Copyright

© 2017 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

Business Making Progress, Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, Deliver More Than Expected, Icenium, Kendo UI, Making Software Work Together, NativeScript, OpenEdge, Powered by Progress, Progress, Progress Software Developers Network, Rollbase, SequeLink, Sitefinity (and Design), SpeedScript, Stylus Studio, TeamPulse, Telerik, Telerik (and Design), Test Studio, and WebSpeed are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. AccelEvent, Analytics360, AppsAlive, AppServer, Arcade, BravePoint, BusinessEdge, DataDirect Spy, DataDirect SupportLink, DevCraft, DigitalFactory, Fiddler, Future Proof, High Performance Integration, JustCode, JustDecompile, JustMock, JustTrace, OpenAccess, ProDataSet, Progress Arcade, Progress Profiles, Progress Results, Progress RFID, Progress Software, ProVision, PSE Pro, SectorAlliance, Sitefinity, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, WebClient, and Who Makes Progress are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

Please refer to the Release Notes applicable to the particular Progress product release for any third-party acknowledgements required to be provided in the documentation associated with the Progress product.

The Release Notes can be found in the OpenEdge installation directory and online at: https://community.progress.com/technicalusers/w/openedgegeneral/1329.openedge-product-documentation-overview.aspx.

For the latest documentation updates see OpenEdge Product Documentation on Progress Communities: (https://community.progress.com/technicalusers/w/openedgegeneral/1329.openedge-product-documentation-overview.aspx).

December 2014

Last updated with new content: Release 11.5.0

Updated: 2017/03/24
# Table of Contents

## Preface
- About this documentation
- User types
- Information on documentation
- Conventions used in this manual
- Product support contact information

## Chapter 1: OpenEdge Business Process Server overview
- Business Process Server components
- How Business Process Server works
- Business Process Server user types

## Chapter 2: Introducing BPM Events
- BPM Events principles
- BPM Events rules

## Chapter 3: BPM Events tutorial
- Using rules to control Business Process Server processes
- Using rules to correlate business process events
- Using rules to monitor and report on business processes
- Using rules to populate infopads
- Using rules to modify monitoring reports
- Using rules to monitor an application
  - Defining a report
  - Events generated by the application
  - More about the value field
  - Understanding event-maps
  - Building the report
- Using rules to correlate multiple events
- Using rules to generate alarms
- Using rules to synchronize Business Process Server processes

## Chapter 4: Understanding BPM Events
- The event model
- Attributes
- Event sources
Chapter 5: The rule language..........................................................................................57

Rule structure.....................................................................................................................58
Syntactical conventions and limitations.................................................................................58
Event variables and expressions............................................................................................59
Common variables, constants, and their scope.........................................................................60
Rule condition as a logical expression.....................................................................................62
Priority and parentheses.........................................................................................................63
Relational expressions and operands.......................................................................................63
Comments in rule language.......................................................................................................65
Rule modules and rule groups.................................................................................................66
Rule modules of an application................................................................................................66
Rule groups.............................................................................................................................68
Current rule base.....................................................................................................................68
Adding and removing rules dynamically................................................................................68
Dynamic loading and unloading of rule modules....................................................................69
Dynamically enabling and disabling rules...............................................................................69
Operational semantics of rules................................................................................................69
Event correlation....................................................................................................................70
Event filtering........................................................................................................................71
General semantics of rule loading and rule execution...........................................................72
Time operations and expressions............................................................................................74
Date expressions....................................................................................................................74
Extracting elements of a date....................................................................................................76
Date conversion functions........................................................................................................76
Time expressions.....................................................................................................................77
Date functions........................................................................................................................77
Business time..........................................................................................................................78
Controlling rule execution........................................................................................................79
Avoiding unnecessary or unwanted rule executions..................................................................79
Canceling multiple evaluations of a rule..................................................................................80
Disabling events as rule triggers.............................................................................................80
Predefined rule actions.............................................................................................................81
println(): Printing values to a file..........................................................................................82
generate(): Generating internal events ................................................................. 84
postDB(): Posting an event into the database event queue ................................... 85
postJMS(): Posting an event into a JMS queue ...................................................... 85
sendMail(): Sending e-mail automatically ........................................................... 85
schedule(): Scheduling a future event ................................................................. 86
Using the event scheduler .................................................................................... 87
 unschedule(): Removing a scheduled event from the scheduler ....................... 88
Discarding infopads and events .......................................................................... 89
Enabling and disabling rules .............................................................................. 91
Controlling the actions of a rule ........................................................................ 92
Syntactic structure of rule actions ..................................................................... 92
Functional action statements ........................................................................... 93
Declaration action statements .......................................................................... 93
Assignment action statements ........................................................................... 93
Conditional action statements .......................................................................... 94
Compound action statements .......................................................................... 95
Iterative action statements .............................................................................. 95
Defining and using global functions................................................................... 96

**Chapter 6: Persistent structures......................................................................... 99**
Infopads .............................................................................................................. 99
Defining Infopads, cells, and slots .................................................................... 100
Creating infopads .............................................................................................. 100
Using infopads ................................................................................................. 104
Persistent maps .................................................................................................. 109
Defining persistent maps, keys, and values ..................................................... 110
Creating persistent map instances .................................................................. 110
Defining persistent map ................................................................................... 110
Accessing persistent maps and map instances .............................................. 111
Associating keys to persistent maps ............................................................... 111
Discarding persistent maps and map instances from memory .................... 111
Using persistent maps ....................................................................................... 112
Updating attributes of a given key ................................................................. 112
Using various operators to perform actions .................................................... 112
Iterating over persistent map keys ................................................................. 113

**Chapter 7: Event channels............................................................................. 115**
The database event channel (DEC) ................................................................. 115
postDB predefined rule action ........................................................................ 116
Configuring the database event channel ....................................................... 116
The JMS event channel .................................................................................... 116
JMS message adapter ...................................................................................... 117
postJMS predefined rule action ...................................................................... 118
# Chapter 8: XML messages

- Namespace declaration .................................................................................................................. 120
- XML type ........................................................................................................................................ 121
- XML data field access syntax .......................................................................................................... 121
  - Dot notation for XML data fields .................................................................................................. 121
  - Bracket notation for XML data fields .......................................................................................... 122
  - Function notation for XML data fields ......................................................................................... 122
- XML data conversion ....................................................................................................................... 122
  - XPath rule condition .................................................................................................................... 122
  - XPath query expression/statement ............................................................................................... 123
- New syntax for iteration ..................................................................................................................... 124
  - For each syntax for iteration .......................................................................................................... 124

# Chapter 9: Accessing Java objects and EJB components

- Rule language extensions ................................................................................................................ 128
  - Mapping data types ..................................................................................................................... 128
  - Object instantiation ...................................................................................................................... 129
  - Static method invocation or property access .............................................................................. 129
  - Instance method invocation or property access .......................................................................... 129
  - Exception catching ....................................................................................................................... 130
  - Exception throwing ...................................................................................................................... 130
  - Using complex Java objects in rules ............................................................................................ 130
  - Using external java objects in rules ............................................................................................. 131
- Searching class files ....................................................................................................................... 132
- Mixing two separate instance of a class ............................................................................................ 132
- EJB component invocation ............................................................................................................. 132
- Dynamic loading ............................................................................................................................. 133
  - Using modified class implementation in BPM Events ................................................................ 133
  - Using modified class implementation in BPM Events when signature of the Java class is changed ....................................................................................................................... 133

# Chapter 10: Administering and operating BPM Events

- The rule development cycle ............................................................................................................. 136
  - Rule compiling .............................................................................................................................. 136
  - Rule Compiler command options ................................................................................................ 137
  - Setting up and monitoring the database persistence tables .......................................................... 137
  - Starting and operating the BPM Events server ............................................................................ 138
  - Loading rules ............................................................................................................................... 141
  - Unloading rules ............................................................................................................................ 142
- Troubleshooting cases ..................................................................................................................... 142
  - Fixing with full recovery .............................................................................................................. 143
  - Fixing by upgrading application .................................................................................................. 143
Fixing with recovery and partial rule upgrade.................................................................144
BPM Events Admin: BPM Events administrator utility..................................................144
Functions.........................................................................................................................144
Starting BPM Events Admin..........................................................................................145
BPM Events Admin main menu.......................................................................................145
Server Monitor menu......................................................................................................145
Loader menu....................................................................................................................146
Rule Manager menu.......................................................................................................147
Persistent Structure Monitor menu................................................................................147
Event Manager menu......................................................................................................149
Scheduler menu............................................................................................................150
File Manager menu.......................................................................................................150
JMS menu.......................................................................................................................151
Event management........................................................................................................151
Processing external and internal events.........................................................................151
Transactional and serial events....................................................................................151
Event scheduling..........................................................................................................152
Event caching and indexing..........................................................................................152
Internal events handling..............................................................................................153
Cache management from the application......................................................................153
Persistence and recovery..............................................................................................154
Infopad persistence........................................................................................................154
Database access failure handling................................................................................155
Keeping the recovery backup up-to-date......................................................................155
Loading and recovering applications..........................................................................156
BPM Events configuration parameters..........................................................................157

Chapter 11: Using the purchase order rule wizard .........................................................161
Applying the purchase order template..........................................................................161
Using the purchase order template..............................................................................166

Chapter 12: Rule syntax..................................................................................................167
Language tokens (Reserved keywords)..........................................................................167
Rule module syntax.......................................................................................................170

Chapter 13: Predefined functions and operators..........................................................175

Chapter 14: Recovery of global variables and file handling.........................................181
Global variables and values..........................................................................................181
Recovering global variables.........................................................................................182
File reopening after recovery.......................................................................................182
Glossary.......................................................................................................185
Preface

For details, see the following topics:

- About this documentation
- User types
- Information on documentation
- Conventions used in this manual
- Product support contact information

About this documentation

This guide is part of the documentation set for Progress OpenEdge Business Process Server.

User types

Progress OpenEdge Business Process Server is a business process management system that can be used by the following types of users:

<table>
<thead>
<tr>
<th>User type</th>
<th>Responsibilities</th>
</tr>
</thead>
</table>
### Information on documentation

This documentation includes information for the entire range of Progress OpenEdge Business Process Server users. In the following table, we recommend the guides that are most relevant to each type of user.

<table>
<thead>
<tr>
<th>If you are the …</th>
<th>Read the …</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application User</td>
<td><em>Business Process Portal User’s Guide</em></td>
</tr>
<tr>
<td></td>
<td><em>First Steps Guide</em></td>
</tr>
<tr>
<td></td>
<td><em>Terminology Guide</em></td>
</tr>
<tr>
<td>Manager</td>
<td><em>Business Process Portal Manager’s Guide</em></td>
</tr>
<tr>
<td></td>
<td><em>Business Process Portal User’s Guide</em></td>
</tr>
<tr>
<td></td>
<td><em>Terminology Guide</em></td>
</tr>
</tbody>
</table>
Conventions used in this manual

This document uses the following conventions and terminology notations.

<table>
<thead>
<tr>
<th>Convention (styles and terms)</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bold</strong></td>
<td>Indicates titles of command buttons, check boxes, options, lists, dialog boxes, and portal page names.</td>
</tr>
<tr>
<td><code>file path</code></td>
<td>Indicates folder paths and file names.</td>
</tr>
</tbody>
</table>
### Convention (styles and terms) and Purpose

<table>
<thead>
<tr>
<th>Convention (styles and terms)</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>italic</em></td>
<td>Indicates book titles.</td>
</tr>
<tr>
<td><em>monospace</em></td>
<td>Represents code segments or examples.</td>
</tr>
<tr>
<td>backslash &quot;&quot;</td>
<td>Indicates the path in Windows environment. For UNIX environment, replace with forward slash &quot;/&quot;</td>
</tr>
<tr>
<td>OEBPS_HOME or %OEBPS_HOME%</td>
<td>Represents the installation folder of Business Process Server, C:\Progress\OpenEdge\oebpm\server.</td>
</tr>
<tr>
<td>STUDIO_HOME or %STUDIO_HOME%</td>
<td>Represents the installation folder of OpenEdge BPM components, C:\Progress\OpenEdge\oebpm\studio.</td>
</tr>
<tr>
<td>JBOSS_HOME or %JBOSS_HOME%</td>
<td>Represents the installation folder of JBOSS server, C:\Progress\OpenEdge\oebpm\jboss.</td>
</tr>
</tbody>
</table>

### Product support contact information

If the product documentation does not provide a solution to your specific issue, or if you need clarification on the issue, then contact our Product Support team. You can contact the team through the Internet, telephone, or postal mail, as per the details provided in the following table.

**Table 1: Product Support Contact Information**

<table>
<thead>
<tr>
<th>To contact by</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web site</td>
<td><a href="http://progresslink.progress.com/supportlink">http://progresslink.progress.com/supportlink</a></td>
</tr>
<tr>
<td></td>
<td>If you are an existing customer, then you can log in to the above site for product support. If you are a first time user, then you need to create an account first.</td>
</tr>
</tbody>
</table>
| Telephone<sup>1</sup> | 1-781-280-4999 for US, Latin America and Canada  
|                     | 1-781-280-4543 for the Product Support Fax Line |
| Postal Address<sup>1</sup> | Progress Software Corporation  
|                     | 14 Oak Park Drive  
|                     | Bedford, MA 01730, USA. |

To enable us to quickly answer your questions, please provide the following information:

- Your name, installation site address and the license key for Business Process Server software.
- Your Business Process Server version and build number.

---

<sup>1</sup> For support telephone numbers and offices in your region, visit the support web site above. This contact information is for customer support only.
• Your operating system, application server and browser, with version and service pack details, if any.
• Your database management system and version, and information on JVM and JDBC used.
OpenEdge Business Process Server overview

Progress Software Corporation is a leading global provider of automated business process management solutions. The company's product, Progress OpenEdge Business Process Server (henceforth referred to as Business Process Server or BPServer), is a comprehensive business process management platform, which enables companies to quickly transform their business processes into flexible and manageable Web applications, distributed over intranets, extranets, and the Internet.
Business Process Server addresses every stage in the business life cycle: define, integrate, publish, monitor, analyze, improve, and control. By adopting an end-to-end approach, Business Process Server incorporates all the key elements required to meet the ever-changing demands of e-business while ensuring e-business success. By providing integrated management tools, Business Process Server lets you monitor operations proactively, modifying automated processes dynamically based on changing external operations online. An overview of a typical automated business process management solutions is shown in Figure 1: Business Process Server overview on page 18.

**Figure 1: Business Process Server overview**

For details, see the following topics:

- Business Process Server components
- How Business Process Server works
- Business Process Server user types
Business Process Server components

Business Process Server is a suite of integrated components that enables you to easily build intranet, extranet, and Internet applications and manage your e-business. Business Process Server consists of the following components as in Figure 2: Business Process Server components on page 19:

Figure 2: Business Process Server components

Table 2: Business Process Server components

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Process Portal Home</td>
<td>The Home module of Business Process Portal is the primary interface for application users, enabling them to interact with Business Process Server applications. Users can complete entries to various tasks and applications, and link to the support infrastructure required to complete these tasks.</td>
</tr>
<tr>
<td>Business Process Portal Management</td>
<td>The Management module enables managers to query, report on, and control processes and resources, visible only to the managers.</td>
</tr>
<tr>
<td>Business Process Portal Administration</td>
<td>The Administration module enables Business Process Server Administrators to modify configuration parameters, manage user or group access control, and install or uninstall Business Process Server applications.</td>
</tr>
<tr>
<td>Component</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>Web services</td>
<td>This component allows application developers to: publish their applications as Web services, and find and convert other available Web services on the Internet into Business Process Server applications.</td>
</tr>
<tr>
<td>BPM Webflow</td>
<td>This component provides a framework for developing and implementing Web-enabled workflow applications.</td>
</tr>
<tr>
<td>Progress Developer Studio for OpenEdge</td>
<td>This is the primary application development tool for Business Process Server, which provides an Eclipse-based integrated development environment in which users can create BPM projects, processes, Web applications, and rule files.</td>
</tr>
<tr>
<td>Business Process Modeler</td>
<td>This tool is used to design templates for basic business processes, and to run simulations of processes and individual worksteps.</td>
</tr>
<tr>
<td>BP Server</td>
<td>This is a flexible, lightweight, scalable workflow process engine for intranets, extranets, and the Internet.</td>
</tr>
<tr>
<td>BPM Events</td>
<td>This open, event-driven rule engine is used to formulate and enforce policies in business applications.</td>
</tr>
</tbody>
</table>
How Business Process Server works

The following figure provides an overview of the interaction between Business Process Server components.

Figure 3: How Business Process Server works

The following explanations correspond to the labels shown in the above figure, and describe how the components operate.

1. Progress Developer Studio for OpenEdge and Business Process Modeler provide an integrated development environment (IDE) for Business Process Server, where you can design and publish business processes. The application developer designs a process template (with the *.spt or *.swt extension) in the IDE that reflects the business flow and other business process requirements. Business rules for the process template can be defined using the Rule Editor, a BPM Events component that is launched with Progress Developer Studio for OpenEdge.


3. Once the process template is defined, Business Process Server Administrators use the Administration module to install the business process on the BP Server. Administrators can also configure Business Process Server components, manage user or group access control, and publish Business Process Server applications as Web services. Once installed, users access applications through servlets that pass the requests over an RMI/IIOP connection to the BP Server within an EJB Container.
4. The EJB Container provides a runtime environment that executes and manages Java-based program components that run on the server side of a client/server network. Within the EJB Container are the BP Server and BPM Events server.

5. The BP Server writes events to event tables in the database. Each Business Process uses JDBC to connect to database server as well as store events in the database. Within the BP Server, BPM Process Store uses JDBC to connect to the database server process and retrieve the events deposited by the BP Server process. BPM Process Store interprets the events and populates the process tables. These populated tables are used by Business Process Portal modules.

6. Once the process template is installed as a Business Process Server application, application users can use the Home module to do the following:
   - Access applications
   - Obtain information to perform their tasks
   - Launch the application to start process instances from the BP Server

7. Once the process template is installed as a Business Process Server application, managers can use the Management module (if they have access privileges) to monitor execution of process instances and create reports. Servlets receive requests from managers and pass them onto the BP Server over an RMI/IIOP connection. Managers use the Report Builder to define management reports that retrieve information through JDBC to the database server.

8. BPM Events is a rule-based event or message processing server that loads application rules and executes them against the BP Server and/or external events or messages. This server persists data in the database for recovery and with the help of JDBC connects to the database.

9. Managed Adapters exchange information between Business Process Server applications and external applications by converting Business Process Server-specific protocol to the protocol of an external system such as a database or ERP system. When users add a Managed Adapter to a work step, they can define complex mapping between Business Process Server dataslots and adapter inputs or outputs of the external application. At runtime when the work step is executed, the Managed Adapter sets the adapter inputs and configuration, and maps the outputs to the appropriate output dataslots.

10. BPM Webflow is a run-time component that executes the presentation flows. This component provides a Model, View, Controller (MVC) paradigm for developing presentation flow-based applications and executing them in a Web container.

11. Business Process Servers Web services component allows BP Server applications to be published as Web services.


Business Process Server user types

There are four user types within Business Process Server, Application users, Managers, Application developers, Business Process Server administrators.

User types

Each Business Process Server user type is defined below:

- **Application users** — Application Users use Business Process Server applications to coordinate specific business tasks with another department within their company, with another company within their organization,
and/or with a business partner in another organization. The Home module in Business Process Portal serves as the primary interface in which Application Users run Business Process Server applications.

- **Managers** — Managers are typically experts in a particular business domain, such as quality assurance or human resources. They might need to work with managers from other groups in automating some of the business procedures that these groups share. The Management module in Business Process Portal serves as the primary Business Process Server interface for business managers to coordinate and integrate business processes, enabling them to exchange information with one another, and to share functionality over such standard communication channels as the Internet or e-mail.

- **Application developers** — Application developers are responsible for analyzing business processes and developing interfaces associated with creating tasks or processes. Application developers are often not domain experts themselves, but work closely with Managers to define business processes and determine the requirements of an application. Application developers use Progress Developer Studio for OpenEdge or Business Process Modeler to define the business process; the resulting process template file is tested, simulated, published, and run as a Business Process Server application.

- **Business Process Server administrators** — Business Process Server administrators are responsible for configuring Business Process Server components, managing user or group profiles and access control, and installing or uninstalling Business Process Server applications. The Administration module in Business Process Portal serves as the primary interface for Business Process Server Administrators to administer applications.

All Business Process Server user types can communicate by using one or more Business Process Server applications. They can also communicate with external applications.
Introducing BPM Events

BPM Events, the rule engine of Business Process Server, enables you to create rules that manage business processes and messages. The BPM Events engine is implemented in Java and runs on any platform supporting a Java Virtual Machine (UNIX, Windows, Linux, etcetera) or native compiler. It has a small footprint and is easily installed and loaded.

In BPM Events, rules in online applications serve the following associated objectives:

- Monitoring business messages/events and generate dynamic reports and alarms.
- Controlling business processes.

The integration of report objects with rule processing is a unique feature of BPM Events.

BPM Events rules implement a wide range of actions, including:

- **Generating new messages/events** — BPM Events can either immediately generate an event (for example, starting or terminating a workstep generates an event) or create an advanced time construct scheduled for a later date (for example, defining business time, time intervals, or scheduling). BPM Events defines rules with a rule language that can detect and describe complex patterns of events spanning long periods of time. The rule language uses an indexed event cache to store events of interest, and relies on event storage and queuing in a database.

- **Generating e-mails or alarms** — BPM Events can trigger the sending of an e-mail or an alarm when specified conditions are met.

- **Calling external adapters and performers** — Adapters to third-party event sources can be easily added and even plugged in at run-time. BPM Events is designed to work with external event handlers and with external management systems that operate efficiently in their particular domain. It can receive and send events from and to Enterprise Management Systems. For example, using an adapter to Unicenter TNG, it can obtain and monitor data from an Unicenter SNMP agent, as well as update MIB variables in a Unicenter MIB database.
• **Providing native JMS support** — BPM Events has a built-in "adapter" that communicates with JMS. The JMS event channel contains JMS listeners, which listen to JMS queues, and map the JMS messages into BPM Eventss.

• **Updating and saving a report** — Special emphasis has been given to the dynamic aspect of online operations: rules and reports can be dynamically created, loaded, saved and modified without shutting down the engine. The recovery function allows for restoring the latest valid state at any time, with minimal overhead on operations.

• **Managing the event cache for improved performance** — Optimization features, including indexing of rules and cross-indexing of events for fast correlation, provide a thorough scalability unequalled by other rule systems less focused on event processing.

• **Supporting failure recovery** — Given the mission-critical nature of business events, BPM Events now contains an advanced failure recovery technology. Whether shutdown occurs normally or accidentally, the BPM Events engine does not lose critical data. Its internal state can be entirely recovered from the repository stored in the database. When restarted, the engine reprocesses all pending events, after having recovered its last stable internal state.

For details, see the following topics:

• **BPM Events principles**

• **BPM Events rules**

### BPM Events principles

BPM Events is designed around the following principles:

• **Business events.** A business event is an object that stores various business objects ranging from a disk failure to a purchase order or a customer record. In order to monitor distributed online systems, we developed a general definition of a business event that captures both the status of the resource and the business transactions it performs. The data model of business events is very simple, yet general enough to contain various business objects and their structure (meta-data).

• **Event-driven Business rules.** In online businesses, business rules are enforced dynamically, and react to business events by selecting the correct action—these actions include sending alarms, building reports, triggering external programs, and generating or scheduling new events. These business rules must handle any type of business event. They also perform advanced correlation of event sequences that can be distant over time.

• **Dynamic business metrics and reports.** In online businesses, the distinction between monitoring and reporting is no longer clear-cut. Reports are now generated in real-time on live business events, as opposed to off-line reports on archived operation data. Dynamic reports act as an extension of monitoring, and provide at any time a snapshot of resources and business transaction flows. Reports are also business objects that in turn are dynamically analyzed by rules, in order to automatically detect abnormal values and react in time.

### BPM Events rules

Rules created in BPM Events interact with BP Server, the Business Process Server’s business process engine, to:
• Capture events generated by the execution of the processes.

• Control the execution of these processes.

These rules enable users to:

• **Monitor a business process.** An advantage in using BPM Events rules is its ability to recognize the business context in which a process executes and to monitor business processes along with their resources. Indeed, monitoring a business process—more than simply monitoring a server load or resource usage—provides powerful support for managing online business processes.

• **Correlate monitoring sources.** Rules also add value by correlating a wide range of monitoring sources. Monitoring of processes (volume, requests, response time, turn-around time) and of their contexts also requires an understanding of how these processes use the IT resources they consume (servers, networks, databases, equipment), as well as how they involve people and their skills.

• **Generate/Analyze reports.** BPM Events rules, as part of their monitoring function, also generate business metrics that are exported as real-time reports, while other rules analyze these metrics and automatically generate alarms or make decisions.

• **Enforce business policies.** BPM Events rules can also be created that represent business management functions, whether these functions are associated with an application or not. Because of the decoupling between rules and processes, it is possible to enforce policies that are not tied to a particular application, but that manage resources and objects across business processes (or shared by several processes). Such business policies concern people involved in these applications, the assignment of various shared resources, the timing and scheduling of tasks, policies that relate to a class of customers or trading partners, and related message exchanges (regardless of the processes with which they may be involved).

• **Dynamically modify rules.** Finally, BPM Events rules that express business policies governing the business process can be easily and dynamically changed because they are not tied to the process definition.

For these reasons, rules are represented separately from business processes in Business Process Server and run by a distinct rule engine. However, they may be associated with these processes under the general umbrella of an "Business Process Server application," which defines the package of all objects and other elements required to publish and run a business application as a unit.
BPM Events tutorial

This tutorial illustrates some functional aspects of BPM Events. All the syntax features presented here are explained in detail in The rule language on page 57, Persistent structures on page 99, and the Interacting with Business Process Server processes topic.

For details, see the following topics:

- Using rules to control Business Process Server processes
- Using rules to correlate business process events
- Using rules to monitor and report on business processes
- Using rules to populate infopads
- Using rules to modify monitoring reports
- Using rules to monitor an application
- Using rules to correlate multiple events
- Using rules to generate alarms
- Using rules to synchronize Business Process Server processes
Using rules to control Business Process Server processes

To illustrate the advantages of using rules to control business processes, consider the following purchase order business process. A new business policy decides to give a rebate of 10 percent to all customers who submit an order for a total amount greater than $500. We could hardcode this policy in the definition of the purchase order process, but it would not be easy to update later. Instead, we will use a rule to express this policy. Rules can easily be modified and activated dynamically, while the application is published and running.

The rule has the following behavior:

• It triggers each time an order is submitted, on reception of an event generated by the process automation component of Business Process Server.
• It checks the total amount of the order, as reported by the notification event.
• If the amount is greater than $500, then it updates the discount attribute associated with this order process, to 10 percent. If not, it leaves this attribute at whatever value it was set initially.

The process automation component of Business Process Server is called BP Server, and it starts a new instance of the PurchaseOrder process each time a new order is submitted. When doing so, it generates an event named "PI_ACTIVATED" that BPM Events receives. The rule that captures this event, and decides if the discount is implemented, is written as follows:

```
rule bold_discount
activated by event1 of BP Server::PI_ACTIVATED {PROCESSTEMPLATE NAME : "PurchaseOrder"}
if (event1.totalamount >= 500)
then {
    val order = BLsession().getPI(event1);
    order.setDataslotValue("Discount", 10);
}
```

The plain English version of the above rule is:

"On reception of an event of type BP Server and value PI_ACTIVATED of PurchaseOrder business process, and if the total amount attribute of this event (event1.totalamount) has a value greater or equal to 500 (>= 500), then identify the corresponding order instance, and set its Discount attribute to 10."

If you generate this rule using the Rule Editor, then notice that there is a single action statement generated, contracting the two statements in the previous `then {}` block:

```
then {
    BLsession().getPI(event1).setDataslotValue("Discount", 10);
}
```

More generally, a rule has the following form:

```
rule <Rulename> activated by <event list> if <conditions> then {<actions>}
```

In the above example:

• the rule name is: bold_discount,
• the condition is: (event1.totalamount >= 500).
• the main action is: order.setDataslotValue("Discount", 10);
This rule is typically an Event - Condition - Action (ECA) rule: that is, on the reception of one or more event(s), if these events satisfy the condition, then execute the actions.

As mentioned earlier, rules are very well suited for implementing business policies that may change over time, or have a temporary effect. Let us assume that the discount policy above is only effective for all orders submitted after December 31st, 2002 and before January 31st, 2003. We will use the following rule, called newyear_discount:

```
rule newyear_discount
   activated by event1 of BP Server::PI_ACTIVATED {PROCESSTEMPLATE NAME : "PurchaseOrder"}
   if (event1.totalamount >= 500)
       and (event1.date > YEAR:2002/MONTH:12/DAY:31)
       and (event1.date < YEAR:2003/MONTH:1/DAY:31)
   then {
       val order = BLsession().getPI(event1);
       order.setDataslotValue("Discount", 10);
   }
```

The rule does exactly the same actions as bold_discount, only it is more restrictive: that is, the actions are executed only if the amount is greater than $500 AND if the date of the order is within the month of January 2003.

### Using rules to correlate business process events

BPM Events rules have the ability to process any event, whether generated from a business process or from an external source. We will call an "external event" any event that, unlike events of type BP Server::PI_ACTIVATED above, is not directly generated by the process execution, but rather reflects some change in the business context (for example, a sales promotion) or the state of a resource needed by the process (for example, the load of a server, the quantity of a product in stock, the price of a commodity). Usually, such events are generated by third party systems.

Let us assume that, in the order provisioning system, some products may be out of stock for an estimated period of time counted in business days. When this happens, the concerned customers should be immediately notified of the estimated delay, and proposals for a substitute product, if any, should be sent. It is assumed that an event of type Alarm::ProductOutOfStock is generated by the inventory system when a product is out of stock.

We can then use the following rule, named notify_if_out_of_stock, to capture such an event:

```
rule notify_if_out_of_stock
   activated by
   event1 of Alarm::ProductOutOfStock,
   event2 of BP Server::PI_ACTIVATED {PROCESSTEMPLATE NAME : "PurchaseOrder"}
   if (event1.product = event2.product)
   then {
       sendMail(event2.customeremail, "customerService@ACME.com",
           "notification of delay",
           "Sorry, your order may be late of " + event1.estimateddelay + 
           " days, due to inventory shortage of product: "+ event1.product + 
           ", Would you like to order instead:" + event1.productalternative );
   }
```

In fact, the rule above is activated by two events: one of type Alarm::ProductOutOfStock, and one of type BP Server::PI_ACTIVATED. We say that such a rule correlates two events. In this case, the correlation is done based on the product name that these events relate to. Note that events may also carry business data in their attributes, like product name, product alternative, and estimated delay.

The plain English version of the above rule is:
"On reception of an event of type Alarm and value ProductOutOfStock, if an event of type BP Server and value PI_ACTIVATED of PurchaseOrder business process that relates to the same product is present in the event log, then send an e-mail notification to the corresponding customer about the delay and possible alternative."

The rule detects all pairs of events that satisfy the correlation condition. Any time an event of the type: Alarm::ProductOutOfStock is generated, the rule correlates this alarm with all current customer orders that are concerned by this product. Each time a successful correlation is found, the rule automatically repeats its action statement for the corresponding customer, which is sending an e-mail.

Please note that:

- The same rule can handle the shortage of any product,
- When triggered by an event—for example, Alarm::ProductOutOfStock—, the rule is able to sort through past events, and select those events that correlate successfully with it.

Once the rule has identified the "out-of-stock" event, and sent mail to all customers concerned by the defaulting product, the same rule is now also triggered by all future customer orders as soon as they appear in the system (events of type BP Server::PI_ACTIVATED {PROCESSTEMPLATE NAME : "PurchaseOrder"}), and treats them similarly. Indeed, the rule above can also read:

"On reception of an event of type BP Server and value PI_ACTIVATED of PurchaseOrder business process, if an event of type Alarm and value ProductOutOfStock that relates to the same product is present in the event log, then send an e-mail notification to the corresponding customer, about the delay and possible alternative."

In other words, the order of the event profiles listed in the "activated by" clause does not matter. The rule may actually trigger on any of the event profiles. Each time a new customer order is activated, the rule tries to correlate its activation event with an alarm event of type Alarm::ProductOutOfStock.

As long as the alarm event (of type Alarm::ProductOutOfStock) has not been explicitly discarded from the event store (which may happen once the inventory is back to normal), the rule succeeds in correlating each new customer order with such an alarm, and generates an e-mail if the ordered product matches the out-of-stock event.

When the inventory level of a product is back to normal, it is sufficient to remove the alarm event for this product. We will define another rule, named stock_back_to_normal, to implement the reception of a "back-to-normal" event.

```
rule stock_back_to_normal
activated by
    event1 of Alarm::ProductInvNormal,
    event2 of Alarm::ProductOutOfStock
if (event2.product = event1.product)
then { discard (event1, event2); }
```

The rule above is also doing correlation, this time for correlating the "inventory back-to-normal" event (of type Alarm::ProductInvNormal) with any previous out-of-stock event for the same product. Once the correlation succeeds, both events are discarded, and the previous rule notify_if_out_of_stock will be unable to trigger successfully (and therefore will not notify customers anymore) for this product.

**Important:** The rules above do not require dynamic adding or removal depending on the inventory status. They are always there: they activate only if the right combination of events occurs. These events then automatically reflect any change in the inventory status, and further adding or removing of rules is not required. This example also shows how important it is for an application to manage the set of events properly, that is, decides when used events are discarded from the event store.
Using rules to monitor and report on business processes

Rules can also monitor business processes. They generate alarms, as well as real-time reports that managers can access online, or that can in turn be analyzed by other rules. This departs from more traditional reporting procedures that generate reports from archived data, by querying or running stored procedures on an operation database. In other words, BPM Events rules allow for generating event-driven reports, meaning the input of such reports is directly obtained from business events and transactions.

Let us assume we want to monitor what kind of request customers are sending from various regions at any time. We may have four types of requests: Product Order, Support, Service Order, and Training.

We want to know how many requests of each type have been served until now. Table 3: Report: Request distribution on page 33 displays how the expected monitoring report looks:

<table>
<thead>
<tr>
<th>Request type/origin</th>
<th>Asia</th>
<th>US</th>
<th>Europe</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProductOrder</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Support</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ServiceOrder</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Training</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In BPM Events, this type of monitoring report is called an infopad. An infopad is structured as a table (of one or two dimensions), with labeled rows and columns. Each element of this table is called a cell. A cell has several attributes. For example, each cell of the report shown in Table 3: Report: Request distribution on page 33 holds two values:

1. A number that represents how many occurrences were recorded for this type of request for this day of the week.
2. A number that represents the average execution time for requests of this type, in the corresponding region.

Each cell of any infopad has a default attribute named count of type integer, that does not require declaring when defining the infopad. We use it to count the number of requests for each cell (req type / region). The infopad is created by executing the following statements:

```plaintext
initialize { 
    ReqStats := new infopad<cell{avgtime:int}>[4][3]("ReqStats");
    ReqStats.addColLabels("Request Origin", list {"Asia", "US", "Europe"});
    ReqStats.addRowLabels("Request Type", list { "ProductOrder", "Support", "ServiceOrder", "Training" });
}
```

The creation of this infopad is done only once, initially when publishing the application. For this reason, the infopad creation statements above are enclosed in an initialization section, in the rule file:

```plaintext
initialize { ... }
```

This initialization section is executed only once, when loading the rule file in the rule engine. The meaning of these statements is as follows:
"When loading the rule file, create an infopad called "ReqStats" of 4 rows and 3 columns, with column labels: "Asia","US","Europe", and with rows labeled "ProductOrder","Support","ServiceOrder","Training". Each element (cell) of the infopad has an attribute (slot) avgtime of the integer type (int), and an attribute count of type int, which is always present by default (need not be specified)."

**Note:** The name ReqStats appears twice in the first line of the initialization section of the infopad creation statement. The first occurrence (left hand side) is the infopad variable, as it is known by the rules. The second occurrence (on right hand side) is the actual persistent name of the infopad, as used in the database and external systems. Usually, the infopad variable and the actual persistent name of the infopad are the same, but this is not required. This allows for writing rules that do not depend on the actual infopad name.

## Using rules to populate infopads

Once the infopad has been created in BPM Events, it is ready for use in a real-time environment. In order to populate the infopad previously described, BPM Events requires:

- Notification each time a new request comes in. The server (or the business process engine) posts a notification event to BPM Events. Such an event is expected to contain the following data: request type, request region.
- Execute a rule that reacts to the events above, and populate the infopad previously created. The rule below is executed each time a BP Server event of type PI_ACTIVATED is generated and BPM Events is notified.

```
rule monitor_startReq
  activated by evt1 of BP Server::PI_ACTIVATED
  then
  { ReqStats [evt1.reqtype][ evt1.region].count++ ;}
```

The plain English version of the rule above is:

"On reception of an event of type PI_ACTIVATED, add 1 to the count attribute of the element of the infopad ReqStats that corresponds to the request type and to the request region, as specified in the event."

**Note:** The rule does not have an if clause here: it is always successfully triggered when any event of type PI_ACTIVATED is notified. The rule then triggers when any instance of any process is activated. This works well if no other process than the customer request handling process is running. Assume, however, that we have two process types running: (1) the previous process named CustomerRequest, and (2) another process named AccountTermination. Both processes generate events of type BP Server::PI_ACTIVATED, but we do not want AccountTermination to trigger the rule. It is then safer to add a restrictive if clause to filter out AccountTermination events. Assume that the process is named "CustomerRequest".

A version of the monitoring rule that provides for two concurrently running processes follows:

```
rule monitor_startReq
  activated by evt1 of BP Server::PI_ACTIVATED {PROCESSTEMPLATE NAME : "CustomerRequest"}
  then
  { ReqStats [evt1.reqtype][ evt1.region].count++ ;}
```

In case each of the four request types described in **Table 3: Report: Request distribution** on page 33 is associated with a different business process, we can either use four rules like the one above or use an if clause of the form:

```
if (evt1.PROCESSTEMPLATENAME = "ProductOrder")or
  evt1.PROCESSTEMPLATENAME = "Support")or
```

---

34

OpenEdge Business Process Server: BPM Events User's Guide
Using rules to modify monitoring reports

Assume we also want to record the average response time of each request type, for each region. In Business Process Server, a process instance always generates an event of type PI_COMPLETED when it completes, in the same way as it generates a PI_ACTIVATED event when it starts. Let us assume that this completion event also contains the actual duration of the request, in the event attribute named responsetime. The previous monitoring rule is modified by a rule named monitor_timeReq, which is defined as follows:

```
rule monitor_timeReq
  activated by evt1 of BP Server::PI_COMPLETED
  then
  { val stat = ReqStats [evt1.reqtype][ evt1.region];
    stat.avgtime ::= avg(toInt(evt1.responsetime), stat.count++, 1);
  }
```

The rule above is executed each time a BP Server event of type PI_COMPLETED is notified.

Its plain English version is:

"On reception of an event of type BP Server::PI_COMPLETED, add 1 to the count attribute of the corresponding element of the infopad ReqStats, AND update the average response time for this request type and this region."

Now each element (or cell) of the infopad must remember two values:

1. A request count for each request type by region (here automatically updated when using the increment operator ++ in the function avg)
2. The average value of all response times for each request type by region (updated here with the avg() function).

In case you use the rule editor to generate this rule, you see that it generates a slightly different syntax for the action part. A single statement does not use the intermediate variable stat, shown as follows:

```
{ ReqStats[evt1.reqtype][ evt1.region].avgtime ::= avg(evt1.responsetime, ReqStats [evt1.reqtype][evt1.region].count++, 1);
```

Using rules to monitor an application

The "Assign_A_Task_V1" application itself does not use any BPM Events rules. This means that although you could assign tasks to assignees, you can not monitor the application. For example, how would you know how many tasks were currently assigned, reviewed, by whom, how many were late, and what was the average completion time?

In this section, you will learn how to monitor an application and generate reports using the BPM Events rule language by creating a sample report. In particular, you will learn how to complete the following:

- Define a report
- Build the report
- View the report
Defining a report

Business Process Server enables you to employ the power of BPM Events to dynamically define, update, observe, store, and print reports, all without writing any Java code or SQL queries. You can consult reports at any time, either by a business manager or automatically by rules. In Business Process Server, reports are dynamic objects and not the result of the offline processing of historical data. Use BPM Events rules to generate alarms, send e-mail, or trigger an action. For example, modify priorities of business processes if abnormal values appear in the report.

You can, for example, define a report called "MyTaskReport" that reports on the total number of tasks assigned and the average completion time for priority level that looks like Table 4: Sample "MyTaskReport" on page 36.

<table>
<thead>
<tr>
<th>Task priority</th>
<th>Volume</th>
<th>Average completion time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical &quot;1&quot;</td>
<td>10</td>
<td>23</td>
</tr>
<tr>
<td>High &quot;2&quot;</td>
<td>27</td>
<td>45</td>
</tr>
<tr>
<td>Medium &quot;3&quot;</td>
<td>58</td>
<td>150</td>
</tr>
<tr>
<td>Low &quot;4&quot;</td>
<td>64</td>
<td>1861</td>
</tr>
</tbody>
</table>

Using the MyTaskReport format, you can define a sample report entitled "MyTaskReport_Ver2" to display the following information:

- The total number of tasks assigned until now.
- The average completion time for each level of priority (Critical, High, Medium, Low).

The dynamic nature of Business Process Server enables it to build online reports, using event-driven rules based on events generated during the application processing. All rules in BPM Events are of the following form:

```
rule <rulename>
  activated by <event header>
  [if <conditions>]
  then "{ <actions> "}
```

The general meaning of such a rule is: "on reception of the specified events, if these events satisfy the conditions then execute the actions."

You should also observe some general practice concerning rules, including the following:

- All names and keywords are case sensitive.
- The `<conditions>` part expresses a condition to be satisfied by incoming events in order for BPM Events to execute the ActionList part. The condition may be a simple one, or it might include several conditions formed using operations such as AND and OR. For more information, refer to Rule condition as a logical expression on page 62.
- The `<actions>` part consists of one or more actions to be performed in the specified sequence. While these actions are capable of performing diverse operations, the sample applications here focus on creating and manipulating reports.
Begin by defining the report structure in BPM Events. The report is implemented using a general structure called an infopad in Business Process Server. An infopad is structured as a table of elements called cells, with each cell containing several attributes. You can define an infopad by using the Rule Editor in BPM Designer to create a rule module, which adds the following statements in the initialize section:

```javascript
initialize {
    MyTaskReport_Ver2 := new infopad<cell{comptime:int}>[4][1]("MyTaskReport_Ver2");
    MyTaskReport_Ver2.addRowLabels("TaskPriority", list {"Critical", "High", "Medium", "Low"});
}
```

The initialize section of the rule module is automatically executed when the module is loaded, typically when the application is installed into Business Process Server. At this time, an image of the report is created in memory, as well as in the database. This persistent image is used for recovery, as well as for access by Business Process Server front-ends.

The above statements:

1. Define each cell as having an attribute comptime of type int (integer).
2. In memory, allocate a table of four rows and one column ([4][1]), resulting in four elements (or cells).
3. Add the title ("TaskPriority") for the row dimension of the report and row titles (or labels): "Critical", "High", "Medium" and "Low".
4. Initialize the data in the cells to 0 (zero). For example, "comptime" and "counter" initialize to 0 for all the cells.
5. Create the infopad in the database. This stores all the infopad meta data information in the database.

**Note:** Each cell of any infopad always contains by default a "counter" attribute or slot. This slot does not require declaration. Additional slots are declared, like "comptime" above.

### Events generated by the application

After defining the report, you must write additional BPM Events rules to populate it. These rules capture events generated by BP Server while executing the application process. It is important to understand these events.

All events, whether sent by BP Server or by an adapter connected to third party software, have the following structure:

```
(type, date, value, context)
```

Table 5: Event elements on page 37 describes the elements of this structure.

**Table 5: Event elements**

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Used as a main discriminator for events. This attribute identifies either the source of the event, such as &quot;BP Server&quot; or the type of a major event, such as &quot;NewSubscription.&quot;</td>
</tr>
</tbody>
</table>

**Note:** Business Process Server includes a few predefined types, two of which are "BP Server" and "initRules." The set of possible values for type is open; you decide on the value of this attribute when creating new events.
The timestamp of the event. You can choose the precision level from day to millisecond.

An attribute that further qualifies the event, and is used as a second-level discriminator for events. For instance, the value of a BP Server event depends on the type of action that generated it such as starting a process, ending a workstep, or changing a dataslot. Typically, for a given event type, there exists a set of predefined values for value.

A list of name-value pairs specific to an event type and value. It represents any additional business data of interest for this event. It may, for example, contain a mapping of complex business objects that we want to associate with the event, such as a purchase order. Some of the information kept in the context field include the following:

- PROCESSINSTANCENAME
- PROCESSTEMPLATE
- WORKSTEPNAME
- PERFORMER

More about the value field

The range of permissible values for the value field is predefined for BP Server events. Table 6: BP Server objects and possible events on page 38 lists the possible events for each BP Server object.

**Table 6: BP Server objects and possible events**

<table>
<thead>
<tr>
<th>Object</th>
<th>Possible events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Templates (P)</td>
<td>P_CREATED, P_INSTALLED, P_REMOVED, P_SUSPENDED, P_RESUMED, and P_REPLACE</td>
</tr>
<tr>
<td>Process Instances (PI)</td>
<td>PI_CREATED, PI_ACTIVATED, PI_SUSPENDED, PI_RESUMED, PI_COMPLETED, PI_REMOVED, PI_PRIORITYSET, PI_CREATORSET, PI_DUEDATESET, PI_ATTRIBUTESSET, and PI_DATASLOTSET</td>
</tr>
<tr>
<td>WorkItems (I)</td>
<td>I_CREATED, I_ACTIVATED, I_AVAILABLE, I_ASSIGNED, I_COMPLETED, I_SUSPENDED, I_REMOVED, I_WAIT, I_PRIORITY, I_DUEDATE, I_PERFORMER, I_TERMINATED, and I_SKIPPED</td>
</tr>
<tr>
<td>Dataslots (S)</td>
<td>S_UPDATED</td>
</tr>
</tbody>
</table>
### Table 7: BP Server event descriptions

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PI_CREATED</td>
<td>Process instance created. This type of event is generated when the process instance is created in BP Server—but not yet started.</td>
</tr>
<tr>
<td>PI_ACTIVATED</td>
<td>Process instance activated. This type of event is generated when the process is initiated. At this time, some predefined initial values are known for some dataslots of the process instance.</td>
</tr>
<tr>
<td>PI_COMPLETED</td>
<td>Process instance completed. This type of event is generated when the process is terminated.</td>
</tr>
<tr>
<td>W_ACTIVATED</td>
<td>Workstep started. Each workstep of the Assign A Task process generates a start event.</td>
</tr>
<tr>
<td>W_PRECONDITION_WAIT</td>
<td>Workstep in wait state on a precondition. This event is generated when a workstep goes in a wait state on evaluation of precondition to false.</td>
</tr>
<tr>
<td>W_ACTIVATION_WAIT</td>
<td>Workstep in wait state waiting to be activated after a certain duration or at a particular time.</td>
</tr>
<tr>
<td>W_DUEDATESET</td>
<td>Due date set for the workstep.</td>
</tr>
<tr>
<td>W_COMPLETED</td>
<td>Workstep completed. At the end of each workstep, this event is generated. Values of output dataslots for this step are added to the context of the event as name/value pairs.</td>
</tr>
<tr>
<td>S_UPDATED</td>
<td>Dataslot updated.</td>
</tr>
<tr>
<td>I_ASSIGNED</td>
<td>Work item assigned.</td>
</tr>
</tbody>
</table>

Imagine assigning John the task of "configure printer" with "medium" priority. The dataslot values for this workstep are as follows:

- "John" – the assignee
- "Configure printer" – the task name
- "Medium" – the priority level

When activating this workstep, BP Server generates a \texttt{W_ACTIVATED} event that reports the value of these dataslots. When the workstep is an Activity workstep, which must be performed by one or more persons, then the workstep generates workitems. A work item is a task unit assigned to a person, as part of a workstep. In such a case, additional events related to workitems are generated: \texttt{I_CREATED}, \texttt{I_ASSIGNED}. 
Understanding event-maps

Each process instance generates a sequence of events called an event-map. An event-map defines the sequence of events from the beginning of the process to the end.

The following demonstrates a selection of events generated by the "Assign_A_Task_V2" application:

Received event:
type=BP Server, value=PI_CREATED, Date=10-Aug-99 12:11:35 PM, DURATION=0,
PILABEL=Assign_A_Task_V2(133),
CLASSNAME=com.tdiinc.BPServer.Server.WFProcessInstance, PRIORITY=Low,
ENDTIME=0, PROCESSINSTANCECLASSNAME=Assign_A_Task_V2#133, Priority=Low,
CREATOR=grace, STARTTIME=0, DueDate=Wed Dec 31 16:00:00 PST 1969,
PROCESSTEMPLATECLASSNAME=Assign_A_Task_V2
---------- Taking appropriate action ----------
Received event:
type=BP Server, value=S_UPDATED, Date=10-Aug-99 12:11:35 PM,
DATASLOTNAME=Assignee, PROCESSINSTANCECLASSNAME=Assign_A_Task_V2#133,
CLASSNAME=com.tdiinc.BPServer.Server.WFDataslot,
DATASLOTTYPE=STRING, DATASLOTVALUE=grace, PROCESSTEMPLATECLASSNAME=Assign_A_Task_V2
---------- Taking appropriate action ----------
Received event:
type=BP Server, value=W_DUEDATESET, Date=10-Aug-99 12:11:47 PM,
PROCESSINSTANCECLASSNAME=Assign_A_Task_V2#133, DUEDATE=936169200000,
WORKSTEPNAME=DoTask, PRIORITY=Low, ENDTIME=0, INSTRUCTION=Please complete
the task according to the instructions given., PERFORMER=Assignee,
Comments=<NULL>, WORKITEMNAME=Assign_A_Task_V2#133::DoTask,
DURATION=0, PERFORMERVALUE=grace, STARTTIME=0,
CLASSNAME=com.tdiinc.BPServer.Server.WFAtomicWS,
PROCESSTEMPLATECLASSNAME=Assign_A_Task_V2
---------- Taking appropriate action ----------
Received event:
type=BP Server, value=S_UPDATED, Date=10-Aug-99 12:11:51 PM,
DATASLOTCLASSNAME=Instructions, PROCESSINSTANCECLASSNAME=Assign_A_Task_V2#133,
CLASSNAME=com.tdiinc.BPServer.Server.WFDataslot, DATASLOTVALUE=Please review all the BP Server documentations.,
PROCESSTEMPLATECLASSNAME=Assign_A_Task_V2
---------- Taking appropriate action ----------
Received event:
type=BP Server, value=PI_PRIORITYSET, Date=10-Aug-99 12:11:51 PM, PRIORITY=Low,
Priority=Low, Assignee=grace, ENDTIME=0, CREATOR=grace,
Instructions=Please review all the BP Server documentations.,
PROCESSINSTANCECLASSNAME=Assign_A_Task_V2#133, STARTTIME=133,
PROCESSTEMPLATECLASSNAME=Assign_A_Task_V2, PILABEL=Assign_A_Task_V2 (133),
DURATION=0, DueDate=Wed Sep 01 00:00:00 PDT 1999,
CLASSNAME=com.tdiinc.BPServer.Server.WFProcessInstance
---------- Taking appropriate action ----------

Note: These events have the Business Process Server pre-defined type "BP Server." You would need to use
the Value field as a secondary discriminator for the events.

Table 8: Predefined BP Server events on page 41 describes each event in the example.
Table 8: Predefined BP Server events

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PI_CREATED</td>
<td>Indicates that a process instance has been created using the process template name &quot;Assign_A_Task_V2.&quot; The event indicates two priorities:</td>
</tr>
<tr>
<td></td>
<td>• The TaskPriority defined by the user</td>
</tr>
<tr>
<td></td>
<td>• The ProcessPriority assigned by BP Server</td>
</tr>
<tr>
<td>S_UPDATED</td>
<td>Indicates that a dataslot named &quot;Assignee&quot; has been updated.</td>
</tr>
<tr>
<td>W_DUEDATESET</td>
<td>Indicates that the due date for workstep named &quot;DoTask&quot; has been set.</td>
</tr>
<tr>
<td>S_UPDATED</td>
<td>Indicates that the dataslot named &quot;Instructions&quot; has been updated.</td>
</tr>
<tr>
<td>PI_PRIORITYSET</td>
<td>Indicates that the priority for the process instance has been set.</td>
</tr>
</tbody>
</table>

Building the report

To populate the report, you need another BPM Events rule that captures the event reporting the priority level in each event-map, and counts the events for each level of priority. This informs you how many task assignments take place. The events you need to capture are of the following form:

```
Type: BP Server
Date: ___
Value: W_COMPLETED
Context:
  PROCESSTEMPLATENAME: Assign_A_Task_V2
  PROCESSINSTANCENAME: Assign_A_Task_V2#xxx
  WORKSTEPNAME: Start
  Priority: ___
```

In particular, for each instance of the "Assign_A_Task_V2" process, you must capture the following data:

- **The priority of the task.** For this, you must catch the W_COMPLETED event of the Start workstep, revealing the priority level of the task.

- **The completion time of the DoTask workstep (the task duration).** This is the elapsed time between the W_ACTIVATED event and W_COMPLETED event of the DoTask process.

Use the data above to capture the following events:

- Event 1: value=W_COMPLETED, WORKSTEPNAME=Start (for priority)
- Event 2: value=W_COMPLETED, WORKSTEPNAME=DoTask (for elapsed time)

**Note:** Instead of using W_ACTIVATED of "Do Task," here we use W_COMPLETED of "Start," which is already caught in order to get the priority. There is no significant elapsed time between these two events. As a general guideline, the smaller the number of events a rule correlates, the faster it executes.
You can write a rule that correlates these two events and updates the corresponding report attributes (including count and average duration) for this priority level. Here is the definition of the rule "update_myTaskReport" that builds the report:

```plaintext
rule update_MyTaskReport
  activated by
  EVT_1 of BP Server::W_COMPLETED{WORKSTEPNAME:"Start", PROCESSTEMPLATENAME:"Assign_A_Task_V2"},
  EVT_2 of BP Server::W_COMPLETED{WORKSTEPNAME:"DoTask", PROCESSTEMPLATENAME:"Assign_A_Task_V2"}
  correlated with INDEX
then{
  val vcell = Assign_A_Task_V2::Assign_A_Task_V2_data::
  MyTaskReport_V2[EVT_1.context.TaskPriority][1];
  vcell.comptime ::= avg(duration(EVT_1.date, EVT_2.date)/1000,
  vcell.count++, 1);
  discard(EVT_1);
  discard(EVT_2);
}
```

The rule "update_myTaskReport" uses two event variables: EVT_1 and EVT_2, both of profile BP Server::W_COMPLETED. This means these events are generated by BP Server (event type: BP Server), at the completion of some workstep (event value: W_COMPLETED). Additional conditions further select the right events for the rule: they must be related to the completion of the "Start" workstep and of the "DoTask" workstep of the "Assign_A_Task_V2" process.

Since there are multiple process instances, we want to make sure that the two events (one signaling the "completion of the Start workstep" and another signaling the "completion of the DoTask workstep") belong to the same process instance. The clause: correlated with INDEX enforces this, as there is an index created automatically by the rule engine for all BP Server events, based on their process ID. The statement indicates that these events are correlated based on the same index entry (here, PID).

The rule "update_myTaskReport" therefore reads:

"Each time a task assignment process instance generates the pair of events (end Start workstep, end DoTask workstep), then increment the report cell that corresponds to its priority, and also update the average completion time for this level of priority."

---

**Note:** The rule is executed only when the last event of the pair is generated, that is at the end of the "DoTask" workstep.

Each time a combination of two events is generated that satisfies this rule, the monitor action executes and modifies the element (cell) of the report that corresponds to the level of priority given by the first event EVT_1. This modification concerns the two attributes (slots) defined for each cell of this report. It performs the following operations:

1. Update the average value for completion time (comptime).
2. Increment the implicit count of the report cell.

The elapsed time between the beginning and the end of a task is obtained by the function \( \text{duration()} \), the first argument of which specifies the desired time unit (here, seconds). The \( \text{avg()} \) function calculates the new average, using the new value (duration), and the default count, the value of which is incremented automatically (++ operator).

Because there is only one such pair of events generated each time a new task is assigned, the rule "update_myTaskReport" and its monitoring action is triggered only once for each instance of the "Assign_A_Task_V2" process. The action of the rule consists of an infopad operation, which increments the report element that corresponds to the level of priority recorded in the event data.
Using rules to correlate multiple events

Event correlation occurs for rules involving more than one event. For example, assume that we want to call on BPM Events to calculate the duration of a business process. Because the PI_COMPLETED event does not contain the actual duration, but only the completion date of the request, BPM Events needs to collect the starting date and completion date of each request. These values are respectively reported in:

- The PI_ACTIVATED event which always contains the starting date of the request.
- The PI_COMPLETED event which always contains the completion date of the request.

In addition to request date, request type, and request region, each of these two events also carries a request ID, which we consider the same as the process instance ID (stored in the PROCESSINSTANCEID attribute of the event). This request ID allows BPM Events to pair off the events related to the same request. The rule to calculate the event duration, named calculate_timeReq, is provided below:

```
rule calculate_timeReq
  activated by
  evt1 of BP Server::PI_ACTIVATED, 
  evt2 of BP Server::PI_COMPLETED
  if (evt1.PROCESSINSTANCEID = evt2.PROCESSINSTANCEID)
  then { val stat = ReqStats [evt1.reqtype][ evt1.region];
    stat.avgtime ::= avg(duration(evt1.date, evt2.date)/60*60*1000, stat.count++, 1);}
```

The rule above executes the monitoring action each time a pair of related events (to the same request) is notified. The rule calculates the request duration by making the difference in hours (HOUR) between the two event dates, using the BPM Events function duration. It then recalculates the average response time. The plain English version of the rule is:

"On reception of an event of type PI_COMPLETED, find a past event of type PI_ACTIVATED, with same request ID, add 1 to the count attribute of the corresponding element of the infopad ReqStats AND calculate the time between the two event dates in hours, then update the average response time for this request type and region."

**Note:** The previous rule only triggers if there is a combination of two events that satisfies its condition.

When a rule does event correlation based on the process instance ID, as shown in the case above, you may write it a shorter way because it is not necessary to express the correlation condition (in the if clause). Indeed, BPM Events is preconfigured so that the BP Server events it received are indexed after their process instance ID attribute. Adding the clause correlated with INDEX to the rule, as shown in the code example below, instructs the rule engine to use this index to correlate events with the same process instance ID.

```
rule calculate_timeReq
  activated by
  evt1 of BP Server::PI_ACTIVATED, evt2 of BP Server::PI_COMPLETED
  correlated with INDEX
  then {
    val stat = ReqStats [evt1.reqtype][ evt1.region];
    stat.avgtime ::= avg(duration(evt1.date, evt2.date)/60*60*1000, stat.count++, 1);}
```

**Note:** Users can define their own message index for each message type. For example, you can create a message index for PurchaseOrder XML messages or for the customer ID field present in the XML body.
Using rules to generate alarms

In addition to generating real-time reports, rules can also detect critical situations and generate alarms. In this example, the measures we store in the `ReqStats` table may show a critical situation. For example, the number of requests handled exceeds a critical limit; or the average response time suddenly degrades beyond an acceptable threshold. Let us assume that the average response time for requests should not exceed 3 hours at any time. To generate an alarm where the response time is greater than three hours, you can upgrade the monitoring rule in the following two ways:

- Update the infopad to associate an alarm-condition to each of its elements. The initialization section that creates the infopad becomes:

```plaintext
initialize {
    ReqStats := new infopad<cell{avgtime:int}>[4][3]("ReqStats");
    ReqStats.addColLabels("Request Origin", list("Asia", "US", "Europe");
    ReqStats.addRowLabels("Request Type", list("ProductOrder", "Support", "ServiceOrder", "Training");
    ReqStats.check("alarm_1", "avgtime", "GT", 3, "transition");
}
```

The rule associates an alarm condition to all cells of the `Support` row of the `ReqStats` infopad. The event generated by an alarm-condition is defined by the `check` statement added to the infopad definition in the initialization section above. It tells BPM Events to trigger an internal event of type "BPEVENT_INFOPAD_ALARM", and of value "alarm_1" each time the average value ("avgtime") is greater than ("GT") the threshold of three hours (3) when updated. No alarm is sent if the threshold was already above 3, a distinction specified by the keyword "transition".

- Add a rule, named `alarm_timeReq`, to handle the generated internal alarm event, and transform it into an external event with appropriate data, which in this example consists of sending an e-mail.

```plaintext
rule alarm_timeReq
    activated by evt1 of BPEVENT_INFOPAD_ALARM::alarm_1
then { sendMail("john@acme.com", "response time too long for support in region:" +
    evt1.column, evt1);}
```

**Note:** In the e-mailing rule above, the region identification is obtained by the corresponding column of the infopad, as reported by the alarm event, and is concatenated (operator "+") to the message in second argument, to form the body of the e-mail.

However, you may further improve the defined monitoring set-up by providing a solution to the following scenario: what if the very first response time used to calculate the Support average response time, in the table `ReqStats`, is 4 hours? An alarm would be triggered, because of a single odd value. Because the alarm condition is associated with an average value, it must not fire off for a single odd measure, as it may happen at the beginning of the monitoring session, when very few values are used in the average. Rather, it must be ensured that the average value is representative of a significant number of support orders, before it is considered for an alarm check. Therefore, the alarm should only consider average values calculated from a significant number of requests. Let us say that at least 100 requests of type `Support` should be logged in order for the average response time to be meaningful. We will improve the `alarm_timeReq` rule by adding this condition:

```plaintext
rule alarm_timeReq
    activated by evt1 of BPEVENT_INFOPAD_ALARM::alarm_1
if (evt1.count >= 100)
then { sendMail("john@acme.com", "response time too long for support in region:" +
    evt1.column, evt1);}
```
The event generated by an alarm-condition automatically contains all the attributes of the faulty element of the infopad (here, the attributes of the element for which the average time attribute passed the 3 hours limit). One of these attributes is the request count. Another is the column ID of the offending cell (+ evt1.column), which represents the region where the request originates.

Using rules to synchronize Business Process Server processes

Let us return to the discount policy rule described in Using rules to control Business Process Server processes on page 30. When writing rules that cooperate with processes, it is often necessary to take additional precautions to make sure that the rules and the process are well synchronized. The discount policy rule that we wrote earlier may not properly control the Purchase Order process in some cases. Indeed, you must design the purchase order process so that it waits for the discount decision before resuming further, that is, before reaching the point where it actually calculates the customer’s bill. The Application Developer’s Guide explains how a process instance can wait for a flag (dataslot) to take a certain value, before proceeding further.

You may design the Purchase Order process so that it waits for an explicit signal from the rules before proceeding further (instead of only waiting for the Discount value to change). In that case, the discount rule must generate this additional signal. This is done by adding the action: `order.flagOn("continue")` after the discount change.

**Note:** The business process instance (identified by the variable order) has defined an attribute named "continue", with initial value to false, and waits until this value is true.

The original **bold_discount** rule now becomes:

```java
rule bold_discount
activated by event1 of BP Server::PI_ACTIVATED {PROCESSTEMPLATE NAME : "CustomerRequest"}
if (event1.totalamount >= 500)
then {
    val order = BLsession().getPI(event1);
    order.setDataslotValue("Discount", 10);
    order.flagOn("continue");
}
```

However, we must also make sure that the PurchaseOrder process resumes, even when the discount does not apply. Remember that each process instance waits for a decision from the rule, whether it is an acceptance or a rejection. The rule above only handles the acceptance case; it subsequently updates the discount rate, and after this tells the process instance to resume its course: `(order.flagOn("continue"))`.

We must also tell the process instance to resume its course when the discount is declined. For this, we can write a second rule, named here **no_discount**, that also handles the non-discount case:

```java
rule no_discount
activated by event1 of BP Server::PI_ACTIVATED {PROCESSTEMPLATE NAME : "CustomerRequest"}
if (event1.totalamount < 500)
then {
    val order = BLsession().getPI(event1);
    order.flagOn("continue");
}
```
For each event of type `BP Server::PI_ACTIVATED`, we are now sure that either the `bold_discount` or `no_discount` rule will be successfully triggered, and that either one of these rules will release or unblock the waiting process instance that triggered it.

There is, however, a simpler way to handle the discount policy. The current version of Business Process Server provides a single rule to handle both the discount case and the no-discount case, which is displayed as follows:

```
whether_discount_or_not,
```

This rule always triggers successfully, that is, executes its action each time it is triggered. It always executes the unblocking action `order.flagOn("continue")` for each Purchase Order activation event, whether the discount applies or not.

```java
rule whether_discount_or_not
  activated by evtl of BP Server::PI_ACTIVATED {PROCESSTEMPLATE NAME : "CustomerRequest"}
  if (evtl.PROCESSTEMPLATENAME = "CustomerRequest")
  {
    val order = BLsession().getPI(evt1);
    if ( evtl.totalamount >= 500) order.setDataslotValue("Discount", 10);
    order.flagOn("continue");
  }
```

**Note:** The discount test is moved out of the main `if` clause of the rule, and into the action part (`then` clause), which also contain if-then conditions.

The `whether_discount_or_not` rule has the following behavior:

- It is triggered each time an order is submitted, on reception of an event `PI_Activated` of type `BP Server`.
- It accesses the process instance handle (`order`), and tests its `totalamount` attribute.
- If the amount is greater than $500, then it updates the discount attribute associated with this order process to 10 percent; otherwise it leaves this attribute at whatever value it was set initially.
- The rule then signals the process to resume its course, even if the discount did not apply.

By using rules, you can control a process in a more sophisticated way. For example, we can define an `earlybird_discount` rule that decides to enable a rebate only for the first 1000 customers of each region. In that case, we will use the previous infopad `ReqStats` which already monitors the customer requests. By adding a condition to the rule that tests the `count` value in this infopad, we filter out any discount candidate once the count for this region is above 1000 requests.

```java
rule earlybird_discount
  activated by evtl of BP Server::PI_ACTIVATED {PROCESSTEMPLATE NAME : "CustomerRequest"}
  {
    val order = BLsession().getPI(evt1);
    if ((evtl.totalamount >= 500) and (evtl.reqtype = "ProductOrder") and
        (ReqStats["ProductOrder"][ evtl.region].count < 1001))
      order.setDataslotValue("Discount", 10);
      order.flagOn("continue");
  }
```

In the `earlybird_discount` rule, we moved all the conditions out of the `if` clause, and put them inside the action part of the rule. The result is that the rule is always triggered for any event of type `BP Server::PI_ACTIVATED` and therefore, executes all the actions, except discount update, which is now directly subject to the restrictions, as expressed in the conditional statement of the action part of the rule.
A final improvement on this rule allows for updating the customer count at the same time we do the discount. For this, we add the increment statement after the `setDataslotValue` statement. The conditional statement in the action part becomes:

```java
... if ( (toInt(evt1.totalamount) >= 500) 
    and (evt1.reqtype = "ProductOrder")
    and (ReqStats["ProductOrder"][ evt1.region].count < 1001))
{ order.setDataslotValue("Discount", 10);
    ReqStats["ProductOrder"][ evt1.region].count++ ;}
...```
Understanding BPM Events

This chapter explains the following types of BPM Events: the event model, the BP Server event, workstep events, work item events, and tracking the parent process of a subprocess.

For details, see the following topics:

- The event model
- The XML event
- The BP Server event protocol

The event model

The BPM Events structure is simple, yet flexible enough to handle various notions of events. An event has four attributes: type, date, value and context that represent business data associated with the event. The context attribute is a complex object with attributes that may vary from one event to the other. The following are two examples of event objects:

```plaintext
type: disk_failure
date: 1998/10/15/3/15/45
value: SCSI
context:
    subnet:xyz
    hostname:klamath
    owner:John
    department:IT
type: insurance_claim
date: 1998/10/15
value: Car
```
Note: Make sure the standard attribute names are in lower case.

The composed attribute context is a list of (name, value) pairs, and is used to map business objects associated with the event.

Attributes

The following sections explain various attribute types, and describe how to test for the presence of an attribute.

Attribute types

Event attributes are found in each element of the context attribute, plus the standard attributes type and value. Events have a very basic but versatile structure: their context is a list of name/value pairs, which is unordered (implemented as hashtable, the elements of which are always accessed by their names in the rule language). Each element of the context list must be of atomic value, mapped to a string value in the rule language.

In other words, when writing rules, any event attribute is handled as a string value, except for the date attribute, which is handled as a date type. In case the attribute is handled as a different type, it is safer to use explicit conversion functions. However, implicit conversion is performed when the expected type is automatically inferred at compile time. In the following example, the function: toInt() converts the attribute Age into an integer, suitable for the comparison.

```plaintext
rule claim_dispatch
  activated by event1 of insurance_claim::Car
  if (toInt(event1.Age) <= 25)
  and (event1.date < YEAR:2001/MONTH:6)
  ...
```

Yet, because the compiler knows that the other operand is an integer constant, it automatically enforces the conversion to integer. The rule above can also be written as:

```plaintext
rule claim_dispatch
  activated by event1 of insurance_claim::Car
  if (event1.Age <= 25)
  ...
```

However, in case the type of the condition operands cannot be inferred at compile time, these operands must be explicitly converted if they are handled other than as string values. In the following rule, the compiler does not know if it should make a comparison of strings or a comparison of numbers. In this case, explicit conversion functions are used:

```plaintext
rule claim_dispatch
  activated by event1 of insurance_claim::Car,
  event2 of insurance_claim::general_policy
  if (toInt(event1.Age) <= toInt(event2.agelimit))
  ....
```

Testing the presence of an attribute

In some cases, we may require testing inside a rule whether an event context contains an attribute or not. This is done by using a condition comparing this attribute with "nil" value (null value).
The condition: \((EVT_1.context.attr = \text{nil})\) is true only if the attribute "\text{attr}\" is not part of the context of the event \(EVT_1\).

The condition: \((EVT_1.context.attr \neq \text{nil})\) is true only if the attribute "\text{attr}\" is part of the context of the event \(EVT_1\), regardless of its value.

Event sources

BPM Events can get events from any source—a business process, IT resource monitoring an external notification, or a database. Events notified from external sources are converted into Business Process Server format using dedicated adapters.

Event Adapters typically convert events, messages or plain data from a third-party application or messaging middleware. These adapters can be gateways to other event managers like those used in the monitoring of IT resources (for example, an Enterprise Management System like HP-OpenView or Unicenter Framework), or to other event channels, like CORBA or third-party messaging products like MQSeries or TIB/Rendezvous. You can also design adapters to read database objects or records and convert them in event format.

An event adapter can be a subscriber to the external event source (in which case, it waits for notification) or pulls event data from an event source (for example, Unicenter Event Manager). The event adapter then pushes these events to the Business Process Server event queue in database. BPM Events pulls events from this queue, in a consumer mode, at a speed consistent with its processing.

The XML event

An xml event has two public attributes:

- A set of \text{properties}. The \text{properties} are set in a hashtable and can be any of the primitive type such as String, int, long.
- An \text{XML body} (\text{xml}). The \text{xml} is a string that contains the XML body of the message.

It also has internal BPM Events attributes such as type and date.

An example of an XML Event is shown in the following section:

```
properties:
    Origin: server1
    Destination: server2
    Description: Purchase Order based on po.xsd

<?xml version="1.0"?>
<purchaseOrder>
    <shipTo type="USAddress">
        <name>Helen Zoe</name>
        <street>47 Eden Street</street>
        <city>Santa Clara</city>
        <state>CA</state>
        <zip>95054</zip>
    </shipTo>
    <billTo type="USAddress">
        <name>Robert Zoe</name>
        <street>47 Eden Street</street>
        <city>Santa Clara</city>
        <state>CA</state>
        <zip>95054</zip>
    </billTo>
    <itemsList>
        <item partNum="833-AA">
            <productName>Lapis necklace</productName>
        </item>
    </itemsList>
</purchaseOrder>
```
The XML schema definition

Each message type should have an xsd file. The xsd file describes the elements as well as their type, which is not the case for dtd files. Typing is primordial in rules for detecting errors at compilation time and for guaranteeing fast execution.

Here is the xsd file that the two XML messages above follow:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Purchase order schema for Example.com.
      Copyright 2000 Example.com. All rights reserved.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
  <xsd:element name="comment" type="xsd:string"/>
  <xsd:complexType name="PurchaseOrderType">
    <xsd:sequence>
      <xsd:element name="shipTo" type="USAddress"/>
      <xsd:element name="billTo" type="USAddress"/>
      <xsd:element ref="comment" minOccurs="0"/>
      <xsd:element name="itemsList" type="Items"/>
    </xsd:sequence>
    <xsd:attribute name="orderDate" type="xsd:date"/>
  </xsd:complexType>
  <xsd:complexType name="USAddress">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="street" type="xsd:string"/>
      <xsd:element name="city" type="xsd:string"/>
      <xsd:element name="state" type="xsd:string"/>
      <xsd:element name="zip" type="xsd:decimal"/>
    </xsd:sequence>
    <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
  </xsd:complexType>
  <xsd:complexType name="Items">
    <xsd:sequence>
      <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="productName" type="xsd:string"/>
            <xsd:element name="quantity">
              <xsd:simpleType>
                <xsd:restriction base="xsd:positiveInteger">
                  <xsd:maxExclusive value="100"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```
Accessing XML body content

The content of the XML body can be accessed using xpath notation. The state from which the order was shipped can be accessed from the purchase order xml event, as shown below:

```java
rule monitor_po
  activated by
    msg of PurchaseOrder
  then {
    println("=============== monitoring_po ===============");
    println("msg.po:purchaseOrder.po:shipTo.po:state = "+
      msg["po:purchaseOrder/po:shipTo/po:state"]);
}
```

Accessing message properties

Properties can be accessed using the `getProperty` method. The properties Origin, Destination and Description can be accessed from the purchase order xml event, as shown below:

```java
rule po_activatedShip
  activated by
    msg of po:purchaseOrder["po:shipTo/po:state = 'NY'"]
  then {
    println("=============== po_activatedShip ===============");
    println("Received JMS Event, shipping state is: " +
      msg["po:purchaseOrder/po:shipTo/po:state"] + " and month is: " +
      msg.date.month+, Origin: "+msg.getProperty("Origin")+", Destination: "+
        msg.getProperty("Destination")+", Description: "+msg.getProperty("Description")");
}
```

The BP Server event protocol

Business processes that are defined with BP Server, Business Process Server’s automation engine, generate particular sequences of events characteristic of the event protocol of BP Server. Features of the BP Server protocol are described in the following section.
Type and value of the event

The type of BP Server event is always "BP Server".

The value field of BP Server events depends on when the event was generated in the process. This field represents a more detailed classification of events of this type: it is usually used as a subtype. The most common events that BP Server processes generate have the values described in Table 9: Event values on page 54.

Table 9: Event values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PI_CREATED</td>
<td>Process instance created. Such an event is generated when the process instance is created in the process engine—but not yet started.</td>
</tr>
<tr>
<td>PI_ACTIVATED</td>
<td>Process instance activated. Such an event is generated when the process is initiated. At this time, some predefined initial values are known for some attributes (dataslots) of the process instance.</td>
</tr>
<tr>
<td>PI_PRIORITYSET</td>
<td>Process instance priority is set.</td>
</tr>
<tr>
<td>PI_DUEDATE</td>
<td>Process instance due date is set.</td>
</tr>
<tr>
<td>PI_ATTRIBUTESSET</td>
<td>Process instance attributes are set.</td>
</tr>
<tr>
<td>S_UPDATED</td>
<td>Slots updated. These events are generated when the dataslots of the process are updated.</td>
</tr>
<tr>
<td>PI_DATASLOTSSET</td>
<td>Process instance dataslots updated. Lists all dataslots, with name, type and value of each, whose value was changed during activation of the process instance.</td>
</tr>
<tr>
<td>PI_COMPLETED</td>
<td>Process instance completed. Such an event is generated when the process is terminated.</td>
</tr>
<tr>
<td>PI_REMOVED</td>
<td>Process instance removed. Such an event is generated when the process instance is removed. This event is similar to the PI_COMPLETED event, but it can happen at any time when the process instance is running and indicates that the process instance was aborted.</td>
</tr>
<tr>
<td>W_ACTIVATED</td>
<td>Workstep started. Each workstep of the process generates an activation event, including the Start workstep of the process.</td>
</tr>
<tr>
<td>W_PRIORITYSET</td>
<td>The priority of a workstep is set.</td>
</tr>
<tr>
<td>W_DUEDATESET</td>
<td>The due date of a workstep is set in a list of events.</td>
</tr>
<tr>
<td>W_COMPLETED</td>
<td>Workstep completed. At the end of each workstep, this event is generated.</td>
</tr>
<tr>
<td>I_CREATED</td>
<td>Work item created, as the result of assigning the workstep to a human performer. Note that a single work step can trigger several work items, in case there are multiple performers.</td>
</tr>
<tr>
<td>I_ASSIGNED</td>
<td>Work item assigned to a performer.</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>I_AVAILABLE</td>
<td>Work item available to a performer.</td>
</tr>
<tr>
<td>I_COMPLETED</td>
<td>Work item completed.</td>
</tr>
</tbody>
</table>

A sequence of events that is generated by the execution of a particular process instance is called the event-map of the process. A detailed description of an event-map is presented in Understanding event-maps on page 40.

Date of the event

The date of a BP Server event is set at the time the BP Server engine generates the event. The date precision is actually set in milliseconds (Java date representation), but is manipulated at second-level maximum precision by BPM Events rules.

Note: The event date is the origination date, not the reception date of the event. This difference may be significant as events transit through the Business Process Server database, and some delay may occur between the current date at which a BPM Events rule processes an event, and the origination date. This type of delay can be caused by the following scenario: BPM Events server restarts after a shutdown and requires processing a backlog of events generated during this time.

Event ID

BP Server events, like any event coming from the Database Event Channel (DEC), are assigned a unique ID by the Business Process Server database server. This ID is reported in the context attribute "ID". It should not be relevant to applications, and should not be relied upon (not used by rules) for identifying events inside the application. It only serves the purpose of internal event management.

Context of BP Server events

The content of the context event attribute depends on the value of the event. However, all BP Server events relating to a process instance have the process template name in their context, plus instance ID attributes:

```
PROCESSTEMPLATENAME:<process_name>
PROCESSTEMPLATEID:<process_id>
PROCESSINSTANCENAME:<process_instance_name>
PROCESSINSTANCEID:<process_instance_id>
RPID:<root process instance id>
```

The `PROCESSINSTANCEID` value is only unique for a given `PROCESSTEMPLATENAME`. The `PROCESSINSTANCENAME` attribute value is unique across applications. It is usually the concatenation of the `PROCESSTEMPLATENAME` and the `PROCESSINSTANCEID` attributes.

The event context contains the names and values of dataslots belonging to the associated process instance. These dataslots are:

- The standard dataslots, identified by their name written with upper-case only characters.
- The application dataslots, with at least one lower case character.
Default attribute values

BP Server assigns default values to undefined dataslots of a process instance. Depending on the dataslot type, as declared in the BP Server process, the default value is:

- INT64 dataslot: -1
- INTEGER dataslot: -1.0
- LOGICAL dataslot: false
- CHARACTER dataslot: "<NULL>"

When writing rules that process dataslots that can be undefined, you are expected to know the original type of the dataslot in BP Server, so that the expected form of an undefined value is tested properly. Remember that because event attributes are untyped, these default values are rendered as string values in a rule; that is, the string "-1" for an event attribute that represents an undefined long dataslot, or the string "false" for an event attribute that represents an undefined LOGICAL dataslot. Default type conversions are done automatically if the context allows. For example, the condition: (EVT_1.mylongDS = -1) is true if the dataslot "mylongDS" was undefined in BP Server. In other cases, explicit use of type conversion functions is recommended.

For more information about process instance events, workstep events and work item events, see the Application Developer's Guide.
The rule language

This chapter provides an overview of the rule language. Many features presented here are described in detail in later sections. The formal grammar and syntax of the rule language is presented in Rule syntax on page 167. The set of predefined functions used throughout the documents are provided in Predefined functions and operators on page 175.

For details, see the following topics:

- Rule structure
- Rule modules and rule groups
- Adding and removing rules dynamically
- Operational semantics of rules
- Time operations and expressions
- Controlling rule execution
- Predefined rule actions
- Controlling the actions of a rule
- Defining and using global functions
Rule structure

Rules are triggered by events pushed to BPM Events. Rules are composed of an event header, a Condition part or if clause (optional), and an Action part or then clause. The general form of a rule is:

```
rule <rulename>
    activated by <event headers>
    [if <conditions>]
    then "{" <actions> "}"
```

The general meaning of such a rule is:

"on reception of the specified events: if these events satisfy the conditions, then execute the actions."

The detailed syntax of a rule is as follows (for a more formal syntax specification, please refer to the language syntax in Rule syntax on page 167):

- **Rulename** — A string of characters with no space in it.
- **Event header** — A list of one or more events, identified by variables, that can trigger—or activate—the rule. If a single event is mentioned, then it means the rule is triggered each time such an event is notified to BPM Events. If more than one event types are mentioned in the header, then the rule triggers for each combination of such events. Combinations of events require access to past events. Each event type in the header is generally specified as:

  `<evt variable> of <evt type>::<evt subtype>{activation filter}`

  For example, the event sent on completion of a Business Process Server process instance specifies as:

  `evt1 of BP Server::W_COMPLETED{WORKSTEPNAME : "Start", PROCESSTEMPLATENAME : "Assign_A_Task_V2"}`

- **Conditions** — Specified in the optional if clause. This is a logical expression that describes additional selection condition(s) on the type(s) of event(s) described in the event header. The conditional expression may combine several simple conditions with the logical connectors and, or and not. If the conditions are satisfied for some incoming event(s), then the rule is said to be successfully triggered.

- **Actions** — The Actions part is a list of actions of possibly very diverse nature: call an external program, send an e-mail, generate a new event—either for internal or external use through an event adapter, update a monitoring report, enable/disable rules, etcetera. It executes only if the conditions are satisfied, that is, if the rule is successfully triggered.

Syntactical conventions and limitations

When writing rules in BPM Events, please use the following conventions:

- Every name and keyword in the rule language is case sensitive.
- When creating rules, do not use restricted Java key words in the rule syntax.
- Some names have limits on their length, primarily due to database restrictions.
- The attribute names of an event are case sensitive (for example, the standard event attributes `type, value, date, should always be lower case`).
- Make infopad names less than or equal to 21 characters, though the Business Process Server database accepts infopads with longer names.
• Make Application names (as they appear in the rule module headers, as well as in the PROCESS TEMPLATENAME attribute of BP Server events) no longer than 18 characters.

• The type field of an event should contain a character string less than 20 characters.

• Make the name of a scheduled item less than 255 characters (see the schedule(): Scheduling a future event on page 86).

• Put any string value (except attribute or variable names/keywords) between double quotes.

Event variables and expressions

This section explains event attributes, their presence, and presents examples.

Event attributes

The Condition and Action parts of a rule can share event variables. Event variables are any string of characters (letters or digits), with no space between characters. Event variables usually appear in event expressions that represent attribute values. For example, given an event variable EVT_1, each expression below returns the value of some event attribute. The type of all these attributes is String, except for the date attribute, which is of type Date. Conversion rules are detailed further.

• EVT_1.type (for example, purchase order event for purchase order XML message)

• EVT_1.value (optional—to refine the type)

• EVT_1.date (date when the event was generated)

• EVT_1.name (an attribute that is specific to this type of event)

In the last example, you may also write the expression: EVT_1.context.name, as the attribute "name" is not a standard event attribute, but actually belongs to the context of the event. This context may vary from one event to the other. The context object is itself a (complex) attribute of the event object, which has in turn possibly several attributes.

Note: The mention of the "context" can be omitted most of the time, as an attribute name that is different from other standard event attributes ("type","value","date") is automatically considered as an attribute of the event context. Context applies only to event, not to XML messages.

The event description in the rule header: <evt variable> of <evt type>::<evt subtype>{activation filter} actually refers to specific attributes of the event object:

• <evt type> must match the "type" attribute of the event (EVT_1.type)

• <evt subtype> must match the "value" attribute of the event (EVT_1.value)

Note: The {activation filter} must match the WORKSTEPNAME and PROCESSTEMPLATENAME of the event.

For example, the event header: EVT_1 of BP Server::W_COMPLETED{WORKSTEPNAME : "Start", PROCESSTEMPLATENAME : "Assign_A_Task_V2" selects every event "EVT_1" such as EVT_1.type = "BP Server", EVT_1.value = "W_COMPLETED".
Checking the presence of an event attribute

You may need to test in the condition part of a rule whether an event attribute is present or not. Indeed, the structure of events is flexible, and even for a given event type, the content of its context may vary from one instance to the other. The following test detects the presence of an attribute (here, named "attr"), regardless of its value:

- The condition: \((EVT_1.attr = \text{nil})\) is true only if the attribute "attr" is not part of the context of the event \(EVT_1\).
- The condition: \((EVT_1.attr \neq \text{nil})\) is true only if the attribute "attr" is part of the context of the event \(EVT_1\), regardless of its value.

Do not confuse the keyword \text{nil} here with the \text{null} value of the event attribute.

Examples of event variables and expressions

Example 1: The following rule records high disk utilization events per disk type, in the infopad \text{mytable} and implements the following monitoring scenario:

"Every time an event of type disk\_alarm and of value overload occurs over the network, record (count) the warning in a table indexed by disk\_type. The monitoring session should last from September 2001 to December 2001."

```
rule record_overload
    activated by event1 of disk\_alarm::overload
    if (event1.date >= YEAR:2001/MONTH:9)
    and (event1.date < YEAR:2001/MONTH:12)
    then { mytable[event1.disk\_type][1].total += 1;}
```

In this rule:

- "event1" is an event variable. Each time an event of type "disk\_alarm" and value "overload" is received, this rule is triggered and the event1 variable is substituted with the new event object.
- The conditions require event date attribute placement in the interval: September 1st, 2001 to November 30th, 2001 inclusive.
- If the event satisfies the rule condition, then the rule is triggered successfully and the action is executed. Here, the action is to increment the total number of disk failures into a table element. The table is indexed by disk type. The type of the failing disk is recorded as an attribute (disk\_type) of the context of the event.

Common variables, constants, and their scope

BPM Events rule language allows for using identifiers (variables, constants) that hold values of any type.

Constant values vs. variables

A constant value is declared using the keyword \text{val}. Such a symbol is intended to be assigned an initial value that does not change later (immutable). This is checked at compile time. Always complete such initialization using the operator \"=\".

Examples:

```
val blserver = getBlserver();
val session = blserver.connect("ebms", "ebms");
```
Variables are declared with the keyword `var`. Initialize them in the same way as a constant value, but modifiable later.

Examples:

```javascript
var mycounter = 1;
var taxrate;
var ratings = new infopad<cell{rating:int}>[10][2]("ratings");
```

**Note:** Although infopad variables are global to a module, they do not require declaring and initializing. Their declaration is implicit. This allows for grouping all infopad-related initial statements in the initial section of a module.

In the examples below, the variable is local to the iteration statement. The identifier `i` is incremented to iterate over the rows of an infopad, and must be declared as a variable.

```javascript
for (var i = 1; i <= employees.rowCount(); i++)
  {
    if (employees[i][1].name = evt1.name)
      employees[i][1].salary += evt1.raise;
  }
```

Except when declared, a variable must be assigned a new value using the operator `:=`. So the statement

```
i++
```

in the previous example could have been written

```
i := i + 1
```

When initialized in the declaration statement, the operator `=` is used, similar to constant values.

```javascript
var empnbr = employees.rowCount(); ...
empnbr := empnbr + newemps;
```

If no re-assignment is required for an identifier, then it is recommended to declare it as a constant value (`val`), instead of a variable (`var`). The compiler enforces this immutability.

### Scope of a value or of a variable

The scope of constant values and variables is basically within the block `{}` in which they are declared.

- If a constant value or a variable is declared inside a rule (action part), then its scope of use and visibility is the rule only. Reuse the same symbol in another rule: it refers to a different object.

- If a constant value or a variable is declared inside a `finalize {}` or `initialize {}` section, then it is only usable inside this section, and is not visible to rules.

- If a constant value or a variable is declared global, at module level, then it is usable by all rules and sections of this module, and its value has the life span of the module (that is, this value is preserved as long as the module loads).

A global constant value or a global variable is definable at module level, so that it is shared by all the rules. For example, this is the case when a file identifier requires reusing by several rules of a module, for writing into the same file. In such a case, the declaration is done immediately after the module statement, before any rule group.

**Example:**

```javascript
application BP_test_suites
module InfopadTest
val outputFile = openFile("e:\\oebpm\\ebmsapps\\BP_test_suites\\rules\\InfopadTest.out");

var taxrate;
group Group1 ...
```
Infopad variables

Infopad variables are automatically global to their module, even if they appear for the first time in the initialize{} section, and not at the top of the module like other global variables. In fact, they are implicitly declared at the global level. When an infopad is created in the initialize{} section, the infopad variable is automatically declared global, and only needs assigning to the new infopad. This explains why it does not appear like a variable declaration, but as a variable assignment, and uses the ":=" operator.

```plaintext
initialize {
  myPad :=
    new infopad < cell{avg_time:int, name:int}>[10][1]("myPad");
}
```

Note: An infopad variable can be referred from another module. If the infopad myPad is declared in module M1, then it references it from module M2 of the same application, by using its module name as qualifier: M1::myPad. In case M1 and M2 belong to different applications, then qualify the reference in addition by the application name M1: Appl::M1::myPad.

Recovery issues for global variables and values

Global variables and values are intended for sharing by all the rules of the module, and their life span covers the time the rule module is active (loaded) in the engine. When the module is loaded, these objects (variables and values) are created and initialized if there is any initialization statement. When BPM Events is shutdown and restarted, the following rules apply to global objects:

- Global values ("val") are recreated and reinitialized.
- Global variables ("var") are recreated, and reinitialized if applicable. If the variable is modified before shutdown, then its last value is not recoverable.

In some cases, this default behavior is not appropriate. For example, an application may expect the following behavior when recovering:

- The latest value of a global variable should be recovered.
- When a global variable holds a file descriptor resulting from an initial "openFile" statement, the file should reopen in "append" mode when recovering, in order to avoid erasing all previous data in the file.

Recovery of global variables and file handling on page 181 gives more details on how to implement these recovery behaviors.

Rule condition as a logical expression

A rule condition is a logical expression built with boolean operators and, or, and not. Using these operators, the rule condition may combine several simple conditions. A simple condition is a relational expression, for example:

```plaintext
(EVT_1.age < 21)
```

Simple conditions may contain event variables, as well as infopad references. They are described in Event variables and expressions on page 59. The value of a relational expression is true or false depending on the content of its event variables.
Example 1: The following rule condition is true when EVT_1 is substituted with an event reporting a subscription, for a person less than 21, from California or Nevada.

(EVT_1.formtype = "subscription") and 
((EVT_1.state = "California") or 
(EVT_1.state = "Nevada")) and 
(toInt(EVT_1.age) < 21)

Priority and parentheses

Usual precedence rules for operators apply (and has higher priority than or). At equal priority, the left-most operator is evaluated first, which means that parentheses apply by default from left to right. Note the additional parenthesis around the or expression, in the previous example. The condition above has the following structure:

Expression 1: c1 and (c2 or c3) and c4

Because the operator and has higher priority than or, a non-parenthesized expression like:

Expression 2: c1 and c2 or c3 and c4

would be interpreted as:

Expression 3: (c1 and c2) or (c3 and c4)

The meaning is quite different from Expression 1, where or is evaluated over the results of the two and sub-expressions, as opposed to only c2 and c3. A fully parenthesized expression equivalent to expression 1 is:

Expression 4: ((c1 and (c2 or c3)) and c4)

Relational expressions and operands

This sections explains the relational expressions and operands you can use in rule language.

Relational expression

Simple boolean expressions are called relational expressions. They are generally built using relational binary operators and are of the form: (operand1 <relational operator> operand2). Relational operators are:

• = (equality)
• != (inequality)
• > (greater)
• < (less)
• >= (greater or equal)
• <= (less or equal)
Arithmetic expression

The operands of a relational expression are arithmetic expressions of type real or integer. Such expressions are built with the standard arithmetic operators (that is, $+$, $-$, $\times$, $\div$, $\%$). Elements of such expressions are shown in Table 10: Arithmetic expression examples on page 64.

Table 10: Arithmetic expression examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant values</td>
<td>3.14</td>
</tr>
<tr>
<td>Numeric attributes of event objects</td>
<td>toInt(EVT_1.context.diskspace)</td>
</tr>
<tr>
<td>Functions returning a numeric type</td>
<td>duration(DAY, EVT_1.date, EVT_2.date)</td>
</tr>
<tr>
<td>Numeric elements of monitor tables</td>
<td>myTable[5][10].cpuload</td>
</tr>
</tbody>
</table>

Type conversions

Arithmetic expressions may combine subexpressions of any numeric type: real number (float) or integer (int). Enforce the conversion of any expression using such type conversion functions as toInt(), toString(), toDate(), toFloat().

Assume the attributes mydeadline and totalamount of the event evt1 contain respectively a String value representing a Java date of year 2000, and the String value "1050.25". The following conditional expressions are syntactically correct, and TRUE:

\[
\begin{align*}
& (\text{toDate} (\text{evt1.mydeadline}) < \text{YEAR:2001}) \\
& (\text{toInt} (\text{evt1.totalamount}) = 1050) \\
& (\text{toFloat} (\text{evt1.totalamount}) \geq 1000) \\
& (\text{evt1.totalamount} + \text{" dollars"} = \text{"1050.25 dollars"}) \\
& \text{toInt} (\text{evt1.totalamount}) + 100 = 1150)
\end{align*}
\]

Default conversions (or type inference) are automatically completed when the expected type is determined at compilation time, as in the case above where one operand of each expression shows the expected type (constants). Therefore, the corresponding pairs of conditions are equivalent:

\[
\begin{align*}
& (\text{evt1.totalamount} = 1050) \\
& \text{is equivalent to:} \\
& (\text{toInt} (\text{evt1.totalamount}) = 1050) \\
& (\text{evt1.totalamount} > 1050.25 - \text{evt2.rebate}) \\
& \text{is equivalent to:} \\
& (\text{toFloat} (\text{evt1.totalamount}) > 1050.25 - \text{toFloat} (\text{evt2.rebate}))
\end{align*}
\]

However, if an integer is divided by an integer, then the result is not a real number (float); it is an integer rounded to the integer value of the quotient.

Use conversion functions in the following situations:
Rule structure

• `toInt()` can take as argument an expression with:
  • a String type, in which case the string is assumed to contain a valid numeric representation (int or float)
  • a Date type, returning an integer that represents a date in terms of milliseconds.

• `toFloat()` can take:
  • a String type, in which case the string is assumed to contain a valid numeric representation (int or float)
  • an Int type, returning a Float (real) representation.

• `toString()` can take:
  • a Date type, in which case the returned value is a string that represents the date in a readable format.
  • an Int type or a Float type, returning a string representation.

• `toDate()` can take:
  • a String type, in which case the string is assumed to contain a long integer which is a valid representation of a Java date (milliseconds since Jan 1st, 1970). Note that an expression like:
    `toDate(toString(event1.date ))` is not equivalent to: `event1.date`, as `toString()` converts the date in a different format intended for human readers.
  • an Int type, returning a date representation. The Int type is assumed to contain a valid date representation.

As a good practice, we recommend that you always use type conversion functions when needed, and to not rely on implicit type conversions, except between numeric types such as `float` and `int`. Indeed, the default type conversions are handled as usually done in programming languages, and makes it possible to mix `int` types and `float` types in the same expression. For example, the following expression is correct, and results in a float value, because at least one of its operands is a float.

`toInt(evt1.totalamount) + 250.75`

The attribute `totalamount` of the event requires a conversion, as event attributes are always accessed as string values (except for the attribute date).

However, implicit type conversion is supported when the expected type is `statically` inferred, that is, at compile time. For example, the following expression compiles successfully and executes properly:

`evt1.totalamount + 250.75`.

String expression

A string expression combines strings using the concatenation operator `+`, or using string-valued functions. For example, the following string depends on an event attribute, known at run-time only.

"response time critical: request type = " + EVT_1.reqtype

Comments in rule language

Add comment lines anywhere in a rule module, and start them with the two characters: "/ /". It is also possible to add a comment—starting with "// "—at the end of a line containing rule syntax.
Rule modules and rule groups

Rules are usually written for a particular application. If we are writing rules for the application "myApp", then there is a standard rule file, named "myApp_rules.txt", associated with this application. When installing the myApp application, Business Process Server automatically looks for this rule file, and loads its rules into BPM Events. (Actually, it looks for the compiled version of this file, which is another file in a special format, called "myApp_rules.bpo".) We also call this file the main rule module of the application myApp. Similarly, when loading the myApp application, BPM Events Admin looks for this rule file and loads its rules into BPM Events.

The rule module usually represents a set of rules that implement a logical unit of the application. The rules inside a rule module are grouped into rule groups. A rule group represents a subdivision in the module. It has no special semantics (this is not a name scope for rules, and there is no specific operation associated with groups). The group organization shows on the Rule Editor, allowing for a tree-like representation of modules.

Rule modules of an application

When editing a Business Process Server application that does not yet have rules with the Business Process Server development tool (Progress Developer Studio for OpenEdge) and then open the Rule Designer, the main rule module file is automatically created. Additional rule modules are developed for an application, with a name that has the form of XYZ_rules.txt.

A rule module is a text file structured as shown in the following example:

```plaintext
application myApp
module myModule
  group myGroupA {
    rule myRuleA1 (evt1 of BP Server::PI_ACTIVATED){PROCESSTEMPLATENAME : "myApp"} if {<action>};
    rule myRuleA2 ...;
    rule myRuleA3 ...;
  }
group myGroupB {
    rule myRuleB1 ...
    rule myRuleB2 ...
  }
group myGroupC {
    rule myRuleC1 ...
    rule myRuleC2 ...
  }
initialize {
  <action1>;
  <action2>;
}
reinitialize {
  <action3>;
}
finalize {
  <action4>;
}
```

For the above example of a rule module:

- The name myApp in the first line (application myApp) must be the same name as the application to which the module belongs. If the module is not attached to any existing application, then this defines a new application name.

- The name myModule in the second line (module myModule) must be consistent with the rule file name: if the module name is monitor_policy_123, then the name of the file containing this module should be monitor_policy_123.txt. In addition, the main rule module of an application must have a module name which is the same as the application name, extended with "_rules". For example, if the application name
is myApp, then name the main module for this application: myApp_rules and store in a file named: myApp_rules.txt. The header of a main module is then as follows:

```
application myApp
module myApp_rules
```

- A module can import other modules. In that case, an import statement is added:

```
module myModule import moduleA, moduleB;
```

- The previous example means that the two modules imported are loaded before myModule, at loading time. Normally, when a module M1 imports a module M2, it is because the execution of M1 rules also requires rules and objects from M2 to be loaded. However, one may also use the import mechanism to automatically load a set of modules along with the main rule module, when an application installs. If the imported module belongs to another application, then qualify it by this application name: myOtherApp::moduleB.

- Rule Groups: After the two header lines above, the module contains a list of rule groups.

  - initialize{} section: This section contains a possibly empty sequence of actions, similar to those definable in the action part of a rule. It must NOT contain rules. These actions are automatically executed when the module is loaded in the rule engine (this happens when the application is installed through the Administration module, or loaded through BPM Events Admin). They usually perform initial setup tasks for the application, like creating infopads and initializing them. An important property of this section is that it never reexecutes during recovery.

  - reinitialize{} section: In general, this optional section is not necessary. It contains a sequence of actions, similar to those in the initialize section. These actions automatically execute when the module is recovered from a previous session, after BPM Events has stopped and restarted. They usually perform initial setup tasks for external systems, that may not have been aware of the last shutdown (in the case of failure), for example, reinitialization of connections, etcetera. Note that initialize section and reinitialize section are mutually exclusive: they are never executed together. See Administering and operating BPM Events on page 135, for more details on the difference between rule loading and recovery.

  - finalize{} section: This section contains a sequence of actions, similar to those defined in the action part of a rule. It must NOT contain rules. These actions automatically execute when the application module is removed from the rule engine. This happens when the application is uninstalled. These actions usually perform final cleanup tasks, like deleting infopads from memory and removing their image in the database.

### Rule module and name scope

The rule module usually represents a set of rules that implement a logical unit of the application. Several modules are required when writing a large application. The module also serves another purpose: it is a name scope for the rule names. A rule name must be unique inside its module, but does not have to be unique across modules of the same application: conflict does not occur between rule r1 in module A and a different rule r1 in module B, when loaded together.

The general structure of a rule module is:

```
application <app_name>
module <module_name>
  [import <gen_module_name> ("," <gen_module_name> )* ";"]
  group <group_name> "["[ "... " ]"]"
  ( rule <file_name> "["[ "... " ]"]")*
  [ [ initialize ] "["<action_list> "]"]
  [ [ reinitialize ] "["<action_list> "]"]
  [ finalize "["<action_list> "]"]
```
The imported modules above may be given only their name, or an application-qualified name if they belong to another application:

```plaintext
<gen_module_name> ::=<module_name>
<app_name> "::" <module_name>
```

In the syntax above, the symbols `<>`, `(*)`, and `[ ]` have the following meaning, similar to their role in conventional regular expressions:

- `<>` denotes an application-specific name
- `( . . . ) *` denotes "repeat expression 0 or more times"
- `[ . . . ]` denotes "optional expression". Note that `initialize` and `finalize` sections are optional

In the current version of BPM Events, a rule file comes in two formats:

- The text rule file, which contains rules written in the format described in this document, and has the suffix ".txt" (for example, `myApp_rules.txt`) or ".bps" (for example, `myApp_rules.bps`).
- The compiled rule file, which contains a compiled, ready-to-load version of the rules. Such files have the suffix ".bpo", as they are coded in a particular format. The compiled version of the previous example is `myApp_rules.bpo`.

**Rule groups**

The rules inside a rule module are grouped into rule groups. A rule group represents a set of rules that achieve a particular function, like managing a particular monitoring structure (infopad), or implementing a business policy. You decide how to group the rules. Usually, small rule modules use only one group. For large modules, rule groups provide a way to structure rules so that they are easier to edit and manage. The notion of group does not have any particular semantics for the rule engine or rule loader. A group contains the syntax:

```plaintext
group <groupname> "{ " <rule1> ";" <rule2> ";" ... <ruleN> ";" "}
```

**Current rule base**

The set of rules loaded in the engine at any given time is called the current rule base. The current rule base results from loading several rule files. For recovery purposes, an image of the current rule base is maintained in the database.

**Adding and removing rules dynamically**

You can dynamically add to or remove rules from the current rule base in the rule engine in two ways: either by loading/unloading rule modules or by enabling/disabling rules. Each of these methods is described in this section, for more details about the operations involved, refer to `Loading rules` on page 141 and `Unloading rules` on page 142, and `Dynamically enabling and disabling rules` on page 69.
Dynamic loading and unloading of rule modules

When installing an application through Business Process Portal Administration or BPM Events Admin, the main rule module of this application is automatically loaded. If the application has more than one rule modules under its rule folder, then those modules that are imported by the main rule module are also loaded.

It is possible, however, to add new modules for this application, separately and at any time. Similarly, these additional modules can be individually unloaded. Complete these operations using the administration utility BPM Events Admin, described in Loading rules on page 141 and Unloading rules on page 142.

**Note:** Any module can import other module(s), and an imported module can in turn import other modules.

Loading/unloading rule modules is done according to the following principles:

- When loading a module, all imported modules are automatically loaded first. This means that their `initialize()` sections execute first, before the `initialize()` section of the importing module.
- When unloading a module, all imported modules automatically unload after they are imported in any other modules. This means that their `finalize()` sections execute after the `finalize()` section of the importing module.
- A module M2 that is imported by another module M1 cannot be unloaded separately from M1. It can only unload when M1 is unloaded.

The BPM Events Admin utility allows you to see, at any time, which current rule modules and current rules load into the engine.

Dynamically enabling and disabling rules

It is sometimes necessary to modify the current rule base at a finer level of granularity than the rule module. Remember that a module is divided into rule groups, which are generally units of rules achieving a common function (that is, some monitoring). You may not add or remove dynamically rules or rule groups to a module that is loaded (unless you modify the module itself, and "replace" it, as previously mentioned), but it is possible to "turn off" some of its rules or some of its rule groups. For example, turning off the rule group inside a module that implements a monitoring option that is no longer necessary, may speed up the processing of a future events. Or, a module may contain several monitoring options, only a subset of which are necessary at any time, but because they all work with the same objects (infopads), we still want them in the same module.

The turning off of a rule or a rule group is called "disabling" the rules. Disabling a rule does not actually remove the rule from the engine, like `unload` does. It only makes the rule(s) invisible to oncoming events. It is then possible to "enable" the rule again. When a module is loaded, all its rules are initially enabled.

The enabling/disabling of rules and rule groups is accomplished through BPM Events Admin, or from other rules through the action statements "enableRule," and "disableRule," described in Dynamically enabling and disabling rules on page 69 later in this chapter.

Operational semantics of rules

The following sections discuss event correlation, event filtering and general semantics of rule loading and rule execution.
Event correlation

By correlating events, rules can detect event patterns spanning long periods of time. Once a combination of events is found that satisfies the event pattern condition, the rule has succeeded in correlating these events.

Example

For example, in order to measure the completion time in business hours of a BP Server process (called here "HelpDesk") we correlate two events: the activation (PI_ACTIVATED) event and the completion (PI_COMPLETED) event of this process instance. The following rule calculates a completion time as the difference between the time stamps of these two events. The resulting time is used to update a value averaging all completion times for the HelpDesk process.

```
rule record_process_time
  activated by
  evt1 of BP Server::PI_ACTIVATED{PROCESSTEMPLATENAME : "HelpDesk"},
  evt2 of BP Server::PI_COMPLETED{PROCESSTEMPLATENAME : "HelpDesk"}
  correlated with INDEX
  then {
    val report = HelpDeskReport[1][1];
    report.avg_time ::= 
      avg(duration(B_HOUR, evt1.date, evt2.date), report.count++, 1);
  }

  {HelpDeskReport := new infopad <cell{avg_time:int}>[1][1]("HelpDeskReport");
  }
```

In the sample above, the mode correlated with INDEX guarantees that every combination of events (evt1 and evt2) is correlated based on the default association criterion, which is the process instance ID associated with each event. In other words, these events are generated by the same process instance. Without this keyword, we must explicitly write the correlation condition in the if clause:

```
{evt1.PROCESSINSTANCEID = evt2.PROCESSINSTANCEID}
```

**Note:** Progress Software Corporation strongly recommend using correlated with INDEX whenever a rule correlates events of type BP Server that generate from the same process instance. If not, the performance of the rule degrades in proportion to the number of events present in cache.

Do not use correlated with INDEX if an index was not defined for an event type. To define an index for your own event type of an XML message, implement the public function get_group_id(). See Event caching and indexing on page 152 for more information.

The triggering event and the correlating event

When a rule like record_process_time above is correlating two or more events, the rule engine attempts to trigger the rule each time an event of any of the profiles described in the activated by clause, is notified. When processing the rule record_process_time, the rule engine exhibits the following behavior, depending on which type of event it receives:

- If an event of profile BP Server::PI_ACTIVATED is received, then try finding in the event cache, an event of profile BP Server::PI_COMPLETED that comes from the same process instance (same INDEX value). Then test the activation filter. If the condition succeeds, then execute the "then" clause. Repeat for each possible PI_COMPLETED event.
- If an event of profile BP Server::PI_COMPLETED is received, then try finding in the event cache, an event of profile BP Server::PI_ACTIVATED that comes from the same process instance (same INDEX value).
Then test the activation filter. If the condition succeeds, then execute the "then" clause. Repeat for each possible PI_ACTIVATED event.

In the first case, the PI_ACTIVATED event is called the triggering event for this rule execution. The other event (PI_COMPLETED) is called the correlating event. In the second case, the reverse is true: the triggering event is the PI_COMPLETED event.

**Note:** When these events are notified according to a known sequence in time, as it is the case here, the triggering event is always the last one of the sequence.

You may prevent a rule from triggering on some event variables, when you are sure that the triggering will not succeed on these event variables. In the previous example, we know the rule triggers only by the PI_COMPLETED event. Do not trigger it when an event of type PI_ACTIVATED is notified, because no corresponding event PI_COMPLETED exists yet in the cache. Refer to Controlling rule execution on page 79 for more information about control.

## Event filtering

When BP Server generates an event, it is posted to the BPMEvent table. BPM Events's Database Event Channel retrieves these events from the BPMEvent table in the database and passes them to the Rule Engine for evaluation.

The Event Filtering mechanism of BPM Events retrieves the relevant events based on the Current Rule Base. This mechanism significantly improves the performance of BPM Events by reducing the number of events that are retrieved and processed.

### Event filter syntax

The Event Filtering mechanism uses an improved syntax to filter events, as shown in the following example.

```plaintext
rule update_MyTaskReport
  activated by
  EVT_1 of BP Server::W_COMPLETED{WORKSTEPNAME : "Start", PROCESSTEMPLATENAME :
  "Assign_A_Task_V2"},
  EVT_2 of BP Server::W_COMPLETED{WORKSTEPNAME : "DoTask", PROCESSTEMPLATENAME :
  "Assign_A_Task_V2"}
correlated with INDEX

{...}
```

The default behavior is to maintain the original semantics of rules, so, if there are weak conditions, for example, BP Server::{}, then it retrieves all the events, therefore disabling other strong conditions for event retrieval. In this case, there is no performance gain. If you still want to use weak conditions, but not enforce retrieval of all possible events (that is, the processing of only filtered events is allowed), then you can achieve processing of only filtered events by using NO_EVENT_FILTERGENERATION, as shown in the following example.

```plaintext
rule update_MyTaskReport
  activated by
  EVT_1 of BP Server::W_COMPLETED{WORKSTEPNAME : "Start", PROCESSTEMPLATENAME :
  "Assign_A_Task_V2"},
  EVT_2 of BP Server::W_COMPLETED{WORKSTEPNAME : "DoTask", PROCESSTEMPLATENAME :
  "Assign_A_Task_V2"}
correlated with INDEX
  : NO_EVENT_FILTER_GENERATION

{...}
```

This means the information extracted from this rule is not used for event filtering from BPMEvent table.
Event filtering example

The following code sample illustrates event filtering.

```java
rule update_MyTaskReport
activated by
EVT_1 of BP Server::W_COMPLETED{WORKSTEPNAME : "Start", PROCESSTEMPLATENAME :
"Assign_A_Task_V2"},
EVT_2 of BP Server::W_COMPLETED{WORKSTEPNAME : "DoTask", PROCESSTEMPLATENAME :
"Assign_A_Task_V2"}
correlated with INDEX
{...}
```

General semantics of rule loading and rule execution

The operational semantics of rule processing are defined by the following principles:

**Principle 1: No defined rule ordering should be assumed**

The set of rules in the engine are called the current rule base. These rules are indexed and organized depending on which events activate them. An event that can trigger several rules triggers them in an order unknown by the user. In other words, do not assume that rules will be executed in the same order as the order in which they are written in a module. When an event triggers two rules, you do not know which one will be triggered first. The only way to enforce the execution order of two rules R1 and R2 when processing an event E, is to chain these rules explicitly, using an internally generated event. Assuming R1 activates before R2, then only R1 should be activated by E. R1 then generates an event E which carries over information from E, and triggers R2.

**Principle 2: The sequencing of module loading and unloading**

When a module of rules loads into the rule engine, if this module imports other modules, then these modules are loaded first. This means that their initialize{} section is executed before the initialize{} section of the importing module. Conversely, when a module unloads, it unloads before unloading its imported modules. This means its finalize{} section is executed first, before the finalize{} section of the imported modules.

**Principle 3: The ordering of events is based on their notification time**

Incoming events execute in the order of their notification and are based on the time they are notified to the engine or to the queuing system, which may be different from their origination time as it appears in their date attribute.

**Principle 4: The processing of external events is serialized**

The processing of an external event (see Event management on page 151 section) includes the evaluation of all the rules this event can trigger. In turn, this includes the execution of corresponding actions, as well as the evaluation of the rules triggered by internal events generated by these actions (closure of rule chaining). We call the set of all these cascading actions the action closure of the event. This principle states that the processing of the next external event does not start before the action closure of the current event is completely processed.

**Principle 5: The rule engine is forward chaining only, breadth-first**

Rules trigger each other through internal event generation. On reception of an event, either external or internal, rules first execute their condition part (if clause), then their action part (then clause). The rule engine is then doing forward chaining, breadth-first. It is, however, possible to write rules so that they emulate backward-like reasoning, like starting from hypothesis in diagnostic systems.
**Principle 6: The execution order of actions in a rule is sequential**

The action part of a rule is a sequence of actions, executed in the same order. The effects of an action (that is, infopad modification) is visible to the next action in the same rule.

**Principle 7: When rules trigger other rules, the action tree is executed breadth-first**

When an action in a rule R generates other actions (by generating an event that trigger other rules), these actions are executed after all remaining actions in R. Assume the sequence of actions of a rule is: A1, A2, A3. When A2 executes, and if it generates an internal event, then the processing of this internal event is likely to trigger rules: the actions entailed by these rules are executed after A3. More generally, the chaining of rules execution defines an *action tree*. This action tree is executed breadth-first, not depth-first. This principle is a corollary of (5) and (6).

**Principle 8: The event correlation tries all possible event combinations, using events older than the triggering event**

When an event E triggers a rule R that does event correlation (a rule using event variables E1, E2), then the processing of R on reception of E tries all possible combinations. If E matches with E1, it tries all possible pairs (E, E2) by searching previous events in the event cache that match the rule conditions on E2. Then the search is done for (E1, E) pairs if E also matches E2 type.

*Note:* You may stop the search at the first successful combination, using the execution mode: "try SINGLE_CASE".

In any case, the triggering event is always correlated with earlier events (that is, events with an earlier reception date) that are already in store.

*Note:* Events received before the triggering event in turn trigger the rule later, so no event combination is forgotten. This is the case for correlated events with an earlier origination date than the triggering event (date attribute), but were received after due to some latency problem.

**Principle 9: Event discarding that occurs during the processing of an event e has no side effect on the entire processing of e**

If an action of the action closure of E is an event discard (of E or of any other correlated event), then this does not affect the set of actions generated in the action closure. In other words, the order in which rules are triggered by an event does not matter with regard to event discarding actions. When an event E1 triggers several rules, the action of discarding an event E2 is delayed until all rules triggered by E1—and possibly correlating E2—have been triggered (E2 can be same as E1).

**Principle 10: Actions other than discard may have a side effect on the evaluation of the next event combination, in a rule doing event correlation**

When an event E1 triggers an event-correlating rule that is then evaluated several times (for each possible combination of events), the effects of any action (that is, infopad modification) are visible to the next evaluation of the condition part for the same rule. Therefore it is recommended that you do NOT write correlating rules that contain actions potentially affecting the outcome of their conditions.

**Principle 11: Only events that can activate correlation rules are placed in the event cache**

These events require discarding explicitly by application developers. Other events are automatically discarded after rule executions.
Time operations and expressions

Each event received by BPM Events is assumed to be dated (that is, has a time stamp). BPM Events also has an internal notion of date for various rule operations (scheduling, comparison with current date, timing of events generated by BPM Events, etcetera). The date assumed by BPM Events is the coordinated universal time (UTC). UTC is normally supported by the Java Virtual Machine, but this depends in turn on the host environment.

Almost all operating systems assume that 1 day = 24 × 60 × 60 = 86400 seconds. In UTC, however, about once every year or two an extra second is added, called a "leap second." The leap second is always added as the last second of the day, and always on December 31 or June 30. Most computer clocks are not accurate enough to reflect the leap second distinction.

Because the notion of time is critical to the processing and correlation of events, BPM Events provides a rich set of time constructs and time operators. The basic principle of time management is that time is a discrete concept. Because the maximum precision, in BPM Events is the second, the time dimension is viewed as an infinite, denumerable sequence of seconds. View a time interval as a finite number of contiguous seconds. With this in mind, we then define the following constructs as part of BPM Events rule language.

Date expressions

This section explains how date expressions are formed in the rule language.

Date values in rule language

Dates are expressed by aggregating date units (year, month, day, hour, minute, second) separated by "/". A few examples of dates are listed below:

- YEAR:1999

The maximum precision of a date is the second. The minimum precision is the year. In case a date is not precise enough for date operations, the missing date units are completed by default with the lowest value allowed by the unit. Each unit of time takes the values shown in Table 11: Time values on page 74.

<table>
<thead>
<tr>
<th>Event object</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>year (YEAR)</td>
<td>Four digit year number</td>
</tr>
<tr>
<td>month (MONTH)</td>
<td>1 to 12</td>
</tr>
<tr>
<td>day (DAY)</td>
<td>1 to 30, 31, 28 or 29, depending on the month</td>
</tr>
<tr>
<td>hour (HOUR)</td>
<td>0 to 23</td>
</tr>
<tr>
<td>minute (MIN)</td>
<td>0 to 59</td>
</tr>
<tr>
<td>second (SEC)</td>
<td>0 to 59</td>
</tr>
</tbody>
</table>
The hour unit (HOUR) takes values between 0 and 23, as the first hour after midnight is 0. The minute (MIN) and second (SEC) units take values between 0 and 59. You may express the date marked by the first second of January 1999 by any of the following date expressions:

YEAR:1999/MONTH:1/DAY:1/HOUR:0/MIN:0/SEC:0
YEAR:1999/MONTH:1/DAY:1
YEAR:1999/MONTH:1
YEAR:1999

**Date unit in rule language**

Each field of a date expression is a date unit. A date unit contains a value and a time type as follows:

time_type:value

**Time types are:**

YEAR (year)
MONTH (month)
DAY (day)
HOUR (hour)
MIN (minute)
SEC (second)

The following code examples are valid date unit expressions (as well as date expressions):

YEAR:1998
MONTH:event1.someattr

**Note:** When using constant numbers to specify the value of a time unit, the "0" digit should not appear at the beginning. For example, write March as MONTH:3, not as MONTH:03.

**General date expressions in rule language**

It is possible to construct a date with expressions in place of the values of the date units. For example:

YEAR:event1.myYear/MONTH:1/DAY:(3 + toInt(event1.myDay))

Parenthesize any composed expression (as in the DAY unit above).

The event attribute date is of type Date, and can therefore be used at any place a date value is authorized. The following condition is true if the time stamp (origination date) of the event event1 is later than or same as January 15th, 2000:

(event1.date >= YEAR:2000/MONTH:1/DAY:15)

Convert an event attribute that represents a Java long format of a date into a type Date with the function toDate() (Refer to Date conversion functions on page 76). In that case, use the following date expression:

(toDate(event1.duedate) >= YEAR:2000/MONTH:1/DAY:15)

**Syntax:**

```
DateExpr ::= 
Year_unit ["/" Month_unit ["/" Day_unit ["/" Hour_unit ["/" Min_unit ["/" Sec_unit]
Year_unit ::= "YEAR:" <numeric_expression>
Month_unit ::= "MONTH:" <numeric_expression>
Day_unit ::= "DAY:" <numeric_expression>
Hour_unit ::= "HOUR:" <numeric_expression>
```
Min_unit ::= "MIN:" <numeric_expression>
Sec_unit ::= "SEC:" <numeric_expression>

The following expression is still a valid date, though the Month unit is missing between Year and Day (replace it by its default value "1"):

YEAR:1998/DAY:15/HOUR:8

Extracting elements of a date

The date object in BPM Events represents a specific instant in time with precision measured down to the second. Given a date object, it is possible to extract each of its fields (year, month, day, hour, minute and second), as an integer value, using projection functions for each field.

For example, if the value of EVT_1.date is:


Then the following numerical equations are true:

EVT_1.date.year = 1998
EVT_1.date.month = 12
EVT_1.date.day = 15
EVT_1.date.hour = 18
EVT_1.date.minute = 30
EVT_1.date.second = 5

Date conversion functions

The section explains the date conversion functions you can use in the rule language.

Converting a date using a long-coded Java date in a rule

BPM Events provides a conversion function toDate(), that generates a Date value from any string that represents a Java internal date representation (long integer, representing the number of milliseconds since January 1, 1970, 00:00:00 GMT). For example, use this function on any event attribute necessary as a date in a rule, except for the date attribute of the event itself (named "date").

The following example illustrates the use of the toDate() function. It assumes that the deadline of a task is coded as a Java date in the event attribute duedate. The rule schedules a reminder event for firing two business days before the due date.

rule reminder
activated by event1 of BP Server::W_ACTIVATED(WORKSTEPNAME : "DoTask"
then {
  schedule("tasktimeout", toDate(EVT_1.context.duedate) - 2*B_DAY, event1,
  type:"TimeOut", value:"TimeOutCheck", messagetype:"reminder");
}

We can use projection functions directly on a date expression:

(toDate(EVT1.deadline).month > 10 )

The condition above is true if the deadline attribute of the event has a month equal to November or December.
Converting a date to long in a rule

BPM Events provides the `toInt` function to convert a Date value into a long integer in a rule. The following example illustrates the use of the `toInt` function, which returns the value of an event date in milliseconds.

```java
val longValue : int = toInt(evt1.date) ;
```

Time expressions

A time expression (or "time") represents a quantity of time, like "seven days" or "2 hours and 30 minutes." Do not confuse it with a Date.

Date functions

The section explains various date functions you can use in the rule language.

Date comparison

The standard arithmetic operators apply for date comparison. For example, the following relational expressions are syntactically correct:

- `YEAR:1997/MONTH:7 < EVT_1.date`
- `EVT_1.date + 15*DAY >= YEAR:1999/MONTH:6/D:15`

equality (=), inequality (!=)

Two dates are equal only if their date-units contain the same value, pair by pair. For example, the condition below is true:


after (>), before (<), after_or_equal (>=), before_or_equal (<=)

The following inequalities hold:


dueDate

The `dueDate` and `duration` functions are based on regular time. The `dueDate` function returns the due date by adding the duration in milliseconds to a given date, as shown in the following example.

```java
dueDate (startDate: date, duration:int): date
```

The following sample returns the due date by adding two hours to the event date:

```java
val dt1 : date = dueDate(evt1.date, 2*60*60*1000) ;
```
duration

The `dueDate` and `duration` functions are based on regular time. The `duration` function returns the duration, or the difference in time between the end date and the start date, in milliseconds. See the following example.

```
duration (startDate: date, endDate: date): int
```

The following sample returns the duration between the event date and the due date (dt1) calculated in the previous `dueDate` sample. It returns a value of two hours, using `java.util.Date` and `java.util.Calendar` classes:

```
val dur1 : int = duration(evt1.date, dt1) ;
```

**NOW**

The function `NOW` returns the current date.

Business time

BPM Events supports a basic notion of business time, which is based on the calendar that is defined and installed in Business Process Portal Administration. The calendar that is installed and defined as the default calendar is also referred to as the system calendar. Business time can also be based on another calendar that is defined and installed in Business Process Portal and is specifically specified; this is referred to as the specified calendar. For more information, see Business Process Portal Administrator’s Guide.

Business time is specified by adding a "b" to the `duration` and `dueDate` functions, as in `bduration` and `bdueDate`. The `bduration` and `bdueDate` functions are based on the business time definition described in the Business Process Portal Administrator’s Guide.

Using the `bdueDate` function

The `bdueDate` function returns the due date by adding the duration in milliseconds to a given date. This calculation is based on the system calendar. If no system calendar is defined, then an error is raised. An example of this function based on the system calendar follows:

```
bdueDate (startDate: date, duration: int): date
```

The following sample returns the due date by adding two hours to the event date. The due date is calculated based on the system calendar defined in the system:

```
val dt2 : date = bdueDate(evt1.date, 2*60*60*1000) ;
```

The following `bdueDate` calculation is based on the given calendar. If the specified calendar is not defined, then an error is raised. An example of this function based on the given calendar follows:

```
bdueDate (startDate: date, duration: int, calendarName: string): date
```

The following sample returns the due date by adding two hours to the event date. The due date is calculated based on the calendar specified in the argument:

```
val dt3 : date = bdueDate(evt1.date, 2*60*60*1000, "sbmcalendar") ;
```
**Using the `bduration` function**

The `bduration` function returns the difference in time between the end date and the start date in milliseconds. This calculation is based on the system calendar. If no system calendar is defined, then an error is raised. An example of this function based on the system calendar follows:

```java
bduration (startDate: date, endDate: date): int
```

The following sample returns the duration between the event date and due date `dt2` calculated in the previous `bdueDate` function. The duration is calculated based on the system calendar defined in the system. It returns a value of two hours:

```java
val dur2 : int = bduration(evt1.date, dt2) ;
```

The following `bduration` calculation is based on the given calendar. If the specified calendar is not defined, then an error is raised. An example of this function based on the given calendar follows:

```java
bduration (startDate: date, endDate: date, calendarName: string): int
```

The following sample returns the duration between the event date and due date `dt3` calculated in the previous `bdueDate` function. The duration is calculated based on the calendar specified in the argument. It returns a value of two hours:

```java
val dur3 : int = bduration(evt1.date, dt3, "sbmcalendar") ;
```

**Controlling rule execution**

The following sections describe means to control rule executions.

**Avoiding unnecessary or unwanted rule executions**

When an event is notified to BPM Events, this event could trigger many rules, that is, it can satisfy the Condition part of these rules. Consider the situation where if a rule evaluation succeeds on an incoming event (EVT_A), then we do not want to pursue the evaluations of other rules that could be triggered by the incoming event. For example, the following set of rules `{Rule_R1, Rule_R2}` handle different critical situations in a similar way (each rule generates the same kind of alarm). But if both rules can apply, we do not want to generate two alarm events. In that case, one can add the clause: *terminated with FIRST_SUCCESS*, immediately after the event header. If the rule is triggered (that is, its condition succeeds, and its action part is executed), then this clause prevents EVT_A from triggering any other remaining rules.

```java
rule myrule1
  activated by
  EVT_A of disk_problem::failure,
  EVT_B of payroll_status::running
  terminated with FIRST_SUCCESS
  if (EVT_A.host = EVT_B.station)
     then {
      sendMail("paul@abc.com","critical disk failure while payroll running",EVT_A);
     }
rule myrule2
  activated by
  EVT_A of disk_problem::failure
  terminated with FIRST_SUCCESS
  then {
      sendMail("paul@abc.com","critical disk failure",EVT_A);
  }
```
Assume that a triggering event EVT_A of type "disk failure" comes in, and that an event which satisfies the condition on EVT_B is already logged in the event cache of BPM Events already. This means that both rules can succeed when EVT_A comes in. However, we want only one to succeed (the first one). The termination clause guarantees that no matter which rule is evaluated first and succeeds, only one alarm is generated: the other rule is not tried. So, if the payroll program was running on weekdays when the failure occurs, then only one alarm is generated. When using this mode, the order of the rules in the rule file is important.

A rule can also select events based on their type alone, not just their combination type::value. In the previous example, in case any event of type disk_problem must trigger the rule (not only disk_problem events reporting a failure), then the rule should be:

```
rule myrule1
  activated by
  EVT_A of disk_problem::,
  EVT_B of payroll_status::running
```

### Canceling multiple evaluations of a rule

You may trigger a rule several times on a single event, if it involves event correlation.

Consider a situation where we try to detect at least one high-priority purchase order currently processing on server S, as soon as the load of server S reaches a critical threshold. If yes, then we take some action, that is, decrease the load of the server. The rule that takes action requires correlating two types of events: (1) server overload notice, (2) open purchase order of high priority. The rule is:

```
rule Server_load_reduce
  activated by
  EVT_A of request::PurchaseOrder,
  EVT_B of ServerStatus::load
  try SINGLE_CASE
  if (EVT_A.priority = "High") and
  (toInt(EVT_B.load) > 10) and
  (EVT_A.server = EVT_B.server)
  then {
    perform(<some action>);
  }
```

When an event triggers the rule on EVT_B (server load over 10), several purchase orders may be currently open on this server, each of which has logged an event that satisfies the condition on EVT_A. In that case, the normal behavior of a rule is to execute its action (perform) as many times as there are event pairs (EVT_A, EVT_B) that satisfy the rule condition (Event pairs may share the same event.). However, we want the rule to execute only once, no matter how many orders are in, provided there is at least one for this server. The clause `try SINGLE_CASE` automatically cancels the evaluation of all remaining pairs as soon as one pair (EVT_A, EVT_B) triggers the rule successfully, thus enforcing a single action evaluation.

### Disabling events as rule triggers

When a rule does event correlation, it can be triggered on any of its event variables. However, because the events correlated by the rule often follow a sequential pattern, we know what should be the last event of the sequence. For example, consider the following rule:

```
rule my_process_completed
  activated by
  EVT_A of BP Server::PI_ACTIVATED{PROCESSTEMPLATENAME : "Assign_A_Task_V2"},
  EVT_B of BP Server::W_ACTIVATED{WORKSTEPNAME : "DoTask", PROCESSTEMPLATENAME : "Assign_A_Task_V2"},
  EVT_C of BP Server::PI_COMPLETED{PROCESSTEMPLATENAME : "Assign_A_Task_V2"}
  correlated with INDEX
```
if (...)  
then {
    perform(<some action>);
}

Clearly, the three events to correlate, EVT_A, EVT_B and EVT_C, are expected to notify BPM Events according to this same sequence. In that case, it is useless for the rule engine to activate the rule on the variable EVT_A or EVT_B, because it will not find in the event store an event of type BP Server::PI_COMPLETED to correlate with or that comes from the same process instance (same INDEX value, or PROCESSINSTANCEID). Indeed, by the time the engine receives EVT_A or EVT_B, no completion event has yet been notified for this process.

The above rule triggers successfully once the BP Server::PI_COMPLETED event is received. It should then only be triggered by events that match EVT_C. We can specify event variables that should NOT be used as triggers on the rule, by prefixing them with "***" (star). The rule my_process_completed_2 below provides the same outcome as the rule my_process_completed above, except that its execution is faster, because it is not unnecessarily triggered each time an event matching EVT_A or EVT_B is notified.

```plaintext
rule my_process_completed_2
activated by
  EVT_C of BP Server::PI_COMPLETED{PROCESSTEMPLATENAME : "Assign_A_Task_V2"},
  *EVT_A of BP Server::PI_ACTIVATED{PROCESSTEMPLATENAME : "Assign_A_Task_V2"},
  *EVT_B of BP Server::W_ACTIVATED{WORKSTEPNAME : "DoTask", PROCESSTEMPLATENAME :
    "Assign_A_Task_V2"}
correlated with INDEX
if (...)  
then {
    perform(<some action>);
}
```

Only after receiving an event matching EVT_C, the rule looks up the event cache for past events matching EVT_A and EVT_B, attempting to correlate with them. We recommend that you put the disabled event profile at the end of the activated by clause—as shown in the rule above—since this has a positive impact on the performance of the rule evaluation.

**Note:** When disabling event variables as rule triggers, the rule engine ignores any sequence of event that uses these variables as the most recent event. Even if some event is normally not expected to be the most recent (the triggering event) of a combination, this may happen sometimes. Indeed, because of event channel latency—especially if events come from various sources—the actual notification order to BPM Events may not always reflect the origination order. In such cases it is prudent not to disable these event variables.

### Predefined rule actions

While the Condition part of a rule consists of detecting some event patterns of interest, the Action part is executing actions that relate to these event patterns. The actions that a rule executes when successfully triggered are specified in the then {...} clause of the rule. The BPM Events rule language provides a set of predefined Actions, which includes:

- `println()`
- `generate()`
- `perform()`
- `sendMail()`
- `schedule()`
Chapter 5: The rule language

- unschedule ()
- discard (), discardIndexEntry(), discardAllIndexEntry()
- enableRule ()
- disableRule ()
- Infopad operations

These actions take most of their input from the following three sources:

- The attributes of the events that triggered the rule
- Infopads
- The output of previous action

Each of these predefined rule actions are described in the following sections.

println(): Printing values to a file

When developing or debugging applications, it is important to be able to trace the rule execution and check the value of the variables and expressions that the rules manipulate.

Printing values in the bpmevents.output file

The action println() takes a primitive value in argument (String, Int, Float, Boolean) and prints its content in a text file, by default in the BPM Events.output file. It prints one line and changes the line automatically for the next call. The general syntax is:

`println( <primitive_value> )`

For example:

```java
rule abc
activated by EVT_A of BP_Server::W_COMPLETED{WORKSTEPNAME : "DoTask",
PROCESSTEMPLATENAME : "Assign_A_Task_V2"}
then {
println("rule activated: abc ");
println("workstep completed:" + EVT_A.WORKSTEPNAME + " performer=" +
EVT_A.PERFORMER);}
```

Note: println(), when used without explicit file definition, is intended as a development and debugging feature. If you want to collect outputs in a production environment, then use the file version described below, where println is used on an object of type file.

It is also possible to print the content of a composite value like an event, by converting it into a string value, using toString().

Printing values in a file

In order to redirect the output to a text file, use the following sequence of statements:
• Open a file. Give the full path of the file. If the file does not exist, then it will be created. The `openFile` function can take a second argument: if `true`, then the file is opened in append mode. If `false` or absent, then the file is opened in rewrite mode (all previous content is lost). The syntax is:

```plaintext
val <file_identifier> = openFile(<file_path>);
val <file_identifier> = openFile(<file_path>, <append_flag>);
```

• Print one or more lines in the file, using:

```plaintext
<file_identifier>.println(<primitive_value>);
```

• Close the file, using:

```plaintext
<file_identifier>.close().
```

You may also flush the file: `<file_identifier>.flush()`. This flushing has the same semantics as in Java and guarantees that the file contents are fully printed from any memory buffer. For example:

```plaintext
rule abc
  activated by EVT_A of BP_Server::W_COMPLETED(WORKSTEPNAME : "DoTask",
  PROCESSTEMPLATENAME : "Assign_A_Task_V2")
  then {
    val myfile = openFile("C:\temp\test.log");
    myfile.println("hello from abc rule");
    myfile.println(2001);
    myfile.println(true);
    myfile.close();
  }
```

In case the output of several rules (or of several activations of the same rule) requires writing to the same file, then a global file variable should be used. Indeed, opening and closing the file should only be done once, otherwise the file would be rewritten from the beginning each time. A variable that is global to a module must be declared and used as follows:

• declare a global file variable immediately after the module declaration statement,

• use `println()` on this variable in any rule that needs to write on the file, and

• close the file in a final rule or in the `finalize{}` section.

For example:

```plaintext
application BP_3_test_suites
module InfopadTest
val outputFile = openFile("e:\BP_test_suites\rules\InfopadTest.out");
group Group1
  rule Rule1
    activated by event1 of Infopad::remove
    then{
      outputFile.println(" ");
      outputFile.println("Rule1 activated.");
      schedule("InfopadCheckRemoval1", NOW + 10*SEC, type: "Message", value: "Scheduling");
      outputFile.flush();
    }
}
initialize {
  outputFile.println(" ");
  outputFile.println(" Start of InfopadTest");
  //---------
  outputFile.println(" Defining Static Infopads.");
  infoPad1 ::= new infopad<cell{strSlot:string, intSlot:int,
```
floatSlot:float>[5][5]"infoPad1";
}

finalize {
  discard(infoPad1);
  outputFile.println("*** InfopadTest done");
  outputFile.println(" ");
  outputFile.flush();
  outputFile.close();
}

Note: Global values like the outputFile value above, are not recoverable because they are transient objects. Only infopads are recoverable. In order to avoid recovery problems, it is preferable to store any global value in an infopad. This is described in Recovery of global variables and file handling on page 181.

Recovery considerations

Use the Global values, like outputFile above, with caution. The initialization statement that is associated with their declaration is re-executed when the module is recovered. This is not always desirable, as variables do not recover their latest state, if they have been modified after initialization.

Use a programming technique based on functions and infopads, as illustrated in Recovery of global variables and file handling on page 181, to recover the value of variables.

Note: When writing in a file (other than bpmevents.log) after recovery, rewrite the file if it has been open as for an initial loading. Recovery of global variables and file handling on page 181 shows how the initial file opening statement may distinguish the context of its execution, whether upon recovery or upon normal module loading.

generate(): Generating internal events

Use the generate() predefined rule action (of type functional statement) to generate events to BPM Events itself. The general syntax is:

generate(
  <list_of_evt_attribute_names_values>,
  event:<evt_var>,)

The arguments are: Values of attributes. These arguments represent a possibly empty list of named values, each of the form: <evt_attribute>:<value>. This list should start with the type of the new event. Each (name:value) pair sets an attribute of the new event. If the event:<evt_var> argument of generate() the reused event is present, then it is used as a basis to generate the new event: the generated event is then initialized as a copy of the reused event. If no reused event is specified (last optional argument), then the list of named values should always include at least the two items: type:... and value:.... The named values contain the following semantics with regard to the reused event:

- If the same attribute name already exists in the reused event, then the named value replaces the value of this attribute in the (context of the) new generated event.
- If no similar attribute exists in the reused event (or if there is no reused event), then the named value initializes a new attribute in the (context of the) generated event.
- "event" argument (optional): "reused event" is an event variable that represents an incoming existing event—captured in the condition part of the rule—to be reused for generating the new event.
Predefined rule actions

Examples of internal event generation are given below.

```java
generate(type: "ApplicationFailure",
    value: "IO",
    hostName: EVT_13.hostName,
    processId: EVT_13.processId,
    diskId: EVT_12.diskId);
generate(
    type: "Timeout",
    task: EVT_1.taskname,
    event: EVT_2);
```

**postDB(): Posting an event into the database event queue**

Use the `postDB()` predefined rule action to post events generated by BPM Events into the database event queue.

```java
postDB {event MyEventType::MyEventValue{Attr1: 123, Attr2: "aString"}};
```

The above is a free format event where any number of attributes can be placed in the event context.

**postJMS(): Posting an event into a JMS queue**

Use the `postJMS()` predefined rule action to post events from BPM Events into a JMS queue. Post and create XML messages as follows:

```java
fun create_jms_event(a: string, b: string, c: string): XML {
    val xml = getXML(a, b, c);
    val type_id = "%XML<"purchaseOrder", "http://example.com/po">";
    return createJMSEvent(xml, type_id, map("a": "z", "b": "1111"));
}
postJMS("queue", "BPEventQueue", create_jms_event("1223.25", "150.00", "NY"));
```

The `postJMS` predefined rule action takes a list of events as parameters, so several events can be posted in the same action call.

```java
postJMS("queue", "queueName", msg1, msg2, ..., msgN);
```

**sendMail(): Sending e-mail automatically**

Use the `sendMail()` predefined rule action to generate an e-mail. The general syntax is:

```java
sendMail(<e-mail address>, <From>, <header>, <content>)
```

The arguments are:

- **Argument 1**: E-mail address.
- **Argument 2** (optional): Originator information or an e-mail address to reply to, if appropriate. If this argument is not given, then default "sender" data from the `oebps.conf` file is used.
• Argument 3: A string value that represents the header of the e-mail.

• Argument 4: The body of the e-mail. Given either as an event or as an expression. If given as a long string expression, introduce line breaks by typing the character "\" followed by a new line character (hit Return key).

Example 1: In this example, the content of the event EVT_1 is converted in text format and becomes the body of the e-mail.

    sendMail("paul@tdiinc.com","order is late",EVT_1);

Example 2: In this example, the body of the e-mail results from the concatenation of an attribute value of the event EVT_1 (order ID), with a message.

    sendMail("paul@acme.com","yourmanager@acme.com", "order is late", EVT_1.orderID + " must be given higher priority." = "\ Signed: your manager");

schedule(): Scheduling a future event

Rules can schedule internal events, which are triggered some time in the future. The predefined rule action used to schedule an event has a syntax similar to the generate() action. The schedule() action specifies in the same way a list of pairs (attribute name, attribute value) that are elements of the future event. The date attribute should be present, and its value should be a future date (later than the current date when the rule is evaluated). Otherwise, the event is immediately triggered.

An example of scheduling an action is:

    schedule("aScaryEvent",
        YEAR:2000/MONTH:1/DAY:1/HOUR:0/MIN:0,
        type:"alarm",
        value:"Y2K_alert",
        mailto:"allIT");

Use the schedule() predefined rule action to schedule an internal event at a specified date. The scheduler keeps track of all scheduled events and activates them when they are due. The general syntax is:

    schedule(<event_name>,
        <date>,
        <list_of_evt_attributes_assignments>,
        event:<evt_var>)

The arguments are:

• Argument 1: <event_name>. This name identifies the event—or a group of events—in the scheduler. Several scheduled events can have the same name. Such identification is useful in case the event(s) are removed from the scheduler (that is, un-scheduled).

    Note: The <event_name> name should be less than 255 characters.

• Argument 2: event date (<date>). This is the date of the scheduled event, that is, when generated. The date is expressed following the UTC standard, which is supported by Java (JDK). In case a date is given that is already past, the event is immediately generated. Often, a date expression that adds some time to the current date is given here, like: NOW + 3*DAY.

• Following arguments: Values of attributes. These arguments represent a list of named values of the form: <evt_attribute>:<value>. They have the same semantics over the reused event (if any) as for the
generate() action. Unless a reused event is specified (last optional argument), the list of named values should always contain at least the two items: type:... and value:....

- Event argument (optional): The "reused event" event:<evt_var> is an event variable that represents an incoming existing event (captured in the condition part of the rule) to be reused for defining the scheduled event.

Note: Like for the generate() call, special attributes of the reused event are not automatically reported in the new event: the event ID attribute and the PROCESSINSTANCEID attribute.

Example 1: The rule below schedules an event for generation three hours after reception of an event marking the beginning of a process. The scheduled event repeats the content of the triggering event, except for its date (3 hours later), its type (now "timeout") and an additional attribute in the context ("notes").

```plaintext
rule myschedule
  activated by EVT_1 of BP Server::PI_ACTIVATED
  if (EVT_1.PROCESSTEMPLATENAME = "Upgrade Memory")
  then {
    schedule("myevent", EVT_1.date + 3*HOUR , type:"timeout",
              value:"timeoutcheck", notes:"upgrade should be finished in 2 hours",
              event:EVT_1);
  }
```

Example 2: In this example, the date is calculated by adding three hours to the time the rule is executed (given by the function NOW). Ideally, if the event is received by BPM Events immediately after being generated, then the event time stamp (its date attribute) is approximately the same as the current date (NOW). So the actions in examples 1 and 2 should be equivalent. In practice, there may be some significant difference between the notification date of an event (the date it is received by BPM Events) and the origination date (the time stamp date actually stored in the event). If what really matters is the difference between the origination times of both events, as in this example (If you want to schedule a check exactly three hours after the Upgrade Memory process started), then use the origination date, as in Example 1. If you want a scheduling based on the current date, then use NOW.

```plaintext
schedule("myevent", NOW + 3*HOUR , type:"timeout", value:"timeoutcheck",
         notes:" upgrade should be finished in 2 hours", event:EVT_1);
```

Example 3: In this schedule action, an absolute date is given. No reused event is given. The scheduling date is independent from the date when the rule actually executes.

```plaintext
schedule("myevent", YEAR:2000/MONTH:1/DAY:1 , type:"alarm",
         value:"alarmcheck", notes:"check for year 2K problems");
```

Using the event scheduler

You can use the scheduler for many purposes. One typical use is the "timeout" event. For example, if an event of type "alarmXYZ" is notified at 3:00 P.M., then we may expect this alarm to be handled in the next three hours. Assume that this handling itself generates an event of type "alarm_handled" to BPM Events. BPM Events’s rules checks that such an event has been notified in time, otherwise the alarm is escalated. This is done in two steps:

1. Schedule a "timeout" event as soon as "alarmXYZ" is received, for activation three hours later.
2. When the timeout event fires off, check if an event "alarm_handled" has been notified in the last three hours. If not, then the rules take action.
unschedule(): Removing a scheduled event from the scheduler

Use the `unschedule()` predefined rule action when a previously scheduled event requires unscheduling, before it goes off. The general syntax is:

```plaintext
unschedule(<item_name>,
<list_of_evt_attributes_assignments>,event:<evt_var>)
```

The arguments are:

- **Argument 1**: The `<item_name>`. All scheduled items (events) with this name are unscheduled. If one wants to select the events to be unscheduled by other properties than the name (see other arguments), then the event-name argument should be the NULL keyword.

- **Following arguments** (optional): Event pattern definition. These arguments represent a list of named values, of form: `<evt_attribute> :<value>`. They have the same semantics over the reused event as for the `generate()` action, that is, they override any attribute value of the event specified in argument 2. If there is no argument 2 (reused event), then these named values define the event pattern.

- **Event argument** (optional): "reused event" is an event variable that represents an incoming existing event (captured in the condition part of the rule) for reusing as an event pattern. All scheduled events that satisfy the pattern are removed from the scheduler. More precisely, in order to be removed, a scheduled event must:
  - Have all the context attribute names that are specified in the reused event.
  - Have attribute values similar to the ones of the reused event, except for the date. If argument 1 (name) is present, then only the scheduled events with this name, which also match the reused event, are unscheduled.

**Note**: As with the `generate()` call, special attributes of the reused event are not automatically reported in the event pattern: the event ID attribute and the PROCESSINSTANCEID attribute. Unscheduling based on event pattern can be combined with unscheduling based on name, if both arguments are present. Only the events with this name AND that match the pattern are removed.

**Example 1**: If the "Upgrade Memory" process is completed in time, then there is no need to activate the scheduled event to trigger the timeout check (see the examples for `schedule()` action above). In order to avoid unnecessary checking and event processing overhead, we unschedule this event as soon as the process completes.

```plaintext
rule job_done
activated by event1 of BP Server::PI_COMPLETED(PROCESSTEMPLATENAME = "Upgrade Memory")
```

We could give a more definitive name to the scheduled event, that is, the ID of the "Upgrade Memory" process itself, which is unique to each process. In that case, the action call would be:

```plaintext
unschedule(event1.PROCESSINSTANCENAME);
```

**Example 2**: The following action removes all the "timeout" events currently scheduled for the "HelpDesk" process. We assume that the process name was stored in the context of the scheduled events.

```plaintext
... then {
unschedule(NULL, type:"timeout", PROCESSTEMPLATENAME:"HelpDesk");
}
```
Discarding infopads and events

The following sections describe how to use the `discard()` action call to discard infopads and events.

**discard(): removing infopads**

When an application (or a rule module) is unloaded, we recommend discarding all its infopads, unless these infopads are shared and still used by other applications. If the application infopads requires archiving (that is, their database image), then complete this before unloading. Discarding infopads is usually done from the `finalize{}` section:

```plaintext
finalize {
    discard(myinfopad1);
    discard(myinfopad2, myinfopad3);
}
```

The `discard()` call can take a variable number of arguments. Both the memory and the database images are discarded.

**Note:** If an application or rule module is unloaded without discarding its infopads, then the database image of these infopads is still present. If the application (or module) is reloaded, then the execution of the infopad creation statements, in its `initialize{}` section, automatically destroys the previous versions of the infopads and recreate them.

**discard(): removing individual events**

Rules can manage the event cache by removing events (discarding). It is common to use an event only once. For example, once an alarm has triggered a rule that handles the appropriate response, there is no point in keeping this event object in memory. However, in other cases, even after the event is used, it is better to keep it in the event cache, as it could be used later by some rule in order to correlate it with some more recent event. The following rule removes an event after using it.

```plaintext
rule clean_up
    activated by EVT of overload::
    then {
        sendMail(...EVT...);
        discard(EVT);
    }
```

The rule detects events of type overload, and uses them for some action (In this example, sending an e-mail). After doing so, it removes the event from the cache. More than one event variable can be given in the argument of `discard()`, for example, `discard(EVT1, EVT2)`. Use the `discard()` predefined rule action mostly to remove events generated by rules (internal events), or generated from a third party source. Events generated from BP Server on the execution of process instances are preferably removed from the internal and external caches using the `DiscardIndexEntry()` or `DiscardAllIndexEntry()` rule actions.

**Note:** As mentioned in Operational semantics of rules on page 69, events are not discarded immediately (that is, while executing the rule), in order to avoid undesirable side effects.
**discardIndexEntry(): removing all events related to the same process**

You may index events stored in the cache. Only one event index is supported, and BPM Events is configured so that this index is automatically implemented for all events coming from the BP Server process engine (event type = BP Server). These events are indexed based on the process instance ID they come from (context.PROCESSINSTANCEID).

Often, at the end of a process instance (marked by the reception of an event of value = "PI_COMPLETED"), all the events generated by this process become useless; that is, the application rules do not need any correlation with them. In that case, the application rules should explicitly discard these events from the cache as well as from the persistent image of the cache.

Each application generally has an "event clean-up" rule that discards the group of events corresponding to a particular index entry, using `discardIndexEntry` with the process instance ID in argument:

```
rule end_process
  activated by evt of BP Server::PI_COMPLETED{PROCESSTEMPLATENAME : "myProcess"} then { discardIndexEntry(evt.PROCESSINSTANCEID); }
```

**Note:** This applies only to BP Server events.

```
rule end_purchaseorder_process
  activated by evt of BP Server::PI_COMPLETED{PROCESSTEMPLATENAME : "PurchaseOrder"} then { discardAllIndexEntry(evt.PROCESSINSTANCEID); }
```

**Note:** It is important for each application to discard events from the cache that have become useless, otherwise these events clutter the cache and affect performance.

**discardAllIndexEntry(): removing all events related to the same process and subprocesses**

When a process P1 generates a subprocess P2, it may happen that the application rules require correlating events across P1 and P2. For example:

- P1 is a product ordering process.
- P2 is a product customization subprocess, that may be triggered by P1 depending on the content of the customer order.

In such a case, even if one of the two processes terminates before the other (for example, the customization subprocess P2), its events may be kept in cache, so that a rule can correlate with these events (that is, a rule calculating the invoice may require data from the customization process). In that case, you can discard the event as follows:

- Rules triggered by P2 events should not discard any P2 events at the end of P2 (that is, do not use `discardIndexEntry`).
- Rules associated with P1 should, at the end of P1, dispose of both P1 and P2 events. Do this in a single call, by using: `discardAllIndexEntry()`.

The call `discardAllIndexEntry(EVT1.PROCESSINSTANCEID)` takes the ID of the parent (or root) process instance (P1 in the above example), and discards from the cache (and from its database image) all events of this process and its subprocess generated so far. Because the root process terminates after its subprocess(es), its completion event triggers the cache clean-up:

```
rule end_purchaseorder_process
  activated by evt of BP Server::PI_COMPLETED{PROCESSTEMPLATENAME : "PurchaseOrder"}
  then { discardAllIndexEntry(evt.PROCESSINSTANCEID); }
```
In case the subprocess outlives the parent process, then the event completion of the subprocess must clean the cache from all events from both parent and subprocess. In order to do this, the subprocess must "remember" the ID of the root process. Each event generated by a subprocess contains, in addition to the ID of this subprocess (PROCESSINSTANCEID attribute), the ID of its root process (the "top" parent process). This "root process" ID is reported in the attribute named RPID of any event generated by a subprocess. The following rule triggers at the completion of any instance of the subprocess "OrderCustomization," and clean up events related to this instance and to its root process instance (of process "PurchaseOrder" in this example):

```java
rule end_customization_process
activated by evt of BP Server::PI_COMPLETED
(PROCESSTEMPLATENAME : "OrderCustomization")
then { discardAllIndexEntry(evt.RPID); }
```

**Note:** This applies only to BP Server events.

### Enabling and disabling rules

When an application is installed, its main module is loaded and, with it, all the imported modules (import statement). It is also possible to load additional module files, or unload modules, using the BPM Events Admin utility. However, such procedures are usually intended for minor application upgrades, when a module requires reloading after some rule change.

Dynamically enabling and disabling rules is a business level operation. A group of rules usually implements such business functions as a monitoring option, a business policy, or a process control. In such cases, we want to enable or disable these functions at run-time, and do this as the result of a decision from other rules.

When a rule module is loaded (that is, when installing an application), all its rules are automatically enabled. It is then possible to disable some of the rules of the module. When some rules of this module are disabled, they remain loaded in the engine memory, but are "turned-off" so that they can no longer be triggered by any event. The logical behavior is the same as if the rule had not been loaded; it is "invisible" to events. When these rules are enabled again, they go back into the state they were in when first loaded: they are "turned-on" and responding again to events.

**enableRule(<application>, <module>, <rulename>)**

This predefined rule action enables a particular rule. This rule was loaded previously (with the module it belongs to). For call effectiveness, the rule must be in a disabled state prior to the call. Indeed, when a module is loaded in BPM Events, all its rules are already automatically enabled by default. The syntax is:

```java
enableRule(<applicationname>, <modulename>, <rulename>)
```

**Note:** These arguments are of String type: they must appear in quotes when given by value.

Example:

```java
enableRule("HelpDesk", "moduleA", "apply_discount");
```

**disableRule(<application>, <module>, <rulename>)**

This predefined rule action disables a particular rule. For call effectiveness, the rule must be in an enabled state prior to the call. The syntax is:

```java
disableRule(<applicationname>, <modulename>, <rulename>)
```
Controlling the actions of a rule

Each rule can have one or more actions to be executed in sequence. An action can consist of the following various forms:

- **Predefined function invocation:** `function_name(argument1, argument2,...);`
- Invocation of the BP Server API, in order to control process instances, get status info, etcetera.
- Generating and sending an e-mail.
- Generating internal events that trigger other rules.
- An infopad operation, that is, value assignment: `ReqStats[2][4].name := event1.name;`
- An administration operation, that is, like discarding used events from the event cache.

Syntactic structure of rule actions

The rule language provides some scripting capability in order to control the execution flow of actions. The Action part of a rule is structured as a sequence of statements:

```plaintext
<ActionStatement> ::= 
    "{" <ActionStatement1> <ActionStatement2>..."}
```

Or more concisely:

```plaintext
<ActionStatement> ::= 
    "{" (<ActionStatement>)* "}
```

There are six classes of Action Statements:

- Functional action statement.
- Declaration action statement.
- Assignment action statement.
- Conditional action statement.
- Compound action statement.
- Iterative action statement.

The general syntax is:

```plaintext
<ActionStatement> ::= 
    <FunctionalStatement> | 
    <DeclarationStatement> | 
    <AssignmentStatement> | 
    <ConditionalStatement> 
    <CompoundStatement> | 
    <IterativeStatement> | 
```
Functional action statements

Functional action statements are of the form \( f(...) \) and represent function call invocations. Most predefined actions listed previously (generate(), schedule(), discard(), etcetera) belong to this class of statements. Their detailed syntax and semantics is given in the following sections. User defined functions can also be defined in BPM Events, but are not supported by the Rule Editor. The general syntax for a functional statement is:

\[
\text{<FunctionalStatement>} ::= \text{<funcname>} \("\ <\text{argument_list}> \)\)\ ";"
\]

For example,

\[
\text{sendMail("paul@acme.com", \"yourmanager@acme.com\", \"order is late\",\nEVT_1.orderID + \" must be given higher priority\")};
\]

Declaration action statements

Use the Declaration statements to declare and initialize identifiers (either variables or constants) to reuse in other actions. The general syntax for the declaration statement is:

\[
\text{<DeclarationStatement> ::= \"val\" <identifier> \"=\" <expression> \";\" \mid \n\text{\"var\" <identifier> \"=\" <expression> \";\"}}
\]

Identifiers declared with \texttt{val} cannot have their initial value changed later. They are constant values. An example is:

\[
\text{val blserver = getBLserver();}
\]

In this example, reuse \texttt{order} as a place holder in other action statements, but not as a new value. In the following example, \texttt{rate} is declared with an initial value of 0.5, but can be updated in future actions, for example, using an assignment of the form: \texttt{rate := 0.6}.

\[
\text{var rate = 0.5;}
\]

The scope of these identifiers is the smallest bloc \{...\} in which they are defined, usually the Action part of a rule, or the \texttt{initialize} or \texttt{finalize} section of a rule module.

Assignment action statements

Use the Assignment statements to assign new values to declared variable identifiers or to infopad cells. The general syntax for an assignment statement is:

\[
\text{<AssignmentStatement> ::= \begin{array}{l}
\text{<identifier> \"=\" <expression> \";\" \mid} \\
\text{<identifier> \"+=\" <numeric_expression> \";\" \mid} \\
\text{<identifier> \"-=\" <numeric_expression> \";\" \mid} \\
\text{<identifier> \"++\" \mid} \\
\text{<identifier> \"--\"} \\
\text{<identifier> \"::=\" <updating_functional_expression> \";\" \mid}
\end{array}}
\]

We provide the following three examples:

1. \texttt{ReqStats["ProductOrder "] [ evtl.region].totalamount := 0;}
In the first example, the value 0 is assigned to the totalamount slot of the infopad cell identified by row: "ProductOrder" and column given in event attribute: evtl.region.

2. ReqStats["ProductOrder"] [ evtl.region].maxamount ::= max(evt1.amount);

In example 2, the maximum between the slot: maxamount of the infopad cell and the event attribute: evtl.amount is calculated, and reassigned to the maxamount slot. The operator "::=" indicates that it does NOT mention the first argument of the function on the right side (that is, max), assumed to be the same as the expression on the left side. You may rewrite Example 2 using the normal ":=" assignment operator, but it would then be longer, as shown in Example 2' below:

(2') ReqStats["ProductOrder"] [ evtl.region].maxamount := max(ReqStats["ProductOrder"] [ evtl.region].maxamount, evtl.amount);

3. ReqStats["ProductOrder"] [ evtl.region].totalamount += evtl.amount;

In Example 3, the operator "+=" is a contraction commonly used in programming languages, for incrementing the left hand side identifier with the value in right hand side. The operator "++" indicates that the value of the left hand side identifier is incremented by 1.

Infopad variables

A special mention is necessary regarding infopad variables. The infopads are created in the initialize section of a module, but their reference must be visible to all rules. Therefore, they should be declared at module level as global identifiers, not only in the initialize section. However, for the sake of easy development, the need for declaring global variables has been avoided in recent Business Process Server versions. You can directly assign an infopad variable in the initialize section as it is automatically and implicitly declared as global to the whole module:

ReqStats :=
new infopad<cell{maxamount:float, totalamount:float}>[4][3]("ReqStats");

Use the ReqStats identifier in any rule of the module, referring to the same infopad. If the same infopad is shared across modules or even applications, then use different variables from one module to the other, provided that the same expression on the right side is used to initialize them, referring to the "internal" infopad name, shown in the above example as "ReqStats".

Conditional action statements

Conditional action statements can execute different actions based on some condition. Note that the condition here is not the same as the rule condition (if clause of the rule); but is a condition used inside the Action part of the rules, which means it is checked only when the rule is successfully triggered.

The general syntax for a conditional statement is:

<ConditionalStatement> ::= 
"if" <condition> ")" <ActionStatement> [ "else" <ActionStatement> ]

Note: The else part is optional. The condition follows exactly the same syntax as for the main condition part of the rule.

In the following example of this rule, a sequence of three action statements exists in the Action part.

rule earlybird_discount
activated by evtl of BP Server::PI_ACTIVATED(PROCESSTEMPLATE_NAME : "CustomerRequest")
then {
    if { (evt1.totalamount >= 500) 
    and (evt1.reqtype = "ProductOrder") 
    and (ReqStats["ProductOrder"]{ evt1.region}.count < 1001)) 
    ReqStats["ProductOrder"]{ evt1.region}.count += 1;
    ReqStats["ProductOrder"]{ evt1.region}.totalamount += evt1.amount;
}

The second one is the conditional statement, which uses a complex condition. It has a functional statement
ReqStats["ProductOrder"]{ evt1.region}.count += 1;

It does not have an else part. The statement:
ReqStats["ProductOrder"]{ evt1.region}.totalamount += evt1.amount;

is the next statement after the conditional statement.

### Compound action statements

Use Compound Action statements to group a sequence of Action statements of any kind into a bloc, as shown by the following notation {...}. Use such a bloc at any place where a single Action statement is allowed, for example in the then or else part of a Conditional statement or inside a loop. The general syntax for a compound statement is:

```plaintext
<CompoundStatement> ::= 
    "{" (<ActionStatement>)* "}"
```

In the following example of a rule, the conditional statement in the Action part uses a compound statement in its then part. The compound gathers the two statements:

```
{ReqStats["ProductOrder"]{ evt1.region}.count += 1; ReqStats["ProductOrder"]{ evt1.region}.totalamount += evt1.amount;}
```

The rule is checking the amount of a purchase order, and decides to apply a discount AND increment some counter, if the amount is more than $500. The last action statement, order.flagOn("continue");, is executed in any case.

```plaintext
rule earlybird_discount
activated by evt1 of BP Server::PI_ACTIVATED(PROCESSTEMPLATENAME : "CustomerRequest")
then {
    if {evt1.totalamount >= 500)
    {ReqStats["ProductOrder"]{ evt1.region}.count += 1; ReqStats["ProductOrder"]{ evt1.region}.totalamount += evt1.amount;}
}
```

### Iterative action statements

Iterations (loops) can execute repetitive actions. They are generally used to iterate over infopad cells.

There are two general syntactical forms for iterative statements. They are:

```plaintext
<IterativeStatement> ::= 
    "for" <index_increment_expr> <ActionStatement> | 
    "do" <ActionStatement> "from" <value_set> "where" <condition>
```
The following statement iterates over the rows of the infopad employees. It adds to the salary of an employee a value contained in the raise attribute of an event evt1. The name attribute of the event contains the name of the employee to select. Note that the ActionStatement of the loop is a conditional statement, embedded inside {...} like compound statements, for readability. Note also that the identifier "i" is declared as a variable (var) as its value changes, while the maximum rows are used as a constant value and therefore are declared as a value (val) to ensure that no action changes it.

```java
val rt= employees.rowCount();
for (var i = 1; i <= rt; i++ )
{ if (employees[i][1].name = evt1.name)
    employees[i][1].salary += evt1.raise; }
```

There is also another way to express an iteration over infopad cells, using the rowKeySet() (or colKeySet()) function that generates the list of indexes for the dimension of interest. Each of these functions actually produces the list of labels (not indexes) associated with the index values of the corresponding dimension. In case no labels were given to the dimension, default labels were generated (string version of the actual index value). In that case, these default labels are returned and used in the iteration.

```java
do employees[xi][yi].salary += evt1.raise;
from xi:employees.rowKeySet()
where (employees[xi][yi].name = evt1.name);
```

A similar loop for the cells of the two-dimensional mypad infopad is as follows. Note that nested loops are not necessary:

```java
do mypad[xi][yi].attstr := evt1.myname
from xi:mypad.rowKeySet(), yi:mypad.colKeySet()
where (mypad[xi][yi].attnum > 100);
```

These last iteration statements require the creation in memory of value sets resulting from the function calls rowKeySet() and colKeySet(). This may be inconvenient in terms of overhead (memory) for large infopads.

## Defining and using global functions

When a specific group of actions must be reused several times by the module rules, it is preferable to abstract these in a function. The function is declared at the module level, between the "module ..." statement, and before the first rule group. Any sequence of statements allowed in the action part of a rule can define the body of a function.

The general syntax is:

```
"fun" <func_name> "(" <list_parameters> ")" [ ":" <returned_type> ] "{" <func_body> "}"  
```

**Note:** Progress Software Corporation recommends not to use duplicate global function declaration in a module. The returned type may or may not be specified, depending whether the function returns a value or not.

A function can return a value using the statement:

```
return <expression_or_var> ";;"
```

In the following example, a function is defined to generate events in the Business Process Server event queue in database:

```java
application Tutorial1
module Tutorial1_rules
// TUTORIAL 1: simulate the creation of three purchase orders processes
```
// (three PI_ACTIVATED events generated as in BP Server processes)
// and decides if a discount applies.
// this function is used to generate BP Server events, and store them in DB queue
fun simulate_BP_Server_evt(i:int, total:float, p:string) {
    postDB(event BP Server::PI_ACTIVATED(ProcessInstanceId: i, RPID:i,
                                          ProcessTemplateName: "PurchaseOrder", totalAmount: total, product: p));
}

]group discount_policy{
    ...

Invoke the function from any place where a rule action is executed: either in the initialize{} or finalize{} section, or inside a rule action part, as any other rule action. The following initial section is logging three events (simulating BP Server events) in the Business Process Server event queue in the database:

    initialize {
        simulate_BP_Server_evt(1, 350.25, "video_monitor");
        simulate_BP_Server_evt(2, 650.45, "personal_computer");
        simulate_BP_Server_evt(3, 150.0, "modem");
    }
    finalize {
}

When returning a value, use a function inside any expression where this value type is expected.

Although generally defined for the rules of a module, use a function—such as global objects as infopads—from other modules of the same application, when qualified by the module name (import this module by the using module).
Persistent structures

BPM Events supports two types of persistent structures — Infopads and Persistent Maps. This chapter describes how rules can create, update and modify infopads and persistent maps.

For details, see the following topics:

• Infopads
• Persistent maps
• Using persistent maps

Infopads

BPM Events rules can generate data—they create, update and externalize dynamic reports as a result of their monitoring activity. These reports are stored in memory data structures called infopads. Infopads can store various kinds of data, other than application reports.

Infopads store any type of global data that the rules may use (that is, write or read), including:

• Application parameters (that is, a list of e-mail addresses of persons to notify).
• Rule parameters (that is, the critical threshold for a resource load).
• A dynamic business report that is modified based on new events.
Defining Infopads, cells, and slots

An infopad is a table of one or two dimensions, which is composed of elements called cells. A cell is a data-structure made up of slots, or attributes. A slot is an atomic value of the following types:

- **String of characters** (*string*)
- **Integer** (*int*)
- **Real number** (*float*)

All the cells of an infopad have the same structure (that is, slot names and types). The following code example represents the cell structure used in monitoring the response time of requests:

```plaintext
avg_resp_time:float,
max_time:int,
current_user:string
```

The previously mentioned cell has the following three declared slots, and one undeclared slot:

- **avg_resp_time**, a number of type real, records the average response time of all requests monitored to date (the time unit depends on the application and is matter of interpretation: it could be seconds, minutes, hours, etcetera).
- **max_time**, a number of type integer, records the maximum response time recorded to date.
- **current_user**, a string of characters, records the user ID of the last monitored request.
- **count**, a number of type integer, is an undeclared slot, that is always present by default in any cell declaration.

This cell definition is used to describe the elements of an infopad. For example, `myMonthlyTable[31][4]` has 31 rows and 4 columns, for a total of 124 elements. Each of these columns are named after a request name, so this table monitors 4 different types of requests. Each row records the monitoring of each day of the month, for each request type.

Creating infopads

The following sections describe how to define infopads, access them across modules, and associate labels and other conditions to them.

Defining infopads

Infopads are defined in the `initialize` section of a module. Although defined in the `initialize` section, they are accessible by all the rules of the module. An infopad is automatically persistent in the Business Process Server database; that is, it recovers from failure like other internal BPM Events objects (rules, events, etcetera.) The state (content) of a persistent infopad is automatically synchronized with its database image, each time the processing of an event is completed by the rule engine.

**Note:** You can also create or edit an infopad through the Infopad Editor in BPM Designer. Creating an infopad in the Infopad Editor automatically generates an initialize section for that infopad. For more information, see *OpenEdge Getting Started: Developing BPM Applications with Developer Studio.*
The following initialize section of a module is creating two persistent infopads:

```plaintext
initialize {
  myPad1 :=
    new infopad < cell{avg_time:int, name:int}>[10][1]("myPad1");
  myPad2 :=
    new persistent infopad<cell{name:string}>[10][4]("myPad2");
}
```

The infopad "myPad1" is created as a vector-like table of size 10 (only 1 column). The infopad "myPad2" is created with 10 rows and 4 columns. The internal name of an infopad—as shown between quotes—should be less than or equal to 21 characters.

The index values for infopad rows or columns always start at 1. For example, when a column dimension is specified as [4], that means the column index values range between 1 and 4.

The arguments of the action-item `defineCell` represent pairs <slot_name, slot_type>. The following are slot types:

- `int` (for integer values)
- `string` (for character strings)
- `float` (for real values)

The general declaration syntax is:

```plaintext
<infopad_creation_stmt> ::=<infoPad_name> "=" "new" <persistent_status>
  "infopad" "<" <cell_definition> "">
  ":[<row_size>] [:<col_size>]" ("myPad1");
  "<cell_definition> ::= "cell" ":" ("<attr_name> ": <attr_value>)" "[
```

The `<infoPad_name>` is the local name of the infopad in the rule module where it is declared, as well as the actual ID of the infopad object, associated with the memory representation as well as the database representation.

**Note:** Infopads have rows and columns. Even if an infopad has a single column, then you declare and manipulate it as a two-dimensional structure.

---

A persistent version of the previous infopad `myMonthlyTable` is declared as:

```plaintext
initialize {
  myMonthlyTable :=
    new persistent infopad<cell{avg_resp_time:float, max_time:int, current_user:string}>[31][4]("myMonthlyTable");
  myMonthlyTable.addRowTitle("Day of Request");
  myMonthlyTable.addColLabels("Request Type", list {"Product Order", "Support", "Service Order", "Training" });
}
```

The optional addition of labels to an infopad (row dimension title, column dimension title, label for each row, label for each column) is not considered as part of the infopad definition. It is supported by separate functions usable at any time after the creation (that is, inside the `Initialize` section, or inside a rule).

**Note:** Labels for rows and columns, as well as dimension titles (row & column), must all be different within the same infopad.

---

If you want to generate an Infopad Report using your infopad in Business Process Portal, then you must ensure that your infopad contains at least two row/columns in one dimension and one row/column in the other.
**Initial values of infopad cells**

After creating an infopad, its cells are automatically allocated and their slots automatically initialized to the value 0 for numeric slot types (int, float), as well as empty string (" ") for string slot types. We recommend, however, you initialize its values (slots of its cells) using an explicit initialization function `setAll()`, or some loop statement, inside the `initialize()` section.

**Accessing infopads across modules**

Rules (from the same application or from another application) may require accessing an infopad defined in a different module. Assume the infopad `myMonthlyTable` is defined in module M1, and that a rule in module M2 requires accessing it. In such a case, write the module M2 as follows:

- It must import the module M1. This is done in the header of M2: `(M2 import M1)`
- Qualify each reference to the infopad in M2 by the imported module name: `M1::myMonthlyTable`. In case M1 and M2 belong to different applications, then qualify each reference in addition to the foreign application name: `App1::M1::myMonthlyTable`

**Note:** In the current Business Process Server version, it is not possible to have "circular" imports, that is, M1 imports M2 and M2 imports M1. This means that if M2 can access infopads defined in M1, then the reverse is not possible.

**Associating labels with infopads**

Infopads can have their dimensions labeled. These labels can identify the cells instead of numerical indices that indicate row and column (for example, [10][4]). The labeling action-item uses the functions `addRowLabels()` or `addColLabels()`. For example, a table with one entry per day of the week may be labeled after the names of the day.

```plaintext
initialize {
    dailyStats := new persistent infopad<cell{avg_resp_time:int,
    avg_req_size:int, max_req_size:int}>[7][24]("dailyStats");
    dailyStats.addRowLabels("DAYS OF WEEK", list ("Monday", "Tuesday",
    "Wednesday", "Thursday", "Friday", "Saturday", "Sunday");)
    dailyStats.addColTitle("HOURS");
}
```

The row labels are specified with the function `addRowLabels()`. The first argument (here, "DAYS OF WEEK") is a dimension title, qualifying all the remaining labels. When no labels other than the title are specified, you can use either `addColTitle(<title>)` or `addRowTitle(<title>)`. Columns and rows are defined similarly, with the list of labels assigned one by one to each column or row of the table, starting with column or row 1 by default.

It is possible to change or set the label of a single row or column, using:

```plaintext
dailyStats.setRowLabel(2, "Day2");
dailyStats.setColLabel(1, "morning");
```

**Attaching alarm conditions to cells**

You can attach an alarm condition to a cell by adding a `check()` statement. The arguments of a `check()` statement include the following generic terms: `conditionname`, `operand1`, `operator`, `operand2`, and `raisemode` where:
• conditionname — Is the name given to this alarm-condition.

• operand1 — Is either the name of a slot involved in the alarm-condition or a constant value. A text constant should be in quotes.

• operator: Is some relational operator. Operators are: GT (greater than), GE (greater than or equal to), LT (less than), LE (less than or equal to), EQ (equal to), NE (not equal to).

• operand2 — Is a slot (of the same cell) or a constant. If it is a slot, then it must follow the syntax: slot[myslotname], where myslotname is the name of the slot. The overall expression must be within quotes, for example, "slot[avgtime]". If it is a text constant, then it should be in quotes, for example, "maxtime". A numeric constant can be written as is, that is, float number 12.5.

• raisemode — Is the mode that specifies when the alarm is raised. If its value is transition, then the alarm will be raised each time the alarm-condition becomes TRUE after being FALSE in the previous check. In the above example, this means that the alarm is raised each time the slot load crosses the "threshold" limit. But no alarm is raised each time the load value changes while staying above the threshold. If the value is absolute, then an alarm is raised each time a change verifies the condition, regardless of what the previous state was of the cell condition.

As an example, assume that we define an infopad to monitor the load of a pool of application servers. Each cell reports about the load of one application server. Each cell has an attribute "load" that should not exceed a particular limit, stored in another attribute "threshold".

When the limit is reached, an alarm is generated. It is possible to associate an alarm-check to the cells of this table using the check() action-item, as illustrated below:

```java
myMonthlyTable.check ("alarm_1","max_time","GT", 120, "absolute");
```

Where "alarm_1" is the condition name, "max_time" is the operand 1, "GT" is the operator, "120" is operand 2, and "absolute" is raise-mode.

Anytime a cell is updated, the alarm-condition set by the check() statement above checks if the slot "max_time" has a value greater than 120. If this is the case, then an event is generated and notified internally to BPM Events.

```java
myMonthlyTable.check ("alarm_2","avg_resp_time","GT", 90,"transition");
```

Anytime a cell is updated, the alarm-condition set by the check() statement above checks if the slot "avg_resp_time" has a value greater than 90. The behavior is however different here, as specified by the keyword "transition". An alarm event is generated only if:

• "avg_resp_time" is greater than 90, and

• its previous value was below or equal to 90.

Such internal alarm events are of the type "BPEVENT_INFOPAD_ALARM". In addition to attributes specific to its type, the event also carries all the slots of the offending cell. Its content is:

Type: BPEVENT_INFOPAD_ALARM
Date: date the event is generated
Value: alarm_1
Context.table: name of the infopad that generated the alarm
Context.row: index of the row of the offending cell
Context.column: index of the column of the offending cell
Context.slot: index/name of the offending slot in the cell
Context.slotvalue: value of the offending slot in the cell
Context.max_time: value of the max_time slot of the offending cell
Context.avg_resp_time: value of the avg_resp_time slot of the offending cell
Context.current_user: value of the current_user slot of the offending cell
Unsetting alarm conditions

Unset an alarm-condition by using `removecheck(condname)`, as shown in the following example:

```
myMonthlyTable.removecheck("alarm_1")
```

Discarding infopads from memory

Discard an infopad from memory using the `discard()` function. The following statement in a `finalize` section deletes the infopad `myMonthlyTable` (along with its database image) when the application is Uninstalled:

```
finalize {
  discard(myMonthlyTable);
}
```

This statement could also be used in the action part of a rule.

Using infopads

This section discusses updating slots in a cell, updating a cell in an event, and iterating over infopad cells.

Updating slots in a cell

Updating slots is also done through action-items. A slot is always referenced with a cell reference as prefix, using the following generic notation:

```
<cell_reference>.<slot_name>
```

`<cell_reference>` is an infopad element that uses an indexed infopad, starting from 1 up to the specified value for the dimension. In the following example, the cell is in row 3, column 2.

For example,

```
myMonthlyTable[3][2].max_time
```

Infopads always have rows and columns, and are therefore considered as two-dimensional, even if their column number or row number is 1. The references to infopad cells should therefore always mention both row and column indexes. A reference to a slot of an infopad cell should always be of the form:

```
infopadname[<rowindex>][<colindex>].slotname
```

**Note:** You can also update slots in an infopad through BPM Designer, using the Infopad Access dialog in the Action Editor. For more information, see OpenEdge Getting Started: Developing BPM Applications with Developer Studio.

Methods to modify infopad contents, whether through operators or action items, are described in the following sections.
### Using operators and functions to perform actions

The operators and functions shown in Table 12: Operators used for infopads on page 105 are commonly used in defining or modifying infopads.

#### Table 12: Operators used for infopads

<table>
<thead>
<tr>
<th>Operators/functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>:=</td>
<td>Assigns any value to a slot in a cell.</td>
</tr>
<tr>
<td>::=</td>
<td>Assigns the result of a functional expression (for example, min, max, avg) to a cell slot, where the first argument of the function is the cell slot itself. This notation enables you to avoid repeating the first argument (in this case, the cell slot itself) which is assumed to be the same as the expression on the left side.</td>
</tr>
<tr>
<td>+=</td>
<td>Adds the value in right hand side to the slot of a cell (numeric type only).</td>
</tr>
<tr>
<td>++</td>
<td>Adds 1 to the slot of a cell (numeric type only).</td>
</tr>
<tr>
<td>-=</td>
<td>Subtracts the value in right hand side to the slot of a cell (numeric type only).</td>
</tr>
<tr>
<td>--</td>
<td>Subtracts 1 from the slot of a cell (numeric type only).</td>
</tr>
<tr>
<td>*=</td>
<td>Multiplies the value in right hand side to the slot of a cell (numeric type only).</td>
</tr>
<tr>
<td>/=</td>
<td>Divides the value in right hand side to the slot of a cell (numeric type only).</td>
</tr>
<tr>
<td>add</td>
<td>Appends a new row for infopad (only for dynamic infopad)s.</td>
</tr>
<tr>
<td>avg</td>
<td>Recalculates an average value, given a new value and its weight.</td>
</tr>
<tr>
<td>check</td>
<td>Associates an alarm check with an infopad.</td>
</tr>
<tr>
<td>max</td>
<td>Calculates the maximum of two numbers.</td>
</tr>
<tr>
<td>min</td>
<td>Calculates the minimum of two numbers.</td>
</tr>
<tr>
<td>remove</td>
<td>Removes the specified entity.</td>
</tr>
<tr>
<td>removeCheck</td>
<td>Removes alarm check from infopad.</td>
</tr>
<tr>
<td>setAll</td>
<td>Initializes all cells of an infopad to a value.</td>
</tr>
<tr>
<td>setColLabel</td>
<td>sets one column label of an infopad.</td>
</tr>
<tr>
<td>setRowLabel</td>
<td>sets one row label of an infopad.</td>
</tr>
<tr>
<td>toFloat</td>
<td>Converts the specified value to float value.</td>
</tr>
<tr>
<td>Operators/functions</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>toInt</td>
<td>Converts the specified value to integer value.</td>
</tr>
<tr>
<td>toString</td>
<td>Converts the specified value to string value.</td>
</tr>
</tbody>
</table>

For more information about the functions, see **Predefined functions and operators** on page 175.

**Updating the value of a slot**

The assignment operator (:=) modifies the value of any slot of a cell. A cell is usually identified with an indexed infopad expression, as in the code example below:

```plaintext
dailyStats[4][3].avg_resp_time := 0;
```

Where the cell corresponding to row = 4 and column = 3 of the `dailyStats` infopad is identified.

You may also use labels instead of index values. The following expression denotes the same cell as the previous one:

```plaintext
dailyStats["Thursday"][3].avg_resp_time := 0;
```

An event attribute is always taken as a string value. If this string represents an index value, and is used as an actual index (and not a label), then you must explicitly convert it to an integer using the `toInt()` function.

Assuming the event attribute: `EVT1.myhours` is equal to the string `[3]`, then the following expression is correct and equivalent to the previous ones:

```plaintext
dailyStats["Thursday"][toInt(EVT1.myhours)].avg_resp_time := 0;
```

Assuming that the date attribute of `EVT1` is: `YEAR:2000/MONTH:4/DAY:15/HOUR:3/MIN:30`, then the following expression is equivalent to the previous ones:

```plaintext
dailyStats["Thursday"][EVT1.date.hour].avg_resp_time := 0;
```

Indeed, the date projection functions return a numeric type, which does not require conversion.

Note that in case `dailyStats` is defined in another module `M1`, then the infopad name is qualified by this module name:

```plaintext
M1::dailyStats[4][3].avg_resp_time := 0;
```

**Note:** If an initial value is not specified, then the default initial value of any numeric (int or float) slot of any cell is zero (0).

**Adding to or subtracting from a slot**

Increment operators `+=` and `-=`, as well as `++` and `--` can be used to add or subtract numeric values to a slot:

The three statements below are equivalent:

```plaintext
myMonthlyTable[NOW.month][EVT1.region].count :=
myMonthlyTable[NOW.month][EVT1.region].count + 1;
myMonthlyTable[NOW.month][EVT1.region].count += 1;
myMonthlyTable[NOW.month][EVT1.region].count++;
```
The three statements below are equivalent:

\[
\begin{align*}
\text{myMonthlyTable}[\text{NOW.month}][\text{EVT1.region}].\text{count} &:= \text{myMonthlyTable}[\text{NOW.month}][\text{EVT1.region}].\text{count} - 1; \\
\text{myMonthlyTable}[\text{NOW.month}][\text{EVT1.region}].\text{count} &:= \text{myMonthlyTable}[\text{NOW.month}][\text{EVT1.region}].\text{count} - 1; \\
\text{myMonthlyTable}[\text{NOW.month}][\text{EVT1.region}].\text{count} &:= \text{myMonthlyTable}[\text{NOW.month}][\text{EVT1.region}].\text{count} - 1;
\end{align*}
\]

**Using max() and min() functions**

The functions \(\text{max}(V)\) and \(\text{min}(V)\) respectively replace the current value \(S\) of a numeric slot with \(\text{max}(S,V)\) and with \(\text{min}(S,V)\). Assuming a cell definition that collects the minimum response time for requests (slot: \(\text{min\_time}\)) as well as the maximum time (slot: \(\text{max\_time}\)), the action-items that achieve this monitoring are:

\[
\begin{align*}
\text{val } x & = \text{myMonthlyTable}[\text{EVT1.date.month}][\text{EVT1.region}]; \\
\text{x.max\_time} & := \text{max(EVT1.time)}; \\
\text{val } x & = \text{myMonthlyTable}[\text{EVT1.date.month}][\text{EVT1.region}]; \\
\text{x.min\_time} & := \text{min(EVT1.time)};
\end{align*}
\]

The operator \(\text{:=}\) allows you to not repeat the slot expression (to the left of \(-\)) in the first argument of the function \(\text{max}\) or \(\text{min}\), and to use a contracted version of these functions: \(\text{max}(V)\) instead of \(\text{max}(S,V)\).

The following statements, therefore, are equivalent:

\[
\begin{align*}
\text{x.max\_time} & := \text{max(x.max\_time, EVT1.time)}; \\
\text{x.max\_time} & := \text{max(EVT1.time)};
\end{align*}
\]

This assignment operator can generally be used on other functions that report the left hand expression (that is, to the left of \(-\)) as their first argument.

**Using the avg() function**

The function \(\text{avg}(V, \text{slot\_name}, \text{weight})\) replaces the current value of a numeric slot with a new averaging value involving \(V\).

\[
\begin{align*}
\text{val } x & = \text{myMonthTable}[\text{EVT1.date.month}][\text{EVT1.region}]; \\
\text{x.avg\_resp\_time} & := \text{avg(EVT1.time, x.count++, 1)};
\end{align*}
\]

The function call: \(\text{avg}(V, \text{countslot}, \text{weight})\) arguments are as follows:

- \(V\). A value element that is involved in the new average value.
- \(\text{Countslot}\) (optional). The slot of the same cell that is used as divisor for calculating the new mean value, usually as a counter of the number of values. The default counter "count" is usually predominate. If an increment operator ++ is used, then the weight is added to the countslot. If not, the value of countslot does not change during the operation.
- \(\text{Weight}\). A positive number assigned to a new element that determines its relative importance for calculating the mean of the values. When an element is not weighted, it is equal to 1. Depending on the mode, the weight is or is not added to the value in countslot.

Assume that the previous value of \(\text{myslot}\) was already an average \(U\) calculated from \(N\) values (mean of \(N\) values). Involving the new value \(V\) with weight 1 requires it to recalculate the new average: its new value assigned to the slot \(\text{myslot}\) is \((N\times U + V)/(N+1)\). The value \(N\) was supposed to be in the slot \(\text{count}\). It updates after the action-item, to \(N+1\), if ++ is on the counter.
Using several averaging statements on the same cell

Consider, for example, a rule that averages two values: request response time and request size. Although two averaging statements are used, incrementing the counter "count" is done once. This rule uses \texttt{avg()} in the following way:

\begin{verbatim}
val x = myMonthTable[EVT1.date.month][EVT1.region];
x.avg_resp_time ::= avg(EVT1.time, x.count, 1);
x.avg_req_size ::= avg(EVT1.time, x.count++, 1);
\end{verbatim}

The two average values \texttt{avg_resp_time} and \texttt{avg_req_size} use the same counter \texttt{(count)}. However, the counter is incremented only once each time the action is executed. The operator \texttt{++} specifies which averaging statement does the update: it is the last one to use the counter in the rule. All other averaging statements with the same counter do not mention the ++ operator.

Assume the value in the counter slot \texttt{count} is \texttt{N} before the action is executed, \texttt{V} is the new value \texttt{EVT1.resp_time} to include in the average, and \texttt{currreqtime} is the value of the slot \texttt{avg_resp_time}. In the example above, the first averaging sets the slot \texttt{avg_resp_time} to: \small{(N* currreqtime + V)/(N+1)}\normalsize. The second averaging uses a similar formula. The weight value (here "1") is used in both averages to calculate the new counter value: \texttt{N' = N+weight}.

Updating a cell with an event

It is possible to automatically update an infopad cell by sending special "cell update" events to BPM Events. No written rules are required to capture these events. They are automatically recognized and interpreted by the engine. These events must be of the form:

\begin{verbatim}
Type:"BPEVENT_INFOPAD_UPDATECELL"
Value: myInfopadName
\end{verbatim}

In the context of the event, the name value pairs identify the cell coordinates:

\begin{verbatim}
Row_Index: index of the row of the cell
Col_Index: index of the column of the cell
\end{verbatim}

Any other name/value pair in the context part of the event is supposed to represent a slot name and its new value. You can add only the slots that you want to update. The event only updates the slots present in the context.

Iterating over infopad cells

The following infopad is used for demonstrating iterations. It has cells containing two attributes: one of the integer type and one of the string type. mypad is a fixed-size infopad.

\begin{verbatim}
initialize {
  mypad := new infopad<cell{attnum:int,attstr:string}>[10][5]("mypadname");
}
\end{verbatim}

\textbf{Note}: You must remember that infopad index values start from 1. For example, when an infopad is declared of dimensions [10][5], this means its row index takes values from 1 to 10, and its column index values go from 1 to 5.

Initializing to iterate on infopad cells

This section describes how a rule—or an initial statement in the \texttt{initialize} section of a rule module—can iterate on the cells of an infopad.
If all the initial values are known for all the slots of a cell, then we can use the following single infopad method call: `setAll()`. All the slots of a cell are specified here (except the default count slot, which is always initialized to 0, unless specified otherwise).

```javascript
mypad.setAll(cell{attnum:0, attstr:"nothing"});
```

Initialization is also done with a loop statement. This allows more control on which slots are initialized, and in which cells. In the statement below for the fixed-sized infopad, the identifiers `rt` and `ct` represent a constant (or value) that is not changeable (only initialized), as it is declared with the keyword `val`. These constant values are initialized to the current row count and column count of the infopad, using the functions `rowCount()` and `colCount()`. The two nested for loops explicitly iterates over the row and column indexes.

```javascript
val rt= mypad.rowCount();
val ct= mypad.colCount();
for (var i = 1; i <= rt; i++ )
  for (var j= 1; j<= ct; j++ )
    { mypad[i][j].attnum := 0; mypad[i][j].attstr := "nothing"; }
```

**General iterations with tests**

You may introduce additional selection conditions in loop statements, as in regular programming languages. A similar loop for the cells of the two-dimension `mypad` infopad are as follows. Note that nested loops are not necessary:

```javascript
do mypad[xi][yi].attstr := evt1.mynname
from xi:mypad.rowKeySet(), yi:mypad.colKeySet()
where (mypad[xi][yi].attnum > 100);
```

These iteration statements require the creation in memory of value sets resulting from the function calls `rowKeySet()` and `colKeySet()`. This may be inconvenient in terms of overhead (memory) for large infopads.

**Using scaled infopads**

Assume that each row of an infopad named `salarystats` is associated with a numeric range representing salary range (in $1000): starting from [0-30], [30-40], [40-50], etcetera until [120-plus]. Each time a new employee is hired, an event is generated that contains his/her salary. We want to record how many employees are in each salary range. We can find the row (and cell) that corresponds to a particular salary, that is, 43, by iterating over the infopad:

```javascript
var uppb = 30;
for (var xi = 1; xi <= 10; xi++)
  { if ( evt1.salary < uppb )
    { salarystats[xi][1].count++; break; }
    uppb += 10;
  }if ( evt1.salary >=120 ) salarystats[11][1].count++;
```

**Persistent maps**

Typically, infopads are used for application monitoring. Infopads have a fixed set of rows and columns. Another variant of infopad -- the dynamic infopad -- has fixed number of columns but rows can be added or removed as required. Both these structures internally use an array. Adding and removing elements in an array structure is not efficient. Additionally, to access a particular value in an infopad you must traverse all the elements in it. These issues can be overcome by using another type of persistent structure, "Persistent Map". A persistent map provides you a dynamic structure, and quick and easy access to values in it.
The persistent framework generates two module files, one for rules and another one for the persistent data. The rule module contains rules whereas the data module contains persistent structures. This allows you to refresh the rule modules without affecting the persistent data. To distinguish between rule and data modules, "rule" and "data" keywords are used.

Defining persistent maps, keys, and values

A persistent map can have a key-value pair. The key is a string and the value is an object. The object can have following different attributes:

- String of characters (string)
- Integer (int)
- Real number (float).

The following code example represents how an attribute and its type is defined:

```plaintext
avg_resp_time:float,
max_time:int,
current_user:string
```

The previously mentioned attributes are explained below:

- `avg_resp_time`, a number of type real, records the average response time of all requests monitored to date (the time unit depends on the application and is matter of interpretation: it could be seconds, minutes, hours, etcetera).
- `max_time`, a number of type integer, records the maximum response time recorded to date.
- `current_user`, a string of characters, records the user ID of the last monitored request.

Creating persistent map instances

The following sections describe how to create persistent map instances, and access them across modules. The creation of persistent maps and declaration of objects is done in the data module.

Defining persistent map

A persistent map is an instance of a persistent map class. So, before instantiating the instance, we must define the persistent class as follows:

```plaintext
pmap_definition() ::= 'persistent' 'class' <ID> 'of' 'map' '<' key_type(),
',' value_type() '>

top_declaration ::= ... |pmap_declaration();
```

The persistent map declaration needs to be done at the top level with global scope. The syntax for instantiating a persistent map from a persistent map class is as follows:

```plaintext
pmap_expr ::= 'new' <ID> [ '(' <string> ')' ] [ '{' item_list() '}' ]
```

For example, defining type of values in the persistent map is done as shown below:

```plaintext
type UserObject = object{avg_resp_time: float, max_time: int, current_user: string};
```
Defining persistent map:

```java
persistent class UserCls of map<string, UserObject>;
```

Creating instance of persistent map,

```java
val ucls0 = new UserCls ("ucls0"){
  "user_key01" : UserObject {avg_resp_time: 4.4, max_time: 8, current_user: "John"},
  "user_key02" : UserObject {avg_resp_time: 5.6, max_time: 4, current_user: "Smith"}
};
```

Accessing persistent maps and map instances

Rules (from the same application or from another application) may require accessing persistent maps / map instances defined in a different module. Assume the infopad UserCls is defined in module M1, and that a rule in module M2 requires accessing it. Both, M1 and M2 are defined for the application A1. In such a case, write the module M2 as follows:

- It must import the module M1. This is done in the header of M2: (M2 import M1)
- Qualify each reference to the persistent map in M2 by the imported module name M1::UserCls. In case M1 and M2 belong to different applications, then qualify each reference in addition to the foreign application name: A1::M1::UserCls.

**Note:** In the current Business Process Server version, it is not possible to have "circular" imports, that is, M1 imports M2 and M2 imports M1. This means that if M2 can access persistent map defined in M1, then the reverse is not possible.

Associating keys to persistent maps

In the following example, user_key01 and user_key02 are the two keys which are associated to the specific object of UserObject.

```java
val ucls0 = new UserCls ("ucls0"){
  "user_key01" : UserObject {avg_resp_time: 4.4, max_time: 8, current_user: "John"},
  "user_key02" : UserObject {avg_resp_time: 5.6, max_time: 4, current_user: "Smith"}
};
```

Discarding persistent maps and map instances from memory

Discard a persistent map from memory using the `discard()` function. The following statement in a `finalize` section of data module deletes the persistent map / persistent map instance UserCls (along with its database image) when the application is uninstalled.

```java
finalize {
  discard(ucls0);
  discard(UserCls);
}
```
Using persistent maps

This section discusses updating attributes of a particular key of a persistent map instance.

Updating attributes of a given key

Updating key is also done through action-items. You can get the value of a particular key using the following statement:

\[ A1::M1::ucls0.get("user_key01"); \]

To get the attributes of this key, use the following syntax:

\[ var \ avg = A1::M1::ucls0.get("user_key01").avg_resp_time; \]

Methods to modify persistent map instance contents, whether through operators or action items, are described in the following sections.

Using various operators to perform actions

The operators shown in Table 13: Operators used for persistent maps on page 112 are commonly used in defining or modifying persistent maps.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>:=</td>
<td>Assigns any value to an attribute of a key.</td>
</tr>
<tr>
<td>::=</td>
<td>Assigns the result of a functional expression (for example, min, max, avg) to an attribute of a key, where the first argument of the function is the attribute itself. This notation enables you to avoid repeating the first argument (in this case, the attribute itself) which is assumed to be the same as the expression on the left side.</td>
</tr>
<tr>
<td>+=</td>
<td>Adds the value in right hand side to the attribute of a key (numeric type only).</td>
</tr>
<tr>
<td>++</td>
<td>Adds 1 to the attribute of a key (numeric type only).</td>
</tr>
<tr>
<td>-=</td>
<td>Subtracts the value in right hand side to the attribute of a key (numeric type only).</td>
</tr>
<tr>
<td>--</td>
<td>Subtracts 1 from the attribute of a key (numeric type only).</td>
</tr>
</tbody>
</table>

Updating the attributes of a given key

The assignment operator (:=) modifies the value of any attribute of a key.

\[ A1::M1::ucls0.get("user_key01").avg_resp_time := 15.4; \]
Adding to or subtracting from the given key attribute

Increment operators += and -=, as well as ++ and -- can be used to add or subtract numeric values of the attributes of the given key:

The three statements below are equivalent:

\[
\begin{align*}
A1::M1::ucls0.get("user_key01").max_time & := \\
A1::M1::ucls0.get("user_key01").max_time & + 1; \\
A1::M1::ucls0.get("user_key01").max_time & += 1; \\
A1::M1::ucls0.get("user_key01").max_time & ++;
\end{align*}
\]

The three statements below are equivalent:

\[
\begin{align*}
A1::M1::ucls0.get("user_key01").max_time & := A1::M1::ucls0.get("user_key01").max_time - \\
1; \\
A1::M1::ucls0.get("user_key01").max_time & -= 1; \\
A1::M1::ucls0.get("user_key01").max_time & --;
\end{align*}
\]

Iterating over persistent map keys

The entries in a persistent map can be enumerated using the for each statement syntax.

\[
\begin{align*}
\text{foreach} & \text{ statement()} \ ::= \text{'}\text{foreach} ( \text{iteratorDef()} (',', \text{iteratorDef()})*) \\
& \quad [\text{'}\text{where} \ \text{expr()}\text{'}] \quad \text{'}\text{'}\text{statement()}\text{'}\text{'}
\end{align*}
\]

\[
\begin{align*}
\text{iteratorDef()} & \ ::= \ <\text{ID}> \quad [\quad (\text{'of'} \ | \ 'of') \ \text{expr()} \quad ]
\end{align*}
\]

For example, the following persistent map is used for demonstrating iterations. The attributes of the keys are avg_resp_time, max_time, current_user which are of type float, integer and string respectively. UserCls is a persistent map.

```scala
fun printpmap(m: map<string, UserCls>){
  foreach e:m {
    val key = e.key;
    val value = e.value;
    pr("key: " + key + " :: value " + value);
  }
}
}
```

Initializing attributes of a persistent map instance

You can initialize the values of an attribute as shown in the following example.

```scala
val ucls0 = new UserCls ("ucls0"){
  "user_key01" : UserObject {avg_resp_time: 4.4, max_time: 8, current_user: "John"},
  "user_key02" : UserObject {avg_resp_time: 5.6, max_time: 4, current_user: "Smith"}
};
```

Persistent map operations

This section explains various functions provided on persistent map.

To access a value associated with a particular key in the persistent map, get function is supported. The returned object is a part of the map. It does not create another object value. If this value is updated, then the map is also updated. For example,

```scala
ucls0.get("user_key01");
```
To update or put a new value into the persistent map, the put function is supported. A runtime exception is raised when trying to assign a nil value to a persistent map. For example,

```javascript
ucls0.put("user_key01", object{avg_resp_time: 4.4, max_time: 8, current_user: "John"});
```

To remove a key, value pair from the persistent map, the remove function is provided. A runtime exception is raised when accessing a key that has been removed. For example,

```javascript
ucls0.remove("user_key01");
```

To check if a particular key is present in the persistent map or not, the containsKey function is provided. If the key is present, then it returns a Boolean value of true. If the key is not present, then it returns a Boolean value of false. For example,

```javascript
ucls0.containsKey("user_key01");
```

To get the number of key, value pairs in a persistent map, the size function is provided. It returns an integer value. For example,

```javascript
ucls0.size();
```

To check if a persistent map is empty, the isEmpty function is provided. The isEmpty function returns a Boolean value of true if there is no key, value pairs in the persistent map. If there is at least one key, value pair, then it return false. For example,

```javascript
ucls0.isEmpty();
```
Event channels

BPM Events can receive messages from different channels. The following two event channels are provided in BPM Events:

• The Database Event Channel.
• The JMS Event Channel.

For details, see the following topics:

• The database event channel (DEC)
• The JMS event channel

The database event channel (DEC)

The Database event channel retrieves events from BPMEvent table in the repository database that implements a queuing mechanism for events (events generated by Business Process Server processes or by third-party systems). BPM Events uses an adapter (the DEC adapter) to pull events from the database and to push them into the BPM Events entry queue. Events can be pulled in batches. BP Server, the process engine of the Business Process Server platform, uses this database event channel to pass events to the BPM Events rule engine.
**postDB predefined rule action**

You can push an event into the database event queue from a rule action using the postDB predefined rule action:

```
postDB(event MyEventType::MyEventValue{Attr1: 123, Attr2: "aString"});
```

You can also post several events to the database queue at the same time:

```
postDB(evt1, evt2, evt3);
```

**Configuring the database event channel**

The Database Event Channel (DEC) does not normally require any attention from developers: its default configuration is suited to most applications. A different configuration may be required for application-specific tuning. The main objective of configuring DEC is to control the flow of events from the database to the BPM Events engine, so that it provides optimal performance. Please note that except for the basic database access configuration parameters (already set in the Business Process Server configuration file), the following is only a tuning issue. In its default configuration, the DEC works fine for most applications.

The Database Event Channel involves two event queues:

- **Database event queue.** Producers: event-generating components, like BP Server. Consumers: BPM Events DEC adapter (and other subscriber components, like BPM Process Store).
- **BPM Events entry queue.** Producer: the DEC adapter of BPM Events. Consumer: BPM Events rule engine.

The configuration parameters (in `oebpsdb.properties`) used in configuring DEC are:

- `oebps.db.<dbname>.url` specifies the URL of the data source from which to receive the events.
- `oebps.db.<dbname>.user` specifies the name of the user of the database.
- `oebps.db.<dbname>.password` specifies the password for the user of the database.

The configuration parameters (in `bpmevents.conf`) used in configuring DEC are:

- `bpmevents.server.eventsource.eventbatchsize` specifies the maximum number of events that are pulled from the database at each fetch (query) operation.
- `bpmevents.server.eventsource.sleeptime` specifies the sleeptime when there are no events in the BPMEvents table.
- `bpmevents.server.eventsource.sleeptime.whenevents` specifies the sleeptime when there are more events in the BPMEvents table.

**The JMS event channel**

The JMS event channel uses JMS infrastructure to get messages. BPM Events receives messages posted to the JMS queue, `jms/BPMEventQueue`. BPM Events uses an adapter (the JMSEC adapter) to listen to JMS messages and to push them into the BPM Events entry queue.
**JMS message adapter**

BPM Events provides a JMS message adapter to post JMS messages to a JMS queue. A sample JMS Adapter `post_po.java` is as follows:

```java
package POExampleDOM.jclass;
import com.savvion.ejb.bizpulse.manager.*;
import com.savvion.sbm.bizpulse.admin.intf.BPMEventsServerAPI;
import java.rmi.RemoteException;
import java.util.Properties;
import java.util.*;
import javax.jms.*;
public class post_po {
/**
 * Runs this example from the command line. Example:
 * <p>
 * <tt>java post_po queueName</tt>
 * <p>
 * @param url  JMS queue name like "queue1"
 */
    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            System.out.println("Usage : java post_po QueueName ");
            return ;
        }
        String queue = args[0] ;
        BPMEventsServerAPI bpc = BPMEventsClient.create() ;
        Hashtable properties = new Hashtable();
        properties.put("myJMSProp1", new Integer(1));
        String xml = new String("<?xml version="1.0"?><purchaseOrder xmlns="http://example.com/po"><shipTotype="USAddress"><name>Fred Zoe</name><street>47 Eden Street</street><city>Santa Clara</city><state>NY</state><zip>95054</zip></shipTo><billTo type="USAddress"><name>Robert Zoe</name><street>47 Eden Street</street><city>Santa Clara</city><state>NY</state><zip>95054</zip></billTo><itemsList><item partNum="833-AA"><productName>Lapis necklace</productName><quantity>1</quantity><USPrice>100.00</USPrice><comment>Want this for the holidays!</comment><shipDate>2001-12-15</shipDate></item><item partNum="423-AA"><productName>Baby Sound Monitor</productName><quantity>2</quantity><USPrice>49.00</USPrice><comment>Got twins!</comment><shipDate>2001-12-17</shipDate></item></itemsList></purchaseOrder>" retained true);
        bpc.postJMSMessage(properties, xml, "queue", queue) ;
    }
}
```

This program expects a valid JMS queue name as an argument. It instantiates a new BPM EventsClient and calls `postJMSMessage()` with arguments as a hashtable consisting of properties, a string XML document, type of destination that is, queue and the string JMS queue name. This posts the XML message into the given JMS queue.

**Running the sample program**

You may post as many messages as you wish. You can again see the results in the file once you unload the application.

To compile and run the sample program.
1. Start the Command Prompt window and compile the program by using the following command.

   `%OEBPS_HOME%/bin\setBPMEventsEnv.cmd`

2. Go to the `ebmsapps` folder, and give the following command.

   `%OEBPS_HOME%/ebmsapps\javac -classpath %oebpscp% POExampleDOM\jclass\post_po.java`

3. Run the program by using the following command.

   `%OEBPS_HOME%/ebmsapps\java -classpath %oebpscp% POExampleDOM.jclass.post_po jms/BPEventQueue`

**postJMS predefined rule action**

You can also post a JMS message into a JMS message queue from a rule action using the postJMS predefined rule action.

```java
fun create_jms_event(a: string, b: string, c: string): XML {
  val xml = getXML(a, b, c);
  val type_id = "%XML<"purchaseOrder","http://example.com/po">";
  return createJMSEvent(xml, type_id, map{"a":"z", "b":"1111"});
}postJMS("queue", "BPEventQueue", create_jms_event("1223.25", "150.00", "NY"));

You can also post several messages to the same queue at the same time: postJMS("queue", "jms/BPEventQueue", msg1, msg2, msg3);
```
As we have seen in Understanding BPM Events on page 49, BPM Events can receive, process and generate XML messages. These messages can be internal or external—for example, coming from the JMS event channel.

BPM Events uses XML oriented architecture for handling external events because of the progress of XML standardization and the increased adoption of XML-based systems.

Previously, XML event support of BPM Events was provided using XML-Java binding based on the Castor technology. In that approach, once an XML document was mapped to a Java object, it was handled like an ordinary Java object. It did not use an XML-based technology to handle XML objects. To overcome the limitations of XML-Java binding approach, BPM Events adopts XPath-oriented architecture for processing XML messages. Though the XML binding using Castor is supported for backward compatibility, it is deprecated in the current release.

BPM Events now provides a new configuration parameter to set the adapter for JMS message processing. The parameter, `bpmevents.server.jms.adapter` in the `bpmevents.conf` file, decides the adapter used by the BPM Events server to handle XML messages received via a JMS channel. In the current release, the default value of this parameter is 'dom' which uses the new XPath-based architecture, while the earlier technology based on Castor can be specified by setting its value to 'castor'. Therefore, if you change the value of this parameter, then you must restart the BPM Events server for the change to take effect. This also implies that rule applications based on XPath does not work with the Castor adapter implementation and vice versa. Application developers should ensure that all the rule applications are developed and published for a particular implementation.

BPM Events now accesses the XML message’s internal data by adapting XPath for rule condition and expression. This XPath-based approach makes XML binding unnecessary. XPath internally replaces the property accessor like `a.b` and there is no need to map XML data to Java classes generated by Castor/XMLBeans. The XPath-based architecture improves the runtime performance as compared to the XML binding approach. It can also support XML data that is not associated with XML schema.
The XPath-based architecture eliminates the need to generate Java classes from .xsd files, setting the classpath, and putting generated classes into proper location in deployment environments. This makes the entire process simple, and increases the ease of use. However, it lacks type checking (message validation) and existence of certain attributes for XPath while compiling.

While using XPath in BPM Events, it is important to integrate XML namespace into BPM Events because the repetitive supply of namespace for XPath is very inconvenient and error prone. Business Process Server introduces namespace declaration and special syntax to incorporate namespace declaration in the environment implicitly.

For details, see the following topics:

- Namespace declaration
- XML type
- XML data field access syntax
- XML data conversion
- New syntax for iteration

### Namespace declaration

There are two types of namespace declarations. One is for declaring the prefix and URI pair whereas the other is for defining the default namespace.

For example,

```java
namespace po="http://example.com/po";
namespace default="http://example.com/po";
```

The prefix is used as a name for the URI in BPM Events. Therefore, even if the prefix is different, if the associated URI value is the same, then they are semantically equivalent. The namespace is used for creating a namespace context in which the XPath is evaluated. When an XPath expression is evaluated, these namespace contexts are implicitly passed to the XPath engine. The scope of a namespace follows the normal block semantics. For instance, in the following code, inner `pol` in separate block represents different namespace, and the context of namespace in each XPath expression consists of two namespaces.

```java
{  
    namespace po="http://example.com/po";
    {  
        namespace pol="http://example.com/pola";
        xml ~~ "pol:purcharseOrder[@a > 10]";
    }  
    {  
        namespace pol="http://example.com/polb";
        xml ~~ "pol:purcharseOrder[@a > 10]";
    }
}
```

**Note:** When dot or array [] notation is used while declaring variables in DOM rules, you must define the default namespace to verify while rule compilation to avoid run-time error on that variable.
XML type

XML type is defined using a notation similar to the one used for XPath. It consists of a prefix and an element name. Its format is as follows:

Prefix "::" elementName

Whenever an XML event is processed by a rule, the correct XML type must be specified.

Rule A triggered by
  Event1 of po:PurchaseOrder
  Then {
  }

This syntax allows narrowing down the scope of XML data, and provides efficient caching framework for both schemed and non-schemed XML data. This notation provides similar expression for conditioned case to XPath notation.

As explained in the namespace description, XML type represents the same type as long as prefixes are bound to the same URI value. For example, type ‘po1:purchaseOrder’ and ‘po2:purchaseOrder’ are of the same type when po1 and po2 are bound to the same URI.

For example,

namespace po1="http://example.com/po";
namespace po2="http://example.com/po";

XML data field access syntax

XPath allows the use of concise query notation for relatively complex conditions. But it is too verbose for simple property access. In order to achieve simple access to the simple field without losing generality, BPM Events supports two types of notations. One is a dot notation, the other is using bracket notation. The field of XML value can be accessed as if it has field property.

Dot notation for XML data fields

Let x be XML data, and a be an element name of the top XML structure.

Instead of writing x.find("a"), you may use x.a as if it is an attribute of the xml object.

Even if there is more than one element with a, it returns only the first element with a as the name.

Dot notation is mainly for non-namespace associated XML data. But it can also be used while declaring the default namespace.

x.a.b is also allowed if x.a is a node element. When x.a is a leaf element or attribute value, it is converted to appropriate primitive values inferred from the content. If there is no such constraint, then it is recommended to use explicit conversion function like, toInt, toFloat. Another way to create constraint is to use the type specification for val declaration like:

Val p: int = item.quantity;

Note that the type of item.quantity is XML type (this is sort of a super class of all XML types like po:purchaseOrder).
So the above conversion does not change this type itself. It only implicitly inserts conversion function which converts an XML value to a primitive value (like \texttt{int}).

For example,

\begin{verbatim}
Val p = item.quantity;
Val s = "abc"+p; // BP compiler implicitly replace p by toString(p) .
Val I = 2 + p; // BP compiler implicitly replace p by toInt(p) .
\end{verbatim}

**Bracket notation for XML data fields**

When a namespace is used for XML data, ordinary dot notation cannot be used unless the default namespace is defined. In this case, you must either declare default namespace, or use bracket notation ‘[ ‘’].

For example, you may use \texttt{item["po:quantity"]} instead of \texttt{item.quantity}. This notation implicitly depends on the namespace context. BPM Events compiler detects the context and passes it to the underlying XPath engine.

Note that this notation uses different semantics depending on the type of the accessed value. Namely, if the item is infopad, then this means index access.

**Function notation for XML data fields**

Instead of using a special notation that is convenient for most of the cases, you may use a function notation for accessing function. \texttt{Find} is a system function of the following signature:

\texttt{Find: XML*string*Map<string, string> -> XML}

This function can be used in the same way we send messages, like \texttt{item.find("po:quantity", ns)}, where \texttt{ns} is defined as:

\begin{verbatim}
val ns = map{
  "po" : "http://example.com/po",
  "aa" : "http://example.com/po",
  "" : http://example.com/po /* default namespace */
};
\end{verbatim}

The advantage of this approach is that it takes the namespace information explicitly. If implicit namespace context is not appropriate, then you may use this approach.

**XML data conversion**

XML data conversion involves XPath Rule condition and XPath query expression/statement. These are explained in the following sections.

**XPath rule condition**

In order to support expressive XPath notation in the rule condition and efficiently extract a precondition that may be used in the lower layer of BPM Events, a new rule condition syntax is introduced. Unless there is correlation, it is sufficient to use only an XPath condition in a rule.
For example,

```java
Rule activated by
    msg of po:purchaseOrder["po:shipTo/po:state = 'NY'"]
then {
    ...
}
```

The condition is specified in a bracket `['',']` as a string. This string is any valid XPath notation of boolean value condition. The XPath's context is under the top element of the XML element type (in the above case, under `po:purchaseOrder`).

This syntax is similar to a valid XPath expression, `po:purchaseOrder[po:shipTo/po:state = 'NY']` except the string quotes.

**XPath query expression/statement**

The current release of BPM Events contains three new expressions for an XPath query. There are corresponding functions for the new syntax, and they take additional parameter(s) for namespace context. They are essentially syntax add-ons to eliminate the need to provide namespace in XPath query where fixed namespace can be assumed.

**SelectPath**

This query returns all the (inner) nodes that satisfy the XPath condition.

```java
*~:
XML*string->list<XML>::
```

this is to get nodes which satisfy the XPath.

**Example1,**

```java
po *~ "po:purchaseOrder/po:itemsList/po:item[po.quantity > 10]";
```

**Example2,**

```java
po.selectPath("po:purchaseOrder/po:itemsList/po:item[po.quantity > 10]", ns);
```

This returns a list of items where the quantity is larger than 10.

**Find**

This query returns a node that satisfies the XPath condition. This notation is already mentioned as field accessor, but it can be used for general XPath.

```java
[]: XML*string->XML: (find)
```

this returns the first element which satisfy the XPath.

**Example1,**

```java
po["po:purchaseOrder/po:itemsList/po:item[po.quantity > 10]"];
```
Example2,

```
po.find("po:purchaseOrder/po:itemsList/po:item[po.quantity > 10]", ns);
```

This returns an item where the quantity is larger than 10.

**Match**

This query returns **TRUE** if there is a node which satisfies the XPath condition; otherwise, it returns **FALSE**.

```
~ ~:
XML*string->boolean: {find}
```

this returns the first element which satisfy the XPath.

Example1,

```
po ~ ~ "po:purchaseOrder/po:itemsList/po:item[po.quantity > 10]";
```

Example2,

```
po.match("po:purchaseOrder/po:itemsList/po:item[po.quantity > 10]", ns);
```

This returns **TRUE** if there is an item where the quantity is larger than 10, otherwise returns **FALSE**.

### New syntax for iteration

Although there are slight syntax differences, it is possible to use the following `do from` statement to enumerate elements.

```scala
namespace po = "http://example.com/po";
val xml = ...

do {
    f(item1.a, item2.b);
}from
    item1 of xml *~ "po:a/po:b/po:c[@d > 100]",
    item2 of xml *~ "po:a/po:b[@x = 'aa']/po:e"
where
    item1.a > item2.b;
```

Although this will work, this syntax is difficult to understand because items are declared in the last part of the statement. In such cases, another syntax—`foreach`—which has a syntax similar to that of `for` can be used.

### Foreach syntax for iteration

The format of the `foreach` syntax is as follows:

```xml
<foreach> ::= 'foreach' <item_decl> (',' <item_decl> )* 
    [ 'where' <expr> ] 
    '{' ( <statement> )* '}'

=item_decl> ::= <ID> [ ':' | 'of' ] <expr>
```
Simple case
An example of a simple case of foreach usage is given below.

```xml
namespace po = "http://example.com/po";
foreach item of po *~ "po:purchaseOrder/po:itemsList/po:item[@a > 10]" { 
    f(item1.a);
}
```

General case
An example of a general case of foreach usage is given below.

```xml
namespace po = "http://example.com/po";
val po = ...;
val xml = ...;
foreach item1 of po *~ "po:purchaseOrder/po:itemsList/po:item[@a > 10]",
    item2 of xml *~ "po:a/po:b[@x = 'aa']/po:e[@a > 10]"
where item1.a > item2.b
{ 
    f(item1.a, item2.b);
}
```
BPM Events can now access Java objects and Enterprise JavaBean (EJB) components. And you can use Java classes and objects to define BPM Events rules, rule modules and events. You can define the mapping between Java types and BPM Event types and directly use external Java classes—local or remote—for:

- **Rule language extension.** BPM Events uses a syntax similar to Java to invoke Java methods in their condition and action statements. This makes it possible to extend the rule language with ad-hoc functions and predicates that serve application-specific needs. Libraries of such language extensions are defined, either for general or application-specific purposes. Static type checking (compile time) is supported for methods and their parameters, as well as for object references assigned to variables.

- **Object instantiation.** BPM Events uses a Java adapter to create Java objects, which have a life-span beyond the rule execution, and references to these objects can be stored in variables (rule scope or module scope). Object instantiation is performed in such bloc statements as functions, rule actions, and initialize and finalize sections.

- **Public method invocation and public property access.** Java class access is possible in the rule condition, the rule action, and in `initialize()` and `finalize()` sections of a rule module.

- **EJB component invocation.** The integration of BPM Events rules and EJB components enable you to use complex Java objects in rules, communicate with an EJB container, invoke both entity and session beans, perform component reference caching, and get results and exceptions.

For details, see the following topics:

- Rule language extensions
- Searching class files
- Mixing two separate instance of a class
- EJB component invocation
Dynamic loading

Rule language extensions

Extending the BPM Events rule language into the Java environment enables you to perform the following:

- **Mapping data types.** Table 14: BPM Events types and Java counterparts on page 128 lists BPM Events types and their Java counterparts.
- **Creating java objects.** Use the "new" constructor operator to create a Java object.
- **Catching and throwing java exceptions.** Catching and throwing Java exceptions is not supported in rule conditions. For more information, see Exception catching on page 130 and Exception throwing on page 130.
- **Creating rules with Vector and other complex Java objects.** Cast operator is introduced.
- **Creating new event types in java.** BPM Events can be defined and named in Java, like any other types, and referenced in declarations or in rule header. We have added an event constructor, using structural notation that accepts complex object attributes.

Mapping data types

The default mapping between Java types and BPM Events types is as shown in Table 14: BPM Events types and Java counterparts on page 128.

Table 14: BPM Events types and Java counterparts

<table>
<thead>
<tr>
<th>BPM Events type</th>
<th>Java type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
<td>Mapping to Integer class is done if an Object is required (for example, Vector add (Object))</td>
</tr>
<tr>
<td>float</td>
<td>Float double</td>
<td>BPM Events float is actually implemented as a double</td>
</tr>
<tr>
<td>string</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>char</td>
<td></td>
</tr>
<tr>
<td>&quot;java&quot;.&lt;Java classname&gt; for example,</td>
<td>Object</td>
<td>The class name is actually a class path, which is resolved using CLASSPATH environment variable.</td>
</tr>
</tbody>
</table>
Object instantiation

As shown in Table 14: BPM Events types and Java counterparts on page 128, BPM Events uses a syntax that is similar to Java. Note that when invoking a Java object, BPM Events uses the full class name, prefixed by the adapter name, "java". The Java adapter is used for qualifying any Java call, and is an abstraction of the Java Virtual Machine (No actual adapter requires writing).

In the following example, a Java object of type Vector is created inside the function "gen()". The scope of the value identifier "vec" is the function only, but this function could return the new Vector object, which could be used inside rules that invoke this function.

Instantiation is possible in any bloc statement, that is, in functions, rule actions, or initialize{} and finalize{} sections. In the example below, new is a constructor operator that creates or instantiates the Java object called Vector().

```
application BP_test_suites
using java.util.Vector;
module jobj_event_test
fun gen(i: int){
  val vec = new ~Vector();
  ...
}
```

Note: The "perform" action is still in use and is reserved for asynchronous calls. Do not use the "perform" action, as currently implemented, to access Java objects.

Static method invocation or property access

The method or property must be public. Note the class name, used with the "qualifier" notation:

```
Val res1 = ~MyClass::MyStaticMethod();
Val res2 = ~MyClass::MyStaticProperty;
...
If (~MyClass::MyStaticProperty > 123)
Then {...}
```

Instance method invocation or property access

To invoke an instance method, the method or property must be public. The following illustrates how object references are handled, and various types of method invocations using them.

```
Val myobj1: ~MyClass = new ~MyClass();
myobj1.MyMethod1();
val v2: int = myobj1.MyMethod2();
myobj1.MyMethod3(v2);
Val v3 = myobj1.MyProperty1;
...
If (myobj1.MyMethod2() > 123)
Then {...}
Exception catching

Exception catching is a procedure that gets exceptions generated by Java object invocation. The syntax is similar to Java, and is used inside blocs (either the action part of a rule, or the `initialize()` and `finalize()` sections). This procedure is shown in the following example:

```
try_statement() ::= "try" compound_statement() ( catch_statement() ) +
catch_statement() ::= "catch" "(" <ID> ":" type() ")"
compound_statement()
throw_statement() ::= "throw" expr() ";"
```

Note that Java exception catching is only supported in complex statements (blocs), that is, not in rule conditions. In order to extend it to rule conditions, we recommend:

- "Wrapping" the method invocations on objects in conditions, inside a BPM Events function defined at module level. This method catches exceptions.
- Writing a function of type boolean (predicate) representing the exception-sensitive condition, in cases when the result of a condition containing a Java access varies based on exceptions raised by the Java call.

Exception throwing

Exception throwing enables BPM Events to generate exceptions, similar to—and in the same format as—Java exceptions. The syntax is similar to Java, and is used inside blocks (either the action part of a rule, or `initialize()` and `finalize()` sections). An example of exception throwing follows:

```
throw_statement() ::= "throw" <exp> ";"
<exp> may be java object (which implements Throwable)
```

Using complex Java objects in rules

BPM Events can manipulate complex Java objects in rules. The following example shows how to use values of type Vector in rules and modules. Note the use of the `cast` operator, which in Java is a program action that converts an object from one type to another. Here, the `cast` operator (shown in **bold**) enforces the mapping to a BPM Events type, and ensures downcasting to a Java type. Downcasting is checked at run-time.

```
application BP_test_suites
using java.util.Vector;
using java.lang;
module jobj_event_test
fun gen(i: int){
  val vec = new ~Vector();
  vec.add(i);
  val evt = event A::B{vec: vec};
  notify evt;
}fun genl(i: int){
  val vec = new ~Vector();
  val v = new ~Vector();
  v.add(222);
  vec.add(v);
  val evt = event A1::B{vec: vec};
  notify evt;
}group jobj_event_test
rule rule_1
  activated by
    evt1 of A::B,
    evt2 of A1::B
```
then {
  val vec: Vector = evt1.vec;
  for (var i = 0; i < 10; i++) {
    vec.add(new Integer(2*i*i));
    vec.add(i*i);
  }
  for (var iter = vec.iterator(); iter.hasNext(); ) {
    val i = cast(int)iter.next();
    println(">> <1>i: "+i);
  }
}

rule rule_2
activated by
evt1 of A::B,
evt2 of A1::B
if ((cast(Vector)evt2.vec).size() > 2)
then {
  val vec: Vector = evt2.vec;
  for (var i = 0; i < 10; i++) {
    val v = new Vector();
    v.add(2*i*i);
    vec.add(v);
  }
  for (var iter = vec.iterator(); iter.hasNext(); ) {
    val v = cast(Vector)iter.next();
    val i = cast(int)v.elementAt(0);
    println(">> <2>i: "+i);
  }
}
println(">> start: job_event_test");
gen1(100);
gen(200);
gen(300);
println(">> done: job_event_test");
}

---

Note: Global Java object references are not recoverable in the current version of BPM Events.

Using external java objects in rules

BPM Events allows the invocation of user-defined Java objects from rules. The location and packaging of these external Java objects is described below.

Note: Make sure the user-defined class does not use any of the BPM Events internal classes. Failure to adhere to this rule may cause problems during the loading of the application.

• Locations common to all applications
  • OEBPS_HOME\ebmsapps
  • OEBPS_HOME\ebmsapps\common\ejbadapters
  • OEBPS_HOME\lib

• Locations specific to an application
  • OEBPS_HOME\ebmsapps\<application name>\lib

In general, BPM Events can load the class files placed directly in the following locations:
  • OEBPS_HOME\ebmsapps
The classes from jar files are loaded from the following locations:

- OEBPS_HOME\ebmsapps\common\ejbadapters
- OEBPS_HOME\ebmsapps\<application name>\lib
- OEBPS_HOME\lib

The lib location is used to search for .jar files as well as class files, but the location ebmsapps\<application name>\lib is used to search only for the jar files.

### Searching class files

The class files are searched in the following order:

1. From jar files located in OEBPS_HOME\ebmsapps\<application name>\lib
2. From jar files located in OEBPS_HOME\ebmsapps\common\lib
3. class files located in OEBPS_HOME\ebmsapps
4. class files located in OEBPS_HOME\ebmsapps\common\ejbadapters

If a class exists at more than one location, then it is loaded from the first location where it is found as per the above order.

### Mixing two separate instance of a class

When a JAR file is stored in an application folder, say app\lib\some.jar, the scope of a class accessed from BPM Events code of application app is limited to that application. For example, if there is a JAR file, app1\lib\some.jar, then a class A included in some.jar is instantiated in a classloader for app and app1 respectively.

Although a Java object app cannot access BPM Events’s Java binding features, it is still possible to get an object allocated by app1 through global variable, or by a function defined in BPM Events module in app1. In this case, there is a possibility of mixing two instances of the ‘same’ class A.

In general, it is not a good practice to import an application using another instance of the class.

### EJB component invocation

Invocation of an EJB from rules is similar to using external classes from BPM Events rules. The following sample application illustrates this process. This application, EJBTest, has a very simple rule file as follows:

```java
application EJBTest
using java.lang.*;
using java.util.*;
using common.externalperformers.*;
module EJBTest_rules
{
    try{
        val eJBClient = new ~EJBTestClient("t3://milpitas.openedge.com,belgium.openedge.com:8562");
        println(eJBClient.getStatus());
    }
}
```
The above application uses a Java class called `EJBTestClient` which looks up the EJB and calls methods on it. The constructor (which takes the URL of the application server) and the `getStatus()` method of `EJBTestClient` are illustrated in the following sample:

```java
public EJBTestClient(String url)
    throws NamingException, CreateException, RemoteException
{
    this.url = url;
    ejbtest_ref = lookupEJBTest(); // ejbtest_ref is the remote reference to EJB kept by EJBTestClient
}
public String getStatus() throws RemoteException
{
    System.out.println("Before calling getStatus");
    String s = ejbtest_ref.getStatus();
    System.out.println("After calling getStatus, return string is " + s);
    return s;
}
```

In order to make the EJB call work:

1. Make the `EJBTestClient.class` part of the `common.externalperformers` package and copy it to the `%OEBPS_HOME%\ebmsapps\common\externalperformers` folder.

2. Note the `using common.externalperformers.*` statement in the rule file, which says that BPM Events will attempt to find `EJBTestClient.class` in this folder.

### Dynamic loading

Since BPM Events is a statically typed language, in order to use updated java classes without recompiling BPM Events code, the signature of property/methods of original class must not be changed. BPM Events binds external Java class reflection objects at application loading time and the server must be restarted in order to use the updated java classes.

### Using modified class implementation in BPM Events

For BPM Events to use the modified class implementations:

1. Stop BPM Events server.
2. Overwrite the existing jar file or class files with the modified jar or class files.
3. Restart BPM Events server.

### Using modified class implementation in BPM Events when signature of the Java class is changed

If the signature of the Java class has changed, you must perform the following steps:
1. Unload the BPM Events application.
2. Stop BPM Events Server.
3. Overwrite the existing jar file or class files with the modified jar or class files.
4. Update the BPM Events application to invoke java class with modified signature.
5. Recompile the BPM Events application.
6. Restart BPM Events server.
7. Load the modified application in BPM Events
Administering and operating BPM Events

Some BPM Events administration operations are done through the Administration module in Business Process Portal. However, a complete set of administration functions are still only available through the BPM Events Admin—a command line, menu-based tool. The important aspects of operating and administering BPM Events through BPM Events Admin are the:

- Management of events
- Management of failure recovery
- Upgrade of applications (rules) as well as of new versions of the system.

Precautions are required because of BPM Events’s highly dynamic environment: by nature, BPM Events interoperates with other systems—inside or outside the Business Process Server system. Changes in configuration or application upgrades may require completion with minimal interruption. BPM Events allows for dynamic, run-time changes in its rule set-up, and provides a comprehensive recovery capability.

For details, see the following topics:

- The rule development cycle
- Troubleshooting cases
- BPM Events Admin: BPM Events administrator utility
- Event management
- Persistence and recovery
- Loading and recovering applications
- BPM Events configuration parameters
The rule development cycle

This section explains various stages of the rule development cycle.

Rule compiling

You may create rules in three ways:

- With the **Rule Editor** (included only in the Business Process Server platform).
- With the **Rule Wizard**.
- With any text editor.

Rules are saved as a text file (also referred to as a rule module) and are associated with an application environment, that is, must be stored under an application folder. For example, assuming the application name is `appName`, you must store the rules for this application under:

```
OEBPS_HOME\ebmsapps\appName\rules
```

The main rule module for the application is named `appName_rules.bps`.

```
OEBPS_HOME\ebmsapps\appName\rules\appNames_rules.bps
```

After a rule module is compiled, there should also be a file with the extension `*.bpo` under the same folder.

```
OEBPS_HOME\ebmsapps\appName\rules\appNames_rules.bpo
```

If you use the Business Process Server RuleEditor to generate the text version of a rule module, then it automatically compiles the rules for you and generates the `BPO` file. If you use any other text editor, then you must compile the rules yourself. To do this, open the `OEBPS_HOME\bin` folder and run the utility:

```
RuleCompiler.cmd (on Windows)
RuleCompiler.sh (on UNIX)
```

You may compile rules in one of the three following ways. Note that the rule modules compiled must be stored under an application rule folder (`OEBPS_HOME\ebmsapps\appName\rules`).

- Compile the main module of your application (here, `appName_rules`), and all the directly or indirectly imported modules, regardless of their previous status (some may have been compiled before (that is, have `*.bpo` files), some may not). Note that this command does not compile the independent modules (if any) that do not have an import relationship with the main rule module of the application. The compile command is:

```
RuleCompiler -a appName
```

- Compile a single module and all the modules it imports. The compile command must use the option `-m`, and requires giving the qualified name of the module. The general syntax of this command is:

```
RuleCompiler -m <appname>::<modulename>
```

For example,

```
RuleCompiler -m appName::mymoduleA
```
• Compile all rule files under an application rule folder:

```
RuleCompiler -a appName -all.
```

**Note:** Module names and file names are case sensitive.

**Note:** If your application uses the BP Server API library of BPM Events, then compile with the `-blapi_lib` option or import the `BP ServerAPI::BP ServerAPI_rules` module into your application by using the following command:

```
import BP ServerAPI::BP ServerAPI_rules;
```

---

**RuleCompiler command options**

The syntax and all options of the Rule Compiler and Rule Compiler Help follow:

```shell
%> RuleCompiler -help
```

**Syntax:**

```
RuleCompiler (-m app_name "::" module_name | -a app_name) (option)*
```

**Option:**

```
option ::= force_option | -all | -warning | -no_warning | -help
```

**force_option ::=**

```
-f | -ff | -nf | -app_level | -all_level | -single_level
```

**Options:**

- `-f` or `-app_level` compile all referred modules in the same application
- `-ff` or `-all_level` compile all referred modules (default)
- `-nf` or `-single_level` compile specified module
- `-all` compile all modules under the application (applied only for `-a`)
- `-blapi_lib` import BP ServerAPI
- `-no_blapi_lib` do not import BP ServerAPI (default)
- `-warning` output warning (default)
- `-no_warning` do not output warning
- `-help` display help menu

---

**Setting up and monitoring the database persistence tables**

Setting up and monitoring the database persistence tables is usually not required when running a published application, but may be useful at development time. When doing rule development between two test runs, you may have to reset the BPM Events tables in a database, to make sure that each test uses a clean database state and is not affected by previous runs.

First, make sure that the database user that is specified in the `oebpsdb.properties` file is actually registered in the database server.
You can then do a complete set-up/clean-up with the command:

```
setupOEBS -c ALL
```

Or, if you do not have Business Process Server:

```
setupBPMEvents -c ALL
```

After this, you can start BPM Events server.

When debugging, be aware of a few useful SQL commands you can do to check the content of BPM Events database tables. Entering these commands tells you how many events are in the database, how many were processed, what is the content of the cache, etcetera. The SQL commands are:

```
// tells how many events are in the BP Server event queue.
SQL> select count(*) from BPMEvent;
// shows the schema (attributes) of a table (for example, BPMEvent)
SQL> desc BPMEvent;
// tells what was the last event successfully processed by BPM Events:
SQL> select * from BPMEventsEventCounter;
// tells what is the highest (latest) event ID generated, in the OEBPS event queue.
SQL> select max(event_ID) from BPMEvent;
// tells what is the content of the cache, for events initially notified via the database queue:
SQL> select * from BPMEvents_EventExtent;
// tells what events are currently scheduled:
SQL> select * from BPMEvents_ScheduledItemExtent;
```

### Starting and operating the BPM Events server

Once a compiled version of the rule modules is produced, these modules are loaded in the BPM Events server. Start the BPM Events server by one of following methods.

#### Starting BPM Events from Business Process Portal

To start BPM Events from Business Process Portal, perform the following steps.

1. From Business Process Portal choose Administration > System > Status to open the System Status page.
2. Select the checkbox for the server that you want to start.
3. Click Start.

#### Starting BPM Events from the Start menu

To start BPM Events from the Start menu:

1. Choose Start > All Programs > Progress > OpenEdge > BP Server > App Servers > Start Portal Server.
2. Choose Start > All Programs > Progress > OpenEdge > BP Server > App Servers > Start EJB Server.
3. Choose Start > All Programs > Progress > OpenEdge > BP Server > BP Servers > Start Business Process Server.
4. From BPM Events Admin, use the Server menu.
Starting BPM Events from the BPM Events component

To start BPM Events from the BPM Events component:

1. Choose Start > All Programs > Progress > OpenEdge > BP Server > BP Servers > Start Business Process Server.
2. From BPM Events Admin, use the Server menu.

Monitoring the server with BPM Events Admin utility

Once the servers are started, you can access the BPM Events Admin utility. BPM Events Admin provides more detailed administration functions, and allows you to monitor the state of your server as well as its internal objects (infopads).

To monitor the server.

- In the Command Prompt window, run the following command.

```
startBPMEventsAdmin
```

It displays the main menu of the BPM Events Admin utility.

```
0) Back
1) Server Monitor
2) Loader
3) Rule Manager
4) Persistent Structure Monitor
5) Event Manager
6) Scheduler
7) File manager
8) JMS
```

Loading rules using BPM Events Admin utility

To load rules.

1. Start the BPM Events Admin utility.
2. Enter 2 to get to the Loader menu.
3. From the Loader menu, enter 3 for "Load Application".
4. Enter the application name.

Viewing an installed infopad using BPM Events Admin utility

To view an installed infopad.

1. Start the BPM Events Admin utility.
2. Enter 5 to get to the "Event Manager" menu.
3. From the "Event Manager" menu, enter 1 for "Show All Event Types".
4. Enter the Infopad name or press Enter to list all installed Infopads.

BPM Events Admin displays the set of all event types currently in cache.
Application upgrade scenarios

When developing rules for an application, one of the following two common upgrade scenarios may occur, which requires manipulation of the database backup:

Application Upgrade Scenario #1:
BPM Events server is running an application that requires replacing. In other words, we want to install a new application that ignores everything from the previous version. The procedure to follow is:

1. Change your application rules.
2. Unload the application.
3. Load the application.

Application Upgrade Scenario #2:
BPM Events is currently running an application that we want to update. But we do not want to lose the current application objects, for example, the current infopads state. Let us assume the update is restricted to some rule change (that is, no change in infopad definition or in any other object) declared in a module. It is then possible to only replace the rules of the module without losing the current state of the running application.

Note: This dynamic upgrade procedure is NOT appropriate for changing infopad definitions. It should only be used for minor changes affecting the rules part of a module. These new rules should be consistent with the previous state of the application objects (infopads, stored events), as saved in the database between shutdown and restart.

Upgrading a dynamic application

To upgrade a currently running application with its current infopad state:

1. Update/Edit the rules in your application module - for example, M3 of application A1 - and recompile this module using the RuleCompiler or Rule Editor, so that a new *.bpo file is generated under ebmsapps\A1\rules.
2. Stop BPM Events server. Do this on the server host, using the BPM Events Admin tool (startBPMEventsAdmin, then select Server Monitor > Stop).
3. Replace the rule module in the backup image in database. The new rules are automatically reloaded by the recovery mechanism when the server is restarted. This replacement is done on the server host, using the following commands.

   unload module
   load module

4. Restart BPM Events. This can be done through the BPM Events Admin tool ("Server Monitor" menu, "Start" command). The server starts in recovery mode. It reloads the same internal state it had when shut down, except for the replaced rule module.
Log files and output files

The log file for the BPM Events server is `bpmevents.log`. This log file captures most messages or errors regarding BPM Events, therefore when a problem occurs check this file first. Each error message reports the rule name and the details of the event that triggered the rule when the error occurred. You may specify the desired level of detail of errors in the `bpmevents.conf` file.

Alternatively, you can also set the log levels, format, file size, and backup archive files using the Log configuration interface in Business Process Portal's Administration module. For more information, see "Chapter 12, Using the Log Viewer" in *Business Process Portal Administrator's Guide*.

When developing and debugging, we recommend relying on the print function (`println()`), with a text file as output. The default output is `bpmevents.output`, located under the BPM Events folder. The `println()` function can print any non-string object, by converting them automatically. It is then easy to print the content of events by giving the event variable as argument.

Loading rules

When loading (or installing) a rule module that imports other modules, all the imported modules are loaded, recursively, prior to the initial module. The `initialize()` section of a module is automatically executed once the loading phase is over. This section performs initial tasks, like creating and initializing infopads. The syntax that specifies these actions is the same as the syntax of the action part of rules.

It is important to use infopad names that does not match with existing infopads. In BPM Events, you are responsible for creating infopads that are unique across applications. Indeed, infopads are sharable across applications. For example, two cases of infopad name collision follow, along with the behavior of the engine:

- Assume an application A1 is currently loaded and creates an infopad "mypad". If a new application A2 loads and creates an infopad with the same name, then the previous infopad in A1 is replaced by the new version, both in memory and in database, and the previous version is lost.

- Assume an application A1 is currently loaded and creates an infopad "mypad", and this application does not discard this infopad in its `finalize[]` section, that is, has no discard (mypad) statement. If this application unloads and reloads, then it attempts to recreate an infopad already created (both in memory and in database). In such a case, the previous version is deleted (both in memory and in database), and the loading enforces the creation and initialization of the new version of the infopad.

Loading is subject to failure due to these initial actions. For example, the creation of a persistent infopad, completed when executing the `initialize[]` section, fails due to a database server connection problem. In such a case, the failure is considered serious enough to cause loading interruption.

Normally, an aborted loading should not affect the consistency of BPM Events. In case the internal state of BPM Events is considered undesirable, it is recommended to run the command `setupBPMEvents -c state` or `setupOEBPS -c bpmevents` (if you have Business Process Server) otherwise the corrupt backup state is discarded. Note, however, that in the current version it is not possible to isolate the backup of one application from other applications loaded in the server: All are discarded at once.

Loading rules using Business Process Portal's Administration module

Once you have a compiled version of the rules, the rules are automatically installed for you every time you install the corresponding application from the Administration module in Business Process Portal.
Loading rules using BPM Events Admin

If you want more flexibility than Business Process Portal's Administration module provides (for example, loading rule modules separately, in case the application was installed in another way, from the BP Server command interface, or if you are only using BPM Events), then you can access the BPM Events Admin utility.

**BPM Events Admin** allows for loading a particular module within an application (*Loader > Load Module*). The module name is given, always qualified by the application name, as in the example below:

```
HelpDesk::external_support
```

Unloading rules

Currently there are two ways to unload (or uninstall) rules:

- From Business Process Portal's Administration module, rules are unloaded whenever you uninstall the corresponding application.
- From BPM Events Admin utility:

Unloading rules from BPM Events Admin utility

To unload rules using BPM Events Admin utility.

1. Start **BPM Events Admin** utility.
2. Enter 2 to go to the *Loader* menu.
3. Enter 6 for "Unload application".
4. Enter the application name.

Points about unloading rules

Note that any infopad created by a module (*initialize{}* section) should normally be discarded by the same module, by a discard statement in the *finalize{}* section:

```
{ discard(<infopad_name>); }
```

If not, then you must remove the corresponding infopad tables from the database, after proper archiving if needed, before reinstalling the same application. Otherwise, reloading (reinstalling) the application again overrides the previous infopad. Indeed, the *initialize{}* section of the rule module (which is always executed when loading the module), tries to create infopads that already exist in the database.

BPM Events Admin allows for unloading a particular module within an application. The module name given is always qualified by the application name:

```
HelpDesk::external_support
```

Troubleshooting cases

In the case of a failure or some event described below, the rule engine may suddenly stop operating, that is, stop reading events from the Event Channel and processing them.
• **Internal exceptions.** Such as division by 0. In such cases, the error is considered serious enough to stop processing the current event. An error is logged (in `bpmevents.log`).

• **Communication failures.** For example, the database connection may be broken, and the retry policy—as specified in the configuration file—has been unsuccessful. Or, calls to the BP Server process interface are failing (as well as the retry policy for them). In such case, an error is also logged. Note that in this case, as in any failure involving a Java invocation, the Java exceptions may be caught at the BPM Events language level. So a rule could catch such failures, and proceed further. If no exception is caught, then this is a case where the server stops.

In such cases, the engine stops processing and the event reader remains in a suspended mode. However, the status of the server, when queried through the BPM Events Admin utility (described below) or through the server API, shows "STARTED", instead of "STOPPED" or "SUSPENDED".

In order to fix the problem, the you can proceed with one of the following scenarios.

**Fixing with full recovery**

**Troubleshooting Scenario #1: To fix with full recovery.**

1. Bring the stalled server to a STOPPED state, with the **BPM Events Admin** utility ("Stop" command). After this operation, the engine state is not lost, but ready for recovery from the database (except for the current event which requires reprocessing).

2. Fix the problem (for example, database server, or process engine availability).

3. Restart the server with the **BPM Events Admin** utility ("Start" command). The server recovers its previous state from the database, and reprocesses the failing event.

**Fixing by upgrading application**

**Troubleshooting Scenario #2: To fix with application upgrade.**

1. Bring the stalled server to a STOPPED state, with the **BPM Events Admin** utility ("Stop" command). After this operation, the engine state is not lost, but is ready for recovery from the database.

2. Discard the persistent image of the server state. This discards the current recovery data so that the server starts from a clean slate.

   • Run this command if you have BPM Events component only:
     ```bash
     setupBPMEvents -c state
     ```

   • Or run this command if you have the Business Process Server:
     ```bash
     setupOEBPS -c BPMEvents
     ```

3. Fix the problem (change and recompile your rule modules).

4. Restart the server using the **BPM Events Admin** utility ("Start" command). The state should show STARTED when queried.
Note: BPM Events Admin utility itself does not require restarting. The engine then restarts with an empty state (no recovery).

5. Reload the application.

Note: Events generated in the database from the previous application run, but not yet processed, are processed then. The user may actually manipulate the "event counter" in the database (see database queries section) so that the engine starts from the desired event, if needed.

Fixing with recovery and partial rule upgrade

In this example, we want recovery from the previous state (infopads, etcetera) but still want some limited changes to the rules, so that the overall recovered state is still consistent with these changes.

Troubleshooting Scenario #3: To fix with recovery and partial rule upgrade.

1. Fix the problem (change and recompile the rule module).
2. Bring the server to a SUSPENDED state, with BPM Events Admin utility ("Suspend" command), so that it stops processing future faulty events.
3. Unload (or disable) the module to be replaced. Note that unload executes the finalize{} section of the module (and of imported modules).
4. Load the new rule module. Note that this executes the initialize{} section, so the state of any declared infopads are reinitialized.
5. Bring the server to a STARTED state, using the BPM Events Admin utility ("Resume" command), so that it starts processing events with the new rules.

BPM Events Admin: BPM Events administrator utility

This section explains you various menu options of the BPM Events Admin utility and their functions.

Functions

BPM Events Admin allows you to:

• Monitor the state of BPM Events server.
• Do most of the operations required when developing and testing an application.
• Observe the internal application objects of the server (rules, events, infopads).

Start the BPM Events server prior to starting BPM Events Admin. Otherwise, BPM Events Admin displays an error message and exits.
Starting BPM Events Admin

Once the server is started, you can access the BPM Events Admin utility, which provides detailed administration functions.

To start BPM Events Admin utility

• startBPMEventsAdmin -ejb

It displays the BPM Events Admin main menu.

BPM Events Admin main menu

The main interface separates the functional areas:

========== Host : localHost ==========
0) Back
1) Server Monitor
2) Loader
3) Rule Manager
4) Persistent Structure Monitor
5) Event Manager
6) Scheduler
7) File Manager
8) JMS
Enter your selection (1 - 8):

• 0) Back: Exits from BPM Events Admin. Note that this does not stop the BPM Events server. It is possible to restart BPM Events Admin to connect again to the same server instance (as an EJB client, in the standard configuration). If you decide to also stop the server, then it should be done independently.

• 1) Server Monitor: Enables you to control the server, and observe whether it is operational.

• 2) Loader: Enables for operations on the rules of an application: load / unload.

• 3) Rule Manager: Shows the rules that are currently loaded and their state.

• 4) Persistent Structure Monitor: Shows the Infopad Monitor and Persistent Map Monitor submenus.

• 5) Event Manager: Shows the events in event cache.

• 6) Scheduler: Shows the current status of scheduled events.

• 7) File Manager: Enables for managing (browsing and compiling) rule modules.

• 8) JMS: Enables for managing listeners to JMS queries.

Server Monitor menu

The Server Monitor menu allows monitoring and controlling the server.

Server Monitor menu:

========== Server Monitor ==========
0) Back
1) Print Status
2) Suspend
3) Resume
4) Stop
5) Start
Enter your selection (0 - 6):

- 0) Back: Exits from Server Monitor and returns you to the main menu.
- 1) Print Status: Shows the server status:
  - STARTED: The server is in a state of processing events. It can then be either stopped ("stop") or suspended ("suspend").
  - STOPPED: The server is stopped. This is equivalent to a shutdown. No state is preserved in memory. If the server is restarted, then its state is reinitialized with the recovery backup. The only available transition from this state is to restart the server ("start").
- 2) Suspend: Suspends the server (status: SUSPENDED): no event is processed from the database queue. The server retains its internal state (loaded applications and objects). The expected transition is then to "resume" or to "stop".
- 3) Resume: Resumes the server processing, after being suspended.
- 4) Stop: "Logically" shuts down the BPM Events server. All internal state is lost.
- 5) Start: Restarts the BPM Events server. This is equivalent to an initial start: the recovery procedure recovers the state the server was in when last stopped or shutdown.

Loader menu

The Loader menu allows for manipulating the rules associated with applications.

============= Loader =============
0) Back
1) Show Loaded Applications
2) Show Loaded Modules
3) Load Application
4) Load All Modules
5) Load Module
6) Unload Application
7) Unload Module
8) Reload Application
9) Reload Module
Enter your selection (0 - 9):

- 0) Back: Returns you to the main menu.
- 1) Show Loaded Applications: Displays all loaded applications.
- 2) Show Loaded Modules: Displays all loaded modules for a given application.
- 3) Load Application: Loads the main rule module associated with an application. For example, loading the application "myApp" loads the main (compiled) module of the application: "myApp_rules.bpo". If this module imports other modules, then these are loaded first. When loading a module, the initialize{} section of the module is automatically executed. All the loaded rules are enabled.
- 4) Load All Modules: Loads all the rule modules under an application folder.
- 5) Load Module: Separately loads additional modules for an application. The module does not require importing by another (already loaded) application module.
• 6) Unload Application: Unloads all the rules associated with an application. Note that this operation is not the exact reverse of Load Application, as the latter only loads the main rule module, while Unload Application unloads ALL modules currently loaded for this application. For example, unloading the application "myApp" unloads the main (compiled) module of the application: "myApp_rules.bpo", as well as any other independent module that is, any module without any import relationship with the main module, that was loaded separately. If an unloaded module imports other modules, then these are unloaded next. When unloading a module, the finalize{} section of the module is automatically executed.

• 7) Unload Module: Unloads a module of a loaded application and executes its finalize{} section. It is not possible to unload a module that is imported by another (loaded) module.

• 8) Reload Application: Loads the application again. This refreshes all the rule modules of the application. It does not refreshes the data modules.

• 9) Reload Module: Loads the specified rule module again. It does not refreshes the data modules.

Rule Manager menu

The Rule Manager menu shows the currently loaded rules, and their status.

----------- Rule Manager -----------
0) Back
1) Show Current Rules
2) Show Enabled Rules
3) Disable Rules
4) Enable Rules
Enter your selection (0 - 4):

• 0) Back: Returns you to the main menu.

• 1) Show Current Rules: Displays currently loaded rules, optionally by groups. For example, the three rules below belong to the application BP_test_suites, the module DateTests.

BP_test_suites::DateTests::printEventDateProjection
BP_test_suites::DateTests::printEventFromInfopad
BP_test_suites::DateTests::testDateProjectionInCondition

The application name is required.

• 2) Show Enabled Rules: Shows only the enabled rules, that is, those currently used for processing events.

• 3) Disable Rule: Disables a rule, a rule module, or an application.

• 4) Enable Rule: Enables a rule, a rule module, or an application.

Persistent Structure Monitor menu

The Persistent Structure Monitor menu shows the Infopad Monitor and Persistent Map Monitor submenus. The Infopad Monitor submenu shows state of currently loaded infopads whereas the Persistent Map Monitor submenu allows you to view and modify map instance details.

---------- Persistent Structure Monitor ----------
0) Back
1) Infopad Monitor
2) Persistent Map Monitor
Enter your selection (0 - 2):
Infopad Monitor submenu

The **Infopad Monitor** submenu allows for checking the status of current infopads. While using these submenu options, wherever you have to specify infopad names, you must specify fully qualified infopad name (for example, **app_name::module_name::infopad_name**).

```
============ Infopad Monitor ============
0) Back
1) Show All Infopads
2) Show Infopad Property
3) Show All Cells of Infopad
4) Show a Single Cell of Infopad
5) Show a Single Slot of a Cell
6) Change Infopad Slot Value
7) Discard Infopad
Enter your selection (0 - 7):
```

- 0) Back: Returns you to the previous menu.
- 1) Show All Infopads: Allows you to select all infopads currently loaded. The infopad content is displayed. If no infopad name is given, then the list of all infopads currently loaded is displayed.
- 2) Show Infopad Property: Shows the definition of an infopad.
- 3) Show All Cells of Infopad: Displays the content of each cell of an infopad—one cell per line.
- 4) Show a Single Cell of Infopad: Displays the content of a selected cell or an infopad, with all slot values.
- 5) Show a Single Slot of a Cell: Displays the single slot of a selected cell of an infopad. Useful in case a cell has many slots or slots with large string values.
- 6) Change Infopad Slot Value: Change the content of a single slot of a selected cell of an infopad.
- 7) Discard Infopad: Discard the given infopad in memory and in the database.

Persistent Map Monitor submenu

The **Persistent Map Monitor** submenu allows for checking the status of current persistent map. While using these submenu options, wherever you have to specify persistent map names/instance names, you must specify fully qualified name (for example, **app_name::module_name::map_name**, **app_name::module_name::map_instance_name**).

```
============ Persistent Map Monitor ============
0) Back
1) Show a Single Persistent Map Instance
2) Show a Single Key Value of Persistent Map Instance
3) Show a Single Attribute Value of Persistent Map Instance
4) Change Attribute Value of Persistent Map Instance
5) Discard Persistent Map Instance
Enter your selection (0 - 5):
```

- 0) Back: Returns you to the previous menu.
- 1) Show a Single Persistent Map Instance: Shows the values of the specified map instance.
• 2) Show a Single Key Value of Persistent Map Instance: Shows all the attribute values of the specified map instance key.

• 3) Show a Single Attribute Value of Persistent Map Instance: Shows the attribute value of the specified attribute name.

• 4) Change Attribute Value of Persistent Map Instance: Allows you to change a particular attribute value of the specified map instance.

• 5) Discard Persistent Map Instance: Removes the specified map instance.

**Event Manager menu**

The Event Manager menu allows for looking up the event cache content.

```
============ Event Manager =============
0) Back
1) Show All Event Types
2) Count Events
3) Discard Events
Enter your selection (0 - 3):
```

• 0) Back: Returns you to the main menu.

• 1) Show All Event Types: Displays the set of all event type currently in cache. Each event type is of the form:

  `<evttype>::<evtvalue>`

  or

  `XMLEvent::<evttype>::<evttype>Event`

• 2) Count Events: Counts the number of events of a particular type. Enter an event type in the form:

  `<evttype>::<evtvalue>`

  or

  `XMLEvent::<evttype>::<evttype>::Event`

  or

  `empty`

• 3) Discard Events: Discards events of a particular type from the cache. Enter an event type of the form:

  `<evttype>::<evtvalue>`

  or

  `XMLEvent::<evttype>::<evttype>Event`

  or

  `empty`
Note: If no entry is provided for the event type (return key), then all events are discarded from the cache. In order to cancel this command, typing any string instead of the event type returns to the Event Manager menu without effect.

Scheduler menu

The Scheduler menu allows for checking the status of currently scheduled events.

========== Scheduler ==========
0) Back
1) Unschedule
2) Print Scheduler Status
Enter your selection (0 - 2):

- 0) Back: Returns you to the main menu.
- 1) Unschedule: Enables you to unschedule one or more events, based on their name. Note that several events can have the same name: all events with the given name are unscheduled.
- 2) Print Scheduler Status: Displays the current list of scheduled events with their properties, including their names.

File Manager menu

The File Manager menu enables you to browse and recompile the rule modules currently stored under the application directories.

========== File Manager ==========
0) Back
1) Show Stored Applications
2) Show Stored Modules
3) Compile Application
4) Compile All Modules of Application
5) Compile Module
Enter your selection (0 - 5):

- 0) Back: Returns you to the main menu.
- 1) Show Stored Applications: Shows all the applications currently available under the root application.
- 2) Show Stored Modules: Shows all the rule modules currently available under an application folder.
- 3) Compile Application: Compiles the given application (that is, the rule file <appName>_rules and all imported modules). This is equivalent to RuleCompiler -a <appName>.
- 4) Compile All Modules of Application: Compiles all the modules under an application folder.
- 5) Compile Module: Compiles a module under an application folder.
JMS menu

The JMS menu enables you to manage JMS listeners and JMS maps.

========== JMS ==========
0) Back
1) Add JMS Map
2) Remove JMS Map
3) Update JMS Map
4) Get the list of JMS Maps
Enter your selection (0 - 9):

• 0) Back: Returns you to the main menu.

• 1) Add JMS Map: Adds a JMS map for a specified message type. Locate the map file in the OEBPS_HOME\BPMEvents\jmsmaps folder. The type should be the same as the top level element of the XML body and starts with a capital letter. It should not contain any special characters like a dash (-). Every time a new message of that type is processed by BPM Events, the mapping file is used to extract the exposed properties from the JMS message.

• 2) Remove JMS Map: Removes the JMS map for a specified message type.

• 3) Update JMS Map: Updates the JMS map for a specified message type.

• 4) Get the list of JMS Maps: Retrieves the list of mapping properties of each message type.

Event management

This section explains how the BPM Events rule engine manages various events.

Processing external and internal events

The rule engine distinguishes two kinds of events, which are managed in two separate event entry queues:

• **External events.** These are events obtained from the Database Event Channel (pulled out of the Business Process Server database event queue by BPM Events using the DEC adapter), events generated by the scheduler, or events coming from JMS queues.

• **Internal events.** These are events generated by BPM Events rules, with the `generate()` action statement. Note that scheduled events are not considered internal events, even if the scheduler is managed by BPM Events. This is because scheduled events are fired and posted independently from the actual rule processing; the scheduling service could as well be managed outside BPM Events.

Both types of events are treated similarly with regard to rules, but with different priorities: internal events are always processed in higher priority, that is, as long as there are events in the internal events entry queue, the rule engine processes these first. Only when the internal queue is empty, the rule engine processes the next event in the external event queue.

Transactional and serial events

This section provides details about the transactional and serialized external events.
Transaction external events

An external event is processed as a single transaction, both from a BPM Events server (internal state) perspective, as well as from a database (recovery) perspective. In other words, if a serious failure occurs during the processing of this event, then BPM Events recovers the state it was in just before processing this external event, which was also the last recovery state recorded in the database. The partial processing is ignored. Note that even if several internal events were generated and processed as a consequence of the processing of this single external event, the processing of all these internal events are rolled back.

More precisely, the processing of an external event includes the evaluation of all the rules this event can trigger. In turn, this includes the execution of corresponding actions, as well as the evaluation of the rules triggered by internal events generated by these actions (closure of rule chaining). We call the set of all these cascading actions the action closure of the event.

Serialized external events

The processing of the next external event does not start before the action closure of the current event is completely processed. That is, all internal events resulting from the processing of this current external event must be completely processed.

Also, when processing an event, external or internal, the engine does not process another event in parallel: the evaluation engine is single-threaded. When BPM Events is busy processing events and more events are loaded in the queue, be aware that it takes more time to process the events.

Note: When a rule action asynchronously activates an external system, the BPM Events engine considers the action over as soon as the call has been transferred to the external system. This call is only part of the action closure, and is usually complete before the actual external action is complete. Thus, external actions may still proceed in external systems, unknown from BPM Events, concurrently with the processing of the next event.

Event scheduling

Rules can schedule events with the schedule() action. Such events are stored persistently and fired at the specified date. A scheduled event, even if it is considered as an external event, are handled at higher priority than regular external events. This is done so that the firing time is as close to the processing time as possible, in order to comply with the expected firing time.

Event caching and indexing

The purpose of the event cache is to provide fast access to past events, for event correlation purpose. The event cache is not required if rules did not perform event correlation. Any event notified to BPM Events, that matches at least one rule activation clause, is automatically added to the cache. This cache is not only a memory structure, but it also has a persistent image in database. When recovering from a shutdown, the original in-memory cache is recovered from this persistent image.

The event cache supports two indexing schemes:

• type ID indexing. All events that are posted to the database event queue, whether from Business Process Server (BP Server) or from external sources, are indexed based on their type ID. The type ID for BP Server events in <evttype>::<evtvalue>. The type ID for XML messages is XMLEvent::<evttype>::<evttype>Event.
• **group ID indexing.** All events of type BP Server are also automatically indexed based on their `PROCESSINSTANCEID` attribute. In other words, each entry of this index contains the list of all events generated by a single process instance. This indexing allows for fast event correlation within a single process instance.

For XML messages, define group ID for each XML message type in the XML Event BPM Events class. The public function `get_group_id() : int` can be implemented. For instance, let's say we have a Purchase Order XML message that contains the customer ID. Use this ID to index this type of purchase order. The xsd follows:

```xml
<xsd:complexType name="PurchaseOrderWithIdType">
  <xsd:sequence>
    <xsd:element name="custId" type="xsd:decimal"/>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="billTo" type="USAddress"/>
    ....
  </xsd:sequence>
</xsd:complexType>
```

In the `<BPM Events>\ebmsapps\XMLEvent\rules\PurchaseOrderWithId class, add the get_group_id function:

```java
public fun get_group_id() : int
{
    return read_jobj().getCustId().intValue();
}
```

We call "cache entry" the set of events associated with one of these index entries. Cache entries are automatically flushed once the cache size reaches its limits. This flushing is done based on a LRU algorithm. If a swapped out cache entry is required later for event correlation, then it is automatically reloaded in memory by the cache manager.

The property `bpmevents.engine.eventcachesize` in the `bpmevents.conf` file specifies the maximum number of events the in-memory cache stores at any time, but not the maximum number of events the cache manager can handle: if the cache size is too small for the actual number of events needed for correlation, then this only affects performance as there is more swapping in and out to the database.

### Internal events handling

Internally generated events—generated by the `generate()` and `schedule()` action statements—are also cached and accessible later for correlation. These events are indexed based on their type (that is, their type and value attributes). We recommend that you discard any of these events as soon as they are not needed anymore, as each index entry for these events may become quite large.

### Cache management from the application

The rules of an applications should explicitly remove from the cache the events generated by this application (or for this application), once they are no longer useful. Once an event is removed from the cache, it is no longer accessible for future correlations. However, the event still remains in the database event queue, until it is flushed or archived. Consider two cases:

- **Events of a type other than BP Server.** Such events are notified either via one of the event channels, or internally, from a `generate()` or `schedule()` action statement. In such cases, a rule of the application decides when to discard them. When an event of this kind has triggered a rule and is not expected to be reused by future correlations, it is discarded by this rule. If it is expected to be reused for future correlations, then it is discarded by the rule that does this correlation. This is the case for the rule `stock_back_to_normal` in the tutorial: this rule discards not only the event that triggered it...
(Alarm::ProductInventoryNormal), but also the past event that correlates with the triggering event (ProductOutOfStock), since this event is no longer relevant, as shown in the following example:

```plaintext
rule stock_back_to_normal
activated by
event1 of Alarm::ProductInventoryNormal,
event2 of Alarm::ProductOutOfStock
if (event2.defaultingproduct = event1.backtonormalproduct)
then { discard (event1, event2); }
```

- **Events generated by Business Process Server business processes (of type "BP Server")**. These event types usually become inoperable once the process instance is completed. In that case, it is appropriate to delete from the cache (both memory and persistent image) all the events that relate to the completed process instance. Because these events are automatically indexed based on the process instance ID, they can be discarded at once, using the action statement: discardIndexEntry(). We then recommend that you write such a rule for each application (here identified as "myprocess"):

```plaintext
rule end_process
activated by evt of BP Server::PI_COMPLETED
if (evt.PROCESSTEMPLATENAME = "myprocess")
then { discardIndexEntry(evt.PROCESSINSTANCEID); }
```

## Persistence and recovery

BPM Events maintains at any time a persistent image of its internal state in database. This persistent state is called the Recovery Backup. Whenever BPM Events server stops, either manually (for example, through Business Process Portal) or by an accident (for example, server crash), the Recovery Backup contains the data corresponding to the server internal state, as it was after the processing of the last external event. When BPM Events is restarted, the server automatically reads the current Recovery Backup and reinitializes its memory state accordingly. It is then ready to process the event next to the last processed event, as this event was also persistent in the Business Process Server event queue. The main elements of the Recovery Backup are:

- Rules,
- Infopads,
- Event cache, and
- Event scheduler.

### Infopad persistence

Infopads are automatically persisted, and their database image is automatically synchronized with their memory image at the end of each external event processing. The persistent infopads are stored in the Business Process Server Infopad Repository in the database. This repository serves three purposes:

- Failure recovery.
- Real-time access from front-end user interfaces, by business users.
- Input of the infopad archiving utility.
When the in-memory processing of an external event EVT is completed successfully by the rule engine (remember that this may have involved possibly several rules, and possibly several internally generated events), then several infopads may have been modified as a result. These modifications are gathered and notified to the Business Process Server Infopad Repository though JDBC calls, as a single transaction, called the synchronization transaction. Only when this transaction succeeds, is the external event EVT considered as successfully processed. Only then is the next external event processed.

Database access failure handling

The synchronization transaction may fail for various reasons:

- The BPM Events database server is down or the database connection is broken. In that case, BPM Events server retries periodically, until the connection problem is fixed or until it is shut down.

BPM Events provides two configuration parameters that handle database access failure:

- `bpmevents.server.eventsource.retryinterval`. This parameter specifies the retry interval that BPM Events uses to check whether the database is up. The default value is 30,000 milliseconds.

- `bpmevents.server.eventsource.maxdbretry`. This parameter specifies the number of times BPM Events tries to establish a connection to the database. By default, the value of this parameter is -1; that is, BPM Events tries to reestablish the connection continually.

- The JDBC call that is generated is incompatible with the database schema. For example, the table that stores an infopad in the Infopad Repository is deleted and the synchronization transaction attempts to update this infopad image. In that case, an error is logged, and BPM Events server is suspended. The fix is likely to require human intervention, either on the database side or on the rule engine side. BPM Events then requires manual shutdown and restarting.

Keeping the recovery backup up-to-date

The operation of maintaining up-to-date elements of the Recovery Backup is called synchronization.

Event cache synchronization

Modifications to the cache are also synchronized with the cache image in the database, within the synchronization transaction. Modifications include the following additions:

- A new external event added to the cache (one for each synchronization transaction).

- Possibly several internally generated events.

Removal of events in cache (discard statement) are also synchronized with the database by the synchronization transaction. The persistent image of the event cache actually does not duplicate the events objects that are already present in the event queue in database: it only contains their references. XML messages that are cached include the actual body of the XML, not a reference.

Rules synchronization

A persistent image of all the rules (for all applications) that are currently loaded in the rule engine is maintained in the database for recovery purpose. Rules are synchronized through a specific transaction, independent from the synchronization transaction, as changes in the rule set-up is usually caused by external agents (for example, a BPM Events Admin command) and is infrequent. The rule changes that are notified to the database are:
• Loading an application or a rule module.
• Unloading an application or a rule module.
• Enabling a rule group or a rule.
• Disabling a rule group or a rule.

Infopad synchronization

Infopads are synchronized, as previously described, by the synchronization transaction. Only the affected cells of an updated infopad are part of the transaction, so no unnecessary overhead is entailed by the synchronization.

Scheduler synchronization

The persistent image of the scheduler in the database updates as a separate transaction, in the following cases:

• Each time an event is scheduled (added to the scheduler).
• Each time a scheduled event is fired (removed from the scheduler).

Event counter synchronization and management

When an external event is successfully processed, the BPM Events Event Counter is also updated in the database. This counter marks in the Business Process Server event queue the last event BPM Events has completely and successfully processed. It is similar to the notion of "cursor" to iterate over tables in databases. The update of the BPM Events Event Counter is part of the synchronization transaction.

Loading and recovering applications

BPM Events automatically keeps a database image of its internal state for recovery purposes. When starting BPM Events, it automatically reloads the latest recovery backup (if any), and processes the event backlog generated in the database event queue while BPM Events was down. This procedure ensures no data is lost because of a shutdown, whether voluntary (operator) or not (server crash).

Do not confuse starting BPM Events (which automatically performs recovery of current applications) with Loading (or Reloading) an application.

• **Starting BPM Events.** Automatically triggers the recovery mechanism for all applications that were loaded during the latest shutdown. The internal state is recovered, as it was at the end of the processing of the latest event before shutdown. If a `reinitialize()` section was specified, then it is executed immediately after recovery. This allows for some initial processing required in the case of recovery (for example, initial set-up of external systems). The event processing proceeds, as if no interruption occurred, with the current internal state. If no application was running before, or if a new application requires starting, then load this application after BPM Events is started.

• **Loading an application.** Involves loading rules using a procedure quite different from recovery. When you start BPM Events, if some applications were recovered, then do not load them again (in fact, you cannot, unless you unload it first). When loading a rule module, the `initialize()` section is automatically executed. It contains initialization statements that create and initialize infopads. This does not happen when recovering an application or a module (in that case, the `reinitialize()` section is executed).

Similarly, stopping BPM Events does not unload the application.
• **Stopping BPM Events.** BPM Events automatically shuts down so that it can restart and resume the processing of events as if there was no interruption. The latest state of all loaded applications is saved to the database, so that after recovery these applications automatically resume without any intervention from the user. Therefore, always consider these applications as loaded, whether the server is stopped or active.

• **Unloading an application.** An unloaded rule module automatically executes its `finalize()` section, and then its execution state is removed from the server. The statements in the `finalize()` section typically deletes application objects (infopads) from both the memory and the database. If you unload an application, then you remove all trace of this application in the rule engine, including in the recovery backup in database. When restarting BPM Events, this application is not recovered. If you reload the application, then it is similar to a new installation; the application state and objects are reinitialized.

---

**BPM Events configuration parameters**

The BPM Events configuration is specified in two files:

- `bpmevents.conf`: This configuration file contains configuration parameters specific to BPM Events.

- `oebps.conf`: This configuration file contains general configuration parameters that are shared across Business Process Server components, or configuration parameters about a specific component (here BPM Events) which is known also from administration tools.

The configuration file (`bpmevents.conf`) is modified through Business Process Portal. The Business Process Server installer automatically sets up some parameters that are installation dependent, like host names. **Table 15: BPM Events configuration parameters** on page 157 describes the configuration parameters found in the `bpmevents.conf` file. The default values are adjustable.

---

**Caution:** Be careful—making adjustments to some of the following values may cause BPM Events to perform in an unstable manner.

---

**Table 15: BPM Events configuration parameters**

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bpmevents.engine.debug</code></td>
<td>If set to &quot;on&quot;, then the debug (trace) information is written to the log file.</td>
<td>off</td>
</tr>
<tr>
<td><code>bpmevents.engine.trace.dbaccess</code></td>
<td>If set to &quot;on&quot;, then the database access information is written to the log file.</td>
<td>off</td>
</tr>
<tr>
<td><code>bpmevents.engine.trace.event</code></td>
<td>If set to &quot;on&quot;, it then generates an entry for each event that is processed in <code>bpmevents.log</code>.</td>
<td>off</td>
</tr>
<tr>
<td><code>bpmevents.engine.trace.filter</code></td>
<td>If set to &quot;on&quot;, then it writes the event filter used by BPM Events to fetch events from database to log file</td>
<td>off</td>
</tr>
<tr>
<td>Parameter name</td>
<td>Description</td>
<td>Default value</td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>bpmevents.engine.eventcachesize</td>
<td>Maximum number of events BPM Events can store in the cache. The cache is partially flushed when this limit is reached. A small cache size increases swapping in/out to database. Format: integer</td>
<td>10000</td>
</tr>
<tr>
<td>bpmevents.engine.emailer.senderaddress</td>
<td>The default sender e-mail address for BPM Events rules, when they use the action sendMail(). Format: e-mail</td>
<td>set by installer</td>
</tr>
<tr>
<td>bpmevents.engine.bizlogic.api.retry</td>
<td>The number of times BPM Events should retry to call BP Server API, when the connection fails. If set to &quot;0&quot;, then it does not try. Format: integer</td>
<td>5</td>
</tr>
<tr>
<td>bpmevents.engine.bizlogic.api.sleeptime</td>
<td>The time interval (milliseconds) that BPM Events should wait before retrying to call BP Server API, in case the connection failed. Format: integer</td>
<td>10000</td>
</tr>
<tr>
<td>bpmevents.engine.bizlogic.api.innewtransaction</td>
<td>If set to &quot;on&quot;, then it executes BP Server API in a new transaction otherwise executes in the BPM Events transaction</td>
<td>off</td>
</tr>
<tr>
<td>bpmevents.engine.ptnamecheckmode</td>
<td>Check whether or not PTName is defined in DB. useful for debugging</td>
<td>on</td>
</tr>
<tr>
<td>bpmevents.server.eventfiltermode</td>
<td>It generates event filter for event fetch optimization.</td>
<td>on</td>
</tr>
<tr>
<td>bpmevents.server.trace.xml</td>
<td>If set to &quot;on&quot;, then it writes all the XML events received on jms/BPEventQueue to log file.</td>
<td>off</td>
</tr>
<tr>
<td>bpmevents.server.autoerrorrecovery</td>
<td>If set to &quot;off&quot;, then the event reader is suspended when an internal error is caught. The server does not plan any new events.</td>
<td>on</td>
</tr>
<tr>
<td>bpmevents.server.eventsource.eventbatchsize</td>
<td>Maximum size of each batch of events fetched from the database. Format: integer</td>
<td>100</td>
</tr>
<tr>
<td>bpmevents.server.eventsource.sleeptime</td>
<td>Period in (milliseconds) between fetching two batches of events when there are no events in BizEvent table.</td>
<td>1000</td>
</tr>
<tr>
<td>bpmevents.server.eventsource.sleeptimewhenevents</td>
<td>Period in (milliseconds) between fetching two batches of events when there are still events in BizEvent table.</td>
<td>0</td>
</tr>
<tr>
<td>Parameter name</td>
<td>Description</td>
<td>Default value</td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>bpmevents.server.eventsource.retryinterval</td>
<td>Period in milliseconds between retrying the database connection.</td>
<td>30000</td>
</tr>
<tr>
<td>bpmevents.server.eventsource.maxdbretry</td>
<td>The maximum number of times BPM Events should retry to connect to database, when the connection fails. By default, the value is -1, which specifies that BPM Events continue trying until connection is reestablished.</td>
<td>-1</td>
</tr>
<tr>
<td>bpmevents.server.eventsource.maxtotalpreparedstatements</td>
<td>Maximum number of opened prepared statements. Format: integer</td>
<td>100</td>
</tr>
</tbody>
</table>
| bpmevents.server.eventsource.maxgapretry          | The maximum retry number when event gap found. the total amount of retry time is  
                      | bpmevents.server.eventsource.sleeptime *                                | 60            |
|                                                   | bpmevents.server.eventsource.maxgapretry                                   |               |
|                                                   | Format: integer                                                             |               |
| bpmevents.server.callback                          | Any change in BPM Events states notified by invoking registered callback adapters. | false         |
| bpmevents.server.jms                               | If set to "on", then it can receive events from the JMS server.            | on            |
| bpmevents.server.jms.adapter                       | Set the adapter for JMS message processing.                                 | dom           |
| bpmevents.server.jms.topic.admin                   | The topic used in monitoring and starting EventReader.                     | BPAdminTopic  |
| bpmevents.server.jms.queue.integration             | The queue used by BPM Events to listen to external messages.               | BPEventQueue  |
| bpmevents.server.jms.wait.interval                 | Time in seconds a JMS Message sender waits for replies from MonitorBeans or EventReaderBean. | 120           |

There are some more parameters that are relevant to BPM Events—and also to other components—but are set at a more general level in oebps.conf. They can also be modified through Business Process Portal.
Using the purchase order rule wizard

The Purchase Order rule wizard is developed using the rule wizard framework of Business Process Server. This rule wizard generates rules that enable you to monitor and analyze incoming purchase orders. For example, you can use infopads to capture relevant purchase order data, trigger alerts based on the total price of a purchase order, calculate the average price of a purchase order, or periodically send summary reports to executives. In the following sections, different options of the Purchase Order template, such as triggering e-mails based on the total price limit of a purchase order, displaying price averages of purchase orders, and calculating the number of purchase orders per each state are reviewed.

For more information on creating new Rule Wizards using the Rule Wizard framework, refer to chapter "Using the rule wizard framework" in the Application Developer’s Guide.

For more information on rule wizards that are available in Business Process Server, see the chapter "Defining Rules with Rule Wizards" in the OpenEdge Getting Started: Developing BPM Applications with Developer Studio.

For details, see the following topics:

- Applying the purchase order template
- Using the purchase order template

Applying the purchase order template

After applying the purchase order template, you compile it, using the rule compiler, and load, using BPM Events Admin, to complete preparation of the template. Refer to Administering and operating BPM Events on page 135 for instructions on compiling and loading rules. The template is then ready to use.

To apply the Purchaser Order rules, complete the following steps:
1. Open the template by running the following commands:

   rulewizard -t \%bpmevents_home%\templates\PurchaseOrder.rtu

   The Rule Wizard for the Purchase Order template is launched, displaying the Description page, as shown in the following figure.

   **Figure 4: Purchase order template, description page**

   ![Purchase Order Description Page](image)

   This rule template can be used to monitor and analyze incoming purchase orders. InfoPads capture relevant purchase order data, alerts are triggered based on the total price of a purchase order, average price of a purchase order is calculated, summary reports are sent periodically to executives.

   2. Read through the description and click **Next** to continue. Click **Cancel**, at any time in the Rule Wizard, to exit the Wizard. Click **Help**, at any time in the Rule Wizard, to open a relevant Help file.

   3. In the Application name page, enter a name for the purchase order program in the field, as shown in the following figure. Click **Next** to continue.

   **Figure 5: Purchase order template, application name page**

   ![Application Name Page](image)

   4. Answer the questions according to your requirements shown in Figure 5: Purchase order template, application name page on page 162 regarding:
• Keeping track of the number of purchase orders issued each month for each state.
• Calculating the number of purchase orders by state based on the shipping or billing address.
• Select both options above, in which case two reports are generated—one for the billing state and one for the shipping state.

Click **Next** to continue.

**Figure 6: Purchase order template, answering the questions**

5. Answer the next question in the next page regarding knowing the average price of a purchase order, as shown in the following figure. Click **Next** to continue.

**Figure 7: Purchase order template, answering the questions**

6. The next page (see **Figure 7: Purchase order template, answering the questions** on page 163) enables you to trigger an e-mail to inform you of the total price of a purchase order if it reaches a certain limit. Click **Next** to continue.
**Important**: In the **Price Limit** field, input the numbers to include the decimal point. For example, for a purchase order limit of $10,000, enter 10000.00.

**Figure 8: Purchase order template, triggering an e-mail**

7. The next page asks you whether you would like a monthly summary report sent via e-mail of purchase order activity, as shown in the following figure. Click **Next** to continue.

**Figure 9: Purchase order template, triggering a monthly report e-mail**
8. The **Summary** page (see the following figure) presents a summary of the purchase order rules that monitor the process. Read through the text to ensure that the rules are correct. Click **Back** to return to the previous pages if you need to modify the rules. Click **Next** if the rules are acceptable.

**Figure 10: Purchase order template, summary page**

![Summary page](image)

9. The **Save Rule File As** page displays the name of the rule file you created. Click **Browse** (…) to browse through files to save the rule file to a different location.

**Figure 11: Purchase order template, save rule file as page**

![Save rule file as](image)

10. Click **Finish** if the file name and path is correct.

The rule file is now generated.
Using the purchase order template

Once the purchase order template is configured, compiled and loaded in the rule engine, you are ready to receive purchase orders. For example, according to the previous example, the limit placed on the purchase order is $5,000.00. When a new purchase order comes in with a total price higher than $5,000, then an e-mail is generated as shown in Figure 12: Purchase order limit e-mail on page 166.

Figure 12: Purchase order limit e-mail

Subject: Purchase Order Limit Reached
Date: Mon, 04 Feb 2002 16:50:37 -0800
From: Robe <robe@apav.com>
To: robe@apav.com

A new PO for a total of $5529.00 was just issued.

You may also arrange to have monthly summary reports (as shown in Figure 13: Monthly summary report e-mail on page 166) of the purchase order activity sent by e-mail.

Figure 13: Monthly summary report e-mail

Subject: Monthly Purchase Order Activity Report
Date: Mon, 25 Jan 2002 15:50:35 -0800 [PST]
From: POmanager@apav.com
To: mary@apav.com

Jan average total price of a purchase order is $ 122.15

Jan Shipping Statistics : number of purchase orders for each state
AK -- 18   AZ -- 17   CA -- 1

Jan Billing Statistics : number of purchase orders for each state
AK -- 18   AZ -- 17   CA -- 1
Rule syntax

This appendix lists the reserved keywords of the rule language and explains the rule module syntax.

For details, see the following topics:

- Language tokens (Reserved keywords)
- Rule module syntax

Language tokens (Reserved keywords)

```
// token
//
| <YACTIVATED: "activated">  
| <YALL: "all"> 
| <YALLCASE: "ALL_CASE"> 
| <YAND: "and"> 
| <YAPPLICATION: "application"> 
| <YARRAY: "array"> 
| <YBOOLEAN: "boolean"> 
| <YBREAK: "break"> 
| <BY: "by"> 
| <YCELL: "cell"> 
| <YCHAR: "char"> 
| <YCONTINUE: "continue"> 
| <YCORRELATED: "correlated"> 
| <YDESC: "desc"> 
| <YDISCARD: "discard"> 
| <YDISTINCT: "distinct"> 
| <YDO: "do"> 
| <YDOUBLE: "double"> 
```
Chapter 12: Rule syntax

| <YELSE: "else"> |
| <YEND: "end"> |
| <YENUM: "enum"> |
| <YEVENT: "event"> |
| <YEXCEPT: "except"> |
| <YEXISTS: "exists"> |
| <YFALSE: "false"> |
| <YFINALIZE: "finalize"> |
| <YFIRSTSUCCESS: "FIRST_SUCCESS"> |
| <YFLOAT: "float"> |
| <YFN: "fn"> |
| <YFOR: "for"> |
| <YFROM: "from"> |
| <YFUN: "fun"> |
| <YGROUP: "group"> |
| <YIF: "if"> |
| <YIMPORT: "import"> |
| <YIN: "in"> |
| <YINSTANCE: "instance"> |
| <YINDEX: "INDEX"> |
| <YINFOPAD: "infopad"> |
| <YINITIALIZE: "initialize"> |
| <YINT: "int"> |
| <YINTERSECT: "intersect"> |
| <YITERATOR: "iterator"> |
| <YLET: "let"> |
| <YLETREC: "letrec"> |
| <YLIKE: "like"> |
| <YLIST: "list"> |
| <YMAP: "map"> |
| <YMODULE: "module"> |
| <YNEXT: "new"> |
| <YNIL: "nil"> |
| <NOT: "not"> |
| <OBJECT: "object"> |
| <OF: "of"> |
| <OR: "or"> |
| <YPERSISTENT: "persistent"> |
| <YPRIVATE: "private"> |
| <YPUBLIC: "public"> |
| <YRETURN: "return"> |
| <YRULE: "rule"> |
| <YSELECT: "select"> |
| <YSET: "set"> |
| <YSINGLECASE: "SINGLE_CASE"> |
| <YSOME: "some"> |
| <YSTRING: "string"> |
| <YTERMINATED: "terminated"> |
| <YTHEN: "then"> |
| <YTRANSIENT: "transient"> |
| <YTRIGGER: "trigger"> |
| <YTRUE: "true"> |
| <YTRY: "try"> |
| <YTYPE: "type"> |
| <YUNION: "union"> |
| <YUNION: "unit"> |
| <VVAL: "val"> |
| <VVAR: "var"> |
| <VOID: "void"> |
| <WHERE: "where"> |
| <WHILE: "while"> |
| <WITH: "with"> |
| <GENERATE: "generate"> |
| <SCHEDULE: "schedule"> |
| <PERFORM: "perform"> |
| <UNSCHEDULE: "unschedule"> |
| <ENDOFKEYWORD: "__end_of_keyword__"> // dummy
| <YEAR: "Y"|"YEA"|"YEAR"> |
| <MONTH: "M"|"MON"|"MONTH"> |
| <DAY: "D"|"DAY"> |
| <HOUR: "H"|"HOU"|"HOUR">
Rule module syntax

```plaintext
translation_unit ::= 
  [ "application" <ID> ] 
  [ "module" [ <ID> ] [ "import" module_name { "," module_name }* ";" ] ] 
  ( declaration )* 
  ( rule_group )* 
  [ ["initialize" ] {" ( statement )* "} ] 
  [ "finalize" "{ ( statement )* "} ] 
<EOF>
module_name ::= <ID> [ ":" <ID> ] //
// rule definition
// rule_group ::= "group" <ID> "(" ( rule )* ")"
rule ::= "rule" <ID> "activated" "by"
  [ rule_param "," rule_param )* ] 
  ( "correlated" "with" "INDEX"
  |"terminated" "with" "FIRST_SUCCESS"
  |"try" ("SINGLE_CASE" | "ALL_CASE"
  )* 
  [ "if" expr ] "then"
  "{" ( statement )* "}"
rule_param ::= ["*"] <ID> "of" [ <ID>] ":[" <ID>] //
// declaration
// declaration ::= function_defintion 
  | value_defintion 
  | var_defintion 
  | type_defintion 
  | enum_defintion
enum_defintion ::= "enum" <ID> "{ <ID> { "," <ID> }* "}"
function_defintion ::= "fun" function_def ( "and" function_def )* 
function_def ::= <ID> fun_args [":" type] compound_statement 
fun_args ::= ( "" [ <ID> [":" type { "," <ID> [":" type] }* ] ")" )+
value_defintion ::= "val" <ID> [":" type] = expr ; 
var_defintion ::= "var" <ID> [":" type] = expr ; 
//
// type definition
// type_defintion ::= "type" <ID> = type ";

//
// type
// type ::= fun_type 
fun_type ::= prod_type [ "->" fun_type ]
prod_type ::= p_type [ "*" p_type ]
p_type ::= primitive_type 
  | id_type 
  | struct_type 
  | object_type 
  | array_type 
  | collection_type 
  | map_type 
  | instance_type 
  | iterator_type 
  | type_var 
  | (" type ")
primitive_type ::= "void" 
  | "int" 
  | "double" 
  | "float" 
  | "char" 
  | "string" 
  | "boolean" 
  | "unit"
id_type ::= qualified_name 
struct_type ::= (" <ID> ":" p_type { "," <ID> ":" p_type }* ")
object_type ::= ("object" | "cell") struct_type
```
array_type ::= ("array" | "infopad") [ "<" type ">"] ("[" [longLiteral | id_type "]"]")+
collection_type ::= ("set" | "list") [ "<" type ">"]
map_type ::= "map" [ "<" type "," type ">"]
instance_type ::= "instance" [ "<" type ">"]
iterator_type ::= "iterator" [ "<" type ">"]
type_var ::= "" (ID | longLiteral)

// statement
//
statement ::= iterate_statement
| declaration_statement
| compound_statement
| expression_statement
| selection_statement
| jump_statement
| event_statement

declaration_statement ::= declaration
expression_statement ::= [ expr ] ";"
selection_statement ::= "if" "(" expr ")" statement [ "else" statement ]
compound_statement ::= "{" ( statement )* "}"
iterate_statement ::= "for" "(" { var_defintion | ";" } [ expr ] ";" [ expr ] ")" statement

| "while" "(" expr ")" ";"
| "do" statement

| "while" "(" expr ")" ";"

| [ fromClause ] [ whereClause ] ";"

jump_statement ::= "break" ";"

| "continue" ";"

| "return" [ expr ] ";"

event_statement ::= "generate" "(" [ event_ctor_body ] ")"
| "perform" "(" [ "adapter" ':' "STRING" "," ] <STRING> "," (event_ctor_body | expr "{"," expr }* ) ")"
| "schedule" "(" expr ("," expr )* )"
| "unschedule" "(" expr ("," expr )* )"
| "discard" "(" expr ("," expr )* )"

event_ctor_body ::= event_actionval ("," event_actionval )*
event_actionval ::= event_actionval_attr ':' expr
event_actionval_attr ::= event_prop_name ( "." event_prop_name )*
event_prop_name ::= <ID>
| "type"
| "event"

// expression

expr ::= assigmentExpr
assigmentExpr ::= orExpr
[ (":" | "+=" | "-=" | "+=" | "/=" | "/=" ) assigmentExpr
| ":=" <ID> fun_arguments ]
orExpr ::= andExpr ( "or" andExpr )*
andExpr ::= quantifierExpr ( "and" quantifierExpr )*
quantifierExpr ::= "for" "all" <ID> "in" expr ";" equalityExpr
| "exists" <ID> "in" expr ";" equalityExpr
| "fn" fun_args [":" type]
(compound_statement
| "=" expr)
equalityExpr ::= relationalExpr ( "=" | "!=" ) [ "some" | "all"] relationalExpr )*
relationalExpr ::= additiveExpr
("<" | ">=" | ">=" | ">=" ) [ "some" | "all"] additiveExpr
| "like" additiveExpr )*
additiveExpr ::= multicativeExpr ( "+" | "-" | "union" | "except") multicativeExpr )*
multicativeExpr ::= inExpr ( "+=" | "+=" | "union" | "except") inExpr )*
inExpr ::= unaryExpr ( "+=" unaryExpr ) ( "+" unaryExpr ";" unaryExpr )*
unaryExpr ::= "+=" | "+=" | "not" ) unaryExpr
| ( "+" | "+=" ) unaryExpr
Chapter 12: Rule syntax

| ("*" ) unaryExpr |
| postfixExpr |
postfixExpr ::= primaryExpr |
| ( |
"." <ID> |
| ("++"|"--") |
| fun_arguments |
| ("[" expr "]") |
| )* |
primaryExpr ::= persistent_ctor |
| new_expr |
| identifier |
| ("let" | "letrec") declaration "in" expr (";" expr )* [";"] "end" |
| ("(" )" |
| (" expr (";" expr )* ")" |
| (" expr (";" expr )+ ")" |
| struct_ctor |
| selectExpr |
| enum_length_expr |
| date_ctor |
| timeunit_ctor |
| literal |
// query exp |
selectExpr ::= "select" ["distinct"] (fieldList | expr) fromClause [ whereClause ] "end" |
fromClause ::= "from" iteratorDef ("," iteratorDef )* |
iteratorDef ::= <ID> (":" expr ) |
whereClause ::= "where" expr |
// struct |
struct_ctor ::= "{" fieldList "}"
fieldList ::= field ("," field )* |
field ::= <ID> (":" expr |
// aggregate exp |
persistent_ctor ::= array_ctor |
| collection_ctor |
| object_ctor |
| map_ctor |
array_ctor ::= array_type |
( |
array_init |
| fun_arguments |
) |
array_init ::= "{" [ array_init_value ("," array_init_value )* ] "}"
array_init_value ::= array_init |
| expr |
collection_ctor ::= collection_type ["{" [ valueList ] "}"] |
object_ctor ::= (id_type | "object" | "cell") ["{" fieldList "}"] |
map_ctor ::= map_type ["{" [ elemMap ("," elemMap )* ] "}"] |
elemMap ::= expr (":" expr ) |
// new exp |
new_expr ::= "new" ["transient" | "persistent"] [ fun_arguments ] persistent_ctor |
fund_arguments ::= "{" [ valueList ] "}" |
valueList ::= expr ("," expr )* |
enum_length_expr ::= 
// date exp |
date_ctor ::= date_unit ("/") date_unit |
| <NOW> |
date_unit ::= ( <YEAR> | <MONTH> | <DAY> | <HOUR> | <MIN> | <SEC> ) "::" primaryExpr |
timeunit_ctor ::= ( <YEAR> | <MONTH> | <DAY> | <HOUR> | <MIN> | <SEC> ) |
// literal |
identifier ::= qualified_name |
qualified_name ::= <ID> ( ":=" <ID> ) |
literal ::= <CHARACTER> |
| "true" |
| "false" |
| "nil" |
longLiteral |
| realLiteral |
longLiteral ::= <DECIMALINT>
realLiteral ::= <FLOATONE> | <FLOATTWO>
In the following text, the symbol ‘x’ is a generic symbol used to denote any expression type, either atomic (for example, int, double) or composed (for example, a structure type like {a:int, b:string }).

The symbols: // are followed by comments.

!= // different
  : (('x'*'x')->boolean)
  : ((int*double)->boolean)
  : ((double*int)->boolean)
%
  // modulo
  : ((int*int)->int)
  : ((double*double)->double)
  : ((int*double)->double)
  : ((double*int)->double)
*
  // multiply
  : ((int*int)->int)
  : ((double*double)->double)
  : ((int*double)->double)
  : ((double*int)->double)
*
  // time expressions
  : ((time*int)->time)
  : ((int*time)->time)
  : ((btime*int)->btime)
  : ((int*btime)->btime)
+
  // addition
  : ((int*int)->int)
  : ((double*double)->double)
  : ((int*double)->double)
  : ((double*int)->double)
+
  // concatenation
  : ((string*string)->string)
  : ((string*any)->string)
  : ((any*string)->string)
+
  // union
  : ((set<'x>*set<'x>)->set<'x>)
  : ((set<'x>*list<'x>)->set<'x>)
  : ((list<'x>*set<'x>)->list<'x>)
Chapter 13: Predefined functions and operators

: ((list<'x>*list<'x>)->list<'x>)
+ // date forward shift
  : ((time*date)->date)
  : ((date*time)->date)
  : ((btime*date)->date)
  : ((date*btime)->date)
+ // time addition
  : ((time*time)->time)
  : ((btime*btime)->btime)
- // minus
  : ((int*int)->int)
  : ((double*double)->double)
  : ((int*double)->double)
  : ((double*int)->double)
- // unary minus
  : (int->int)
  : (double->double)
- // date backward shift
  : ((date*time)->date)
  : ((date*btime)->date)
- // time subtraction
  : ((time*time)->time)
  : ((btime*btime)->btime)
/ // division
  : ((int*int)->int)
  : ((double*double)->double)
  : ((int*double)->double)
  : ((double*int)->double)
/ // time division
  : ((time*int)->time)
  : ((int*time)->time)
  : ((btime*int)->btime)
  : ((int*btime)->btime)
< // less than
  : ((int*int)->boolean)
  : ((double*double)->boolean)
  : ((int*double)->boolean)
  : ((double*int)->boolean)
< // earlier than
  : ((date*date)->boolean)
< // duration less than
  : ((time*time)->boolean)
  : ((btime*btime)->boolean)
< // belongs to
  : (\'x*set<\'x>)->boolean)
  : (\'x*list<\'x>)->boolean)
<= // less than or equal to
  : ((int*int)->boolean)
  : ((double*double)->boolean)
  : ((int*double)->boolean)
  : ((double*int)->boolean)
<= // earlier or at same date as
  : ((date*date)->boolean)
<= // duration less than or equal to
  : ((time*time)->boolean)
  : ((btime*btime)->boolean)
= // equal to
  : (\'x*'x)->boolean)
  : ((int*double)->boolean)
  : ((double*int)->boolean)
> // greater than
  : ((int*int)->boolean)
  : ((double*double)->boolean)
  : ((int*double)->boolean)
  : ((double*int)->boolean)
> // later than
  : ((date*date)->boolean)
> // duration greater than
  : ((time*time)->boolean)
  : ((btime*btime)->boolean)
>= // greater or equal to
: ((int*int)->boolean)
: ((double*double)->boolean)
: ((int*double)->boolean)
: ((double*int)->boolean)

>= // later than or same date as
: ((date*date)->boolean)

>= // duration greater than or equal to
: ((time*time)->boolean)
: ((btime*btime)->boolean)

abs // absolute part of
: (int->int)
: (double->double)

add // add element to set/list. return false if already in set/list
: ((set<'x>*'x)->boolean)
: ((list<'x>*'x)->boolean)

add
: (array<'x>[]->int)
: (array<'x>[][]->int)
: ((array<'x>[]*'<x>)->int)
: ((array<'x>[][]*array<'x>[][])->int)

addAt // insert element at
: ((list<'x>*int*'x)->void)

addColLabels // add labels to each column of an infopad. First is column title.
: ((array<'x>[][]*string*list<string>)->void)

addColTitle // add title to columns.
: ((array<'x>[][]*string)->void)

addLabels
: ((array<'x>[][]*string*list<string>)->void)
: ((array<'x>[][]*string*list<string>)->void)

addRowLabels // add labels to each row of an infopad. First is row title.
: ((array<'x>[][]*string*list<string>)->void)

addRowTitle // add title to rows.
: ((array<'x>[][]*string)->void)

addTitle
: ((array<'x>[]*string)->void)
: ((array<'x>[]*string)->void)

avg // re-calculate an average value, given a new value and its weight
: ((int*int*int*int)->int)
: ((double*double*int*double)->double)
: ((double*double*int*int)->double)
: ((double*int*int*double)->double)
: ((double*int*int*int)->double)

check // associates an alarm check with an infopad
: ((array<'x>[]*string*string*string*int*string)->void)
: ((array<'x>[]*string*string*string*string*string)->void)
: ((array<'x>[][]*string*string*string*string*string)->void)
: ((array<'x>[][]*string*string*string*string*string)->void)

close // close a file
: (output->void)

colCount // return the number of columns in an infopad
: (array<'x>[][][]->int)

colKeySet // return the list of column labels
: (array<'x>[][][]->list<string>)

day // extract the day (number) from a date
: (date->int)
: (time->int)
: (btime->int)

disableRule // disable a rule currently loaded
: ((string*string*string)->boolean)

discardAllIndexEntry // discard all events of a process and subprocess in the cache index
: (string->void)

discardIndexEntry // discard all events of a process in the cache index
: (string->void)

dueDate // calculate the due date based on the specified calendar
: ((btime*date)->date)
: ((btime*date*string)->date)

duration // elapsed time between two dates
: ((time*date*date)->int)
: ((btime*date*date)->int)
: ((btime*date*date*string)->int)
Chapter 13: Predefined functions and operators

elementAt // return Nth element of a list
: ((list<'x>*int)->'x)
elements // iterator for list elements
: (set<'x>->iterator<'x>)
: (list<'x>->iterator<'x>)
enableRule // enable a rule currently loaded
: ((string*string*string)->boolean)
except // set difference
: ((set<'x>*set<'x>)->set<'x>)
findColIndex // return index given the column label, if label is not found, then return -1
: ((array<'x>[][]*string)->int)
findIndex
: ((array<'x>[][]*string)->int)
: ((array<'x>[][]*string)->int)
findRowIndex // return index given the row label, if label is not found, then return -1
: ((array<'x>[][]*string)->int)
first // first elt of a list
: (list<'x>->'x)
flush // flush pending outputs to a file
: (output->void)
getColIndex // return index given the column label
: ((array<'x>[][]*string)->int)
getColLabel // return label given the column index
: ((array<'x>[][]*int)->string)
getDateProperty // return a field of a date
: ((array<'x>[][]*int)->string)
getEventDate // return date of an event
: ((EVENT->date)->int)
getEventDateProperty // return a field of a date of an event
: ((EVENT*string)->int)
getEventProperty // return a field of an event
: ((EVENT*string)->string)
getIndex
: ((array<'x>[][]*string)->int)
: ((array<'x>[][]*string)->int)
getInfopadProperty
: ((INFOPAD*string)->any)
getLabel
: ((array<'x>[][]*int)->int)
: ((array<'x>[][]*int)->int)
getProcessTemplateId // given the process template name, return its id
: ((array<'x>[][]*int)->int)
getRowIndex // return index given the row label
: ((array<'x>[][]*string)->int)
getRowLabel // return label given the row index
: ((array<'x>[][]*int)->string)
hasNext
: (iterator<'x>->boolean)
hour // extract the hour field (number) from a date
: (date->int)
: (time->int)
: (btime->int)
index // return element at index
: ((list<'x>*int)->'x)
: ((string*int)->char)
intersect // set intersection
: ((set<'x>*set<'x>)->set<'x>)
last // last elt of a list
: (list<'x>->'x)
like
: ((string*string)->boolean)
max // max of two numbers
: ((int*int)->int)
: ((double*double)->double)
: ((int*double)->double)
: ((double*int)->double)
min // min of two numbers
: ((int*int)->int)
: ((double*double)->double)
: ((int*double)->double)
: ((double*int)->double)
minute // extract the minute field (number) from a date
  : (date->int)
  : (time->int)
  : (btime->int)
month // extract the month field (number) from a date
  : (date->int)
  : (time->int)
  : (btime->int)
next
  : (iterator<'x>->'x)
not // negation
  : (boolean->boolean)
openFile // open a file
  : (string->output)
  : (string*boolean->output)
print // print to standard output
  : (boolean->void)
  : (int->void)
  : (double->void)
  : (char->void)
  : (string->void)
print // print to file
  : (output*boolean->void)
  : (output*int->void)
  : (output*double->void)
  : (output*char->void)
  : (output*string->void)
println // print a line to standard output
  : (boolean->void)
  : (int->void)
  : (double->void)
  : (char->void)
  : (string->void)
println // print a line to file
  : (output*boolean->void)
  : (output*int->void)
  : (output*double->void)
  : (output*char->void)
  : (output*string->void)
remove
  : ((set<'x>*'x)->boolean)
  : ((list<'x>*'x)->boolean)
remove
  : ((array<'x>[]*int)->void)
  : ((array<'x>[][]*int)->void)
  : ((array<'x>[][][]*int)->void)
  : ((array<'x>[][][][]*int)->void)
removeAt
  : ((list<'x>*int*'x)->'x)
removeCheck // remove alarm check from infopad
  : (array<'x>[][][]->int)
rowCount // return the number of rows in an infopad
  : (array<'x>[][][]->int)
rowKeySet // return the list of row labels
  : (array<'x>[][][]->list<string>)
second // extract the second field (number) from a date
  : (date->int)
  : (time->int)
  : (btime->int)
sendMail // send a mail
  : (string*string*string->void)
  : (string*string*EVENT->void)
  : (string*string*string*string->void)
  : (string*string*string*EVENT->void)
sellAll // initialize all cells of an infopad to some value
  : (array<'x>[][][]->void)
  : (array<'x>[][][][*'x]->void)
sellAt
  : ((list<'x>*int*'x)->void)
sellColLabel // set one column label of an infopad
setLabel
: ((array<'x>[][*int]*string)->void)
setLabel
: ((array<'x>[][*int]*string)->void)
setLabel
: ((array<'x>[][*int]*string)->void)

setRowLabel // set one row label of an infopad
: ((array<'x>[][*int]*string)->void)

size
: (array<'x>[]->int)
: (array<'x>[][*int]*string)->int)
: (list<'x>->int)
: (set<'x>->int)
: (map<'x, 'y>->int)

toDate // convert to a date
: (string->date)
: (date->date)
: (int->date)

toFloat
: (string->double)
: (int->double)
: (double->double)

toInt
: (string->int)
: (double->int)
: (int->int)
: (date->int)

toString
: (any->string)

union
: ((set<'x>*set<'x>)->set<'x>)

year // extract year field (number) from a date
: (date->int)
: (time->int)
: (btime->int)
Recovery of global variables and file handling

This appendix explains how to recover global variables and their values, and how to reopen recovered files.

For details, see the following topics:

- Global variables and values
- Recovering global variables
- File reopening after recovery

Global variables and values

In a module, global variables and values are intended for sharing by all the rules of the module. Their life span covers the time the rule module is active (loaded) in the engine. Global variables and values are declared immediately after the "module" statement, and before the first rule group. You can initialize them in the same declaration statement. In the following example, the value: file_name is initialized to a file path, and cannot be modified later (declared as "val"). The variable mycounter (declared as "var") is initialized with "1", and can be modified later by any rule.

```java
application BP_test_suites
module test_util
val file_name = "e:\obms\log\test_util.log";
var mycounter = 1;
```

When recovering, the rule engine reloads the module with specific behavior that ensures the initialize{} section is not executed again. Also, all global variables are recreated and reinitialized.
Recovering global variables

The only data objects that are fully recoverable are infopads. It is possible to use an infopad for storing the value of a global variable, so that it will be recovered. In the example below, the variable \( x \) is supposed to be initialized to 0 when the module is initially loaded, but is initialized to some infopad value when recovering. This infopad value holds the value of the variable when BPM Events is shutdown.

```plaintext
application BP_test_suites
module test_util
var persist_x = new infopad<cell{xval:int}>[1][1]("persist_x");
var xval = 0;
fun x_init(){ xval := persist_x[1][1].xval; return xval; }
var x = (is_recovery_mode())? x_init(): 0;
```

The predefined boolean function `is_recovery_mode()` returns `true` if the initialization statement is executed in the context of a recovery, and `false` if executed when loading the module in a normal way. In order to make sure the infopad `persist_x` always stores the latest value of \( x \), it must update each time \( x \) is updated in a rule (an assignment statement at the end of the action part of the rule is sufficient: `persist_x[1][1].xval := x;`).

File reopening after recovery

In the following sample module, the output file is opened appropriately when the global variables are initialized, for example, either in append mode or in rewrite mode.

```plaintext
application BP_test_suites
module test_util
val file_name = "e:\\ obms\\log\\test_util.log";
val out_file: output = openFile(file_name, is_recovery_mode());
fun fprintln(s: string) {
  out_file.println(s);
}group abc {
rule myrule1 activated by ... then {fprintln("rule 1");}
rule myrule2 activated by ... then {out_file.println("rule 2");}
}initialize {
  ...
}finalize {
  if (out_file != nil) out_file.close();
}
```

The function `openFile()` was overloaded to accept a second argument (boolean). If it is equal to `true`, then the function opens the file in append mode. If it is equal to `false` or absent, then the function opens the file in rewrite mode. Here, the predefined function `is_recovery_mode()` decides whether the file should be appended (in the case of recovery) or rewritten (in the case of normal loading).

The following example shows some of a file-writing function, either from inside the module or from outside. It is usually good practice to manage such resources as files, in a single module that is imported by other modules, so that this resource management does not require replicating across modules. Note that in the module below, the global variable `start_time` was not made recovery proof.

```plaintext
application BP_test_suites
module test_util
val file_name = "e:\\ obms\\log\\test_util.log";
val out_file: output = openFile(file_name, is_recovery_mode());
var start_time: date;
fun fprintln(s: string) {
  out_file.println(s);
}fun set_start_time(msg: string) {
  fprintln("\n>> started: " + msg);
  start_time = date("01/01/1970 00:00:00");
}initialize {
  ...
}finalize {
  if (out_file != nil) out_file.close();
}
```
start_time := NOW;
)fun display_consumed_time(msg: string) {
    fprintln(">> "+msg+": consumed time: "+duration(NOW, start_time)/1000);
)initialize {
}finalize {
    if (out_file != nil) out_file.close();
}-----
// from other modules, fprintln can be used as well as other functions:
// (Note that the test_util module needs to be imported by the using modules)
{ test_util::fprintln(">> create_instances<3>");
    test_util::display_consumed_time("Test_CrossCorrel_A<1>");
    test_util::set_start_time("Test_CrossCorrel_A");
}
**Glossary**

**ACL manager**
In Business Process Server, Access Control List Manager provides a finer, more precise control over user access rights for resources and actions.

**Activity workstep**
In Business Process, the basic unit of work; must be performed by one or more human performers (valid individual user, multiple users or user group).

**Adapter**
A Java class that integrates remote, third party classes and actions with Business Process. An adapter can automate certain functions and tasks performed by a remote server or other external systems.

**Administration**
A module in Business Process Portal enabling the administrator to perform tasks such as installing/uninstalling applications, modifying configuration parameters controlling Business Process operations, and manage users, groups and access control. The Administration module is visible only to application users who have permissions to access it.

**Application**
In Business Process, an application is an installed, executable business process that automates a business flow.

**Balanced scorecard**
A management application in the Management module that measures performance by analyzing how an organization’s business activities help it achieve its strategic goals. The Balanced Scorecard provides an analysis from a range of perspectives.

**BAM**
Business Activity Management combines Business process management with strategic and analytical information on specific business performance indicators, providing real-time status information and identifying critical events to assist senior management in making informed business decisions.
BPM Events

A Business Process Server component that provides an open event-driven rule engine to formulate and enforce policies in business applications.

BPM Webflow

A Business Process Server component that enables users to develop customizable, sophisticated presentation flows for business processes, install them as Web applications, and execute them on their Web browsers.

BPEL

BPEL (Business Process Execution Language) for Web services is an XML-based language designed to enable task-sharing for a distributed computing or grid computing environment - including across multiple organizations - using a combination of Web services.

BPMN

BPMN (Business Process Modelling Notation) provides businesses with the capability of defining and understanding their internal and external business procedures through a Business Process Diagram giving organizations the ability to communicate these procedures in a standard manner.

BP Server

A Business Process Server component that provides a flexible, lightweight, scalable workflow process engine for intranets, extranets, and the Internet.

Business calendar

A Business Process Server feature that accurately calculates the Due Date of tasks, and provides support for multiple business calendars across different time zones.

Business flow

The logical sequence of process activities, related to one another by a triggering activity, to achieve an outcome. It represents a business process that begins with a commitment and ends with the termination of that commitment. In Business Process Server, business flow includes Workflow (the flow of all human-performed activities), integration flow (the flow of activities performed by systems) and presentation flow (from a user’s viewpoint, the flow of data from one Web page to the next).

Business logic

The control flow and information flow among worksteps that define a business process.

Business object

A representation of an activity in the business domain, including its name, definition, attributes, behavior, relationships and constraints.

Business process

A process involving multiple worksteps in the form of operations, interactions and notifications performed by a user, group of users, an external adapter, or a script.

Business Process Server application

An application is an implementation of a business process. It can contain one or more process templates, performers, adapters, customized forms or rules. An application can be published, installed and run on BP Servers. In Business Process Server, an application is an installed, executable business process that automates a Workflow.
Business Process Server Web services
A Business Process Server component that allows application developers to: a) publish their applications as Web services, and b) find and convert other available Web services on the Internet into Business Process Server applications.

Business Process Portal
A Business Process Server component that offers users, managers, administrators and developers a unified, customizable portal for single sign-on access to all Business Process Server functionalities to which they are granted permission.

Business process management
The concept of guiding work activities through a multi-step business process in order to improve performance and reduce costs within and across functional business units.

Business Process Modeler
A stand-alone component that enables users to design templates for basic business processes.

Business rule
A combination of elements, including validation edits, logon verifications, database lookups, policies and transformations, that represent an enterprise’s way of doing business.

Control flow
The sequences of worksteps and workstep conditions, as defined in a process template in Progress Developer Studio for OpenEdge or Business Process Modeler.

Dashboard
A Business Process Server feature that provides a graphic overview of the status of several business processes on a single Web page, enabling users to monitor the progress of each process. Users can view business processes across all applications or for a selected application.

Dataslot
A data placeholder that persists through the entire process and defines the information flow of the business process. Dataslots are associated with processes, where they can add information into (Input type) or out of (Output type) worksteps, and appear as editable or read-only fields on a user’s interface.

Expression editor
A Business Process Server tool that enables users to define complex conditional expressions within a Decision gateway to support their business requirements.

Group
In Business Process Server, an entity that has as members valid users or other groups who perform related work and have authorized access to specific components.

Heatmap
A Business Process Server feature that provides a convenient, graphical tool for managers to visually locate the bottlenecks in the process execution. It helps managers to get an overview of the status of the currently active instances, identify suspended instances, and analyze the history of the completed instances.

Home
A module in Business Process Portal through which users interact with Business Process Server. Using the Home module, users complete entries to various tasks and applications, update profile, set preferences, and link to the support infrastructure required to achieve these tasks. The Home module is the primary interface for application users.
Infopad
In Business Process Server, a data structure used to capture business metrics, typically displayed as a table with one or two dimensions.

Instance
An individual object within a specific class. In Business Process Server, a self-contained unit that is created each time you use a process template to run a Business Process Server application.

KPI
Key Performance Indicator, used in the Balanced Scorecard system, that provides the data translating enterprise goals into a set of measurable objectives.

Managed Adapter
In Business Process Server, a Managed Adapter is an implementation of an adapter interface that facilitates data exchange between Business Process Server processes and external applications.

Management
A module in Business Process Portal enabling the managers to query, report, and control processes and resources for application users. The Management module is visible only to application users who have permissions to access it.

Migration
The process of moving from the use of one operating environment to another operating environment that is typically seen as improvement. Migration can involve moving to new hardware, new software, or both. It may involve a new application, another type of database, or a redesigned network. Migration is also used to refer simply to the process of moving data from one storage device to another. Business Process Server supports data migration as well as application migration.

Performer
An entity that executes a workstep. Depending on the workstep type, the performer can be a human user, a group of users, an adapter or other external performer, or a script.

Presentation flow
The flow of information and user input from one interface to the next. Typically related to a single Activity workstep in the process and generated in a BPM Webflow environment.

Process engine
Orchestrates the execution of business processes and also coordinates conversations among process engines based on public processes, which forms the backbone of global business collaboration.

Process refresh
A Business Process Server feature for replacing the installed process without versioning, facilitating the running process instances to refresh and seamlessly adapt to the new Workflow.

Process template
In Business Process Server, a model of business flow that includes worksteps, connectors and dataslots. After users publish and install it as an application in Business Process Server folder structure, they can use the application to create process instances.

Progress Developer Studio for OpenEdge
An Integrated Development Environment for Business Process Server that enables application users to develop and publish a Business Process Server application without leaving the development environment.
Role

The actions and activities assigned to a valid application user who is a member of a group. In Business Process Server, only members of a group can be assigned a role. A role indicates the relationships of the user in a group context.

Rollback

In Business Process Server, a feature that restarts the Workflow from a workstep previously selected as the rollback point in the process, performed automatically in the event of a failure.

Rule wizard

An interactive utility that enables application users to quickly develop rules that can be applied to a business process.

Swim lanes

Used in Workflow diagrams to organize complex processes across functional boundaries. For example, seen as horizontal lines on a process map, swim lanes can be used to place individual task steps into different categories that depend on task ownership.

Task

In Business Process Server, a performer is assigned one or more work items that the performer sees as tasks. There are two types of tasks: Assigned, which are assigned specifically to you; and Available, which are available to be performed by you or other members of your user group.

User

In Business Process Server, a valid human performer with authorized access to specific modules.

Workflow

The logical sequence of activities performed by human performers. Workflow includes the tasks, procedural steps, organizations or people involved, required input and output information, and tools needed for each activity in a business process.