

Example

The following example adds 12 hours to the current time and displays the result:

```
display "time in 12 hours will be: " +
    addHours('now', i : 12)
```

addMicroSeconds

Adds a specified number of microseconds to a Time object.

Arguments

- Time - the Time object to which microseconds will be added.
- Int i - the number of microseconds to be added to the Time object.

Returns

The Time object resulting from adding the specified number of microseconds to the given time.

Example

The following example adds 500 microseconds to the current time and displays the result:

```
display "time in 500 microseconds will be: " +
    addMicroSeconds('now', i : 500)
```

addMilliseconds

Adds a specified number of milliseconds to a Time object.

Arguments

- Time - the Time object to which milliseconds will be added.
- Int i - the number of milliseconds to be added to the Time object.

Returns

The Time object resulting from adding the specified number of milliseconds to the given time.

Example

The following example adds 50,000 milliseconds to the current time and displays the result:

```
display "time in 50000 milliseconds will be: " +
    addMilliseconds('now', i : 50000)
```

addMinutes

Adds a specified number of minutes to a Time object.

Arguments

- Time - the Time object to which the milliseconds will be added.
- Int i - the number of minutes to be added to the Time object.

Returns

The Time object resulting from adding the specified number of milliseconds to the given time.

Example

The following example adds 30 minutes to the current time and displays the result:

```
display "time in 30 minutes will be: " +
    addMinutes('now', i : 30)
```

addMonths

Adds a specified number of months to a Time object.

Arguments

- Time - the Time object to which the months will be added.
- Int i - the number of months to be added to the Time object.

Returns

The Time object resulting from adding the specified number of months to the given time.

Example

The following example adds 6 months to the current time and displays the result.

```
display "time in 6 months will be: " +
    addMonths('now', i : 6)
```

addSeconds

Adds a specified number of seconds to a Time object.

Arguments

- Time - the Time object to which the seconds will be added.
- Int i - the number of seconds to be added to the Time object.

Returns

The Time object resulting from adding the specified number of seconds to the given time.

Example

The following example adds 50 seconds to the time object and displays the result:

```
display "time in 50 seconds will be: " +
    addSeconds('now', i : 50)
```

addWeeks

Adds a specified number of weeks to a Time object.

Arguments

- Time - the Time object to which the weeks will be added.
- Int i - the number of weeks to be added to the Time object.

Returns

The Time object resulting from adding the specified number of weeks to the given time.

Example

The following example adds 50 weeks to the current time and displays the result:

```
display "time in 50 weeks will be: " +
    addWeeks('now', weeks : 50)
```

addYears

Adds a specified number of years to a Time object.

Arguments

- Time - the Time object to which the years will be added.
- Int i - the number of years to be added to the Time object.

Returns

The Time object resulting from adding the specified number of years to the given time:

```
display "time in 10 years will be: " +
    addYears('now', i : 10)
```

add

Adds a specified interval of time to a Time object.

Arguments

- Time - the Time object to which the interval of time will be added.
- Interval i - interval of time to add to the Time object.

Returns

The Time object resulting from adding the specified interval to the given time.

Example

The following example adds 5 days, 15 hours, and 30 minutes to the current time and displays the result:

```
display "time in 5 days 15 hours and 30 minutes: \n\n" +
    add('now', interval : '5d15h30m')
```

between

Determines if the given Time object is between two Time objects.

Arguments:

- Time - the given Time object.
- Time from - the upper bound of the time period in which to search the given time, exclusive.
- Time to - the lower bound of the time period in which to search the given time, inclusive.

Returns

true - if the given Time object is contained in the specified period.

false - if the given object is not contained in the specified period.

Example

The following example finds out if the current time is in between the first day and last day of year 2004:

```
display "is today between 2004-01-01 and 2004-12-31? " +
  between('now', from : '2004-01-01 12:00:00',
    @to : '2004-12-31 12:00:00')
```

daysSince

Calculates the days passed between a given time and another time.

Arguments

- Time - the Time object.
- Time t - the other Time object.

Returns

An Int value representing the number of days between the two given times.

Example

The following example defines a Time variable birthdate, calculates the days passed between 'now' and birthdate and displays the result:

```
birthdate as Time
birthdate = '1979-02-19'
display "days passed since birthdate: " +
  daysSince('now', t : birthdate)
```

formatDate

Formats the Time object with the default mask.

Arguments

- Time - the Time object to be formatted.

Returns

A String containing the representation of the Time object formatted with the default mask.

Example

The following example displays the current time formatted with the default mask:

```
display formatDate('now')
```

formatDate

Formats the Time object with the specified date formatting style for the default locale.

Arguments

- Time - the Time object to be formatted.
- Int style - The formatting style. Available styles are Time.DEFAULT, Time.FULL, Time.LONG, and Time.SHORT.

Returns

A String containing the representation of the Time object formatted with the specified style.

Example

The following example displays the current time with the four possible formatting styles:

```
defaultDate as String = "Default format --> " +
```

```

        formatDate('now', dateStyle : Time.DEFAULT)

fullDate as String = "Full format --> " +
    formatDate('now', dateStyle : Time.FULL)

longDate as String = "Long format --> " +
    formatDate('now', dateStyle : Time.LONG)

shortDate as String = "Short format --> " +
    formatDate('now', dateStyle : Time.SHORT)

display defaultDate + "\n\n" + fullDate + "\n\n" +
    longDate + "\n\n" + shortDate + "\n\n"

```

formatTimeOnly

Formats this Time object as a time only, with no time zone correction.

Arguments

- Time - The Time object to be formatted.

Returns

A String containing the representation of the Time object formatted as time only.

The following example displays the current time formatted as time only. It should display something similar to 12439d 16:58:58:

```
display formatTimeOnly('now')
```

formatTimeOnly

Formats this Time object as a time only, with no time zone correction, applying the specified style.

Arguments

- Time - The Time object to be formatted.
- Int - The formatting style. Available styles are Time.DEFAULT, Time.FULL, Time.LONG, and Time.SHORT.

Returns

A String containing the representation of the Time object formatted as time only, applying the specified style.

Example

The following example only displays the time representation of the current time in the four available styles:

```

defaultTime as String = "Default format --> " +
    formatTimeOnly('now', intervalStyle : Time.DEFAULT)

fullTime as String = "Full format --> " +
    formatTimeOnly('now', intervalStyle : Time.FULL)

longTime as String = "Long format --> " +
    formatTimeOnly('now', intervalStyle : Time.LONG)

shortTime as String = "Short format --> " +
    formatTimeOnly('now', intervalStyle : Time.SHORT)

display defaultTime + "\n\n" +
    fullTime + "\n\n" +
    longTime + "\n\n" +
    shortTime + "\n\n"

```

formatTime

Formats the time component of a Time object with a specified style.

Arguments

- Time - The Time object to be formatted.
- Int timeStyle - The formatting style. Available styles are Time.DEFAULT, Time.FULL, Time.LONG, and Time.SHORT.

Returns

A String containing the time component of the Time object formatted with the specified style.

Example

The following example displays the time component of the current time formatted with the four available styles:

```
defaultTime as String = "Default format --> " +
    formatTime('now', timeStyle : Time.DEFAULT)

fullTime as String = "Full format --> " +
    formatTime('now', timeStyle : Time.FULL)

longTime as String = "Long format --> " +
    formatTime('now', timeStyle : Time.LONG)

shortDate as String = "Short format --> " +
    formatTime('now', timeStyle : Time.SHORT)

display defaultTime + "\n\n" +
    fullTime + "\n\n" +
    longTime + "\n\n" +
    shortTime + "\n\n"
```

format

Formats a Time object with the default mask.

Arguments

- Time - the Time object to be formatted.

Returns

A string containing the representation of the Time object formatted with the default mask.

Example

The following example displays the current time formatted with the default mask.

```
display format('now')
```

format

Formats a Time object with a specified date style and time style.

Arguments

- Time - The Time object to be formatted.
- Int dateStyle - The style to be applied to the date component of the Time object. Available styles are Time.DEFAULT, Time.FULL, Time.LONG, and Time.SHORT.
- Int timeStyle - The style to be applied to the time component of the Time object. Available styles are Time.DEFAULT, Time.FULL, Time.LONG, and Time.SHORT.

Returns

A string containing the representation of the Time object whose date was formatted with the specified date style, and whose time was formatted with the specified time style.

Example

The following example displays the current time with its date in full format style and its time in short format style first, then the same time with its date in short format style and its time in full format style:

```
fullDateShortTime as String = "Full date, short time: " +
    format('now', dateStyle : Time.FULL,
           timeStyle : Time.SHORT)

shortDateFullTime as String = "Short date, full time: " +
    format('now', dateStyle : Time.SHORT,
           timeStyle : Time.FULL)

display fullDateShortTime + "\n\n" + shortDateFullTime
```

format

Formats a Time object by applying a specified formatter.

Arguments

- Time - The Time object to be formatted.
- Java.Text.DateFormat formatter - The formatter to be applied in order to format the Time object.

Returns

A String containing the representation of the Time object formatted by applying the specified formatter.

Example

The following example displays the current Time formatted with the formatter passed by arguments:

```
display format('now', formatter : DateFormat.getInstance())
```

format

Formats a Time object with a specified formatter using the provided time zone and locale.

Arguments

- Time - The Time object to be formatted.
- Java.Text.DateFormat formatter - The formatter to be applied in order to format the Time object.
- Java.Util.TimeZone timeZone - The time zone to apply to the Time object when formatting it.
- Java.Util.Locale locale - The locale to apply to the Time object when formatting it.

Returns

A String containing the representation of the Time object formatted by applying the specified formatter and the time zone and locale provided.

Example

The following example displays the current Time formatted applying the formatter passed by arguments, GMT-10 time zone and French locale:

```
display format('now', formatter : DateFormat.getInstance(),
              timeZone : TimeZone.getTimeZone(arg1 : "GMT-10"),
              locale : Java.Util.Locale.FRANCE)
```

format

Formats a Time object with a specified mask.

Arguments

- Time - The Time object to be formatted.
- String mask - A string containing the mask to apply in order to format the Time object.

Returns

A String containing the representation of the Time object formatted with the specified mask. The String should be written according to the patterns and rules described below.

The following pattern letters are defined (all other characters from 'A' to 'Z' and from 'a' to 'z' are reserved):

Letter	Date or Time Component	Presentation	Example
G	Era designator	Text	AD
y	Year	Year	1996; 96
M	Month in year	Month	July; Jul; 07
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day in week	Text	Tuesday; Tue
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800

Pattern letters are usually repeated, as their number determines the exact presentation:

- Text: For formatting, if the number of pattern letters is 4 or more, the full form is used. Otherwise, a short or abbreviated form is used if available. For parsing, both forms are accepted, independent of the number of pattern letters.
- Number: For formatting, the number of pattern letters is the minimum number of digits, and shorter numbers are zero-padded to this number. For parsing, the number of pattern letters is ignored unless it's needed to separate two adjacent fields.
- Year: For formatting, if the number of pattern letters is 2, the year is truncated to 2 digits. Otherwise, it is interpreted as a number.

Example

The following example displays the current time formatted with the mask defined by the String passed by arguments. It should display something like: Thu 22 01 2004 04:31:48:975 PM ART:

```
display 'now'.format("E dd MM yyyy hh:mm:ss:SS a z")
```

getDateFormat

Returns an appropriate DateFormat based on the parts needed and a style.

Arguments

- Int parts - The needed parts. These could be Time.DATE_ONLY, Time.TIME_ONLY, or Time.DATE_TIME.
- Int style - The formatting style. Available styles are Time.DEFAULT, Time.FULL, Time.LONG, and Time.SHORT.

Returns

The format based on the required parts and style.

Example

The following example uses the function `getDateFormat` with different needed parts and style to format the current time:

```
fullDateOnly as String

fullDateOnly = "Date only - Full format: " +
  'now'.format(Time.getDateFormat(
    parts : Time.DATE_ONLY,
    style : Time.FULL))

shortTimeOnly as String
shortTimeOnly = "Time only - Short format: " +
  'now'.format(Time.getDateFormat(
    parts : Time.TIME_ONLY,
    style : Time.SHORT))

longDateTime as String
longDateTime = "Date time - Long format: " +
  'now'.format(Time.getDateFormat(
    parts : Time.DATE_TIME,
    style : Time.FULL))

display fullDateOnly + "\n\n" + shortTimeOnly + "\n\n" +
  longDateTime
```

getDateFormat

Returns an appropriate `DateFormat` based on the required parts, style, and locale.

Arguments

- `Int parts` - The needed parts. These could be `Time.DATE_ONLY`, `Time.TIME_ONLY`, or `Time.DATE_TIME`.
- `Int style` - The formatting style. Available styles are `Time.DEFAULT`, `Time.FULL`, `Time.LONG`, and `Time.SHORT`.
- `Java.Util.Locale` - The locale the date formatter will have to apply.

Returns

Returns an appropriate `DateFormat` based on the required parts, style, and locale.

Example

The following example uses the function `getDateFormat` with different needed parts and style, and french locale, to format the current time:

```
fullDateOnly as String
fullDateOnly = "Date only - Full format: " +
  'now'.format(Time.getDateFormat(
    parts : Time.DATE_ONLY,
    style : Time.FULL, Java.Util.Locale.FRANCE))

shortTimeOnly as String
shortTimeOnly = "Time only - Short format: " +
  'now'.format(Time.getDateFormat(
    parts : Time.TIME_ONLY,
    style : Time.SHORT, Java.Util.Locale.FRANCE))

longDateTime as String
longDateTime = "Date time - Long format: " +
  'now'.format(Time.getDateFormat(
    parts : Time.DATE_TIME,
    style : Time.FULL, Java.Util.Locale.FRANCE))

display fullDateOnly + "\n\n" + shortTimeOnly + "\n\n" +
```

```
longDateTime
```

Time.getEaster

Calculates Easter day for a specified year.

Arguments

y

The year for which you would like to know Easter's date.

Returns

A Time object containing Easter date.

Example

The following example displays Easter date for the year 2007:

```
display Time.getEaster(y : 2007)
```

This will show:

```
Apr 8, 2007 12:00:00 AM
```

Time.getMonthName

Returns the name of the specified month.

Arguments

monthNum

The number which identifies the month from which you would like to know the name. This number may vary between 1 (January) and 12 (December).

Returns

The name of the specified month.

Example

The following example displays the name of month 2 (February):

```
display Time.getMonthName(monthNum : 2)
```

Time.getWeekdayName

Returns the name of the specified weekday.

Arguments

- **Int dayNum** - The number which identifies the day from which you would like to know the name. This number may vary between 0 (Sunday) and 6 (Saturday).

Returns

The name of the specified day.

Example

The following example displays the day 3 (Wednesday):

```
display Time.getWeekdayName(dayNum : 3)
```

max

Returns the greater time between two Time objects.

Arguments**Time1**

A Time object

Time2

Another Time object

Returns

The greater Time object.

Example

The following example displays the greater Time between the current time and the date the man landed on the moon. The current time is returned:

```
manOnMoon as Time
manOnMoon = '1969-07-21 02:56:00 -00'
display max(manOnMoon, b : 'now')
```

min

Returns the smaller Time object between two Time objects.

Arguments

- Time a - A Time object.
- Time b - Another Time object.

Returns

The smaller Time object.

For example:

```
manOnMoon as Time
manOnMoon = '1969-07-21 02:56:00 -00'
display min(manOnMoon, b : 'now')
```

monthsSince

Returns the number of months elapsed since a given Time object.

Arguments**Time**

A Time object

Returns

An Int value with the number of whole months elapsed since the time specified by the Time object to the present time. For example, there are two whole months between January 15 and March 15, but only one whole month between January 15 and March 14.

For example:

```
xMas2007 as Time
xMas2007 = '2007-12-25'
display monthsSince('now', t : xMas2007)
```

roundToSeconds


Rounds a Time object to the nearest second.

Arguments**Time**

A Time object

Returns

A Time object with the value of Time rounded to the nearest second.

 **Note:** The value is rounded, not truncated, so 499 milliseconds or less will be rounded down, while 500 milliseconds or more will be rounded up.

Example

The following example creates a Time object called `timeStart` with seconds and milliseconds, then it displays the value rounded to the nearest second:

```
timeStart as Time
time = '2007-04-16 17:21:30.235'
display roundToSeconds(timeStart)
```

This displays:

```
Apr 16, 2007 2:21:30 PM
```

Interval Attributes

Attribute	Description
ONE_DAY	Interval value representing an interval of one day.

Example:

```
interval as Interval
interval = '25d5h1m'
display "original interval: " + interval
interval = interval + Interval.ONE_DAY
display "after adding a day: " + interval
```

Attribute	Description
ONE_HOUR	Interval value representing an interval of one hour.

Example:

```
interval as Interval
interval = '25d5h1m'
display "original interval: " + interval
interval = interval + Interval.ONE_HOUR
display "after adding an hour: " + interval
```

Attribute	Description
ONE_MINUTE	Interval value representing an interval of one minute.

Example:

```
interval as Interval
interval = '25d5h1m'
display "original interval: " + interval
interval = interval + Interval.ONE_MINUTE
display "after adding a minute: " + interval
```

Attribute	Description
ONE_MONTH	Interval value representing an interval of one month.

Example:

```
interval as Interval
interval = '2M25d5h1m'
display "original interval: " + interval
interval = interval + Interval.ONE_MONTH
display "after adding a month: " + interval
```

Attribute	Description
ONE_SECOND	Interval value representing an interval of one second.

Example:

```
interval as Interval
interval = '25d5h1m7s'
display "original interval: " + interval
interval = interval + Interval.ONE_SECOND
display "after adding a second: " + interval
```

Attribute	Description
ZERO	Interval representing the zero value. When added to another interval, the interval does not change its value.

Example:

```
interval as Interval
interval = '1M20d3h40m5s'
display "original interval: " + interval
interval = interval + Interval.ZERO
display "after adding ZERO: " + interval
```

Attribute	Description
daysOnly	Int value representing the days component of an interval.

Example:

```
interval as Interval
interval = '1M20d10h'
display "days component of an interval: " +
interval.daysOnly
```

Attribute	Description
days	Int value representing the days component of an interval.

Example:

```
interval as Interval
interval = '1M20d10h'
display "days component of an interval: " +
interval.days
```

Attribute	Description
hoursOnly	Int value representing the hours component of an interval.

Example:

```
interval as Interval
interval = '1d20h30m'
display "hours component of an interval: " +
interval.hoursOnly
```

Attribute	Description
hours	Int value representing the hours component of an interval.

Example:

```
interval as Interval
interval = '1d20h30m'
display "hours component of an interval: " +
interval.hours
```

Attribute	Description
microSecondsOnly	Int value representing the microseconds component of an interval.

Example:

```
interval as Interval
interval = Interval("20.000003s")
display "microseconds component of an interval: "+
interval.microSecondsOnly
```

Attribute	Description
microSeconds	Total number of microseconds contained in the interval.

Example:

```
interval as Interval
interval = Interval("20.000003s")
display "total microseconds of an interval: " +
interval.microSeconds
```

Attribute	Description
milliSecondsOnly	Int value representing the milliseconds component of an interval, without including microseconds.

Example:

```
interval as Interval
interval = '20.250320s'
display "milliseconds component of an interval: " +
interval.milliSecondsOnly
```

Attribute	Description
milliSeconds	Int value representing the milliseconds component of an interval plus the microseconds contained in it.

Example:

```
interval = '20.250320s'
display "millisecons component with microseconds: " +
interval.milliSeconds
```


Attribute	Description
minutesOnly	Int value representing the minutes component of an interval.

Examples:

```
interval as Interval
interval = '2h20m10s'
display "minutes component of an interval: " +
interval.minutesOnly
```

Attribute	Description
minutes	Total number of minutes contained in the interval.

Examples:

```
interval as Interval
interval = '2h20m10s'
display "total minutes of an interval: " +
interval.minutes
```

Attribute	Description
monthsOnly	Int value representing the months component of an interval.

Examples:

```
interval as Interval
interval = '1Y2M3d20h'
display "months component of an interval: " +
interval.monthsOnly
```

Attribute	Description
months	Total number of months in an interval.

Examples:

```
interval as Interval
interval = '1Y2M3d20h'
display "total months of an interval: " +
interval.months
```

Attribute	Description
secondsOnly	Int value representing the seconds component of an interval.

Examples:

```
interval as Interval
interval = '1h20m35s'
display "seconds component of an interval: " +
interval.secondsOnly
```

Attribute	Description
seconds	Total number of seconds in an interval.

Examples:

```
interval as Interval
interval = '1h1m5s'
display "total seconds of an interval " +
interval.seconds
```

Attribute	Description
totalMicroSeconds	Int value representing the total number of microseconds contained in the interval.

Examples:

```
interval as Interval
interval = '20.250320s'
display "total microseconds of an interval: " +
interval.totalMicroseconds
```

Attribute	Description
yearsOnly	Int value representing the years component of an interval.

Examples:

```
interval as Interval
interval = '2Y10M15d'
display "years component of an interval: " +
interval.yearsOnly
```

Attribute	Description
years	Total number of years contained in the interval.

Examples:

```
interval as Interval
interval = '25M15d'
display "total years of an interval: " +
interval.years
```

Interval Functions

abs

Returns the absolute value of a given interval.

Arguments

- Interval - The given Interval object.

Returns

The absolute value of a given Interval object.

Example

The following example calculates an Interval by subtracting the current time from the first day of year 2000. Then, it displays the absolute value of the resulting interval:

```
year2k as Time
year2k = '2000-01-01 00:00:00'
interval = year2k - 'now'
display interval
display abs(interval)
```

addDays

Adds a specified number of days to an Interval object.

Arguments

- Interval - The interval to which days will be added.
- Int i - The number of days to be added to the Interval object.

Returns

An Interval object resulting from adding the specified number of days to the given Interval object.

Example

The following example creates an Interval variable named `holidays` and another named `newHolidays`, which is the result of adding 10 days to `holidays`. Then, it displays both the original variable and the one resulting from adding 10 days to the original variable:

```
holidays as Interval
holidays = '15d20h00m'

updatedHolidays as Interval
updatedHolidays = addDays(holidays, i : 10)

display "original holidays: " + holidays +
        "\n\n updated holidays: " +
        updatedHolidays
```

addHours

Adds a specified number of hours to an Interval object.

Arguments

- Interval - The interval to which hours will be added.
- Int i - The number of hours to be added to the Interval object.

Returns

An Interval object resulting from adding the specified number of hours to the given Interval object.

Example

The following example creates an Interval variable named `deliveryTime` and another named `newDeliveryTime`, which is the result of adding 12 hours to `deliveryTime`. Then, it displays both the original variable and the one resulting from adding 12 hours to the original variable:

```
deliveryTime as Interval
newDeliveryTime as Interval

deliveryTime = '1d00h00m'
newDeliveryTime = addHours(deliveryTime, i : 12)

display "original deliveryTime: " + deliveryTime +
        "\n\n new deliveryTime: " + newDeliveryTime
```

addMicroSeconds

Adds a specified number of microseconds to an Interval object.

Arguments

Interval

The interval to which microseconds will be added.

Int

The number of microseconds to be added to the Interval object.

Returns

An Interval object resulting from adding the specified number of microseconds to the given Interval object.

Example

The following example creates an Interval variable named `retry` and another named `largerRetry`, which is the result of adding 222 microseconds to `retry`. Then, it displays both the original variable and the one resulting from adding 222 microseconds to the original variable:

```
retry as Interval
retry = '1m30.600s'

largerRetry as Interval
largerRetry = addMicroSeconds(retry, i : 222)

display "old retry: " + retry + "\n\nnew retry: " +
    largerRetry
```

addMinutes

Adds a specified number of minutes to an Interval object.

Arguments

Interval

The interval to which minutes will be added.

Int

The number of minutes to be added to the Interval object.

Returns

An Interval object resulting from adding the specified number of minutes to the given Interval object.

Example

The following example creates an Interval variable named `breakTime` and another named `newBreakTime`, which is the result of adding 25 minutes to `breakTime`. Then, it displays both the original variable and the one resulting from adding 25 minutes to the original variable:

```
breakTime as Interval
breakTime = '1h20m00s'

newBreakTime as Interval
newBreakTime = addMinutes(breakTime, i : 25)

display "old break-time: " +breakTime +
    "\n\n new break-time: "+ newBreakTime
```

addMonths

Adds a specified number of months to an Interval object.

Arguments

- Interval - The interval to which months will be added.
- Int i - The number of months to be added to the Interval object.

Returns

An Interval object resulting from adding the specified number of months to the given Interval object.

Example

The following example defines an Interval `fishingSeason`. Then, it creates a new Interval named `newFishingSeason`, which is the result of adding a month to `fishingSeason`. Both the original interval and the result of the addition are displayed:

```
fishingSeason as Interval
fishingSeason = '1M20d'

newFishingSeason as Interval
newFishingSeason = addMonths(fishingSeason, i : 1)

display "original fishingSeason: " + fishingSeason +
        "\n\nnew fishingSeason: " + newFishingSeason
```

addYears

Adds a specified number of years to an Interval object.

Arguments

- Interval - The Interval object to which years will be added.
- Int i - The number of years to be added to the Interval object.

Returns

The Interval object resulting from adding the specified number of years to the given Interval.

Example

```
licensePeriod as Interval
licensePeriod = '1Y6M'

newLicensePeriod as Interval
newLicensePeriod = addYears(licensePeriod, i : 1)

display "original license period: " + licensePeriod +
        "\n\nnew license period: " + newLicensePeriod
```

format

Returns a String representation of the given interval. This String representation is based on the current locale.

Arguments

- Interval - The given Interval object.

Returns

A String representation of the given Interval based on the current locale.

Example

```
interval as Interval
interval = '2Y6M15d12h30m30s'
display format(interval)
```

intValue

Returns the Int value of an Interval objects. This value represents the number of seconds in the interval.

Arguments

- Interval - The Interval object.

Returns

An Int representing the number of seconds in the given interval.

Example

The following example displays the Int representation, that is to say, the number of seconds in an interval of 1 hour, 30 minutes, and 20 seconds:

```
display intValue('1h30m20s')
```

max

Returns the greater of two Interval objects.

Arguments

- Interval - The given Interval object.
- Interval b - Another Interval object.

Returns

The greater of the two Interval objects.

Example

The following example displays the greater Interval between 1 hour and 20 minutes and 1 hour and 35 minutes:

```
display max('1h20m', b : '1h35m')
```

min

Returns the smaller of two Interval objects.

Arguments

- Interval - The given Interval object.
- Interval b - Another Interval object.

Returns

The smaller of the two Interval objects.

Example

The following example displays the smaller Interval between 1 hour and 20 minutes and 1 hour and 35 minutes:

```
display min('1h20m', b : '1h35m')
```

sleep

Causes the current BP-method to sleep for the specified number of time. Note that you cannot sleep past the current timeout. While the method is *sleeping* the timeout period is still running, therefore the sleep is applicable for the defined interval or until timeout, whatever happens first.

Arguments

Interval - Time to wait

Example

The following example pauses the execution for 5 seconds:

```
sleep('5s')
```

Array Overview

An array is a collection of values of the same type. Each element of the array is identified with an *index* or a *key*. Any type that can be used to declare a variable can be used to declare an array, even another array.

Types of arrays

Studio supports the following types of arrays:

- [Indexed Arrays](#) on page 255
- [Associative Arrays](#) on page 256

Indexed arrays are indexed by consecutive positive integers, starting from 0 (zero).

Associative arrays may be indexed by any type, although certain types are better suited for use as indexes.

For further information about Arrays, please refer to these sections:

- [Manipulating Arrays](#) on page 257
- [Array Functions](#) on page 258
- [Array Attributes](#) on page 261
- [Array Procedures](#) on page 262
- [Mapping Array Members](#) on page 263

Indexed Arrays

Indexed arrays are arrays that store a set of values in a sequence of positions which are specified by an *index*, which is an integer number. PBL uses zero-based arrays, meaning that the index of the first position of the array is zero rather than one. Zero-based arrays are used in Java and Visual Basic as well.

Declaration

Indexed arrays are declared using square brackets:

```
ages as Int[]
```

The code above declares an indexed array named `ages`, which is of type `Int`.

Initializing an array

You can use in-line arrays for initialization, specifying the values separated by commas:

```
ages as Int[]
ages = [23, 42, 29]
```

The code above initializes the empty array `ages`, with the integer values 23, 42, and 29.

You can add array elements at the end of the array, but must not skip any index values. That is, if the array has six elements, with indexes ranging from 0 to 5, the next assigned value must be with index 6:

```
ages as Int[]
codes = [505, 607, 404, 405, 307, 806]
codes[6] = 306
```

If you skip an index value, an *Index out of bounds* error is thrown.

Accessing elements

The elements of an indexed array can be accessed by the index:

```
ages as Int[]
ages = [23, 42, 29]

display ages[0]
```

If you pass an index which is higher than the last available index (in the example, the last index is 2), an *Array index out of bounds* error results.

Expressions are allowed in the index, so long as they result in an integer within the array bounds:

```
codes as Int[]
codes = [505, 607, 404, 405, 307, 806]
i = 2

display codes[1 + 2]
display codes[i]
display codes[i * 2]
```

Changing Array Values

You can change the value of any element in the array by specifying its index. This example uses a string array:

```
Names as String[]
names = ["Bill", "Ed", "Alfred"]

names[1] = "Edward"
display names
```

Associative Arrays

Associative arrays are arrays that map (or *associate*) a set of keys to a set of values. The data type of the keys need not be an integer, so descriptive strings, for instance, may be used. Keys must be unique, but need not be contiguous, or even ordered.

Declaration

Associative arrays are declared almost the same way as indexed arrays, with the difference that you must specify the data type of the key. For example:

```
ages as Int[String]
```

The code above declares an associative array named `ages`, which is of type `Int` and is indexed by string keys.

Initializing an array

You can use in-line arrays for initialization. They are similar to indexed arrays, but you must specify a key for each key-value pair, since the key is arbitrary:

```
ages as Int[String]
ages = ["John" : 23, "Peter" : 42, "Mary" : 29]
```

The code above initializes the array `ages`, associating the value 23 to the key "John", the value 42 to the key "Peter", and the value 29 to the key "Mary".

You can use keys to add elements to an existing associative array. For example:

```
ages as Int[String]
ages["John"] = 23
ages["Peter"] = 42
ages["Mary"] = 29
```

The code above also initializes the empty array `ages`, and then adds associated key-value pairs. The value 23 is assigned to the key "John", the value 42 to the key "Peter", and the value 29 to the key "Mary". If a key which already exists in the array is used again with a new value, this value *replaces* the old value for that key.

Accessing elements

The elements of an associative array can be accessed by key. The key is specified between square brackets, as an index would be for an indexed array:

```
ages as Int[String]
ages = ["John" : 23, "Peter" : 42, "Mary" : 29]

display ages["John"]
```

If you pass a nonexistent key, a null value is returned.

Ordered arrays

An associative array can be automatically ordered by key. To do so, the array must be declared using the ordered option before the index declaration, as follows:

```
ages as Int[ordered String]
```

The following example will result in an ordered array, even though the array keys are initialized out of order:

```
ages as Int[ordered String]
ages = ["John" : 23, "Peter" : 42, "Mary" : 29]
for key in ages do
  display key
end
```

The keys will be displayed in ascending order, using the key data type sort order, which in this case is alphabetical.

Manipulating Arrays

Change Elements

To change an array element, you can just access it with its list number and assign it a new value:

```
products = ["A", "B", "C"]
products[2] = "D"
```

This changes the value of the element with the list number two (remember, arrays start counting at zero, so the element with the list number two is actually the third element.) As we changed "C" to "D", the array now contains "A", "B" and "D".

Add Elements

Now, suppose that we want to add a new element to the array. We can add a new element in the last position by just assigning a value to the next position. If it does not exist, it is added at the end:

```
products = ["A", "B", "C"]
products[3] = "D"
```

Now, the array has four elements: "A", "B", "C" and "D".

It is not necessary to know the array length to add an element at the end. In order to do it, the add ([]) operator or the extend method can be used.

Example, the following method is equivalent to the previous one:

```
products = ["A", "B", "C"]
extend products // add at the end
```

```
using "D"
```

Delete Elements

Elements can be deleted from an array by using the delete operator:

```
products = ["A", "B", "C"]
delete products[0] // delete first element
```

Now, the array has two elements "B", "C".

Find Elements

The 'in' operator can be used to check if an element is contained in an array. The following code checks whether "A" is contained in the array products:

```
products = ["A", "B", "C"]
if "A" in products then
    display "'products' contains the element 'A'"
end
```

Now, if you want to get the index of the first occurrence of an element:

```
products = ["A", "B", "A", "C"]
index = indexOf(products, "A")

if index != -1 then
    display "'A' is located at position : " + index
end
```

Last examples will show the index 0. Instead, if you want to find the last occurrence, 'lastIndexOf' can be used:

```
products = ["A", "B", "A", "C"]
index = lastIndexOf(products, "A")

if index != -1 then
    display "'A' is located at position : " + index
end
```

Array Functions

avg

Calculates the average value of the data contained in an array. Its behavior is defined for numeric element types only. The return type will be the same as the type of the array.

Arguments	Array array to be averaged
Returns	The average value of all numeric elements of the array. If the array is empty, returns 0 (zero). If the array contains null elements, they are ignored in the calculation. With an array containing only null elements, the function will also return 0 (zero).

The following will display 19.42:

```
array as Decimal [] = [10.49, 13.78, null, 33.99]
display avg(array)
```

count

Counts the number of non-null elements contained in an array.

Arguments	Array array to be counted for non-null elements
Returns	An Integer value with the index. If the element is not found, returns -1. If the array is empty, returns -1.

The following will display 3:

```
array as Int[]=[22,33,null,55]
display count(array)
```

indexOf

This function returns the index of the first occurrence of an element in the array. The array index starts at zero for the first element.

Arguments	Array array to be searched for matching elements Any the element to be searched
Returns	An Integer value with the index. If the element is not found, returns -1. If the array is empty, returns -1.

The following will display 1:

```
array as String[] = ["Hello","!!!","world","!!!"]
display indexOf(array, "!!!")
```

lastIndexOf

This function searches the array for matching elements, and returns the index of the element's last occurrence. The array index starts at zero for the first element.

Arguments	Array array to be searched for matching elements Any the element to be searched
Returns	An Integer value with the index. If the element is not found, returns -1. If the array is empty, returns -1.

The following will display -1:

```
array as String[] = ["Hello","world","!!!"]
display indexOf(array, "happy")
```

The following will return 2:

```
array as String[] = ["Hello","world","!!!"]
display indexOf(array, "!!!")
```

The following will display 3:

```
array as String[] = ["Hello","!!!","world","!!!"]
display lastIndexOf(array, "!!!")
```

length

This function returns the length of the array.

Arguments	Array array from which the length will be obtained
Returns	An Integer value with the number of elements in the array. If the array is empty, returns 0 (zero).

The following will return display 4:

```
array as Int[] = [7,8,9,10]
display length(array)
```

max

This function returns the maximum element of the array. The element type must have a defined sorting order. The return type will be the same as the type of the array.

Arguments	Array array from which the maximum value will be obtained
Returns	The maximum element of the array. If the array is empty, returns a null value.

For example, the following will display "D":

```
array as String[]=["A","B","C","D"]
display max(array)
```

min

This function returns the minimum element of the array. The element type must have a default ordering defined. The return type will be the same as the type of the array.

Arguments	Array array from which the minimum value will be obtained
-----------	---

Returns	The minimum element of the array. If the array is empty, returns a null value.
---------	--

For example, the following will display 22:

```
array as Int[]=[22,33,44,55]
display min(array)
```

sum

Calculates the sum of all the elements of the array. This function is defined for numeric element types only.

Arguments	Array array to be summed
Returns	The sum of all the elements of the array. If the array has no elements, returns 0 (zero).

For example, the following should display 4394:

```
array as Int[]=[112,3233,454,595]
display sum(array)
```

Array Attributes

first

This attribute returns the first element of an array. The datatype of the attribute depends on the data type of the array on which it is applied. For example, the first element of a String array will always be a String.

Returns

- the first element of the array, or null if the array is empty.

Example

```
array as String[]= ["Hello","World","!!!"]
display array.first
```

The expected result is "Hello".

last

This attribute returns the last element of an array. As with first, the datatype of the attribute depends on the data type of the array on which it is applied.

Returns

- the last element of the array, or null if the array is empty.

Example

```
array as String[]= ["Hello","World","!!!"]
display array.last
```

The expected result is "!!!".

Array Procedures

Array procedures operate on a given array. Unlike functions, they do not return a value. Rather, they modify the actual array which is supplied to the procedure.

sort

Sorts the elements of the array. The array's element type must have a defined sort order.

Arguments	Array array to be sorted
Result	The array with the elements sorted in ascending order. If the array contains null elements, an error will be thrown.

The following will display [2, 3, 5, 7]:

```
myArray as Int [] = [7, 3, 5, 2]
sort myArray
display myArray
```

clear

Clears all the elements of the array.

Arguments	Array array to be cleared
Result	The array with no elements.

The following will result in an empty array:

```
myArray as Int [] = [7, 3, 5, 2]
clear myArray
```

insert

Inserts an element in the specified position.

Arguments	Array array in which an element will be inserted
	Integer index after which the element is to be inserted
	Any a value of the same type as the array
Result	The array with no elements.

The following will result in myArray being changed to [2, 9, 7, 3, 4]:

```
myArray as Int[] = [2, 7, 3, 4]
insert myArray
```

```

    using int = 1,
    value = 9
display myArray

```

Mapping Array Members

All of the attributes of an array's element type are mapped to the array. This means that if you have an array of **MyComponent** and **MyComponent** has an attribute called **name** of String type, you can do the following:

```

array as MyComponent[]
names as String[]
// ... initialize array ...
names = array[].name

```

This has the effect of having an attribute called **name** on the array of String type.

The equivalent code without using member mapping is as follows:

```

array as MyComponent[]
names as String[]
// ... initialize array ...
for each component in array do
    names[] = component.name
end

```

The following is an example using Interval:

```

intervals as Interval[]
intervals = ['1d2h', '2d3h', '5d6h']
display intervals[].hours

```

This displays an Int array containing [2, 3, 6].

Variables Overview

Provides a description of the types of variables that can be used in a PBL program.

Variables are locations in memory (and sometimes in a database) in which data can be stored. Each variable has a name, description, type, and value. Most variables only store data of one particular type. For example, if you define a variable to store a number, it would not normally be used to store a text string.

Because data types are commonly fixed, the compiler will check to ensure that you are using correct type for the values your code assigns to them. If a statement is meant to process integers, the compiler will detect when you inadvertently try to use it to process another datatype, such as a string.

Variable Naming

Valid PBL variable names must start with an alphabetic character or an underscore character followed by zero or more alphanumeric characters. The following are valid variable names:

```

participantName = "John Doe"
iso9001 = "..."
_ugly_var_name_ = 1

```

The following PBL variable names are not valid:

```

//Variables cannot begin with numbers
4ever = true

```

```
//display is a reserved word
display = true
```

Reserved Words

Certain names cannot be used as variable names in a PBL program. These are called reserved words. Reserved words are dependant on the current language skin. To use a reserved word as a variable name, you must escape it with the '@' character.

```
// This is legal because the reserved word
// has been escaped
@display = true

// References to it must also include
// the escape character
display = @display
```

In this example, the '@' sign is not part of the variable name. It is just a way to tell the compiler that you are not referring to the reserved word but that you want to refer (in this case) to the variable.

Variable Declaration

The exact variable declaration syntax within a PBL program is dependant on the skin you are using. The following examples demonstrate how to declare variables using different programming styles:

Using the PBL Style:

```
name as String
temp as Any
```

Using the Visual Basic style:

```
Dim name as String
Dim temp as Any
```

Using the Java style:

```
String name;
Any temp;
```

Variable Scope

The term *scope* is used to denote the applicability or availability of a variable within your project. For example, variables defined within the code of a method have a local scope and are therefore called *local variables*. This means that they can only be seen from within the method where they are declared.

ALBPM variables may have one of the following scopes:

- Local
- Instance
- Project
-

Instance Variables

Instance variables are variables attached to an instance. They contain information that refers to the instance. Instance variables can be accessed from the activities as the instance flows through a process from the Begin activity to (and including) the End activity.

All activities, except Global Creation, Global and Global Automatic can access instance variables. Examples of instance variables that may be found in a shipping order management process include: `invoiceNumber`, `customerName`, `customerNumber`, `orderStatus`, `orderAmt` and `shipStatus`.

Instance variables are called in any BP-Method by the variable's name. If there is any possibility for a naming conflict, for example, you have a SQL component and the table has a field with the same name as the instance variable, then you should refer to the instance variable as `this.instancevariablename` within the BP-method.

Instance variables have a special property called *Category*, which can be set to one of the following:

1. Normal
2. Separated
3. External

Normal Instance Variables

Most of the Instance variables are defined as *normal*. Only those that have some special characteristics need to be categorized in a different way.



Note: The total storage size of all the normal instance variables cannot exceed 2MB.

Separated Instance Variables

Instance variables used to store a large amount of data (10 KB or more) need to be categorized as *separated*.

The BPM system separately stores these variables on the database to avoid performance problems. Large BPM Objects or serialized Java objects are examples of instance variables defined as separated.



Note: The size of a separated instance variable is limited to 2MB.



Note: In Split-N activities, *separated instance variables* values are not automatically copied to the separated instance variables of the copies. They have to be copied manually in the Split-N BP-Method.

External Instance Variables

Sometimes instance variables contain information about a process instances which needs to be shared across other processes or be used as a query filter from Workspace. In order to expose a variable in this way, it can be set as *project*. When a variable is defined as external, it becomes available as an *project variable* across the project.

For example, the instance variable `CustomerName` used in a process within a project also needs to be used by all other processes across the project. Therefore, define it as external and, in each process that needs to use it, declare the same variable (with the same name and type). All variables defined in this way will be references to the same external variable.

External variables are also available for display in Workspace, as they are saved within the Studio database.

From Workspace, you can search for instances by setting filtering conditions on the external variable.

Variable Initialization

- Variables defined as a numeric type (integer, real, decimal, etc) have an initial default value of 0 (zero).
- Boolean variables have the initial default value set to False.
- String variables are initialized as an empty string (" ").
- Time variables are initialized as 'now'.
- Interval variables are initialized as '0s'.
- BPM Objects are initialized if they have a constructor method which does not require parameters.
- BPM Object attributes are initialized according to the attribute properties specified at design time, so they may be set to null or to a default value.

Using BPM Objects as Instance Variables

See How to use a BPM Object as an instance variable for more information.

SQL Components as Instance Variables

See SQL Components as Instance variables for more information.

Project Variables

Explains how and why to use project variables

What are Project Variables?

Project variables are a special kind of instance variable which has more visibility than a normal instance variable. Project variable values can be searched for, and can be displayed in AquaLogic BPM WorkSpace views.



Note: Project variables were formerly called External variables.

Project variables are used to save information across processes. For example, in ALBPM WorkSpace, you can search for all processes for any instances corresponding to an invoice number.

You can add a new project variable or define an instance variable as a project variable.

A given project variable can be made visible to any process within a Project. To do this, define an instance variable of the same name and type as the project variable in each process which will use it. Then, define a project variable with the same name, and all the instance variables will refer to this project variable.

Once the instance variable has been defined as a project variable, the project variable appears in the **Project** section of the **Variables** view.

Any change on that variable has to be done on the Project variable directly and all instance variables that reference to it are automatically updated.



Important: Project Variables are special variables and should be defined thoroughly during the development phase. Once a project variable has been defined and used during user testing or production, the recommendation is *not to change* its type nor use its name if it has previously been used for one purpose and later on you want to save a different data type.



Restriction: Each project variable requires an additional column in the instance table of the Process Execution Engine database, which means it incurs a performance cost. While there is no set limit, as a rule of thumb fewer than ten project variables are recommended, and in most cases a well-designed project should not require more than about five or six.

Working with Project Variables

Project variables are added from Studio. For detailed instructions see [Creating Project and Instance Variables](#) on page 99.

Once the variable has been created you can:

- Convert the variable to a Business variable
- Use the Project variable in the process: an instance variable is created in that process defined as project.



Note: Project variable names may have up to 16 characters.

Argument Variables

Argument variables act as an interface. You can transform instance variables with argument variables so that they may be passed to or from external components or between processes. See Argument Mapping for further information on passing information between processes and their calling/called or notifying/notified activities.

Argument variables can be defined as:

- In
- Out
- In/Out

For each activity, the valid options depend on how arguments can be treated in each case. Therefore, arguments can be in each activity as follows:

Activity	Type
<ul style="list-style-type: none"> • Interactive • Grab • Automatic • Global • Global Creation 	In/Out
<ul style="list-style-type: none"> • Subflow 	Out for the create BP-method In for the wait BP-method.
<ul style="list-style-type: none"> • Process creation • End • Process Notification 	Out
<ul style="list-style-type: none"> • Termination wait, Begin, Join, Notification Wait 	In
<ul style="list-style-type: none"> • Split • Split-N • Conditional 	Not Available

Argument variables can be considered as *external* or as *internal* based on what they were defined for.

External Argument Variables

External argument variables are those defined in the activities (Begin, End and Notification Wait) that can deal with external components of the process. These external components could be an HTML form, a Java program, another process, Web Services, and so on. External Argument variables are the interface to a process. They are responsible for passing variables from a process to an external component and receiving variables from external components into the process. Arguments can be grouped in argument sets.

Two BPM processes that need to communicate with each other use external argument variables to pass the variables between the processes. One process, the parent process, passes variables through a Subflow activity to another process (a subprocess or child process). The child process's Begin activity receives these argument variables and maps them to instance variables. When the child process has finished processing the instance, it passes the instance variables to the End activity, which then maps the instance variables back to argument variables to pass back to the Subflow activity in the parent process.

Internal Argument Variables

The second type of argument variable is internal to a process. It is available for BP-Methods used within the process (for example, a BP-Method that receives arguments and can be invoked from another BP-Method assigned to an activity task).

Using Argument Variables

Fully qualified argument variable names are written: `arg.myArgument`

The following sets an argument variable equal to an instance variable:

```
arg.myArgument = myInstance
```

The following sets an instance variable equal to an argument variable as follows:

```
myInstance = arg.myArgument
```



Note: The usage of the `arg` keyword is optional. However, it is required when you need to differentiate from a local or instance variable that has the same name as an argument variable.

For example, suppose that there is a method that contains a local variable called `orderNo`. The method also receives an argument called `orderNo`, and the process has an instance variable with the same name as well. In order to identify the proper variable, you will need to qualify it.

To refer to the instance variable you must type:

```
this.orderNo
```

To refer to the argument variable you must type:

```
arg.orderNo
```

Unqualified variables are assumed to be local, so they do not require qualification.

The Copy Argument Variable

There is a predefined argument called `copy` which is only available when writing a Join activity BP-Method. The `copy` argument represents the instance-copy that arrived at the `Join` activity.

In the corresponding Split of the Split-Join circuit, if copies are created, they can be processed in the Join activity upon arrival. The way to reference copies is by using the `copy` argument keyword.

In other words, `copy` is used to access the values of the variables of the copies of an instance in the Join activity.

For example:

```
// Set the original instance variable with
// the copy instance variable.
orderTotal = copy.orderTotal
```

See Split-Join activity for further information.

Local Variables

You define and use *local variables* inside a single process method. The scope of this type of variable is the method itself, which means that it cannot be seen from outside. Once method execution ends, the value stored in a local variable is lost.

Creating a Local Variable

Local variables can be created by performing the following actions:

- Adding a local variable in the Studio Variable tab and clicking on + in the Local variables section,

- Declaring it from the Fix option.

Example:

```
" var = 5 "
```

var is not a variable yet, so the compiler gets an error.


By right-clicking on the error, a **ix** option appears showing a possible solution for the problem.


Predefined Variables

Predefined variables are global to all parts of the process and, just as their name implies, have already been defined. Most of the predefined variables deal with instances as they relate to activities and are used to keep track of an instance and its status as it flows through a process. Some predefined variables are modifiable and others are not. The following table lists the predefined variables included in Studio.

List of Studio Predefined Variables

Predefined Variable	Permitted Values	Type	Description
action	Modifiable : OK ; FAIL ; RELEASE ; CANCEL ; REPEAT ; ABORT ; BACK ; SKIP ; NONE	Fuego.Lib.Action	Marks the action to be performed on a process instance as a result of previous BP-Method statements. See below <i>Using the action variable</i> .
activity	Read only	Fuego.Lib.Activity	Returns an object that is the current activity.
activity.deadline	Read only	Time	Set automatically if there is a Due transition going from the current activity (receptionTime + the due transition time interval.)
activity.source	Read only	Fuego.Lib.Activity	The activity from which the current instance came into the current activity. Null in the case of Global and Begin activities.
attachments	Read only	Fuego.Lib.Attachment[]	Array of the instance attachments.
children	Read only	String[ordered Object]	Array of the Ids of the instances that are children of this instance, ordered by the list of activities that created them.
creation. participant	Read only.	Participant	Participant that created the process instance.
creation.time	Read only	Time	Creation time of the instance.
currentException	Read only	String	Name of the exception, if the instance is within an exception flow, which was given to it by the throw statement.
deadline	Modifiable.	Time	The instance will expire if it is not completed before the specified time.

Predefined Variable	Permitted Values	Type	Description
description	Modifiable: Any string that does not contain special characters	String	Name of the instance that appears in WorkSpace. By default, it is ProcessName+id.number.
id.id	Read only	String	An instance's identifier, which uniquely identifies an instance. It includes the deployed process name, organization, organizational unit and the instance Id (including thread id).
id.number	Read only	Int	A number that uniquely identifies an instance within an Engine.
id.copy	Read Only	.	The current instance thread number.
notes	Read only	Int	An array of all notes added to an instance.
organization	Read Only	String	Name of organization where the process is running.
organiztionalUnit	Read Only	String	Name of the process' organizational unit where the process is running, such as Marketing or Finance.
parent.id	Read only	String	The Id of the parent instance of the current instance if there is one; NULL otherwise. In the case of a Procedure, it contains the id of the calling process
parent.copy	Read only	Int	The parent instance thread number.
parent.number	Read only	Int	A number that uniquely identifies the parent instance within an Engine.
participant	Read Only. Any participant	Fuego.Lib.Participant	The human participant stored in directory services that is currently processing the instance.
participant.locale	Read Only	Fuego.Util.Locale	The language, country and variant, if applicable, the participant has set.
participant.next	Modifiable: Any participant	Fuego.Lib.Participant	The participant to which the instance will be sent next.
 Note: If the instance is grabbed, the <code>participant.next</code> is cleared. If the instance is sent BACK or SKIP from an Exception flow or from an Interruption, the only way to reset the <code>participant.next</code> is using the Unselect method from the Participant			

Predefined Variable	Permitted Values	Type	Description
			component. See Participant component documentation for further information.
participant.sticky	Modifiable: Bool	Fuego.Lib.Participant	<p>If set to true, the participant.next is set as the preferred participant. Each time the instance moves to an activity and the participant.next is enabled to work with it (belongs to the role where the activity is), the instance is submitted to the participant.next work queue. In the Split and SplitN activities, all generated copies maintain the participant.sticky value from the original instance. In the SplitN method you can change its value.</p> <p> Note: If the participant.next is changed during the process, be sure that participant.sticky is reset unless the new participant.next is considered the preferred participant.</p>
priority	Modifiable: 1 Lowest ; 2 Low ; 3 Normal ; 4 High ; 5 Highest	Int	Priority of the instance.
process	Read Only. Any process	Fuego.Lib.Process	The process the instance belongs to.
process.id	Read Only	String	The Identifier of the deployed process containing the deployed process' name, its organization and organizational unit.
process.idNumber	Read Only	Int	A number that identifies the process inside an Engine
process.name	Read only	String	The name of the process of the instance.
receptionTime	Read only	Time	The time that the current instance was received at the current activity.
result	Modifiable: Any string	String	It contains the result that you set in the BP-Method. For example, you can set a result and in the outgoing conditional transition of the activity ask for that value. See below for more information.
status	Read Only. RUNNING, EXCEPTION, SUSPENDED,	ProcessInstanceState	Current status of the process instance.

Predefined Variable	Permitted Values	Type	Description
	GRABBED, COMPLETED, ABORTED, ACTIVITY_ COMPLETED		
timeout	Modifiable: String indicating an interval, enclosed in single quotes: '5m'	Interval	The length of time that a BP-Method or an external component has to complete before the engine cancels its execution. By default this variable is set to 5 minutes.
totalCopies	Read only	Int	Number of threads or copies of an instance.

Using the Action Variable

When a BP-Method is executed, multiple types of output can result from the execution. BP-Method execution status is indicated by the value of the predefined variable action.

Depending on the value of action, the engine responds accordingly and saves or undoes (commit or rollback) the changes invoked when the BP-Method is executed.

Valid values for the action variable

The valid values for the **action** variable are listed in the following sections.

Action.OK or Action.NONE

- Indicates that the BP-Method execution was successful.
- The default value for action variable. Therefore, it is not mandatory to set a value to the action variable.

Action.FAIL

- The way to indicate that the BP-Method has failed. The instance data is reversed and, if there is a rollback BP-Method, it is executed.
- Can be used in *any kind of BP-Method, and any kind of activity*.
- If a rollback BP-Method is included for Interactive activities, it is executed. If an Automatic activity fails, the engine retries the BP-Method until it succeeds or invokes the maximum number of retry times and it is routed to an exception handling activity. Maximum number of retry times is set in the Engine Properties option on the **Run** menu.



Note: A component exception is treated like `action=FAIL` by the engine. NO error is logged in the engine log. If you want to log a message, you will have to use the `logMessage` statement inside the Method before the failure.

Action.CANCEL

- The way to cancel the BP-Method. The instance data is reversed but the rollback BP-Method, if it exists, will not be executed.
- Can be used in activities of **Interactive, Grab or Global Creation** type.
- If the activity type is any other (i.e., **Automatic**, etc.), this value is ignored, as if it had never been set.
- No trace of the BP-Method failure or execution appears in the Audit Trail.

Action.RELEASE


- Ends the BP-Method execution successfully.
- If the activity type is **Interactive, Grab or Automatic** the instance is released to the next activity without processing any of the BP-Methods in the current activity, even if they are marked Mandatory.
- If the activity type is **Join**, the original instance will be released and all the active copies will be aborted.
- If the activity type is any other (i.e., **Global**, etc.), this value is ignored, as if it had never been set.


Action.REPEAT

- If the activity type is *Interactive* or *Grab*, REPEAT indicates that, although the task was successfully executed, it will remain pending. Therefore, the participant will be able to execute it again. Note that if the task is *Mandatory*, the participant will have to execute it repeatedly until the task appears as completed.
- If the activity type is any other (i.e., **Global**, **Automatic**, etc.), this value is ignored, as if it had never been set.

Action.ABORT


- Aborts the instance. The instance is sent to the End activity and marked as aborted.
- Used in activities of **Interactive**, **Grab** or **Automatic** type.
- Used in activities of **Join** type. The original instance will be aborted and therefore, all copies will be aborted.
- If the activity type is any other (i.e., **Global**, etc.), this value is ignored, as if it had never been set.

 **Note:** Instances that are aborted cannot be recovered.

 **Note:** The transaction is committed. Therefore, any change performed in the BP-method is persisted.


Action.BACK

- Used in an activity in an exception handling flow to send an instance back to the activity where the exception occurred. Normally, the exception flow is used to fix the exception and, after fixing, sends the instance back.
- Sends the instance back to the activity where the exception occurred.
- Used in activities of *Interactive*, *Automatic* and *Grab* type, when they are in an exception handling flow.
- If the activity type is any other (i.e., *Global*, etc.) or the activity is not an exceptions handler, this value is ignored, as if it had never been set.
- Within procedures or atomic groups, the instance goes back to the *Begin* activity of the procedure or the group.

 **Note:** The transaction is committed. Therefore, any change performed in the BP-method is persisted.

Action.SKIP

- Used in activities belonging to an exception handling flow to send an instance back to an activity in the main flow of the process. The instance goes to the point of BP-Method failure and is immediately released to the next activity without re-performing the BP-Method that caused the failure.
- Sends the instance to the activity immediately AFTER the one where an exception occurred.
- Used in activities of *Interactive* or *Automatic* type, when they are exceptions handlers.
- If the activity type is any other (i.e., *Grab*, *Global*, etc.) or the activity was not an exceptions handler, this value will be ignored as if it had never been set.
- Within procedures and atomic groups, the instance goes to the *next activity* that follows within the procedure or group.

 **Note:** The transaction is committed. Therefore, any change performed in the BP-method is persisted.

Example: action variable

The following BP-Method example shows how you can manually set the **action** variable if a Boolean expression evaluates to true:

```
if selectedButton == "Yes" then
  action = OK

elseif selectedButton == "Abort" then
  action = ABORT

else
  action = BACK
```

Using the result variable


The **result** variable contains the result that you set in the BP-Method. For example, you can set a result and in the outgoing conditional transition of the activity ask for that value. Further in the process this value will no longer be available. The result variable is persistent between calls to the BP-Methods and is reset when the instance has already flown through the corresponding transition to another activity.

As well if the action variable is not used in the BP-Method, the predefined variable result values are mapped to the variable action. For example, if the BP-Method has the line `result = "fail"`, this is equivalent to `action = Action.FAIL`.

If the BP-Method uses the action variable, then the result can have any value, either predefined or not. For example:

```
result = "release"; action = Action.FAIL';
//in this case the BP-Method fails
result = "abort"; action = RELEASE';
// in this case the BP-Method does not fail
```

The action variable is reset at every BP-Method invocation to the default value while the result variable is reset after the instance flows into the next activity.

 **Note:** The predefined variable result is used to maintain compatibility to previous version.

Using the "timeout" variable

The timeout variable defines the time that a BP-Method has to complete its execution. If this time is exceeded, the Engine cancels the BP-Method execution and assumes that it has failed.

There is a maximum timeout defined for all BP-Methods of all processes within the engine. This value is defined in the Process Administrator or Studio Engine Preferences.

The timeout set within a BP-Method (timeout variable) cannot exceed the Maximum timeout set in the Process Administrator. If you set a greater value, a runtime exception is thrown and the BP-Method execution is aborted.

See Timeout for further information.

Examples

Case01: Suspend Action

Find the project **ActionsCase01.fpr** in the installation directory, in the studio/samples directory to study an example about handling the SUSPEND action and assigning the participant who suspended the instance.

To test this example, use the participant: **test**.

Initializing Variables

Default Values

In Studio, all variables are automatically initialized when first used if there is a suitable default value for the variable's type. The following table summarizes the default values that are used for each type:

Variable Type	Default Value
Numeric	0
String	""
Any	null
Time	'now'
Interval	'0s'

Initialization

Variables can be initialized during declaration as in the following example:

```
name as String = "Hello"
```

Variables can also be initialized after declaration by using an Assignment Statement as in the following example:

```
name as String
name = "Hello"
```

Operator Types Overview

An operator is a symbol that performs a function on one, two or three operands. An operator that requires one operand is called a *unary* operator. If they require two or three operands, they are called *binary* or *ternary* operators.

The following types of operators are provided in the Process Business Language:

- Arithmetic: Performs mathematical operations.
- Relational: Compares two or more values.
- Logical: Performs logical operations.

Arithmetic Operators

The following table lists the operators that can be used in a method to perform mathematical calculations:

PBL Style	Java Style	Visual Basic Style	Description
+	+	+	Addition
-	-	-	Subtraction
*	*	*	Multiplication
/	/	/	Division
rem	%	Mod	Modulo (Remainder)

Order of Precedence

When multiple operators are used within an expression, the following order of precedence is followed:

- operators within parentheses are evaluated first
- multiplication, division, and modulo are evaluated from left to right
- addition and subtraction are evaluated from left to right

Using Variables with Arithmetic Operators

Variables of Int, Decimal, Real, Time, and Interval types may be used as operands of arithmetic operators. The result of a given expression is based on the following rules:

- If any of the operands is Real, the result is Real
- If any of the operands is Decimal and none are Real, the result is Decimal with sufficient places to hold the operation's sum, difference, multiplication, or remainder
- If both operands are Integer, the result is Integer
- Monadic '-' (change of sign) and '+' identity are supported
- The result always has at least the precision of the operand with the highest precision in the expression

Relational Operators

Relational operations yield a Boolean result. Relational operators must be applied to homogenous variable types. For example, comparing an integer to another integer passes the check. However, comparing an integer to a Boolean results in an error. Any variable of a numeric type is considered homogenous.

Logical Operators

Logical operators can be used on or between Boolean variable types to obtain a boolean result.

The following table lists the operators that are available in BPL:

Operator	Meaning	Order of Operation	Evaluation Criteria
not	Logical not	5	Negates the operand value so if the value is true, it becomes false and vice versa.
and	Logical and	6	Yields true if and only if both operands are true. Yields false if any of the operands are false. Uses evaluation by need, so if the first operand is false, the evaluation process is stopped and the result is false.
or	Logical or	7	Yields true if either operand is true. Yields false if and only if both operands are false. Uses evaluation by need, so if the first operand is true, the evaluation process is stopped and the result is true.

NOT Example:

```
if not found then
    // do something
end
```

AND Example:

```
if orderAmt < 200 and paymentType = "Credit" then
    // do something
end
```

OR Example:

```
if shipStatus = "Received" or shipStatus = "Pending") then
    // do something
end
```

Statements Overview

There are other statements, such as assignment statements, which allow you to set values to variables, and interactive statements, which allow you to interact with the current participant.

Control Statements

Control statements are used to control the sequence of statement executions.

Top-down, sequential execution is the simplest sequence of method execution. The method starts to execute on the first line of the code, goes to the next and continues until the final statement in the code has been executed. This works fine for very simple tasks but tends to lack real-world usefulness since such a method can only handle one situation. Most programs need to be able to decide what to do in response to changing circumstances. By controlling the execution sequence according to different situations, a specific piece of code can then be used to handle more than one situation.

Statement Timeout

To protect the Process Execution Engine against Method tasks that are not behaving as expected (such as Method with an infinite loop) and remote components, every Method has a timeout property that controls the maximum duration of Method execution. By default, this property is set to a five-minute ('5m') interval, which starts counting from the moment the Method begins to execute.

```
i = 1
while i > 0
  // ... code here
  i = i + 1
end
```

If you run the Method as displayed above without some protection mechanism, the Method would run forever (or until the engine is shut down). Besides locking the engine, infinite Method tasks also lock the instance so that no other user is able to process it. To keep locking from occurring, the timeout property invokes and ends the Method in five minutes.

The timeout property is checked when the following conditions occur:

- Method enters a loop or iteration.
- A remote component is invoked.

Changing Method Timeouts

To change a Method timeout:

```
// Set the maximum time the Method can run

timeout = '10m'

// Execute a loop or iteration for 10 minutes at most
for each i in myArray do
  // statements
end
```

If the Method task is in an *Interactive* activity, consider using *relay to*. The *relay to* statement automatically ends Method execution for the activity when you expect that a user will take a certain amount of time to finish the execution. When the user finally responds, the response is routed to an alternate method designated in the *relay to* statement.

Relay to statements



Note: From this version onwards, consider using Screenflows instead of "relay to" invocations since Screenflows have all the benefits of relay to invocations with the simplicity of process-like design.

Controlling long running statements with relay to

When developing a Method, consider the impact the code will have on your process. For example, code in interactive activities often requires some kind of response from a human end user.

By nature, humans are unpredictable. A human end user may start his or her task in Workspace but then something may cause him or her to forget to complete the task until a later time.

Uncompleted tasks lock resources in the BPM Engine. Locked resources not only lock the method, but also decrease the scalability of the engine, which means that the engine cannot accommodate as many users as it can do under normal circumstances. To ensure that resources do not lock while waiting for end user input, you can use the relay to in your code.

When you use relay to in an interactive method, the engine immediately ends the execution of the method and does not wait for end user input. This frees the resources to handle new instances. When the end user finally enters his or her input, the engine routes the instance with the end user input to the method designated by relay to.

Relay to example

In the following example, a simple input is requested. The relay to statement is then invoked.

```
input "Enter something in the box here: " name
  using title = "Relay To Example"
  relay to CilReachedFromRelayTo
    using relayToName = name
```

The method in the CilReachedFromRelayTo task is as follows:

```
// This is the standalone Method reached by an input
// statement in some Method in the process.
// Usually, the only Method you see in it is setting
// instance variable(s) from the Method's incoming
// argument variable(s).

name = arg.relayToName

display "This is the standalone Method reached from another
  Method's input statement with a \"relay to\".
  You entered: " + name
```

For further information, refer to Using Relay To.

Input Statement

Input statements allow you to invite the current participant to enter information that is required by the method's logic.

The following types are available:

- *Basic input*, which builds a simple form for entering data.
- *Interceptor input*, which intercepts one or more web pages fetching data from the different form fields.
- *BPM Object input*, which displays a presentation of a BPM Object.

Basic input and BPM Object input can be implemented with screenflows. This is strongly recommended. See Screenflow's documentation for detailed information.



Note: If you use the input statement in a BP-Method in a task, when the design is checked, a warning is thrown. It is recommended to use a screenflow or the relay to option, as the input statement is not applicable if you run the process in an EJB based Engine.

Syntax

Basic input

```
input "field label" basicReference
  [({option}[={value}], ...)]
```

```

        [ in [{valid values}] ] [, ...]
    [using
        title = "{title}",
        buttons = [{button labels}],
        cancelButton = "{button label}"
    ]
    [returning
        {selected button} = selection
    ]

```

Interceptor input

```

input "field label" basicReference
    [ in [{valid values}] ] [, ...]
    [using
        title = "{title}",
        buttons = [{button labels}],
        htmlForm = "{initial URL}",
        until = "{stop condition}",
        links = "{ intercept | popup | clear }",
        userControl = { true | false },
        cookies = [ { cookies } ]
    ]
    [returning
        {selected button} = selection,
        {cookie map} = cookies
    ]

```

The main functionality of the Web Interceptor can be simplified in these 3 topics:

- Interact with existing Web Applications.
- Allow BP-Method access forms in HTML, JSP, ASP.
- Support any form control as Lists, Text Areas, Radio Buttons, Fields.

How does the Web Interceptor work?

Web Interceptor basically works over HTTP(S), FTP and FILE resources.

The referenced URL (**htmlForm**) can be:

- http://...
- https://...
- ftp://...
- file://...

BPM Object input

```

input basicReference
    [using
        selectedPresentation = "{presentation name}"
    ]
    [returning
        {selected button} = selection
    ]

```

Arguments

Attribute: title

Type: String

Description: The input form's title.

Attribute: buttons

Type: String[]

Description: An array containing the labels of the buttons you want to be displayed on the form.

Attribute: cancelButton

Type: String

Description: button that acts ignoring all changes and avoiding the input of required fields.

Attribute: htmlForm

Type: String

Description: Full or relative URL to the initial page to be intercepted.

For relative files as **.html**, **.jsp** or **.asp** to be intercepted have to be copied into the installation directory, in studio/webapps/portal/**dirname**.

The **htmlForm** attribute is composed by: **http://host:port/projectname/dirname/file.[html/jsp/asp]**", where:

host:port is the host and port in which you run the Workspace.

projectname is the name of your project,

dirname is the name of the directory you created in **studio/webapps/portal/**

For example:

htmlForm = "**http://localhost:9595/InterceptorCase01/TestInterceptor/classRegister.html**",

InterceptorCase01 is the name of the project,

TestInterceptor is the directory in studio/webapps/portal, and

classRegsiter.html is the html page to be intercepted.

Attribute: until

Type: String

Description: Stop condition that indicates the interceptor when to stop intercepting pages. It is of the form: **pattern@location**

Where **location** is a full or partial url that marks the end of interceptions and **pattern** is a string that is sought for when **location** is reached. (pattern@URL or pattern@file name)

If not found, interception continues.

Some examples for stop conditions are:

- post.asp?id=query
- http://mysite/Poster/post.asp
- Congratulations@post.asp
- Congratulations@http://mysite/Poster/post.asp

Attribute: links

Type: String

Description: This parameter determines what happens when the intercepted page contains links and the user clicks on one. It takes one of the following values:

- popup: the content of the link will be displayed in a new window.
- clear: the link will be removed.
- intercept: the link will be intercepted and displayed in the same window.

It will continue to intercept until the specified criteria has been met or the user hits "Stop" in the navigational control (see: `userControl` attribute.)

Attribute: `userControl`

Type: `Bool`

Description: If true, the navigational control will be displayed in the intercepted page.

One of the purposes of using Web interceptor is to set and get information from the pages you are intercepting or navigating. This basically implies that you are able to go through a sequence of HTML, JSP or ASP pages intercepting and collecting information that will later be used in the BP-Method logic. The navigational tool allows the end user go back and forward through the pages until the stop criteria is reached.

In order to show the navigational tool in the intercepted pages you need to set “`userControl`” to true. If you set true to **links**, then **`userControl`** takes the value true automatically and the navigational toolbar is displayed.

The buttons showed in the navigational panel are:

- Go Back : Goes to the previous intercepted page.
- Finish Interceptor : Stops the interception and returns the control to the BP-Method.
- First page: It starts again the interception from the first page.

Attribute: cookies

Type: `java.Object[java.Object]`

Description: Associative array that contains a collection of cookies used during interception. The keys and values are of String type.

Attribute: selection

Type: `String`

Description: Returns the label of the button that caused the form to be dismissed.

Attribute: `selectedPresentation`

Type: `String`

Description: The name of the desired presentation for the Object, if this attribute is not specified, the default presentation will be used

Field options

The following table contains a list of the **options** that can be passed to an input field:

Option	Required Type	Description
date	Time	displays a Time as date-only.
time	Time	displays a Time as time-only.
readonly	Any	the field is displayed, but cannot be modified.
password	String	the field is displayed as a password field.
required	Any	The field cannot be null.
textarea	String	displays an area to enter a text.

Remarks

For the BPM Object input, the `selectedPresentation` attribute is only valid if the `basicReference` is a BPM Object. If it is not, a field label will be synthesized and the input will behave as a basic input.

If the `selectedPresentation` attribute is missing, the default presentation of the BPM Object will be displayed.

When you specify a partial URL in any of the fields that take one, the URL is relative to the portal in which the input statement is displayed.

For an Interceptor input, the field name must match the name of a field in the form being intercepted. If it does not match, the variable will be left empty.

Input Examples

Basic Input

```
creditCardNo = ""
acceptedTypes = ["visa" : "Visa", "master" : "Mastercard",
  "amex" : "American Express"]
creditCardType = "visa"
firstName = ""
lastName = ""
expiration = 'now'
comments = ""
input "First Name:" firstName (required),
  "Last Name:" lastName (required),
  "Credit card type:" creditCardType (required)
    in acceptedTypes,
  "Credit card No.:" creditCardNo,
  "Expiration Date:" expiration,
  "Additional Comments:" comments (textarea)
using
  title = "Enter payment info",
  buttons = ["Ok", "Cancel"]
```

Interceptor Input

```
googleQuery = ""
input "q" googleQuery
  using
    htmlForm = "http://www.google.com",
    links = "clear"
```

Refer to Web Interceptor for more examples.

BPM Object Input

```
//Order is a BPM Object with a presentation called
//'auditView'
input order
  using
    selectedPresentation="auditView"
```

Display Statement

The display statement, as its name implies, allows you to display information to the user and to get feedback based on the choice of buttons the user selected.

The **Display** can be implemented with screenflows. This is strongly recommended. See Screenflow's documentation for detailed information.



Note: If you use the **display** statement in a BP-Method in a task, when the design is checked, a warning is thrown. It is recommended to use a screenflow, as the display statement is not applicable if you run the process in an EJB based Engine.

Syntax

Basic Display

```
display {object}
```

```
[ using
  [title = "{title}", ]
  [type = "{error | question | warning | info | plain}",]
  [options = {options}, ]
  [default = {default button}]
]
[ returning {selected button} = selection ]
```

BPM Object Display

```
display {fuego object}
  [ using selectedPresentation = "{presentation name" ]
```

This form of display shows a BPM Object presentation as read-only.

Arguments

The following is the list of the arguments that can be passed to a display statement:

Argument: title

Type: String

Description: Title of the display window/page.

Argument: type

Type: String

Description: Kind of display. The icon will be chosen based on this argument. The default value is "plain".

Argument: otions

Type: String[]

Description: Array of strings containing the labels of the buttons you want to display.

Argument: default

Type: String

Description: Which of the buttons is returned in case the display is canceled.

Argument: selectedPresentation

Type: String

Description: Name of the presentation used to display a BPM Object. If left unspecified, the default presentation will be used.

Display Examples

Basic Display

```
selectedButton as String
display "Should we try again?"
  using title = "Confirm",
    type = "question",
    buttons = ["Yes", "No"],
    default = "No"
  returning selectedButton = selection

if selectedButton = "Yes" then
  //Retry
end
```

BPM Object Display

```
//Order is a BPM Object with a presentation
//called 'auditView'
input order
  using
    selectedPresentation="auditView"
```

Compound Statement

A *compound statement* groups other statements in a logical unit. It defines a scope and allows you to handle exceptions or execute statements that must always be executed, regardless of the outcome of the block.

Syntax

The most basic form of a compound statement simply encloses some statements together. This is also called a code block:

```
do
  //Your statements here
end
```

You can also handle exceptions that are thrown inside the block:

```
do
  //Your statements here
on excep as Java.Exception
  //Handle Exception here
end
```

You can add some code to be executed when your block finishes, regardless of whether it has finished by exception or not:

```
do
  //Your statements here
on exit
  //Do something that must be done always, such as
  //releasing external resources
end
```

Exception and end of block handling can be combined. For example:

```
do
  //Your statements here
on ex as Java.Exception
  //Handle Exception here
on exit
  //Do something that must always be done, such as
  //releasing external resources
end
```

About Loops and Methods

Loops and methods define special-purpose blocks, and these can also be used to handle exceptions.

Loops

Loops are used to execute a given code block several times, usually with one or more variables which hold different values in each iteration of the loop. These values may be updated by the code block within the loop, or as a result of the loop definition itself.

For example, the following loop will iterate three times, where the `name` variable will successively adopt the values of "John", "Peter", and "Mary":

```
names as String[]
names = ["John", "Peter", "Mary"]
for each name in names do
    // Do something
on ex as Java.Exception
    // Handle your exceptions
end
```

which is equivalent to:

```
names as String[]
names = ["John", "Peter", "Mary"]
do
    for each name in names do
        // Do something
    end
on ex as Java.Exception
    // Handle your exceptions
end
```

Methods

Methods define a compound statement that contains the entire body of the method. Each method is contained by an implicit `do/end` block. For example, the following statement:

```
on Exception
```

is semantically equivalent to:

```
do
on Exception
end
```

This block is labeled after the method's name, so you can add exit and exception handlers directly in the method as in the following:

```
...
...
on ex as Java.Exception
    //Handle Exception here
on exit
    //Execute required code, such as releasing external resources
...
```

The choice between using an explicit `do/end` block or handling exceptions directly within a method depends on the scope of the method you are writing. Also, code readability should be taken into account when choosing which style to use.

Simple Conditional Statements (if-then-else)

Describes purpose and variants of the if-then-else statement

If-then-else

The if-then-else statement evaluates a boolean (true/false) expression. If the expression yields true, the statement block following then is executed. Else, if the expression is false, execution goes to the else statement block, if present. Execution then continues normally after the end statement.

The syntax is as follows, with the else clause being optional:

```
if <condition> then
    <statements>
[else
    <statements>]
end
```

The following example is used with display and input statements to capture end user feedback. This particular example evaluates the variable `selected` and sets the predefined action variable to `FAIL`:

```
if selected = "Cancel" then
    action = FAIL
end
```

The following example evaluates the variable `orderTotal`. If the order is greater than \$5,000, the Boolean variable `checkCredit` is set to `true`:

```
if orderTotal > 5000 then
    checkCredit = true
end
```

We can go one step further with the previous example. We can also check whether the order is a credit order or a cash order by using the logical operator and to verify the two conditions. As before, the code checks if `orderTotal` is greater than \$5,000 and now also requires that `paymentType` be set to "Credit":

```
if orderTotal > 5000 and paymentType = "Credit" then
    checkCredit = true
end
```

The final example shows the use of the `or` logical operator and the `else` clause. This example checks whether the variable `lollipop` is "cherry" or "raspberry". If so, `eat` is set to `true`. If not, `eat` is set to `false`:

```
if lollipop = "cherry" or lollipop = "raspberry" then
    eat = true
else
    eat = false
end
```

Elseif

Some times you will need to handle more than two alternatives. To support this situation you can use the optional `elseif` clause:

```
if <condition> then
    <statements>
[elseif <condition> then
    <statements>]
...
[elseif <condition> then
    <statements>]
[else <condition> then
    <statements>]
end
```

In effect, each `elseif` concatenates two `if-then-else` statements. An arbitrary number of `elseif` blocks can be included, as well as a single optional `else` clause at the end. The example below uses the `elseif` clause and the `else` clause. This way, you can continue adding conditions indefinitely:

```
if selected = "Cancel" then
    action = FAIL
elseif selected = "Process" then
    orderStatus = "Reviewed"
    financeStatus = "Check"
else
    orderStatus = "In Review"
end
```

For situations where program flow must follow several possible options based on only one parameter, it is better to use the [Case Statement](#) on page 287 statement.

Case Statement

Describes the case multipath conditional statement

This statement is an alternative to the *Simple Conditional Statements (if-then-else)* on page 285 conditional statement. The case construct is more efficient and easier to read when you need to execute one block of code among many, based on the value of a single parameter.

Syntax:

```
case <expression>
when <case1> then
    <statements>
[when <cases2> then
    <statements>]
...
[when <caseN> then
    <statements>]
[else
    <statements>]
end
```

There can be zero or more when-then statement blocks, though normally there will be more than one.

A given when expression can check for more than one value, where each value is placed in a comma-delimited list:

```
case x
when 1 then
    display "x is equal to one"
when 2,3,4,5,6 then
    display "x is a value between two and six"
else
    display "x is greater than six or less than one"
end
```

The else block is optional. It can be used to implement a default action if no when conditions are met.

The following case example sets the string `shortWeekday` based on the value of `dayNumber`. There are seven weekdays, so if `dayNumber` is not 1, 2, 3, 4, 5, 6, or 7, then its value is considered to be invalid.

Example:

```
case dayNumber
when 1 then
    shortWeekday = "Sun"
when 2 then
    shortWeekday = "Mon"
when 3 then
    shortWeekday = "Tue"
when 4 then
    shortWeekday = "Wed"
when 5 then
    shortWeekday = "Thu"
when 6 then
    shortWeekday = "Fri"
when 7 then
    shortWeekday = "Sat"
else
    shortWeekday = "This is not a valid weekday!"
end
```

Bounded Loops

Describes range and key iteration bounded loops

Bounded loops allow you to execute a set of statements a number of times which is known before entering the loop. The number of times might be determined by a range or a collection.

Range iteration (for in)

This loop iterates over an integer range. That is, on each iteration, an integer variable increments by one:

```
for <id> in <rangeStart>..rangeEnd> do
  <statements>
end
```



Note: *rangeStart* and *rangeEnd* can be expressions. The range limits are inclusive. Modifying the variable inside the loop does not have any effect on the loop's execution.

For example, this loop displays the numbers from 1 to 3, inclusive:

```
for i in 1..3 do
  display i
end
```

Key iteration (for in)

The `for in` loop can also iterate over the *keys* or *indexes* of an array:

```
for <id> in <expression> do
  <statements>
end
```



Note: *expression* must yield an array.

This example displays all the keys in the `ages` array:

```
ages as Int[String]
ages = ["John" : 23, "Peter" : 42, "Mary" : 29]

for name in ages do
  display name
end
```

Element iteration (for each in)

The `for each in` loop iterates over the *values* of an array. Some of them may be excluded by adding a "where" clause:

```
for each <id> in <expression> [where <condition>] do
  <statements>
end
```



Note: *expression* must yield an array.

This example displays all the values in the `ages` array:

```
ages as Int[String]
ages = ["John" : 23, "Peter" : 42, "Mary" : 29]

for each age in ages do
  display age
end
```

The following example shows only the ages above 25:

```
ages as Int[String]
ages = ["John" : 23, "Peter" : 42, "Mary" : 29]

for each age in ages
  where age > 25
do
  display age
end
```


Stopping a Loop

Loop execution can be stopped by using an Exit Statement. Execution will continue from the first statement following the end of the loop.

Unbounded Loops

Unbounded loops are useful when the programmer does not know in advance how many times the loop will be traversed.

This loop repeats an action until an associated test condition returns false or until an exit statement is executed. The condition is evaluated before entering the loop and after each iteration.

Syntax

```
while <condition> do
  <statements>
end
```

Example

Continue asking until the user chooses "Ok":

```
selection as String = ""
while selection /= "Ok" do
  display "Are you sure ?"
  using buttons = ["Ok", "Cancel"]
  returning selection = selectedButton
end
```

Exit Statement

A method or a loop in a method can be interrupted by using the exit statement:

- If the loop is labeled and the exit statement refers to that label, the exit statement exits the labeled loop.
- If the name provided is the method name, the execution of that method ends.
- If no name is provided, execution exits the innermost loop of the method, or the method itself, if it is outside a loop.

Syntax:

```
exit [<label>] [when <condition>]
```



Note: *label* can be the method name or a labeled loop. The *condition* can be used to avoid cluttering your code with a conditional statement.

The following example finds the first e-mail containing a specific subject then exits the loop:

```
// order is a variable of type Fuego.Net.Mail

order as Mail
url as String = ""

for each mail in MailServer(url, false).messages do
  if mail.subject = "New Order" then
    order = mail
    exit
  end
end
```

```
// if there is no order to process,
// stop method execution exit when
// order is null.
```

The second example finds a participant, assigns it to the `nextParticipant` variable, and ends the method execution. The name of the method is `findParticipant`:

```
participantName = "John"

for each p in activity.role.participants do
  if p.name = participantName then
    // participant found!
    nextParticipant = p
    exit : findParticipant
    // findParticipant is the name of the method
  end
end
end
```

For further information refer to topics:

Labeled Statement

Labels provide a name to identify and reference a statement. Labels are used by the exit statements to specify the statement to which the exit applies.

Note that all methods have an enclosing label which has the same name as the method.

Syntax

```
<label> : <statement>
```

Example

```
// order is a variable of type Fuego.Net.Mail
order as Mail
url as String = ""

mainLoop: for each mail in
  MailServer(url, false).messages do
    if mail.subject = "New Order" then
      order = mail
      exit : mainLoop
    end
  end
end

// if there is no order to process, stop method execution
exit when order is null
```


Throw Statement

The throw statement lets you raise an exception, thus breaking the execution until:

- The exception is handled.
- The method finishes and the process' exception handling procedures gain control.

Syntax

```
throw <expression>
```

 **Note:** * **expression** must yield a `Java.Lang.Throwable`.

Example

```
throw Java.Exception("Something is wrong")
```

Logging Statement

The logging statement is used to log a message in the log files maintained by the Process Execution Engine. Log messages are useful to debug the behavior of your code, especially in automatic activities.

Syntax

```
logMessage "message to log"
  [using
    [severity = <severity>]
    [, detail = <detail>]
  ]
```

Severity levels

The following are severity levels that can be used with the logging statement:

- DEBUG
- INFO
- WARNING
- SEVERE
- FATAL

You can choose the severity level you want to display in the Engine logs in the Execution Console.

Example

```
orderName = 1
customer = 1

logMessage "Order " + orderNumber + " from customer "
  + customer + "was aborted" using severity = SEVERE
```

Regular Expression Overview

A *regular expression* is a pattern or template for string matching. Regular expressions are written with a very specialized, powerful syntax which can perform complex string comparisons, extract desired substrings, or perform advanced search and replace operations on strings.

PBL provides support for regular expressions using syntax compatible with Perl regular expression syntax.

Regular Expression Syntax

Regular expressions are enclosed between forward slashes:

```
{regular expression}
```

The simplest regular expression is a single word to search for in a string. A regular expression consisting of a word matches any string that contains that word. So, the following regular expression matches any string that contains the word "hello" anywhere in the string:

```
/hello/
```

A regular expression is a pattern which is written to match single characters or multiple characters. In the case above, each one of the characters in `/hello/` is simply matching itself. The next simplest matching character is the dot ".", which will match any single character except newline "\n". For example, the expression `/c.t/` will match "cat", "cut", or any other three-character string starting with c and ending with t, so long as the middle character is not a newline.

If, however, you specifically wish to match only "cat" and "cut", you can use a character class, which is a pattern of characters within square brackets. The following matches "cat" and "cut", but no other word:

```
/c[au]t/
```

Many other characters and expressions are possible. They are described in various topics throughout this section.

Using Regular Expressions

In PBL, you must write regular expressions between a single quotes as follows:

```
'/{regular expression}'
```

In order to actually use a regular expression to do something in PBL, you must use a function which works with regular expressions. For instance, to find out if a particular string contains a word, you need to use the `contains` function, as follows:

```
myString.contains('/Hello/')
```

This line of code will search for the word "Hello" in the string that is contained in `myString`. It returns true if "Hello" is found and false if it is not.

You can also apply function calls to literal strings instead of variables. For example:

```
"Hello world!".contains('/Hello/')
```

returns true because "Hello world!" does contain "Hello".

Regular Expressions in Functions

In Studio, you use regular expressions by calling functions (methods) used on string objects. The following string functions support regular expressions:

Function	Purpose	Returns
<code>contains()</code>	Matches a substring contained in the string.	Bool
<code>isMatch()</code>	Matches the string completely.	Bool
<code>indexOf()</code>	Gets the first index location where the regular expression matches the string.	Int
<code>lastIndexOf()</code>	Gets the last index where the regular expression matches the string.	Int

Function	Purpose	Returns
match()	Attempts to match and return the substring(s) that matched the regular expression. This is useful for extraction and parsing.	String[] - The array of subexpressions (groupings) matched. When the g modifier is used, the array of matched occurrences is returned instead.
split()	Splits a string using the given regular expression as a separator.	String[] - The array of fields (pieces of the string that were separated by the given separator.)
count()	Gets the number of substrings (within the string) that the given regular expression matches.	Int
replace()	Replaces pieces of the string with new strings.	String - the new modified string.

Search and Replace

Regular expressions can also be used to replace parts of a string by a different string. In Studio, you can do this using the replace function. The first argument to replace is the regular expression search and the second argument is the new string. For example:

```
myString = "We played 1 on 1"
myString.replace('/1/', "one")

display myString
```

In the second line, 1 is replaced by one. Notice that only the first occurrence of 1 is replaced when you try the code with the debugger. In order to replace all occurrences, you must use the g modifier. For example:

```
myString.replace('/1/g', "one")
```

The following code strips any zero digit on the left end of a string. For example, to change 005422 to 5422, use:

```
myString.replace('/\b0+/g', "")
```

A powerful feature of the replace function is that you can use backreferencing in the replacing string. You do so by using \$x, where x is the grouping number.

For example, to swap the first two words in a string:

```
myString.replace('/(w+) (\w+)/', "$2 $1")
```

To convert a MM-DD-YYYY date to DD/MM/YYYY, use:

```
myString.replace('/(\d+)-(\d+)-(\d+)/', "$2/$1/$3")
```

To remove quotes surrounding any word:

```
myString.replace('/"(\w+)"/g', "$1")
```

Modifiers

You may have already noticed that matching is case sensitive. This means that the regular expression will only match a substring if both the regular expression and the substring have the same upper/lowercase characters.

In order to make matching case insensitive, you need to use a modifier. Regular expression modifiers allow you to change the default behavior of the matching.

To add a modifier, you extend the basic syntax of the regular expression as follows:

```
'/regular expression/modifier(s)'
```

Each modifier is just a single character that is specified between the last slash and the quote. The modifier for insensitive case matching is `i`. So, the following regular expression matches `Hello`, `hello`, `HeLiO`, and `hELLO`:

```
'/hello/i'
```

Some other modifiers are listed in the following table:

Modifier	Meaning
i	Searches in a case insensitive manner.
g	Matches the regular expression as many times as possible. Matches all substrings globally.
s	Treats the string as a single line.
m	Treats the string as multiple lines.

More than one modifier can be used at the same time.

Metacharacters and Character Sets

The power of regular expressions is based on the fact that they can contain special characters that perform special functions. These characters are not treated as regular characters and are not matched literally. They are known as *metacharacters*.

Some of the metacharacters that make pattern matching more generic are as follows:

```
[ ] . ( ) * ^ $ ? \
```

The `[` and `]` characters are used to specify a set of characters that you wish to match. Characters can be listed individually or a range of characters can be indicated by listing two characters separated by a dash (`-`). For example, `[aeiou]` matches any of the vowels. The set `[a-d]` is equivalent to `[abcd]`.

If you specify the `^` character right after the opening bracket, it matches the complement of the set. In other words, any character that is not in the set. For example, `[^0-9]` matches any non-digit character.

The following table lists some examples of regular expressions using sets:

Regular Expression	Strings that Match
'/[cr]at/'	cat, rat
'/[0-9]/'	Any digit
'/[0123456789]/'	Any digit
'/[A-Z]/'	Any uppercase letter

Regular Expression	Strings that Match
'/[0-9][0-9]'	Any two digits together (like 01, 42, 27...)
'/[aeiou]'	Any of the vowels

Fortunately, there are some shortcuts for some of the common character sets. For example, `\d` denotes "any digit" the same way that `[0-9]` does. `\D` means "any non-digit" like `[^0-9]`. The following table lists some other common shortcuts.

Shortcut Sequence	Equivalent to
<code>\d</code>	<code>[0-9]</code>
<code>\D</code>	<code>[^0-9]</code>
<code>\w</code>	<code>[a-zA-Z0-9_]</code> Any alphanumeric character including the underscore.
<code>\W</code>	<code>[^a-zA-Z0-9_]</code> Any non-alphanumeric character.
<code>\s</code>	Any whitespace character.
<code>\S</code>	Any non-whitespace character.

These shortcuts can be included inside a character class (set.) For example, `[\da-fA-F]` is a character class that will match one hexadecimal digit. The tab, new line, and return characters are specified with `\t`, `\n` and `\r`, respectively.

Another special metacharacter is the dot (`.`). A dot within a regular expression matches any character (except the `\n` character, unless the `s` modifier is used).

Special Cases

What happens if you want to search for some of the metacharacters, like `[` or `?`? You can escape these characters with a backslash (`\`) just before the character. For example, `\.`, `\`, and `\[` match a literal dot, backslash, and opening bracket, respectively.

Go back to the Method example and try these metacharacters with different regular expressions. Experiment with character classes and the shortcuts available. The only way to learn to use regular expressions is to build some.


Matching Repetitions

In the previous examples, you were only matching expressions consisting of a few generic characters and literal words. To help write more expressive patterns, you use a quantifier metacharacter. The quantifiers are as follows:

`? * + { }`

The quantifiers allow you to determine the number of repeats of a portion of a regular expression you consider a match. Quantifiers are located immediately after the character, character class or grouping that you want to match. The following table defines each quantifier and its meaning.

Quantifier	Definition
<code>a?</code>	Match 'a' one or zero times.
<code>a*</code>	Match 'a' zero or more times (any number of times.)
<code>a+</code>	Match 'a' one or more times (at least one time.)
<code>a{n,m}</code>	Match 'a' at least n times, but not more than m times
<code>a{n, }</code>	Match 'a' at least n or more times.
<code>a{n}</code>	Match 'a' exactly n times.

 **Note:** 'a' in the previous table can be any character, character class or grouping. You will learn more about groupings later but basically, you can group a part of a regular expression using parenthesis.

The following are some examples using matching repetitions:

Matching repetition	Description
'/\w+/'	Any alphanumeric word (one or more alphanumeric characters together.)
'/-?\d+/'	A number (one or more digits) optionally prefixed by a hyphen.
'/[a-z]+\t\d{1,5}/'	Any lowercase word, followed by a tab, followed by 1 to 5 digits
'/\w+/'	Any alphanumeric word (one or more alphanumeric characters together.)
'/The.*dog/'	Any line that is followed by anything and then dog. Examples: The nice dog The quick brown fox jumped over the lazy dog The WhateverHeredog Thedog

Now, you have enough tools to create some useful regular expressions. For example, let's build a simple regular expression to match 10 digit telephone numbers. You may start with:

```
'/\d{10}/' // 10 digits (no more, no less)
```

This regular expression matches any 10 digit number but it has some weaknesses. For instance, what happens if you want to accept numbers that contain dashes (-) in between, such as 321-123-1234? In that case, you can do the following:

```
'/\d{3}-\d{3}-\d{4}/'
```

This is fine, but what if you want the dashes to be optional? Try this:

```
'/\d{3}-?\d{3}-?\d{4}/'
```

This is better, but still there are some improvements to be made. You might not want to allow a zero (0) as the first digit of the number. Thus, the first digit must be in the class [1-9] instead of \d as follows:

```
'/[1-9]\d{2}-?\d{3}-?\d{4}/'
```

Did you understand it? Let's study it in parts:

- 1. First, a digit between 1 and 9: [1-9]
- 2. Next, two digits (from 0 to 9): \d{2}
- 3. Then, an optional dash: -?
- 4. Three digits: \d{3}
- 5. Another optional dash: -?
- 6. Finally, four digits: \d{4}

Try it in the debugger:

```
phone as String
input "Enter your phone number:" phone

if phone.isMatch('/[1-9]\d{2}-?\d{3}-?\d{4}/') then
  display "OK, a valid phone number"
else
  display "ERROR, invalid phone number"
end
```


Anchors

Describes purpose and use of anchor metacharacters.

When you use the contains function, it returns true if the regular expression matched anywhere in the string. However, sometimes you would like to specify where in the string the regular expression should try to match. To do this, you use *anchor* metacharacters.

Common Anchor Metacharacters

The two most common anchor metacharacters are '^' and '\$'. The '^' anchor means match at the beginning of the line and the '\$' anchor means match at the end of the line. The following examples show how they are used:

```
display "rock and roll".contains('/and/')
// displays true
display "rock and roll".contains('/~np~^~/np~and/')
// displays false
display "rock and roll".contains('/~np~^~/np~rock/')
// true
display "rock and roll".contains('/rock$/')
// false
display "rock and roll".contains('/roll$/')
// true
display "rock and roll".contains('/nd roll$/')
// true
display "rock and roll".contains('/\~np~^~/np~rock$/')
// false
display "rock and roll".contains('/\~np~^~/np~rock and roll$/')
// true
display "rock and roll".isMatch('/rock and roll/')
// true
```

The second regular expression does not match because '^' constrains and to match only if it is at the beginning of the string. The fifth regular expression does match, since the '\$' constrains roll to match only at the end of the string.

Look at the last two examples. If you use both the '^' and '\$', you mean that the regular expression must match both the beginning and the end of the string. In other words, the regular expression matches the whole string. Note that both examples are equivalent since the isMatch will always look for a complete match. The '^' and '\$' anchors are irrelevant when using isMatch.

Difference between '^' and '^A' and between '\$' and '\$Z'?

Usually, you will only use '^' and '\$', but when using the m modifier, there is a small difference. If the string contains newline (\n) characters, then the '^' and '\$' match, just after and just before, the new line. However, '^A' and '\$Z' only match at the start and end of the whole string. So, using the m modifier and replacing the space in "and roll" with a newline you get the following:

```
display "rock and~np~\~/np~nroll".contains('/and$/m')
// true
display "rock and~np~\~/np~nroll".contains('/and~np~\~/np~Z/m')
// false
display "rock and~np~\~/np~nroll".contains('/^roll$/')
// true
display "rock and~np~\~/np~nroll".contains('/~np~\~/np~Aroll~np~\~/np~Z/')
// false
display "rock and~np~\~/np~nroll".contains('/~np~\~/np~Arock/')
// true
```

The following table describes modifier behavior:

Modifier	Behavior
<i>none</i>	Default behavior. '.' matches any character except '\n'. '^' only matches at the beginning of the string and '\$' only matches at the end of the string.
<i>s</i>	Treat string as a single long line. '.' matches any character, even '\n'. '^' only matches at the beginning of the string and '\$' only matches at the end or before a new line at the end.
<i>m</i>	Treat string as a set of multiple lines. '.' matches any character except '\n'. '^' and '\$' are able to match at the start or end of any line within the string.
<i>m and s</i>	Treat string as a single long line but detect multiple lines. '.' matches any character, even '\n', '^', and '\$'. However, they are able to match at the start or at the end of any line within the string.

Alternations

Sometimes, you need a regular expression to match different possible words or character strings. This is possible by using the alternation metacharacter (`|`). So, if you want to match any substring that contains the word `hi` or the word `hello`, then use the following expression:

```
'/hi|hello/'
```

Bear in mind that the expression tries the alternative choices from left to right trying to match the regular expression at the earliest possible point in the string. The following are some examples:

```
"black and white".contains('/black|gray|white/')
// matches black
"black and white".contains('/white|gray|black/')
// matches black. Even though white is the first
// alternative in the string, black matches
// earlier in the string.
"Bye!".contains('/b|by|bye|bye!/i')
// matches b
"Bye!".contains('/bye!|bye|by|b/i')
// matches bye!
```

The last example suggests that if some of the alternatives are prefixes of the others, they put the longest alternatives first. Otherwise, they will never match.

In some way, you can think of character classes as character alternations. So `'/[aeiou]'` behaves like `'/a|e|i|o|u/'`.

Grouping

Parts of a regular expression can be grouped so that they are treated as a single unit. Parts of a regular expression are grouped by enclosing them with parenthesis (`()`). Grouping a subexpression has many uses such as for alternation on part of the whole regular expression, for repetitions, for text extraction and for backreferencing. (Extraction and backreferencing are discussed in later sections.)

Some grouping examples are displayed in the following table:

Regular Expression

```
'/(straw|blue|rasp)berry/'
```

String that Matches: strawberry, blueberry, or raspberry

Regular Expression

```
'/Blah( blah)*/'
```

String that Matches: Blah, Blah blah, Blah blah blah, Blah, blah, blah, blah,...

Regular Expression

```
'/^a|b/'
```

String that Matches: Matches either a or b at the beginning of the line (note that `‘/^a|b/’` would match a at the beginning or any b anywhere).

Regular Expression

```
'/y(es)?i'
```

String that Matches: Y, y, or any case insensitive version of ‘yes’.

Extraction

In regular expressions, *extraction* refers to the storage of strings matched by one part of the regular expression with the purpose of using them elsewhere in the expression. This is very useful for parsing and for general text processing.


An extraction group is delimited by parenthesis. For each grouping, the part of the string that matches inside the parenthesis goes into a particular position within an array of matched groupings. In PBL, the extraction can be done with the match function, which returns the array of substrings for each grouping.

For example, suppose that you have a string with the current time, in *hh:mm:ss* format. You can build a basic regular expression for matching times in that format, such as `/\d\d:\d\d:\d\d/`. However, you want to know what the value of just one element, such as the hour, is. To obtain it, group each element with parenthesis. For example, `/(\d\d):(\d\d):(\d\d)/`. The following example shows how to display hours, minutes and seconds using the index numbers of the array:

```
time as String
matches as String[]
input "Enter a time (hh:mm:ss):" time

matches = time.match('/(\d\d):(\d\d):(\d\d)/')

if matches is not null then
    display "Hours: " + matches[1] + "\n" +
           "Minutes: " + matches[2] + "\n" +
           "Seconds: " + matches[3]
else
    display "Invalid time!"
end
```

 **Note:** When a regular expression is matched against a string, the whole part of the string that matches is stored in position 0 (zero) of the array.

For the previous example, if you enter "12:40:23", the array will contain the following:

Position	Value
1	12:40:23
2	12
3	40

Position	Value
4	23

Positions are assigned to each group from left to right.

Extraction Example

The following is a real world example of extraction. Suppose that you need to interpret a text file with lines with the following format:

property = value

The file can also have comment lines, which begin with the pound sign (#). A sample of the file follows:

```
# Configuration parameters
adminEmail=admin@yoursite.com
serverHost=server.yoursite.com
serverPort=12345

# some preferences
soundEnabled=false
fontSize=12

# colors
background = white
foreground = blue
```

It would be useful if you could create an associative array, for simple access to each property. For example, to get the value of the serverPort property defined in the file we would use:

```
port = properties["serverPort"]
```

First, you need to define the regular expression to interpret a valid line in the file. As mentioned before, lines can be in property = value format or they may start with a pound (#) sign. In the latter case, the line must be ignored.

The assignment lines can be matched with `/\w+=\w+/. This looks for a word (\w+) and equals sign (=) and another word (\w+).`

The following allows optional white space around the equals sign:

```
/\w+\s?=\s?\w+/
```

Now you need to group the left side word (before the equals) and the right side word (after the equals sign) so that you can extract the values:

```
/(\w+)\s?=\s?(\w+)/
```

One more detail is required. Let's force the regular expression to match the whole string. You achieve this by adding the ^ and \$ anchors:

```
/^(\w+)\s?=\s?(\w+)$/
```

The following code fragment tests the expression:

```
input "Enter a line:" line
m = line.match('/^(\w+)\s?=\s?(\w+)$/')
if m is not null then
    display "Property: " + m[1] + "\nValue: "
        + m[2]
else
    display "ERROR, invalid line!"
end
```

A comment is easy to match by using the following regular expression (remember comment lines begin with the pound sign # in the sample text file):

```
/^#.*
```

The expression `/^#.*/` means a line beginning with `#` and followed by any number of characters. An alternation will allow comment lines to match and test the Method again:

```
input "Enter a line:" line
m = line.match('/(^#.*$)|^(\\w+)\\s?=\\s?(\\w+)$/')
if m is not null then
  if m[1] = "" then
    display "Property: " + m[3] + "\\nValue: "
      + m[4]
  else
    display "Comment line found: " + m[0]
  end
else
  display "ERROR, invalid line!"
end
```

Now that you have tested the regular expression, you can remove the display statements and write the code that builds the associative array. Instead of reading the lines from an input, we read them from a file:

```
for each line in TextFile("/tmp/test.txt").lines
  m = line.match('/(^#.*$)|^(\\w+)\\s?=\\s?(\\w+)$/')
  if m is not null then
    // if m is not a comment
    if m[1] = "" then
      props[m[3]] = m[4]
    end
  else
    // erroneous line - ignore it
  end
end
display props
```

Replace `tmp/test.txt` with a valid file name and location before testing the code.



Note: The `TextFile` component contains a built-in function for creating an associative array from a properties file. This example just shows you how to use regular expressions in a real problem. If the file were compatible with a Java properties file, then the `Textfile.loadPropertiesFrom` component is the easiest solution.

The following examples show regular expression solutions to common problems.

Example 1

Obtain the path from the filename of a fully-qualified UNIX path and filename such as `/usr/utilities/reader/readme.txt`. This requires two extractions, as follows:

```
/(.*)\\/(^[^\\/]*)$/
```

Position [1] will contain the path (`usr/utilities/reader`), and position [2] will contain the name of the file (`readme.txt`).

Example 2

Obtain the user ID and the host name from an e-mail address such as `support@bea.com`. This requires two extractions, as follows:

```
/([\\w\\.]+)@([\\w\\.]+)/
```

Position [1] will contain the user ID (`support`), and position [2] will contain the host name (`bea.com`).

Example 3

To extract the parts of a URL such as `http://www.bea.com:80/index.html`. We require the protocol, host name, port number, and resource:

```
/((\\w+):\\/\\/)(^[^\\/:]+)(:(\\d+))?(\\/.*)?/
```

The following values will be obtained:

Position	Value
1	http
2	www.bea.com
3	:80
4	80
5	/index.html

Note that to obtain the port number both with and without the colon, a nested extraction was used.

Backreferencing

A substring matched by a grouping can also be referenced within the regular expression, which is known as backreferencing. Backreferencing allows you to make matches later in a regular expression depending on what matched earlier in the regular expression. You can reference a previous grouping with `\x`, where `x` is the grouping position.

The following table provides some backreferencing examples:

Regular Expression	Matching String
<code>/(\~np~\~/np~w+) \~np~\~/np~1/</code>	The same word repeated twice. For example, the echo ha ha...
<code>/(\~np~\~/np~w+)\~np~\~/np~1/</code>	Words with repeated parts. For example, mama, papa, coco...
<code>/(\~np~\~/np~d)(\~np~\~/np~d)(\~np~\~/np~d)\~np~\~/np~2~np~\~/np~1</code>	Any five-digit palindrome number. For example, 12321, 83638, 91119...

Objects Overview

Studio makes extensive use of components, and it includes a large library of built-in components for common tasks. You can write your own components inside Studio (Business Objects), and you can include different technologies as components in the component catalog.

Components define a type, which can be used to declare variables. All components can be used to declare a local variable or an argument variable, but not all components can be used as instance variables.

This happens because instance variables are usually persisted (a process instance variable) or transferred (a Presentable Business Object instance variable). And, for persistence or instance variable transference to work, the content of such variables must support serialization. Some components do not support serialization.

A component can be identified by its casing. Component names always begin with an uppercase letter. For further details, refer to the [General Naming Conventions](#) on page 310 topic. For further information on component usage in Studio, please refer to the following topics:

- Implementing Business Objects using BPM Objects
- Introducing BPM Objects into the BPM Project Catalog

Creating an Object

As noted in the [Variables Overview](#) on page 263, all variables have a type.

Variables of primitive types (such as String, Int, Real, and so on) always have a default value (as described in [Initializing Variables](#) on page 274). On the other hand, variables which have a non-primitive type have special initialization rules.

In this section, we will discuss how to explicitly initialize non-primitive variables.

Constructors

For initialization, we can group components in two categories:

- Instantiable components
- Components that must be obtained from another component

The difference between the two categories is that the first has a special method called a *constructor*, which is used to create new instances of the component. The second does not.

Constructors are methods that are named after the component, and may or may not have arguments. If the constructor of a component does not have arguments, it is called the *default constructor*.

The syntax to initialize a variable is the following:

```
variable = [Module.]ComponentName([[{argument name}:{value}[,...]])
```

Note that the names and types of arguments depend on the component and on the constructor you are calling.

Consider the following example:

```
configFile as TextFile
configFile = TextFile(name : "/home/config.props")
for each line in configFile.lines do
    //Process the lines...
end
```

In the example above, on the first line, a local variable named `configFile` of type `TextFile` is declared, and on the second line it is initialized using `TextFile`'s constructor, passing a file name as an argument.

Duplicating an Object

Describes the clone function, used to duplicate objects.

Clone Function

Sometimes you want to create an exact copy of an object.

Simply assigning a component to a variable *does not* create a copy of it. Rather, it creates an additional *reference* to the same object. If any property of the original object changes, the new reference will also show these changes, since it is still pointing to the same object.

In order to actually create a new duplicate of the object, you can use the clone function:

```
// Create an instance for each
// participant in the role

for each person in activity.role.participants do
    copy = clone(this)
    copy.participant.next = person
end
```

Function Behavior

The clone function behaves differently depending on how the object to be cloned is implemented. To be able to respond to different conditions, the function follows the following steps:

1. If the object you are trying to clone has a method named clone and implements the interface Cloneable, that method is used to obtain a copy.
2. If the object implements the Serializable interface, it attempts to serialize it and deserialize it to obtain a copy of the object.
3. Otherwise, it attempts to dynamically create a copy of it.

Calling an Object

Current and Default Instances

Object instance behavior under PBL

Default Instance

AquaLogic BPM has the concept of a default instance, which is an instance associated to a component.

Only components that have default constructors have default instances. Such instances are accessible within the method scope. That is, a default instance that has been created while running a specific method only exists through that method's execution.

For example:

```
show Menu
    using entries = ["Apples", "Oranges", "Chocolate"],
        title = "Which do you like best?"
```

In the example above, the reference to the component Menu is using its default instance, that is, an instance is automatically created the first time that a reference to Menu appears.

Current Instance

Typically, when you want to refer to your current instance, you use the keyword this (or Me, in Visual Basic style):

```
update this using date = 'now'
```

Suppose that the code above belongs to a component named MyComponent. In this case, you could also write it as follows:

```
update MyComponent using date = 'now'
```

Here, the default instance of MyComponent is the same as the this, so the current instance is used as the default instance.

Object Cleanup

Studio automatically releases the memory used by components when they are no longer used. It is not necessary to 'release' or 'clean' the components used in a method. However, there are certain components that require some kind of cleaning before ending the execution. They must be cleaned by using an 'exit' block to ensure that they are always cleaned up. For example:

```
do
```



```
// use components here
on exit
  // clean used components here
end
```

Note that the code enclosed in the **on exit** (Java style: **finally**, VB style: **Finally**) part of the block is executed even if an exception occurs. Next, after the execution, the original exception is thrown, unless it is masked by an exception thrown in the **on exit** block.

Code Conventions Overview

Provides a general description of code conventions

In any computer language, *code conventions* are a set of rules that should be followed when writing program code. They are called conventions because they are not enforced by the compiler, as they are not a part of the language syntax itself. For example, a variable can be named `lastName` or it can be named `lstn01`. The first choice is easier to read for humans, but to the compiler either is valid. Think of code conventions as a set of best practices, which under normal circumstances should be adhered to as closely as possible.

As a general rule, the purpose of code conventions is to improve readability and prevent bugs. To the extent that everybody adopts the same conventions when programming, each individual will be able to understand the work of others, and fewer mistakes will be made. This is even the case when a single individual reads his own code many months or years after having written it.

One way to improve readability is to include comments in your code. On the other hand, for the most part code conventions guide the way the code itself is structured. There are a number of ways to make code more readable:

- Adhere to variable and object naming conventions
- Use explicit variable names
- Indent code according to depth within code blocks
- Express logical statements in the simplest way possible
- Use whitespace to separate program segments

The Studio editor can help you adhere to some of these code conventions using two commands: `Indent` and `Refactor`.

Improving Code Readability

Nested conditional statements

Nested conditional statements are automatically grouped in one statement with both conditions.

Before

```
a as Int
if a > 2 then
  if a < 10 then
    a = 5
  end
end
```

After

```
a as Int
if a > 2 and a < 10 then
```

```
a = 5
end
```

Identifiers for Exceptions

Identifiers for exception handlers are automatically added when they are not specified.

Before

```
message as String
do
  message = "Ok"
on Exception
  message = Exception.message
end
```

After

```
message as String
do
  message = "Ok"
on e as Exception
  message = e.message
end
```

Bounded loop instead of unbounded loop

Unbounded loops are converted to bounded loops when possible.

Before

```
i = 0
while i <= 10 do
  i = i + 1
end
```

After

```
for i in 0..10 do
end
```

Conditional Exit

Conditional statements with an exit statement are transformed to conditional exits.

Before

```
array as Int[] = [10, 20, 30]
for each e in array do
  if e = 20 then
    exit
  end
end
```

After

```
array as Int[] = [10, 20, 30]
```

```

for each e in array do
    exit when e = 20
end

```

Redundant negation

Redundant negations are removed.

Before

```

a as Int = 5

if not a != 2 then
end

```

After

```

a as Int = 5

if a = 2 then
end

```

Conditional statement inside else blocks

Before

```

a as Int = 2
if a < 2 then
    a = 2
else
    if a > 5 then
        a = 5
    end
end
end

```

After

```

a as Int = 2

if a < 2 then
    a = 2
elseif a > 5 then
    a = 5
end
end

```

Check for null value

Before

```

s as String

if s != null then
end

```

After

```

s as String

if s is not null then
end

```

Right order for 'is not'**Before**

```
s as String

if not s is null then
end
```

After

```
s as String

if s is not null then
end
```

Redundant equality**Before**

```
found as Bool

if found = false then
end
```

After

```
found as Bool

if not found then
end
```

Explicit argument names**Before**

```
open TextFile using "", ""
```

After

```
open TextFile using name = "",
                    lineSeparator = ""
```

Unneeded parenthesis

This refactory is applied to 'if' and 'while' conditions in Process Business Language (PBL).

Before

```
a as Int = 2

if (a > 2) then
end
```

After

```
a as Int = 2

if a > 2 then
```

```
end
```

Legacy multi path conditional statements

Before

```
a as Int = 2

switch a in
  case 2:
    display "Two"
  case 4:
    display "Four"
end
```

After

```
a as Int = 2

case a
when 2 then
  display "Two"
when 4 then
  display "Four"
end
```

Functions

Functions are rewritten using functional syntax.

Before

```
a as Int
a = a.abs()
```

After

```
a as Int
a = abs(a)
```

Wrong symbols

In PBL, some invalid symbols (e.g: &&, ||, !, ==) are accepted but they are automatically fixed when the code is rewritten.

Before

```
a as Int = 2
b as Int = 4

if ((a > 2 && a < 10) || b == 4) then
end
```

After

```
a as Int = 2
b as Int = 4

if (a > 2 and a < 10) or b = 4 then
end
```

Misspelled member names**Before**

```
Open TextFile using name = "",
                    lineSeparator = ""

Mail.content_type = ""
```

After

```
open TextFile using name = "",
                    lineSeparator = ""

Mail.contentType = ""
```

Methods equals() and toString()**Before**

```
if object.equals(this) then
    string = object.toString()
end
```

After

```
if equals(object, arg1 : this) then
    string = toString(object)
end
```

General Naming Conventions

Names representing types or modules must be nouns and they must be written in mixed case starting with upper case:

```
Line, FilePrefix
```

Variable names must be in mixed case starting with lower case:

```
line, filePrefix
```

Makes variables easy to distinguish from types and effectively resolves potential naming collision as in the declaration `Line line`.

Names representing constants (or enum values) should be all uppercase using underscore to separate words:

```
MAX_ITERATIONS, RED, MONDAY
```

Names representing methods must be verbs and they must be written in mixed case starting with lower case:

```
find(), computeTotalWidth()
```

This is identical to variable names but, in Studio, methods are already distinguishable from variables by their specific form.

Abbreviations and acronyms should not be uppercase when used as a name:

```
exportHtmlSource();      // NOT: exportHTMLSource();
openDvdPlayer();         // NOT: openDVDPlayer();
```

Using all uppercase for the base name will trigger conflicts with the naming conventions given above. A variable of this type should be named dVD, hTML, and so on, which is obviously not very readable. Another problem is illustrated in the examples above. When the name is connected to another, the readability is seriously reduced. The word following the acronym does not stand out as it should.

Variables should not have prefixes or suffixes.

Given the fact that the PBL Editor has already colored variables in a different way depending on their scope (local, instance, and so on), it is not necessary to add prefixes:

```
length as Int
this.length = length
```

Generic variables should have the same name as their type:

```
assignTopic (topic : Topic)

NOT assignTopic (value : Topic)
NOT assignTopic (aTopic : Topic)
NOT assignTopic (x : Topic)
```

Reduces complexity by reducing the number of terms and names that are used. Also, this makes it easier to deduce the type given a variable name only.

If, for some reason this convention does not seem to fit, it is a strong indication that the type name is badly chosen.

Non-generic variables have a role. These variables can often be named by combining role and type:

```
startingPoint as Point
centerPoint as Point
loginName as Name
```

All names should be written in English.

```
fileName NOT  nomArchivo
```

English is the preferred language for international development.

Variables with a large scope should have long names, and variables with a small scope can have short names.

Scratch variables used for temporary storage or indexes are best kept short. A programmer reading such variables should be able to assume that its value is not used outside a few lines of code. Common scratch variables are:

integers	i, j, k, m, n, i1, i2
booleans	b,b1,b2
reals	x,y,z,w
Strings	s,str

The name of the object is implicit and should be avoided in a method name.

```
line.length
```

```
NOT line.length
```

The latter seems natural in the class declaration but it proves superfluous in use, as shown in the example.

Specific Naming Conventions

Describes naming conventions

The term "compute" can be used in methods in which something is computed.

```
computeAverage valueSet
computeInverse matrix
```

Gives the reader the immediate clue that this is a potential time-consuming operation and, if used repeatedly, he/she might consider caching the result. Consistent use of the term enhances readability.

The term "find" can be used in methods where something is looked up.

```
findNearest Vertex
findMinElementIn matrix
NOT getMinElementIn matrix
```

Gives the reader the immediate clue that this is a simple look up method with a minimum of computations involved, *but* more expensive than a simple getter. Consistent use of the term enhances readability.

The term "initialize" can be used where an object or a concept is established.

```
initializeFontSetFor Printer
```

The American spelling of "initialize" should be used instead of the English "initialise". Abbreviation of "init" must be avoided.

"n" prefix should be used for variables representing a number of objects.

```
nPoints, nLines
```

The notation is taken from mathematics, where it is an established convention to indicate a number of objects.

If "number of" is the preferred statement, numberOf prefix can be used instead of just n. The **num** prefix must not be used.

The "No" suffix should be used for variables representing an entity number.

```
tableNo, employeeNo
```

The notation is taken from mathematics, where it is an established convention for indicating the number of an entity.

Complementary names should be used for complementary concepts or actions:

- add/remove
- create/destroy
- start/stop
- insert/delete
- increment/decrement
- old/new
- begin/end

- first/last
- up/down
- min/max
- next/previous
- old/new
- open/close
- show/hide
- get/set

Reduce complexity by symmetry, and avoid abbreviations in names wherever possible. For example, use `computeSalary`, rather than `compSal`.

Exception classes should be suffixed with `Exception`:

```
AccessException
```

Exception classes are really not part of the main design of the code. Naming them like this makes them stand out relative to the other classes.

Functions (methods returning an object) should be named after what they return, and procedures (void methods), after what they do.

Increase readability. Makes it clear what the unit should do and, especially, what it is not supposed to do. Again, this makes it easier to keep the code free from causing undesired side effects.

Negation

Negated boolean variable names must be avoided. For example, `isError` is better than `isNotError`.

The reason is that a readability problem arises when the logical not operator is used, and double negative arises. It is not immediately clear what `not isError` means.

Abbreviations

When considering the use of an abbreviation, think of which kind of word you are using. Common words listed in a language dictionary should almost never be abbreviated. Avoid writing *pt* instead of *point*, *comp* instead of *compute*, *init* instead of *initialize*, and so forth.

On the other hand, there are also domain-specific phrases that are more naturally known through their acronym or abbreviation. These phrases should be kept abbreviated. For example, don't write: *Hypertext Markup Language* instead of *HTML*, or *Central Processing Unit* instead of *CPU*.

Creating Statements

Variables




Tip: Variables should be initialized where they are declared and they should be declared in the narrowest possible scope. This ensures that variables are valid at any time.

Sometimes, it is impossible to initialize a variable to a valid value where it is declared. In these cases, it should be left uninitialized rather than initialized to some phony value.




Tip: Variables must never have dual meaning.

Enhance readability by ensuring that all concepts are uniquely represented. Reduce the chance of error by side effects.

 **Tip:** Variables should be kept alive for as short a time as possible.

By keeping the operations on a variable within a narrow scope, it is easier to control the effects and side effects of the variable.

Loops

 **Tip:** Loop variables should be initialized immediately before the loop.


```
ready as Bool = true
while ready do
    //Do something
end
```

Not:

```
ready as Bool = true


// Other stuff....

while ready do
    //Do something
end
```

 **Tip:** The use of break and exit should be minimized.

These statements should be used only if they prove to give a higher readability than their structured counterparts.

Conditionals

 **Tip:** Complex conditional expressions should be avoided. Introduce temporary boolean variables instead.

For example, the following code:

```
if (elementNo < 0) or (elementNo > maxElement)
    or elementNo = lastElement then
    //Do something
end
```

Should be replaced by:

```
isFinished as Bool = (elementNo < 0) or (elementNo > maxElement)
isRepeatedElement as Bool = elementNo = lastElement

if isFinished or isRepeatedElement then
    //Do something
end
```

By assigning boolean variables to expressions, the program gets automatic documentation. The construction will be easier to read and debug.


 **Tip:** The nominal case should be put in the if-part and the exception in the else-part of an if statement.

```
isError as Bool = readFile (fileName)

if not isError then
    //Do something
else
    //Handle the error
```

```
end
```

Makes sure that the exceptions do not obscure the normal path of execution. This is important for both the readability and performance.

 **Tip:** Executable statements in conditionals must be avoided.


```
file = openFile (fileName, "w")
if file /= null then
    //Do something
end
```

Not:

```
if (file = openFile (fileName, "w")) is not null then
    //Do something
end
```

Conditionals with executable statements are very difficult to read. This is especially true for new programmers.

Miscellaneous


 **Tip:** The use of magic numbers in the code should be avoided.

If the number does not have an obvious meaning by itself, the readability is enhanced by introducing a named constant instead.

```
d = s / SECONDS_PER_DAY
timeout = DEFAULT_TIMEOUT
```

Not:

```
d = s / 86400
timeout = 1000
```

 **Tip:** Real and Decimal constants should always be written with a decimal point and at least one decimal:

```
total = 0.0
speed = 3.0

sum = (a + b) * 10.0;
```

Not:

```
total = 0;
speed = 3;

sum = (a + b) * 10;
```

This emphasizes the different nature of integer and floating point numbers, even if their values might happen to be the same in a specific case.

Moreover, as in the last example above, it emphasizes the type of the assigned variable (sum) at a point in the code where this might not be evident.



Tip: Real and Decimal constants should always be written with a digit before the decimal point.

```
total as Real = 0.5f
```

Not:

```
total as Real = .5f
```

The number and expression system in Studio is borrowed from mathematics and one should adhere to mathematical conventions for syntax wherever possible. In addition, 0.5 is much more readable than .5. There is no way it can be mixed with the integer 5.

Code Layout and Comments

Indentation

Indentation enhances readability, particularly within loops and conditional statements. Under PBL, standard practice is to indent four spaces per level.

```
for i in 1..10 do
  a[i] = 0
end
```

To facilitate this, the Tab key inserts four spaces within the code editor.

The if-then-else class of statements should have the following form:

```
if condition then
  // statements
end
```

or:

```
if condition then
  statements
else
  statements
end
```

or:

```
if condition then
  statements
elseif condition then
  statements;
else
  statements;
end
```

Not:

```
if condition
then
```

```

    statements
end

```

And not:

```

if condition then statements end

```

The chosen approach is considered to be better since each part of the if-else statement is written on separate lines of the file. This should make it easier to manipulate the statement, for instance, when moving else clauses around.

A bounded loop should have the following form:

```

for i in set do
    //statements
end

```

An unbounded loop should have the following form:

```

while condition do
    //statements
end

```

A Multipath conditional statement should have the following form:

```

case condition
when ABC
    // statements
when DEF
    // statements
else
    // statements
end

```

A Compound statement should have the following form:

```

do
    statements;
on e as Exception exception
    statements
end

```

or:

```

do
    statements;
on e as Exception
    statements
on exit
    statements
end

```

White Space in Expressions

- Conventional operators should be surrounded by a space character.
- Reserved words should be followed by a white space.
- Commas should be followed by a white space.
- Colons should be followed by a white space.

- Semicolons for statements should be followed by a space character.

```
a = (b + c) * d
```

These rules make the individual components of the statements stand out and enhance readability. It is difficult to give a complete list of the suggested use of white space in Studio code. However, the examples shown above should give a general idea.



Note: Logical units within a compound statement should be separated by one blank line. This enhances readability by introducing a white space between logical units of a block.

Comments



Remember: Tricky code should not be commented on but rewritten.

In general, the use of comments should be minimized by making the code self-documenting through appropriate name choices and an explicit logical structure.



Tip: All comments should be written in English.

In an international environment, English is the preferred language.



Tip: Minimize the use of multi-line comments.

```
// Comment spanning
// more than one line
```

Since nested multi-line comments are not supported, using single line comments ensures that it is always possible to comment out entire sections of code for debugging purposes, among others.



Remember: Comments should be indented in relation to their position in the code.

```
while true do
  // Do something
  something()
end
```

Not:

```
while true do
  // Do something
  something()
end
```

This is to prevent comments from breaking the logical structure of the program.

Files



Tip: File content must be kept within 100 columns.



Tip: The incompleteness of split lines should be made obvious.

```
totalSum = a + b + c +
          d + e

function (param1, param2,
          param3)
```

```
passingText ("Long line split" +
            "into two parts.")
```

Split lines are required when a statement becomes too wide to read comfortably, or exceeds the column limit given above. It is difficult to provide strict rules for how lines should be split, but the examples above can serve to illustrate the guidelines shown below.

In general it is good practice to:

- Break after a comma.
- Break after an operator.
- Align the new line with the beginning of the expression on the previous line.

Embedded SQL Overview

Introduces embedded SQL

Studio supports embedded SQL (ANSI-92 Entry Level), written directly in the code. This section describes the supported syntax and provides some examples.



Note: In the syntax and examples in the topics of this section, SQL keywords appear in all-caps. This is an SQL convention and it is not required by Studio. However, it is a useful convention which helps differentiate SQL commands from regular code.

For further information about introspecting SQL components, refer to the [SQL Database](#) on page 183 external resource reference.

SQL Operators

Describes common SQL operators

SQL operators allow you to control query selection criteria and the values returned. The following is a list of operators supported in PBL.

LIKE operator

The **LIKE** operator searches for strings that match a specific pattern. The percent sign "%" matches any string and the underscore "_" matches any single character. The following example returns any row where `fname` starts with "J":

```
for each e in
  SELECT *
  FROM employees
  WHERE fname LIKE "J%"
do
  // do something here
end
```

Concatenation operator (||)

In SQL statements, the `||` operator is used to concatenate two values of any type. For example:

```
for each e in
  SELECT lname || ", " || fname AS fullname
  FROM employees
do
  display "full name: " + e.fullname
end
```



Note: In PBL and Java style, the `||` symbol means "or" and it is used in conditional expressions. For further information, please see [Logical Operators](#) on page 276.

IN operator

The IN operator matches a column value against a set of literal values:

```
column_name IN (<value1>, <value2>, ...)
```

For example:

```
for each e in
  SELECT lname, fname
  FROM employees
  WHERE salary IN (20000, 25000, 30000)
do
  display "name: " + e.lname
end
```

This statement is equivalent to the following:

```
for each e in
  SELECT lname, fname
  FROM employees
  WHERE salary = 20000 or salary = 25000 or salary = 30000
do
  display "name: " + e.lname
end
```

IS operator

The IS operator locates a record that does or does not have a null value for a particular column:

```
<column_name> IS [NOT] NULL
```

For example:

```
for each e in
  SELECT lname, fname
  FROM employees
  WHERE address IS NOT NULL
do
  display "name: " + e.lname + ", address: "
    + e.address
end
```

BETWEEN Operator

The BETWEEN operator allows you to select records that are between two values:

```
<column_name> [NOT] BETWEEN <value1> AND <value2>
```

The expression `a BETWEEN b AND c` is equivalent to `a >= b AND a <= c`. For example:

```
for each e in
  SELECT lname, fname
  FROM employees
  WHERE salary BETWEEN 20000 AND 30000
do
  display "name: " + e.name + ", salary: "
    + e.salary
end
```

SQL Keywords

Lists SQL keywords

Some words in an SQL statement have a special meaning and cannot be used as regular identifiers. These keywords include the following:

```
ALL
AND
AS
ASC
AVG
BETWEEN
BY
COUNT
DELETE
DESC
DISTINCT
FROM
GROUP
HAVING
IN
INSERT
INTO
IS
LIKE
MAX
MIN
NULL
OR
ORDER
SELECT
SET
SUM
UPDATE
VALUES
WHERE
```



Note: These keywords can be used as regular identifiers *outside* SQL statements. Also, in compliance with SQL standards, keyword case is ignored within SQL statements, so `SELECT`, `Select`, and `select` are all accepted.

INSERT Statement

Describes the SQL `INSERT` statement and common options

The `INSERT` statement is used to add one or more rows to a table. Columns may be specified by position or by name. If specified by position, the order of the values must match the position of the corresponding columns in the table. If columns are specified by name, they may be listed in any order.

In general, it is recommended practice to specify columns by name, since the table may be modified and column positions may change, breaking your code. Name references are position independent, and a `SELECT` statement with column names is more explicit and easier to read.

Columns not specified in the column list are set to their default values or to null. If the column value cannot be set to null (if it is defined as `NOT NULL` in the database) and it has no default value, an error will result and the `INSERT` will fail.



Note: In the syntax and examples below, SQL keywords appear in all caps. This is an SQL convention and it is not necessary in Studio. However, it is a useful convention to differentiate SQL statements from regular code.

There are two alternative ways to supply data for the `INSERT` operation. One is to specify a list of values directly:

```
INSERT INTO <table_name> [(<col_name1>, <col_name2>, ...)]
VALUES (<value1>, <value2>, ...)
```

The other alternative is to specify a `SELECT` query. In this case, the rows obtained from this query will be inserted into the table, so long as the column values from the query match the column values required by the `INSERT`:

```
INSERT INTO <table_name> [(<col_name1>, <col_name2>, ...)]
<select statement>
```

The following example specifies a set of values and will insert one row:

```
INSERT INTO employees(fname, lname, salary)
VALUES ("John", "Smith", 25000)
```

The value set can also include expressions:

```
firstname = "John"
salary = 20000

INSERT INTO employees(fname, lname, salary)
VALUES(firstname, "Smith", salary + 5000)
```

The following example uses `INSERT` with a `SELECT` statement:

```
INSERT INTO students
SELECT *
FROM employees
WHERE salary > 30000
```

UPDATE Statement

Describes purpose and syntax of the `UPDATE` statement

The `UPDATE` statement modifies a set of field values in each row which satisfies a given search condition. If no row matches the condition, the `UPDATE` will have no effect.



Note: In the syntax and example sections below, SQL keywords appear in all caps. This is an SQL convention and it is not required by Studio. However, it is useful to help differentiate SQL commands from regular code.

Syntax:

```
UPDATE <table_name>
SET <column_name1> = <value-expression1>,
    <column_name2> = <value-expression2>,
    ...
[WHERE <condition>]
```

For example, the following `UPDATE` increases the salary by 10% for all employees who earn less than \$25,000:

```
UPDATE employee
SET salary = salary * 1.1
WHERE salary < 25000
```



Note: If the `WHERE` condition is not specified, *all* the rows will be updated.

DELETE Statement

Describes purpose and syntax of the SQL `DELETE` statement

`DELETE` removes all rows that satisfy a given condition from a table:

```
DELETE FROM <table_name>
[WHERE <condition>]
```

The following example deletes all employees whose first name is "John" and last name is "Smith":

```
DELETE FROM employees
WHERE fname = "John" and lname = "Smith"
```

SELECT Statement

Describes the SELECT statement and basic query options

The SELECT statement finds and retrieves rows, columns, and derived values from one or more tables of a database. The SELECT statement is flexible, with many options, and accepts column specifications, search conditions, ordering instructions, and other parameters. The powerful and complex SELECT statement is a core feature of SQL and a thorough description of it well exceeds the scope of this document. This section outlines basic SELECT syntax and clauses which suffice for most simple queries.

Syntax

A SELECT operation may retrieve one or many records, so in PBL it is commonly placed within a for each loop, as follows:

```
for each <variable> in
  SELECT [DISTINCT | ALL] <column1>, <column2>, ...
  FROM <table1>, <table2>, ...
  [WHERE <condition>]
  [GROUP BY <grouping-column1>, <grouping-column2>, ...]
  [HAVING <group-selection-condition1>]
  [ORDER BY <ordering-col1> [ASC | DESC],
            <ordering-col2> [ASC | DESC], ...]
do
  // ...
end
```

The columns to be retrieved are delimited by commas or, alternatively, an asterisk (*) may be used to retrieve all columns from every table queried. It is recommended that you specify each column you need rather than retrieving all of them, as this will improve performance, substantially if the table is large and contains many columns you do not need.

You can request that a column be returned under an alias by using the AS clause:

```
SELECT clientId, fn AS firstName FROM clients
```

This selects `clientId` and `fn` from the database, but it will return the columns as `clientId` and `firstName`. In this way, you will be able to access the column using `firstName` in your code rather than `fn`, so it will be easier to read.

A column can also be an expression or an aggregate function. Aggregate functions combine values from every row into a single value, such as a sum or an average, and are discussed below.

You may use the ORDER BY clause to sort the results of the query according to a given value. Ordering may be ascending (ASC), or descending (DESC). Ascending order is the default and need not be specified. You can order by one or more columns, delimited by commas. Sorting is first done on the first ORDER BY column, and subsequent ORDER BY columns are used when the previous column contains equal values.

The following example displays the name of every employee with a salary higher than 25,000:

```
for each e in
  SELECT *
  FROM employees
  WHERE salary > 25000
  ORDER BY lname
do
  display "employee name: " + e.lname + ", " + e.fname
end
```

Using Aggregate Functions

The following example selects the maximum salary in the employee table using the MAX function. The `row.1` term is used to specify the first column. The variable `salary` is used to store the result of the maximum salary in the table:

```
for each row in
  SELECT MAX(salary)
  FROM employees
do
```

```
    salary = row.1  
end
```

The following example returns the average salary of employees grouped by `depnumber`, but does not return employees where `depnumber` is equal to 3 or 4, or cases where 5 or fewer employees have the same `depnumber` value:

```
for each e in  
    SELECT depnumber, COUNT(*), AVG(salary)  
    FROM employees  
    WHERE depnumber != 3 and depnumber !=4  
    GROUP BY depnumber  
    HAVING COUNT(*) > 5  
    ORDER BY depnumber  
do  
    // ...  
end
```

Stored Procedures

Describes support for stored procedures

Any procedure that you have developed in your database system (such as Oracle or Microsoft SQL Server) is added to the catalog during introspection. The stored procedure can be treated as a method and used in your code. Any procedure that uses vendor specific features, such as rowtype in Oracle, is not supported. Only standard SQL procedures are added to the catalog.