



Developing BPL Processes

Version 2025.1
2025-06-03

Developing BPL Processes

PDF generated on 2025-06-03

InterSystems IRIS® Version 2025.1

Copyright © 2025 InterSystems Corporation

All rights reserved.

InterSystems®, HealthShare Care Community®, HealthShare Unified Care Record®, IntegratedML®, InterSystems Caché®, InterSystems Ensemble®, InterSystems HealthShare®, InterSystems IRIS®, and TrakCare are registered trademarks of InterSystems Corporation. HealthShare® CMS Solution Pack™, HealthShare® Health Connect Cloud™, InterSystems® Data Fabric Studio™, InterSystems IRIS for Health™, InterSystems Supply Chain Orchestrator™, and InterSystems TotalView™ For Asset Management are trademarks of InterSystems Corporation. TrakCare is a registered trademark in Australia and the European Union.

All other brand or product names used herein are trademarks or registered trademarks of their respective companies or organizations.

This document contains trade secret and confidential information which is the property of InterSystems Corporation, One Congress Street, Boston, MA 02114, or its affiliates, and is furnished for the sole purpose of the operation and maintenance of the products of InterSystems Corporation. No part of this publication is to be used for any other purpose, and this publication is not to be reproduced, copied, disclosed, transmitted, stored in a retrieval system or translated into any human or computer language, in any form, by any means, in whole or in part, without the express prior written consent of InterSystems Corporation.

The copying, use and disposition of this document and the software programs described herein is prohibited except to the limited extent set forth in the standard software license agreement(s) of InterSystems Corporation covering such programs and related documentation. InterSystems Corporation makes no representations and warranties concerning such software programs other than those set forth in such standard software license agreement(s). In addition, the liability of InterSystems Corporation for any losses or damages relating to or arising out of the use of such software programs is limited in the manner set forth in such standard software license agreement(s).

THE FOREGOING IS A GENERAL SUMMARY OF THE RESTRICTIONS AND LIMITATIONS IMPOSED BY INTERSYSTEMS CORPORATION ON THE USE OF, AND LIABILITY ARISING FROM, ITS COMPUTER SOFTWARE. FOR COMPLETE INFORMATION REFERENCE SHOULD BE MADE TO THE STANDARD SOFTWARE LICENSE AGREEMENT(S) OF INTERSYSTEMS CORPORATION, COPIES OF WHICH WILL BE MADE AVAILABLE UPON REQUEST.

InterSystems Corporation disclaims responsibility for errors which may appear in this document, and it reserves the right, in its sole discretion and without notice, to make substitutions and modifications in the products and practices described in this document.

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)

Tel: +1-617-621-0700

Tel: +44 (0) 844 854 2917

Email: support@InterSystems.com

Table of Contents

1 Introduction to BPL Processes	1
1.1 BPL Features	1
1.2 A BPL Business Process Can Be Reusable	2
1.3 See Also	2
2 Listing the Business Processes	5
2.1 The Business Process List Page	5
2.2 See Also	5
3 Creating a New BPL Business Process	7
3.1 Using the Business Process Wizard	7
3.2 See Also	7
4 Editing a BPL Business Process	9
4.1 BPL Designer Toolbar	9
4.2 BPL Diagram	10
4.2.1 Editing Tips	11
4.2.2 BPL Diagram Shapes	12
4.2.3 BPL Diagram Connections	14
4.2.4 BPL Diagram Layout	15
4.2.5 Drilling Down into a BPL Diagram	15
4.3 Adding Activities to a BPL Diagram	17
4.3.1 Adding a Call Activity	19
4.4 BPL Designer Property Tabs	19
4.4.1 Setting General Properties of the BPL Business Process	20
4.4.2 Defining the context Object	20
4.4.3 Setting BPL Diagram Preferences	21
4.5 See Also	21
5 Variables Available in BPL (Execution Context)	23
5.1 Introduction	23
5.2 The context Object	23
5.3 The request Object	24
5.4 The response Object	24
5.5 The callrequest Object	24
5.6 The callresponse Object	24
5.7 The syncresponses Collection	25
5.8 The synctimeout Value	25
5.9 The status Value	25
5.10 The process Object	26
6 BPL Syntax Rules	27
6.1 References to Message Properties	27
6.2 Literal Values	27
6.2.1 XML Reserved Characters	28
6.2.2 Separator Characters in Virtual Documents	28
6.2.3 When XML Reserved Characters Are Also Separators	28
6.2.4 Numeric Character Codes	28
6.3 Valid Expressions	29
6.4 Indirection	29

7 List of BPL Elements	31
7.1 Business Process	31
7.2 Execution Context	31
7.3 Control Flow	32
7.4 Messaging	32
7.5 Scheduling	33
7.6 Rules and Decisions	33
7.7 Data Manipulation	33
7.8 User-written Code	34
7.9 Logging	34
7.10 Error Handling	34
8 Handling Errors in BPL	37
8.1 System Error with No Fault Handling	37
8.1.1 Event Log Entries	38
8.1.2 XData for This BPL	38
8.2 System Error with Catchall	39
8.2.1 Event Log Entries	41
8.2.2 XData for This BPL	42
8.3 Thrown Fault with Catchall	42
8.3.1 Event Log Entries	44
8.3.2 XData for This BPL	44
8.4 Thrown Fault with Catch	45
8.4.1 Event Log Entries	47
8.4.2 XData for This BPL	47
8.5 Nested Scopes, Inner Fault Handler Has Catchall	48
8.5.1 Event Log Entries	50
8.5.2 XData for This BPL	51
8.6 Nested Scopes, Outer Fault Handler Has Catchall	51
8.6.1 Event Log Entries	54
8.6.2 XData for This BPL	54
8.7 Nested Scopes, No Match in Either Scope	54
8.7.1 Event Log Entries	56
8.7.2 XData for This BPL	57
8.8 Nested Scopes, Outer Fault Handler Has Catch	58
8.8.1 Event Log Entries	59
8.8.2 XData for This BPL	60
8.9 Thrown Fault with Compensation Handler	60
8.9.1 Event Log Entries	62
8.9.2 XData for This BPL	63
9 BPL Business Process Example	65

1

Introduction to BPL Processes

BPL business processes are a form of [business logic](#) you can use within [interoperability productions](#). A BPL business process is written in the Business Process Language (BPL) and is derived from `Ens.BusinessProcessBPL`. It is identical in every way to a class derived from `Ens.BusinessProcess`, except that it supports BPL.

Use the **Business Process Designer** to create BPL classes in the Management Portal. To access the **Business Process Designer**, select **Interoperability > Build > Business Processes**.

- Click **New** to create a BPL business process using the [Business Process Wizard](#).
- Click **Open** to edit an existing BPL business process using the [Business Process Designer](#).

Note that there is overlap among the options available in business processes, data transformations, and business rules. For a comparison, see [Comparison of Business Logic Tools](#).

1.1 BPL Features

BPL is a language used to describe executable business processes within a standard XML document. BPL syntax is based on several of the proposed XML standards for defining business process logic, including the Business Process Execution Language for Web Services (BPEL4WS or BPEL) and the Business Process Management Language (BPML or BPMI).

BPL is a superset of other proposed XML-based standards, in that it provides additional elements whose purpose is to help you build integration solutions. These additional elements include support for the following:

- Execution flow control elements such as `<branch>`, `<if>`, `<switch>`, `<foreach>`, `<while>`, and `<until>`. For information about how to use these and other BPL syntax elements, see [Business Process and Data Transformation Language Reference](#).
- Generation of executable code from business process logic.
- Embedding SQL and custom-written code into the business process logic.

You can use ObjectScript or Python as the scripting language.

- The **Business Process Designer**, a full-featured, visual modeling tool for graphically viewing and editing business process logic. This tool includes complete round-trip engineering between the visual and BPL representations of the business process. Changes to one representation are automatically reflected in the other.
- Automatic support for both asynchronous and synchronous messaging between business processes and other members of an integration solution. BPL streamlines this difficult and error-prone programming task.

- Persistent state. BPL permits a long-running business process to automatically suspend execution—and efficiently save its execution state to a built-in, persistent cache—whenever it is inactive; for example, when it is waiting for an asynchronous response. InterSystems IRIS automatically manages all state preservation and the ability to smoothly resume processing.
- Rich and varied data transformation services, including SQL queries embedded within the business process.

You can create a BPL business process using the Management Portal or [your IDE](#).

1.2 A BPL Business Process Can Be Reusable

A *business process component* or *BPL component* is a BPL business process that a programmer wishes to identify as a modular, reusable sequence of steps in the BPL language. A BPL component is analogous to a function, macro, or subroutine in other programming languages.

Only another BPL business process can call a BPL component. It does this using the BPL `<call>` element. The BPL business process component performs tasks, then returns control to the BPL business process that called it.

The production architecture already allows one BPL business process to call another BPL business process. The optional *component* designation simply provides convenience. It allows you to classify certain BPL business processes as simpler, lower-level components that:

- Are not intended to run as stand-alone business processes (although nothing in the architecture prevents this)
- May be reusable (in the sense of a function, macro, or subroutine in the BPL language)

Business processes that are not components are assumed to have more complex, special-purpose designs, and to operate at a higher conceptual level than components. It is expected that BPL non-components call BPL components to accomplish tasks.

Important: There is no requirement that you use the component designation for any BPL business process. It is available as a convenience for any BPL programmer who prefers it.

You make a business process into a component by setting an attribute of the top-level `<process>` container for the BPL business process. The attribute is called *component* and you can set it to 1 (true) or 0 (false). For syntax details, see [Business Process and Data Transformation Language Reference](#).

To set the value of the *component* attribute, you can do either of the following:

- In the **General** tab of the **Business Process Designer**, select **Is component** to include this process in the Component Library.
- Edit the BPL `<process>` element within the XData BPL block in the class code in [your IDE](#).

To set up a `<call>` to a component from a BPL business process, see [Adding a Call Activity](#).

1.3 See Also

- [Comparison of Business Logic Tools](#)
- [Listing the Business Rule Processes](#)
- [Creating a New BPL Business Process](#)

- [Editing a BPL Business Process](#)

2

Listing the Business Processes

This page describes how to list the [business processes](#) available for [interoperability productions](#) in the current namespace. From this list, you have options to create, edit, and see the activity of business processes.

2.1 The Business Process List Page

To list the business processes available in the current namespace, display the **Business Process List** page (available only for interoperability-enabled namespaces.) To navigate to this page in the Management Portal, select **Interoperability > List > Business Processes**.

BPL business processes are displayed in blue; you can double-click one to open it in the **Business Process Designer**. Business processes displayed in black are custom classes you must edit in [your IDE](#).

You can select a business process class to be the target of one of the following commands in the ribbon bar:

- **New**—Click this to launch the [Business Process Wizard](#).
- **Open** (*BPL classes only*)—Click this to edit the selected business process in the [Business Process Designer](#).
- **Export**—Click to export the selected business process class to an XML file.
- **Import**—Click to import a business process that was exported to an XML file.
- **Delete**—Click to delete the selected business process class.
- **Instances**—Click to list any current instances of the business process in the running production. If a business process has completed its work, there is no entry for it on this page.
See [Monitoring Productions](#).
- **Rule Log**—Click to view the business rule log for rules invoked by this business process.
See [Monitoring Productions](#).

You can also export and import business process classes as you do any other class in InterSystems IRIS, from [your IDE](#) or on the **Classes** page of the Management Portal (**System Explorer > Classes**).

2.2 See Also

- [Creating a New BPL Business Process](#)

- [Editing a BPL Business Process](#)
- [Monitoring Productions](#)

3

Creating a New BPL Business Process

This page describes how to get started creating a new BPL [business process](#) for interoperability [productions](#).

3.1 Using the Business Process Wizard

The **Business Process Wizard** enables you to create a BPL business process class inherited from `Ens.BusinessProcessBPL`.

To access the wizard, display the [list of business processes](#) and then click **New**. The wizard then displays a dialog box where you can specify the following preliminary characteristics of your BPL business process:

Package

Enter a package name to contain the business process class or select from a list of packages in the namespace.

Name

Enter a name for your BPL business process class.

Description

(Optional) Enter a description for the business process; this becomes the class description.

When you complete the wizard by clicking **OK**, the start and end points of the BPL diagram display in the [Business Process Designer](#), ready for you to add activities to your BPL business process.

3.2 See Also

- [Listing the Business Processes](#)
- [Editing a BPL Business Process](#)

4

Editing a BPL Business Process

This topic describes how to use the Business Process Designer, which enables you to create a BPL [business processes](#) for interoperability [productions](#). This visual editor generates a class description—that is, a text document—in correct BPL syntax. The BPL diagram and the BPL document are equally valid descriptions of the same BPL business process class.

When you open a BPL business process in the Management Portal or [your IDE](#), or when you create a new BPL business process using a wizard, its BPL diagram displays in the Business Process Designer. To the right of the diagram is a pane containing a set of property tabs; you can expand and collapse this right pane as desired using the double arrow icons.

Important: After a period of inactivity, the InterSystems Management Portal may log you out and discard any unsaved changes. Inactivity is the time between calls to the InterSystems IRIS server. Not all actions constitute a call to the server. For example, clicking **Save** constitutes a call to the server, but typing in a text field does not. Consequently, if you are editing a business process, but have not clicked **Save** for longer than **Session Timeout** threshold, your session will expire and your unsaved changes will be discarded. After a logout, the login page appears or the current page is refreshed. For more information, see Automatic Logout Behavior in the Management Portal.










4.1 BPL Designer Toolbar

The ribbon bar of the Business Process Designer page contains the options and commands that form the BPL designer toolbar:



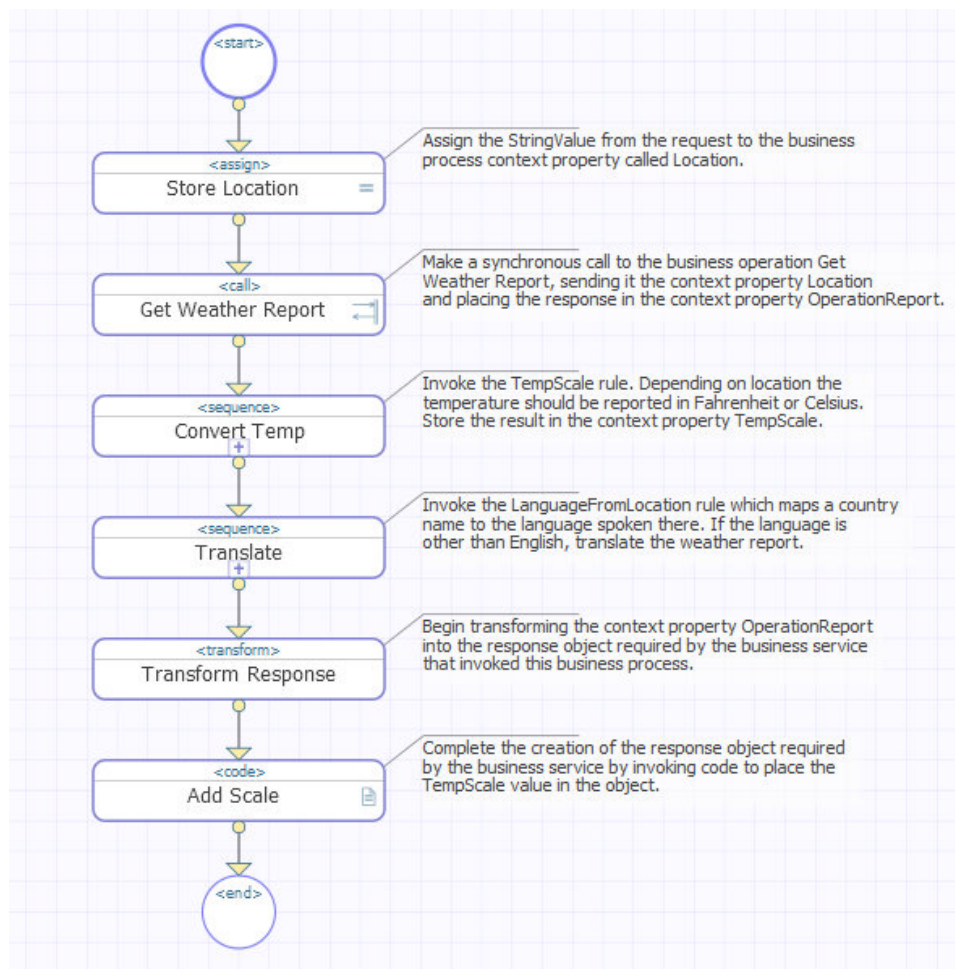
Commands that are not valid at the focus of the diagram appear dimmed. The following table describes the action you initiate for each command and, if applicable, provides a link to a detailed description of the process.

Command	Description
New	Launch the Business Process Wizard to create a new BPL business process.
Open	Launch the Finder Dialog to choose an existing BPL business process class to load and begin editing using the Business Process Designer.
Save	Save any changes you have made to the business process diagram.
Save As	Save your changes as a new BPL business process class.
Compile	Compile the BPL business process class.

Command	Description
100%	Choose from a list of percentage values to shrink or enlarge the size of the BPL diagram. Choose a large factor to view details, a small factor to gain an overview.
Add Activity	Choose a BPL element from the list of activities to add to your BPL diagram as described in Adding Activities to BPL .
Group Items	Group the selected items to form a group diagram, into which you can drill down for details; the group is represented as one shape on the higher level diagram. You can select whether to group the selected elements as a <sequence>, as a specific type of loop: <foreach>, <while>, or <until>, or as a <scope> or <flow> element.
	Undo your most recent action, such as adding, moving, or editing an activity.
	Remove the selected item from the diagram. You can select one activity at a time by clicking on it. The selected activity changes color to yellow.
	Cut the selected items and places them on the BPL clipboard.
	Copy the selected items to the BPL clipboard.
	Paste items from the BPL clipboard at the selected location.
	Display the BPL diagram of the activity details that you have grouped together. This drills down into a nested diagram. Only available for a shape representing a group (you can also click the plus sign in at the bottom of the shape).
	Display the current group as a single shape in the higher level diagram. This drills up the nested diagram. Only available if you have previously drilled down into a group.
	Arrange the layout of items in the diagram; this aligns shapes in the diagram without changing the underlying BPL document.
	Display a printer friendly version of the diagram in a new browser page.

4.2 BPL Diagram

The left pane below the ribbon bar displays the BPL diagram which consists of shapes that correspond to activities in a BPL file, with additional shapes and connections that correspond to logic in the BPL file. The following is a sample BPL diagram.



The following topics provide details about the diagram and how it represents the different elements of BPL.

- [Editing Tips](#)
- [BPL Diagram Shapes](#)
- [BPL Diagram Connections](#)
- [BPL Diagram Layout](#)
- [Drilling Down in a BPL Diagram](#)

4.2.1 Editing Tips






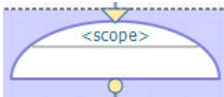





When using the Business Process Designer to work with a BPL diagram, you can:

- Select a shape by clicking it.
- Select multiple shapes by holding down the **Ctrl** key while selecting.
- Connect one element to another by clicking on its input or output connection point and dragging to the desired element. The Business Process Designer does not allow you to make an illegal connection.
- Display or edit the properties of an element by selecting it and viewing its properties in the **Activity** tab to the right. You can also select a connector to see its properties; click the other property tabs to see the properties of the process itself.










- Insert and connect a new shape in one operation: select the connector between the two elements where the new shape should go, then add an activity. The new shape appears between the existing elements with connections automatically in place.
- Automatically validate activities as you add them to the diagram. If InterSystems IRIS® detects an element with a logical error, it displays a red warning on the **Activity** tab for the element along with the reason for the error.

4.2.2 BPL Diagram Shapes

A BPL diagram uses certain shapes to indicate that a BPL element is present in the code.

BPL Shape	Meaning	Example
	Activity *	<assign>, <call>, <sync>, and most others.
	Loop	<foreach>, <while>, or <until>. Reveal the loop details by clicking on the arrow at the bottom of the shape, or by clicking 
	Sequence	<catch>, <catchall>, or <sequence>. Reveal the sequence by clicking on the plus sign at the bottom of the shape, or by selecting the shape, or by clicking 
	Scope	The start of a BPL <scope> for error handling purposes. A shaded rectangular background encloses all the BPL elements that fall within this <scope>. If the <scope> includes a <faulthandlers> element, the rectangle includes a horizontal dashed line across the middle; the area below this line displays the contents of the <faulthandlers>. For examples, see the topic Handling Errors in BPL .
	Decision	The start of an <if>, <switch>, or <branch>.
	Special	<alert>, <reply>, or <label>.
	Split	The start of a BPL <flow> element, where various logical paths diverge from a single point
	Join	The end of any branching element—<if>, <flow>, <branch>, <scope>, or <switch>—where all possible paths come together.
	Start/End	The start or end of a BPL diagram.

* Many activity shapes display an icon, as shown in the right portion of the <call> activity box. The following table lists and describes the meaning of these icons.

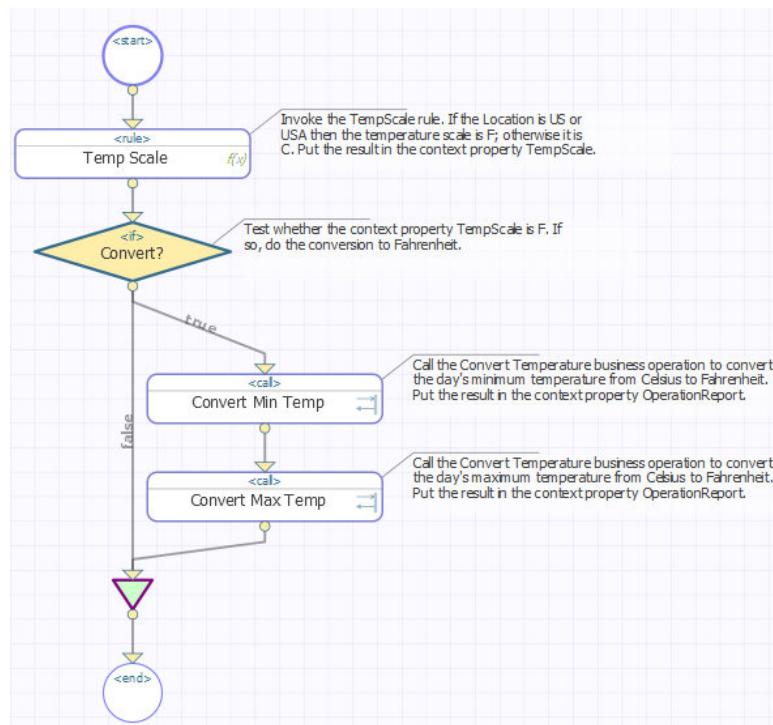
Icon	BPL Element	Icon	BPL Element	Icon	BPL Element
=	<assign>		<code>		<sql>
	Asynchronous <call>		<delay>		<sync>
	Synchronous <call>		<milestone>		<trace>
	<catch>	$f(x)$	<rule>		

In general the interior color of a BPL diagram shape is white, with a blue outline. If the shape is in error, its outline is red. If the shape is disabled, its interior color is gray, with a gray outline.

When you click a shape in a BPL diagram, it becomes selected. Its attributes display in the **Activity** tab, where you can edit their values. Its interior color changes to yellow. If it is in error, its outline remains red; if not, its outline changes to a bolder blue. When you select a disabled shape it shows a dotted outline.

You can select multiple shapes by holding down the **Ctrl** key while clicking on shapes. To clear the selection of a shape, click on it while it is selected.

When a shape represents a complex activity such as <if> or <switch> that has multiple branches, joins, or other types of related shapes elsewhere in the BPL diagram, clicking on one of these shapes highlights the related shapes in green with a purple outline. Clicking on a <sync> element highlights the <call> elements that it synchronizes. Clicking on an <if> shape highlights the Join where the <true> and <false> branches come together, and so on. For example:



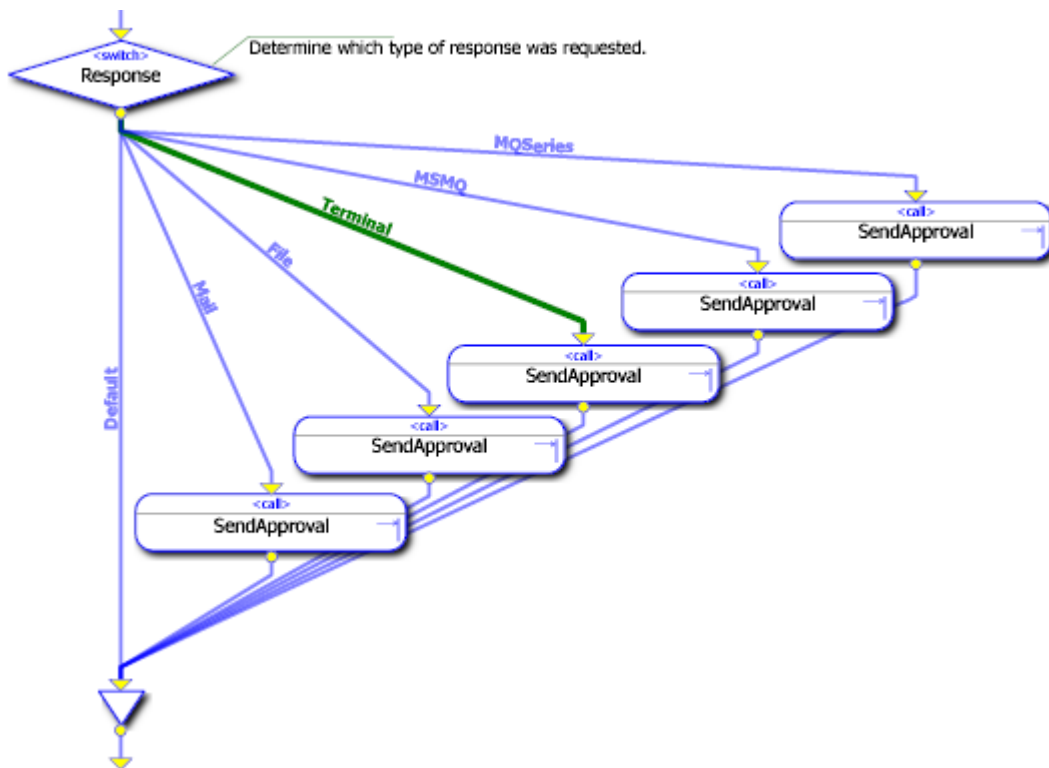
4.2.3 BPL Diagram Connections

In a BPL diagram, the lines between shapes specify logical relationships and sequencing among the elements. These lines are called connections. The start of each line is a circular *nub* and the end is a triangular point. One triangular *input* nub and one circular *output* nub are built into each shape that you add to the BPL diagram.

You can connect one shape to another by clicking on its input or output nub and dragging the cursor to the desired shape. When you release the mouse, a connection appears. Another way to connect shapes is to insert and auto-connect a new shape in one step. Select the two elements on either side of where the new shape should go. You can select multiple elements by holding down the **Ctrl** key while clicking on the shapes. If two elements are selected with no existing connection between them, you can add a new shape and it appears between the existing elements, with connections automatically in place. To add a new shape between two connected elements, click on the connection to highlight it, then add the new element. The new shape appears between the existing elements, with connections automatically in place.


Once two shapes are connected, the connection is preserved no matter where you drag the respective shapes. You can drag shapes to any layout position you wish, within the same diagram. Connections reroute automatically, and the underlying BPL document is not changed. On the other hand, if you change the logic of the connections, for example to reorder calls, create loops, or cut and paste, then the underlying BPL document *does* change to reflect your actions in the Business Process Designer.

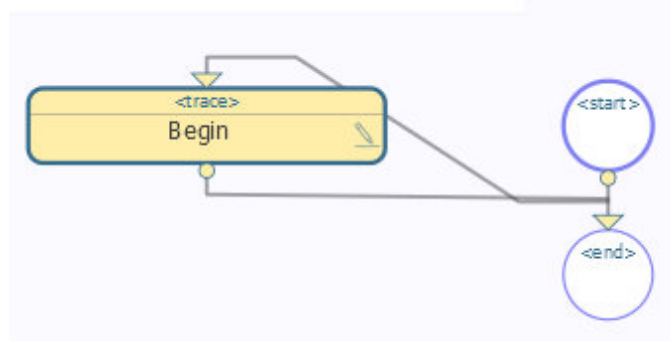
Within a <switch> activity, each possible path is automatically labeled with the corresponding <switch> value. All of the possible paths from a <switch> activity converge at a Join shape before a single arrow connects from the Join shape to the next activity in the BPL diagram.



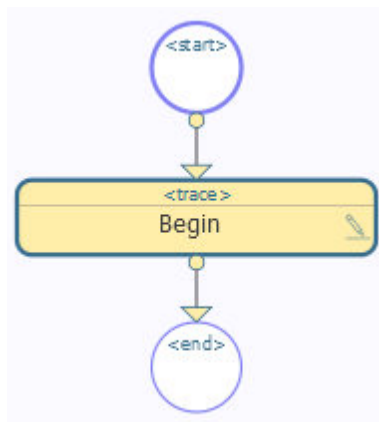
The Business Process Designer provides many types of validation of your diagram as you work on it. One useful validation feature is that the editor detects if the output branches of an <if>, <flow>, or <switch> element are connected to the wrong Join shape in the diagram. If so, the connector that is in error displays in red until you correct the diagram.

4.2.4 BPL Diagram Layout

After you add shapes or create new connections, you can tidy the diagram by clicking the arrange icon  on the tool bar. For example, if you do not have the auto arrange feature set in your preferences, when you add a shape to a BPL diagram it looks something like the following figure.



When you click the auto arrange tool  the shapes are aligned as shown in the following figure.

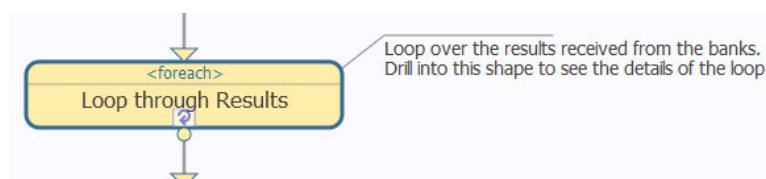




If you want your diagrams to always use this type of structured layout, select the **Auto arrange** check box on the **Preferences** tab.

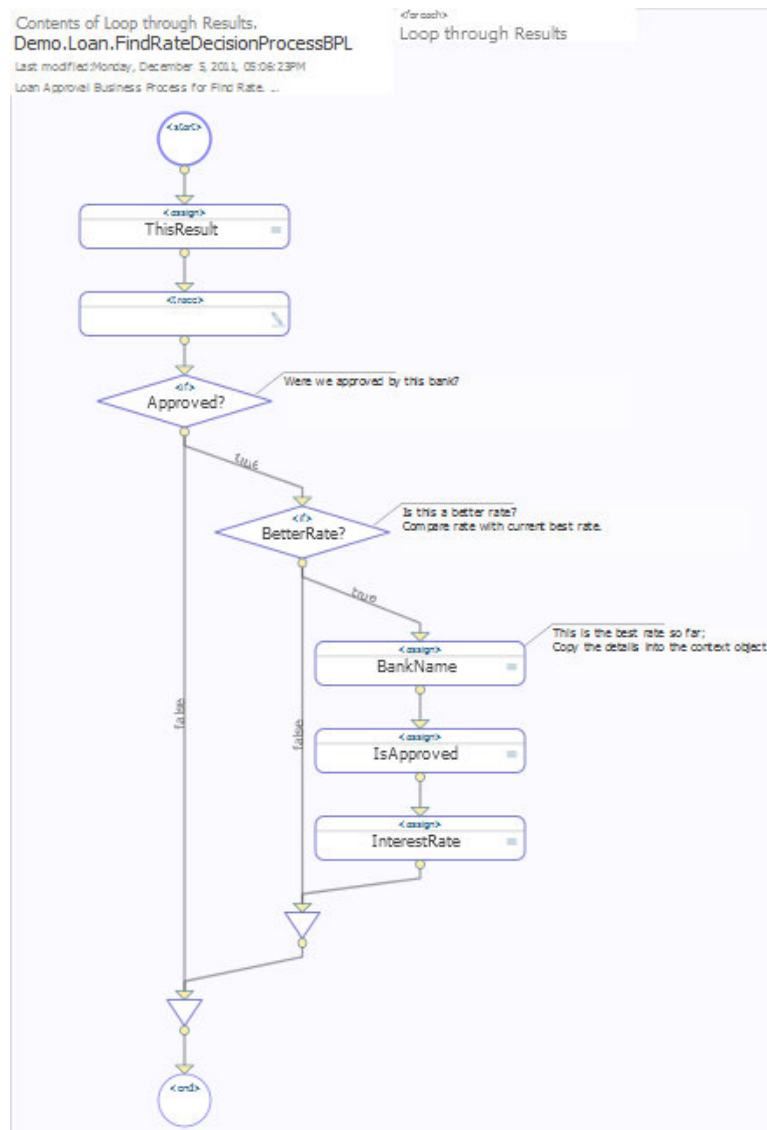
By default, when you open a BPL diagram in the Business Process Designer for the first time, the auto arrange feature is enabled. This choice may or may not be appropriate for a particular drawing. You can disable automatic arrangement to ensure that your diagram always displays with exactly the layout you want by clearing the **Auto arrange** check box on the **Preferences** tab. This way, when the diagram is displayed in the Business Process Designer, it does not take on any layout characteristics except what you have specified.


4.2.5 Drilling Down into a BPL Diagram

A loop activity displays a cyclic arrow to indicate that it provides drill-down details. The following is an example of the `<foreach>` loop activity. Others include `<while>` and `<until>`. For example:

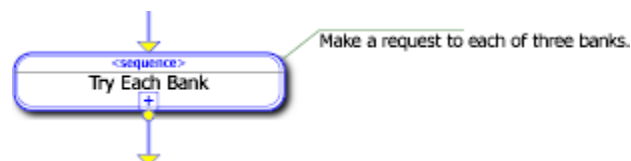



If you select the loop activity and then click the  tool or click the  in the loop shape, a BPL diagram of the loop displays. This is a full BPL diagram showing all the logic between the start and end of the loop. For example:

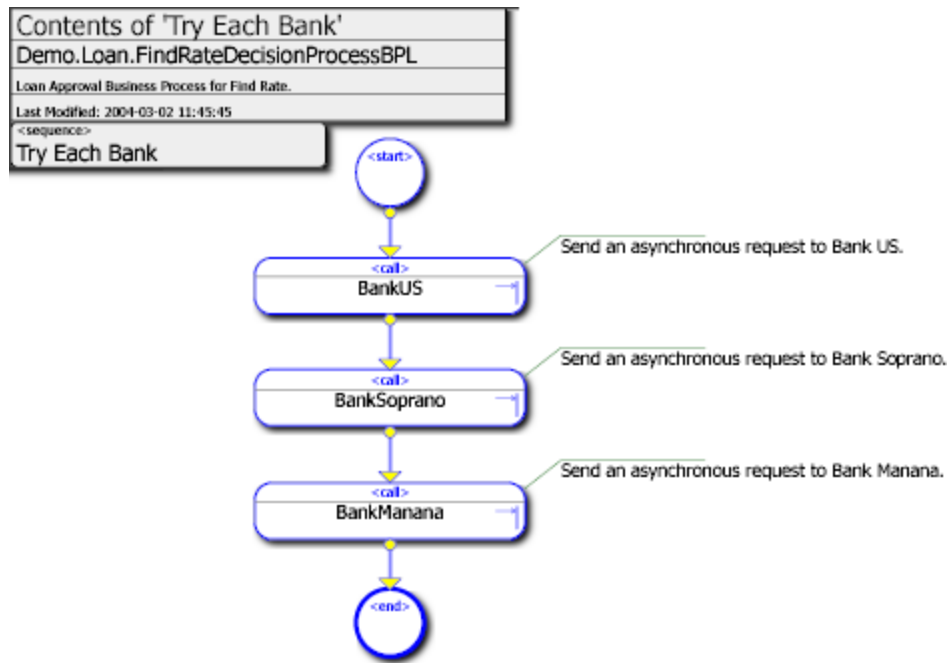


To return to the higher logical level, where one shape represents the entire loop, click the  tool.

A sequence displays a plus sign to indicate that it, too, can provide drill-down details. For example:



When you drill down into a sequence, the resulting BPL diagram shows all the logic between the start and end of the sequence. To return to the higher logical level, where one shape represents the entire sequence, click the  tool. For example:



If there is an error anywhere in a lower-level diagram, the Business Process Designer highlights the group shape (<foreach>, <sequence>, <while>, or <until>) in red. To fix the error, you must drill down into the group shape to see the activity that has the error highlighted in red in the lower-level diagram.

4.3 Adding Activities to a BPL Diagram

The **Add Activity** list is available in the Business Process Designer toolbar whenever you have a BPL diagram open in the Management Portal or [your IDE](#). When you click on an item in this list, you add its shape to the BPL diagram. The list is divided into the following categories:

- Activities

Adds this Activity	View documentation for this element
Alert	<alert>
Assign	<assign>
Break	<break>
Call	<call>
Code	<code>
Continue	<continue>
Delay	<delay>
Empty	<empty>
Reply	<reply>
Rule	<rule>
SQL	<sql>

Adds this Activity	View documentation for this element
Sync	<sync>
Trace	<trace>
Transform	<transform>
XPATH	<xpath>
XSLT	<xslt>

- Decisions and Placeholders

Adds this Decision	View documentation for this element
If	<if>
Switch	<switch>
Branch	<branch>
Label	<label>
Milestone	<milestone>

- Logic

Adds this Logic	View documentation for this element
Flow	<flow>
Join	(required for the diagram, no corresponding BPL element)
Scope	<scope>
Sequence	<sequence>

- Loops

Adds this Loop	View documentation for this element
ForEach	<foreach>
While	<while>
Until	<until>

- Error Handling

Adds this Loop	View documentation for this element
Throw	<throw>
Catch	<catch>
Catch All	<catchall>
Compensate	<compensate>
Compensation Handler	<compensationhandlers>

When you add an element to the diagram, the **Activity** tab displays with the properties applicable to the element. At the top of the tab is the corresponding BPL element and a description along with an active link to the BPL reference entry for that element. This is the most accurate place to get information about the settings that follow.

Except for the <start> and <end> shapes, the following settings are common to all elements:

Name

Enter a name for the caption inside the shape.

x

The x axis coordinate for the location of the selected shape in the diagram.

y

The y axis coordinate for the location of the selected shape in the diagram.

Disabled

Select this check box to disable the activity; clear it to enable. The default is enabled.

Annotation

Enter text to appear as comments next to the shape in the diagram.

Note: The <start> and <end> shapes only have the x and y coordinates, so you can move them manually if you wish.

4.3.1 Adding a Call Activity

A common task in a BPL business process is to add a Call activity. The following information is necessary to properly create a new <call> to one of the available business processes or business operations in the production:

- Input
- Output
- Name
- Target
- Request

4.4 BPL Designer Property Tabs

To the right of the BPL diagram is a pane containing a set of property tabs; you can expand and collapse this right pane as desired using the double arrow icons. Three of the tabs relate to the BPL business process itself and one relates to the selected shape:

- **General**—contains settings for the overall definition of the BPL business process. See [Setting General Properties of the BPL Business Process](#).
- **Context**—provides the interface for [defining the context object](#) for this BPL business process.
- **Activity**—contains settings for the selected item in the BPL diagram; see [Adding Activities to a BPL Diagram](#).
- **Preferences**—contains settings pertaining to the appearance of the BPL diagram. See [Setting BPL Diagram Preferences](#).

4.4.1 Setting General Properties of the BPL Business Process

The **General** tab contains the following settings that apply to the BPL business process:

- **Language**—Can be **Python**, **ObjectScript**, or **Basic**.
- **Layout**—Select either **Automatic** or **Manual** for the size of the diagram. If you select **Manual** you can enter a **Width** and **Height**.
- **Annotation**—Enter text to include in the class description.
- **Includes**—An optional comma-delimited list of ObjectScript include file names, so that you can use macros in your `<code>` segments.
- **Python From / Import statements**—An optional list of Python `from / import` statements, one per line. Use this so that Python code within this business process can refer to these modules.
- **Version**—Enter an optional version number of the BPL diagram
- **Is component**—If true, include this process in the component library where it can be called by other processes.

See [<process>](#) for details on these properties.

4.4.2 Defining the context Object

You can define the context object of a BPL business process from the **Context** tab of the Business Process Designer. You can click the magnifying glass to launch the Finder Dialog for each of the following fields:

- **Request Class**—Choose the class of the incoming request for this process.
- **Response Class**—Choose the class of the response returned by this process.
- **Context Superclass**—Use this option to provide custom context properties, in a different way than adding to the **Context properties** list, described next. To use **Context Superclass**, create a custom subclass of `Ens.BP.Context`. In this subclass, define class properties to use as context properties. Use the name of this class as the value of **Context Superclass** in the business process. Then when you create `<assign>` actions, for example, you can choose these custom properties in addition to the standard properties of the *context* object.

You can add to the list of **Context properties** by clicking the plus sign to launch the Business Process Context Property wizard. Then enter values in the following fields:

- **Property Name**—Must be a valid identifier.
- Choose if the property data is one of the following: **Single Value**, **List Collection**, or **Array Collection**
- **Property Type**—Type of this property including parameters.
Enter a data type class name in the **Type** field or click the magnifying glass to browse for a class you want to use as a data type.
- **Default Value** (ignored for collections)—Enter an initial expression for a single value data type.
- **Instantiate**—Select this check box for object-valued properties if you want the object to be instantiated when it is created.
- **Description**—Enter an optional description of the context property.

Click **OK** to save your changes, **Cancel** to discard them. The Business Process Designer generates the necessary `<context>` and `<property>` elements in the BPL code.

To set property parameters such as `MINVAL`, `MAXVAL`, `MINLEN`, `MAXLEN`, or others in the *Business Process Designer*, add data type parameters to a *context* property when you first add the property, or at any subsequent time, by inserting a

comma-separated list of parameters enclosed in parentheses after the data type class name. That is, rather than simply entering `%String` or `%Integer`, you can enter data types such as:

```
%String(MAXLEN=256)
%Integer(MINVAL=0,MAXVAL=100)
%String(VALUELIST=" ,Buy,Sell,Hold")
```

The Business Process Designer generates the necessary `<parameters>` element in the BPL code.

Once you have defined properties on the *context* object, you can refer to them anywhere in BPL using ordinary dot syntax and the property name, as in: `context.MyData`


For reference details, see the following resources:

- Typically you choose the property type from the system library of data types described in *Data Types*. These include `%String`, `%Integer`, `%Boolean`, etc.
- System data types have optional parameters. For details, see *Parameters*. These include the *MINLEN* and *MAXLEN* parameters that set the minimum and maximum allowed lengths of a `%String` property. The default maximum `%String` length is 50 characters; you can reset this by setting the *MAXLEN* for that `%String` property to another value.

By default, the *ruleContext* passed to the rule is the business process execution context. If you specify a different object as a context, there are some restrictions on this object: It must have a property called `%Process` of type `Ens.BusinessProcess`; this is used to pass the business process calling context to the rules engine. You do not need to set the value of this property, but it must be present. Also, the object must match what is expected by the rule itself. No checking is done to ensure this; it is up to the developer to set this up correctly.

4.4.3 Setting BPL Diagram Preferences

The **Preferences** tab contains the following settings that apply to the appearance of the BPL diagram:

- **Gridlines**—Select one of the following choices for the appearance of the grid lines on the diagram: **None**, **Light**, **Medium**, or **Dark**.
- **Show annotations**—Reveal or conceal the text notes that explain each shape. When you reveal annotations, they appear to the upper right of each shape that has an `<annotation>` element in the BPL document.
- **Auto arrange**—Cause each new shapes in the diagram to automatically conform to a structured arrangement without needing to select  after adding each shape.

Changing the position of shapes does not change the underlying BPL code. Only a change to the connecting lines changes the code.

4.5 See Also

- [Listing the Business Processes](#)
- [Creating a New BPL Business Process](#)

5

Variables Available in BPL (Execution Context)

This page introduces the variables that are available within a BPL [business process](#) for interoperability [productions](#).

Tip: For detailed information about BPL syntax on BPL elements such as `<process>`, `<context>`, and `<call>`, see [Business Process and Data Transformation Language Reference](#), which also provides reference information for the context variables.

5.1 Introduction

The life cycle of a business process requires it to have certain state information saved to disk and restored from disk, whenever the business process suspends or resumes execution. This feature is especially important for long-running business processes, which may take days or weeks to complete.

A BPL business process supports the business process life cycle with a group of variables known as the *execution context*. InterSystems IRIS automatically save the variables in the execution context and restores them each time the BPL business process suspends and resumes execution. These variables are available to every BPL business process; that is, to every business process class that inherits from `Ens.BusinessProcessBPL`.

Some of the execution context variables are available to every activity within a BPL business process. Others are generally available, but go in and out of scope, depending on the type of activity that the business process is executing at the time.

Important: Custom business processes that inherit from `Ens.BusinessProcess` do not have access to a built-in execution context and must handle similar issues using custom code.

5.2 The context Object

The *context* object is available to a BPL business process anywhere inside the `<process>` element. *context* is a general-purpose container for any data that needs to be persisted during the life cycle of the business process. You define each data item as a property on the *context* object when creating the BPL business process. See [Defining the context Object](#) for the recommended procedure.

Once you have defined properties on the *context* object, you can refer to them anywhere in BPL using ordinary dot syntax and the property name, as in: `context.MyData`

5.3 The request Object

The *request* object contains the properties that were in the original request message object—the incoming message that first caused this business process to be instantiated. This is known as the *primary request*.

The *request* object is available to a BPL business process anywhere inside the `<process>` element. You can refer to the properties of the *request* object using dot syntax and the property name, as in: `request.OriginalThought`

5.4 The response Object

The *response* object contains the properties that are required to build the final response message object to be returned by this business process instance. The business process returns this final response either when it reaches the end of its life cycle, or when it encounters a `<reply>` activity.

The *response* object is available to a BPL business process anywhere inside the `<process>` element. You can refer to the properties of the *response* object using dot syntax and the parameter name, as in: `response.BottomLine`

5.5 The callrequest Object

The *callrequest* object contains any properties that are required to build the request message object to be sent by a `<call>`.

A `<call>` activity sends a request message and, optionally, receives a response. A BPL `<call>` element must include a `<request>` activity to put values into the properties on the request message object. In order to accomplish this, the `<request>` provides a sequence of `<assign>` activities that place values into properties on the *callrequest* object. Typically, some of these values are derived from properties on the original *request* object, but you are free to assign any value.

As soon as the `<assign>` activities inside the `<request>` are completed, the message is sent, and the associated *callrequest* object goes out of scope. *callrequest* has no meaning outside its associated `<request>` activity; it is already out of scope when the associated `<call>` begins processing its next activity, the optional `<response>`.

Within the scope of the relevant `<request>` element, you can refer to the properties on *callrequest* using dot syntax, as in: `callrequest.UserData`

5.6 The callresponse Object

Upon completion of a `<call>` activity, the *callresponse* object contains the properties of the response message object that was returned to the `<call>`. If the `<call>` was designed with no response, there is no *callresponse*. Similarly, if you use `<sync>` to wait for a response, but the response does not return within the timeout period specified by the `<sync>` element, there is no *callresponse*.

Every `<call>` that expects a response must provide a `<response>` activity within the `<call>`. The purpose of the `<response>` activity is to retrieve the response values and make them available to the business process as a whole. The *callresponse* object is available anywhere inside the `<response>` activity. However, as soon as the `<response>` activity completes, the

associated *callresponse* object goes out of scope. Therefore, if you want to use the values in *callresponse* elsewhere in the business process, you must [assign](#) these values to properties on the *context* or *response* objects, and you must do so before the end of the `<response>` activity in which they were received.

You can refer to the properties on *callresponse* using dot syntax, as in: `callresponse.UserAnswer`

5.7 The syncresponses Collection

syncresponses is a collection, keyed by the names of the `<call>` activities being synchronized by a `<sync>`.

When a `<sync>` activity begins, *syncresponses* is cleared in preparation for new responses. As the `<call>` activities return, responses go into the collection. When the `<sync>` activity completes, *syncresponses* may contain all, some, or none of the desired responses (see *synctimeout*). *syncresponses* is available anywhere inside the `<sequence>` that contains the relevant `<call>` and `<sync>` activities, but goes out of scope outside that `<sequence>`.

To refer to the response value from one of the synchronized calls, use the syntax: `syncresponses.GetAt ("name")`

Where the relevant `<call>` was defined as: `<call name="name">`

5.8 The synctimeout Value

synctimeout is an integer value that may be 0, 1, or 2. *synctimeout* indicates the outcome of a `<sync>` activity after several calls. You can test the value of *synctimeout* after the `<sync>` and before the end of the `<sequence>` that contains the calls and `<sync>`. *synctimeout* has one of three values:

- If 0, no call timed out. All the calls had time to complete. This is also the value if the `<sync>` activity had no *timeout* set.
- If 1, at least one call timed out. This means not all `<call>` activities completed before the timeout.
- If 2, at least one call was interrupted before it could complete.

synctimeout is available to a BPL business process anywhere inside the `<sequence>` that contains the relevant `<call>` and `<sync>` activities, but goes out of scope outside that `<sequence>`. Generally you will test *synctimeout* for status and then retrieve the responses from completed calls out of the *syncresponses* collection. You can refer to *synctimeout* with the same syntax as for any integer variable name, that is: `synctimeout`

5.9 The status Value

status is a value of type `%Status` that indicates success or failure.

Note: Error handling for a BPL business process happens automatically without your ever needing to test or set the *status* value in the BPL source code. The *status* value is documented here in case you need to trigger a BPL business process to exit under certain special conditions.

When a BPL business process starts up, *status* is automatically assigned a value indicating success. To test that *status* has a success value, you can use the macro `$$$ISOK(status)` in ObjectScript. If the test returns a True value, *status* has a success value.

As the BPL business process runs, if at any time *status* acquires a failure value, InterSystems IRIS immediately terminates the business process and writes the corresponding text message to the Event Log. This happens regardless of how *status* acquired the failure value. Thus, the best way to cause a BPL business process to exit suddenly, but gracefully is to set *status* to a failure value.

status can acquire a failure value in any of the following ways:

- *status* automatically receives the returned %Status value from any `<call>` that the business process makes to another business host. If the value of this %Status indicates failure, *status* automatically receives the failure value. This is the most common way in which *status* is set, and it happens automatically, without any special statements in the BPL code.
- An `<assign>` activity can set *status* to a failure value. The usual convention for doing this is to use an `<if>` element to test the result of some prior activity, and then within the `<true>` or `<false>` element use `<assign>` to set *status* to a failure value when failure conditions exist.
- Statements within a `<code>` activity can set *status* to a failure value. The BPL business process does not perceive the change in the value of *status* until the `<code>` activity has fully completed. Therefore, if you want a failure *status* to cause an immediate exit from a `<code>` activity, you must place a quit command in the `<code>` activity immediately after setting a failure value for *status*.

To test that *status* has a failure value, use the macro `$$$ISERR(status)` in ObjectScript. If the test returns a True value, *status* has a failure value. You will be able to perform this test only within the body of a `<code>` activity before it returns to the main BPL business process, since the business process will automatically quit with an error as soon as it detects that *status* has acquired a failure value following any `<call>`, `<assign>`, or `<code>` activity.

status is available to a BPL business process anywhere inside the `<process>`. You can refer to *status* with the same syntax as for any variable of the %Status type, that is: *status*

CAUTION: Like all other execution context variable names, *status* is a reserved word in BPL. Do not use it except as described in this topic.

5.10 The process Object

The *process* object represents the current instance of the BPL business process object. The *process* object is provided so that you can invoke any business process method, such as `SendRequestSync()` or `SendRequestAsync()`, from any context within the flow of the BPL business process, for example from within the text block of a `<code>` activity.

The *process* object is available to a BPL business process anywhere inside the `<process>` element, but is typically needed only within the `<code>` activity. You can refer to methods of the *process* object using dot syntax and the method name, as in: `process.SendRequestSync()` or `process.ClearAllPendingResponses`

6

BPL Syntax Rules

This topic describes the syntax rules for referring to properties and for creating expressions within various activities in a [business process](#) in an interoperability [production](#).

6.1 References to Message Properties

In activities within a BPL process, it may be necessary to refer to properties of the message. The rules for referring to a property are different depending on the kind of messages you are working with.

- For messages other than virtual documents, use syntax like the following:

```
message.propertyname
```

Or:

```
message.propertyname.subpropertyname
```

Where *propertyname* is a property in the message, and *subpropertyname* is a property of that property.

- For virtual documents other than XML virtual documents, use the syntax described in [Syntax Guide for Virtual Property Paths](#).
- For XML virtual documents, see [Routing XML Virtual Documents in Productions](#).

6.2 Literal Values

When you assign a value to a property, you often specify a literal value. Literal values are also sometimes suitable in other places, such as the value in a **trace** action.

A literal value is either of the following:

- A numeric literal is just a number. For example: 42.3
- A string literal is a set of characters *enclosed by double quotes*. For example: "ABD"

Note: This string cannot include XML reserved characters. For details, see [XML Reserved Characters](#).

For virtual documents, this string cannot include separator characters used by that virtual document format. See [Separator Characters in Virtual Documents](#) and [When XML Reserved Characters Are Also Separators](#).

6.2.1 XML Reserved Characters

Because BPL processes are saved as XML documents, you must use XML entities in the place of XML reserved characters:

To include this character...	Use this XML entity...
>	>
<	<
&	&
'	'
"	"

For example, to assign the value Joe's "Good Time" Bar & Grill to a property, set **Value** equal to the following:

```
"Joe&apos;s &quot;Good Time&quot; Bar &amp; Grill"
```

This restriction does not apply inside `<code>` and `<sql>` activities, because InterSystems IRIS® automatically wraps a CDATA block around the text that you enter into the editor. (In the XML standard, a CDATA block encloses text that should not be parsed as XML. Thus you can include reserved characters in that block.)

6.2.2 Separator Characters in Virtual Documents

In most of the virtual document formats, specific characters are used as separators between segments, between fields, between subfields, and so on. If you need to include any of these characters as literal text when you are setting a value in the message, you must instead use the applicable escape sequence, if any, for that document format.

For information on these separators, see:

- [EDIFACT Separators](#)
- [X12 Separators](#)

6.2.3 When XML Reserved Characters Are Also Separators

- If the character (for example, &) is a separator and you want to include it as a literal character, use the escape sequence that applies to the virtual document format.
- In all other cases, use the XML entity as shown previously in [XML Reserved Characters](#).

6.2.4 Numeric Character Codes

You can include decimal or hexadecimal representations of characters within literal strings.

The string `&#n;` represents a Unicode character when *n* is a decimal Unicode character number. One example is `é` for the Latin e character with acute accent mark (é).

Alternatively, the string `&#xh;` represents a Unicode character when *h* is a hexadecimal Unicode character number. One example is `¿` for the inverted question mark (¿).

6.3 Valid Expressions

When you assign a value to a property, you can specify an expression, in the language that you selected for the BPL process. You also use expressions in other places, such as the condition for an `<if>` activity, the value in a `<trace>` activity, statements in a `<code>` activity, and so on.

The following are all valid expressions:

- Literal values, as described in the [previous section](#).
- Function calls (InterSystems IRIS provides a set of utility functions for use in business rules and data transformations. For details, see [Utility Functions for Use in Productions](#).)
- References to properties, as described in [References to Properties](#).
- Any expression that combines these, using the syntax of the scripting language you chose for BPL process. Note the following:
 - In ObjectScript, the concatenation operator is the `_` (underscore) character, as in:


```
value="prefix"_source.{MSH:ReceivingApplication}_"suffix"
```
 - To learn about useful ObjectScript string functions, such as `$CHAR` and `$PIECE`, see ObjectScript Reference.
 - For a general introduction, see [Using ObjectScript](#).

6.4 Indirection

InterSystems IRIS supports indirection in values for the following BPL element-and-attribute combinations only:

- `<call name=`
- `<call target=`
- `<sync calls=`
- `<transform class=`

The *at sign* symbol, `@`, is the indirection operator.

For example, the `<call>` element supports indirection in the values of the *name* or *target* attributes. The *name* identifies the call and may be referenced in a later `<sync>` element. The *target* is the configured name of the business operation or business process to which the request is being sent. Either of these strings can be a literal value:

```
<call name="Call" target="MyApp.MyOperation" async="1">
```

Or the `@` indirection operator can be used to access the value of a context variable that contains the appropriate string:

```
<call name="@context.nextCallName" target="@context.nextBusinessHost" async="1">
```

For information on `@` indirection syntax, see [<call>](#), [<sync>](#), and [<transform>](#).

Important: BPL and DTL are similar in many ways, but DTL does *not* support indirection.

7

List of BPL Elements

This topic divides elements of BPL [business process](#) into functional groups and explains the purpose of each element.

7.1 Business Process

A BPL document consists of a [<process>](#) element and its various child elements. The [<process>](#) element is the container for the business process.

7.2 Execution Context

The life cycle of a business process requires it to have certain state information saved to disk and restored from disk, whenever the business process suspends or resumes execution. This feature is especially important for long-running business processes, which may take days or weeks to complete.

A BPL business process supports the business process life cycle with a group of variables known as the *execution context*. The variables in the execution context are automatically saved and restored each time the BPL business process suspends and resumes execution. The variables are:

- *context*
- *request*
- *response*
- *callrequest*
- *callresponse*
- *syncresponses*
- *synctimedout*
- *status*

Most of the execution context variables are automatically defined for the business process. The exception to this rule is the general-purpose container object called *context*, which a BPL developer defines by providing [<context>](#), [<property>](#), and [<parameters>](#) elements at the beginning of the BPL document.

For complete details on these variables, see [Business Process and Data Transformation Language Reference](#).

Also see documentation of the [<assign>](#) element and see [Business Process Execution Context](#).

7.3 Control Flow

Within a business process, activities are either executed sequentially or in parallel:

- Sequential execution is specified using the [<sequence>](#) element
- Parallel execution is specified using the [<flow>](#) element in combination with [<sequence>](#) elements.

BPL includes a number of control flow elements that you can use to control the order of execution inside a BPL business process.

BPL Element	Purpose	Description
<branch>	Branch	Conditionally cause an immediate change in the flow of execution.
<break>	Loop	Break out of a loop and exit the loop activity.
<continue>	Loop	Jump to the next iteration within a loop, without exiting the loop.
<flow>	Group	Perform activities in a non-determinate order.
<foreach>	Loop	Define a sequence of activities to be executed iteratively.
<if>	Branch	Evaluate a condition and perform one action if true, another if false.
<label>	Branch	Provide a destination for a conditional branch operation.
<sequence>	Group	Organize one or more calls to other business operations and business processes. Structures parts of the BPL diagram.
<switch>	Branch	Evaluate a set of conditions to determine which of several actions to perform.
<until>	Loop	Define a sequence of activities to be repeatedly executed <i>until</i> a condition is true.
<while>	Loop	Define a sequence of activities to be repeatedly executed <i>as long as</i> a condition is true.

Note: BPL business process code can initiate a sudden, but graceful exit by setting the business process execution context variable *status* to a failure value using an [<assign>](#) or [<code>](#) statement.

7.4 Messaging

BPL includes elements that allow you to make synchronous and asynchronous requests to business operations, and to other business processes.

BPL Element	Purpose
<code><call></code>	Send a request and (optionally) receive a response from a business operation or business process. The call may be synchronous or asynchronous.
<code><request></code>	Prepare the request for a call to another business operation or business process.
<code><response></code>	Receive the response returned from a call to another business operation or business process.
<code><sync></code>	Wait for a response from one or more asynchronous calls to other business operations and business processes.
<code><reply></code>	Return a primary response from the business process before execution of the process is fully complete.

7.5 Scheduling

The `<delay>` element can be used to delay execution of a business process for a specified duration or until a future time.

7.6 Rules and Decisions

The `<rule>` element executes a business rule. This element specifies the business rule name, plus parameters to hold the result of the decision and (optionally) the reason for that result.

The parameters for the `<rule>` element can include any property in the general-purpose execution context variable called *context*. Therefore, a typical design approach, for a business process that invokes a rule, is to ensure that the business process accomplishes the following:

1. Provides `<property>` and `<context>` elements so that the *context* object contains properties with appropriate names and types.
For example, if the rule determines eligibility for a state education loan, you might add properties such as Age, State, and Income.
2. Gathers values for the properties in whatever way you wish, for example by sending requests to business operations or business processes, and as responses return, assigning values to the properties in *context*.
3. Provides a `<rule>` element that invokes a business rule that returns an answer based on these input values.

For details on the execution context variables, see [Business Process Execution Context](#).

For information on creating business rules, see [Developing Business Rules](#).

7.7 Data Manipulation

BPL includes several elements that allow you to move data from one location to another. For example, a typical business process makes a series of calls to business operations or other business processes. To set up these calls, as well as to process the data they return, the business process shuffles data between the various execution context variables—*context*, *request*, *response*, and others. This shuffling and other data manipulation tasks are accomplished using the elements described below.

BPL Element	Purpose
<code><assign></code>	Assign a value to a property.
<code><sql></code>	Execute an embedded SQL SELECT statement.
<code><transform></code>	Transform one object into another using a data transformation.
<code><xpath></code>	Evaluate XPath expressions on a target XML document.
<code><xslt></code>	Execute an XSLT transformation to modify a data stream.

7.8 User-written Code

For cases where BPL is not expressive enough to solve a specific problem, InterSystems IRIS® provides mechanisms to embed user-written code within the automatically generated business process code.

BPL Element	Purpose
<code><code></code>	Allows you to specify the required code within a CDATA block.
<code><empty></code>	Performs no action; acts as a placeholder until code can be written.

7.9 Logging

BPL includes elements you can use to log informational and error messages.

BPL Element	Purpose
<code><alert></code>	Write a text message to an external alert mechanism.
<code><milestone></code>	Store a message to acknowledge a step achieved by a business process.
<code><trace></code>	Write a text message to a console window and to the Event Log.

7.10 Error Handling

BPL includes elements that you can use to throw and catch faults, and perform compensation for errors or faults. These elements are closely interrelated. For details, see [Handling Errors in BPL](#). The list of elements is as follows:

BPL Element	Purpose
<code><catch></code>	Catch a fault produced by a <code><throw></code> element.
<code><catchall></code>	Catch a fault or system error that does not match any <code><catch></code> .
<code><compensate></code>	Invoke a <code><compensationhandler></code> from <code><catch></code> or <code><catchall></code> .
<code><compensationhandler></code>	Perform a sequence of activities to undo a previous action.
<code><compensationhandlers></code>	Contain one or more <code><compensationhandler></code> elements.

BPL Element	Purpose
<code><faulthandlers></code>	Provide zero or more <code><catch></code> and one <code><catchall></code> element.
<code><scope></code>	Wrap a set of activities with its fault and compensation handlers.
<code><throw></code>	Throw a specific, named fault.

8

Handling Errors in BPL

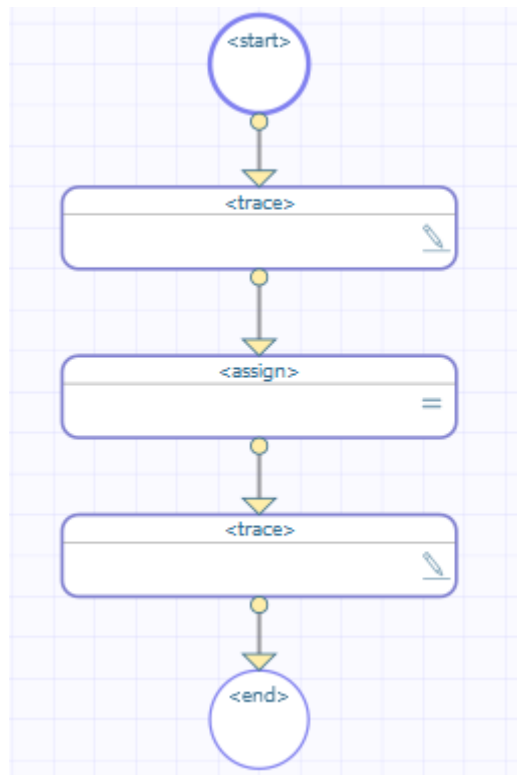
This topic explains how BPL [business processes](#) support error handling. BPL provides fault handlers that allow your business process to throw and catch errors, and compensation handlers that allow your business process to specify how it recovers from errors by undoing the actions that led to the error condition.

Important: When you use this error handling system with `<call>` statements that communicate with other business hosts, make sure that the target business hosts return an error status in the case of an error. If the target component returns success even in the case of an error, the BPL process will not trigger `<catchall>` logic.

The BPL elements involved in error handling are `<scope>`, `<throw>`, `<catch>`, `<catchall>`, `<compensate>`, `<compensation-handlers>`, `<compensationhandler>`, and `<faulthandlers>`. This topic introduces these elements and explains how they work together to support the different error handling scenarios.

8.1 System Error with No Fault Handling

The following is an example of a BPL business process that produces an error condition and provides no error handling:



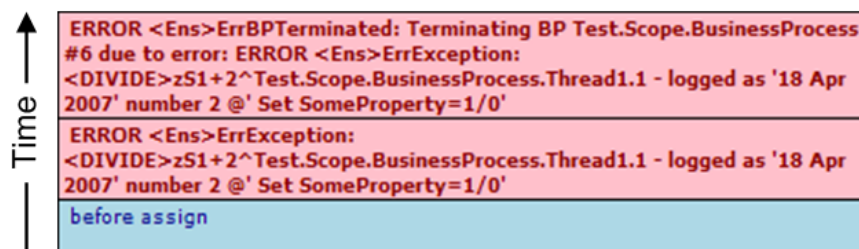
This BPL business process does the following:

1. The first <trace> element generates the message before assign.
2. The <assign> element tries to set SomeProperty equal to the expression 1/0. This attempt produces a divide-by-zero system error.
3. The business process ends and sends a message to the Event Log.

The second <trace> element is never used.

8.1.1 Event Log Entries

The corresponding Event Log entries look like this.



For background information, see [Event Log](#).

8.1.2 XData for This BPL

This BPL is defined by the following XData block:

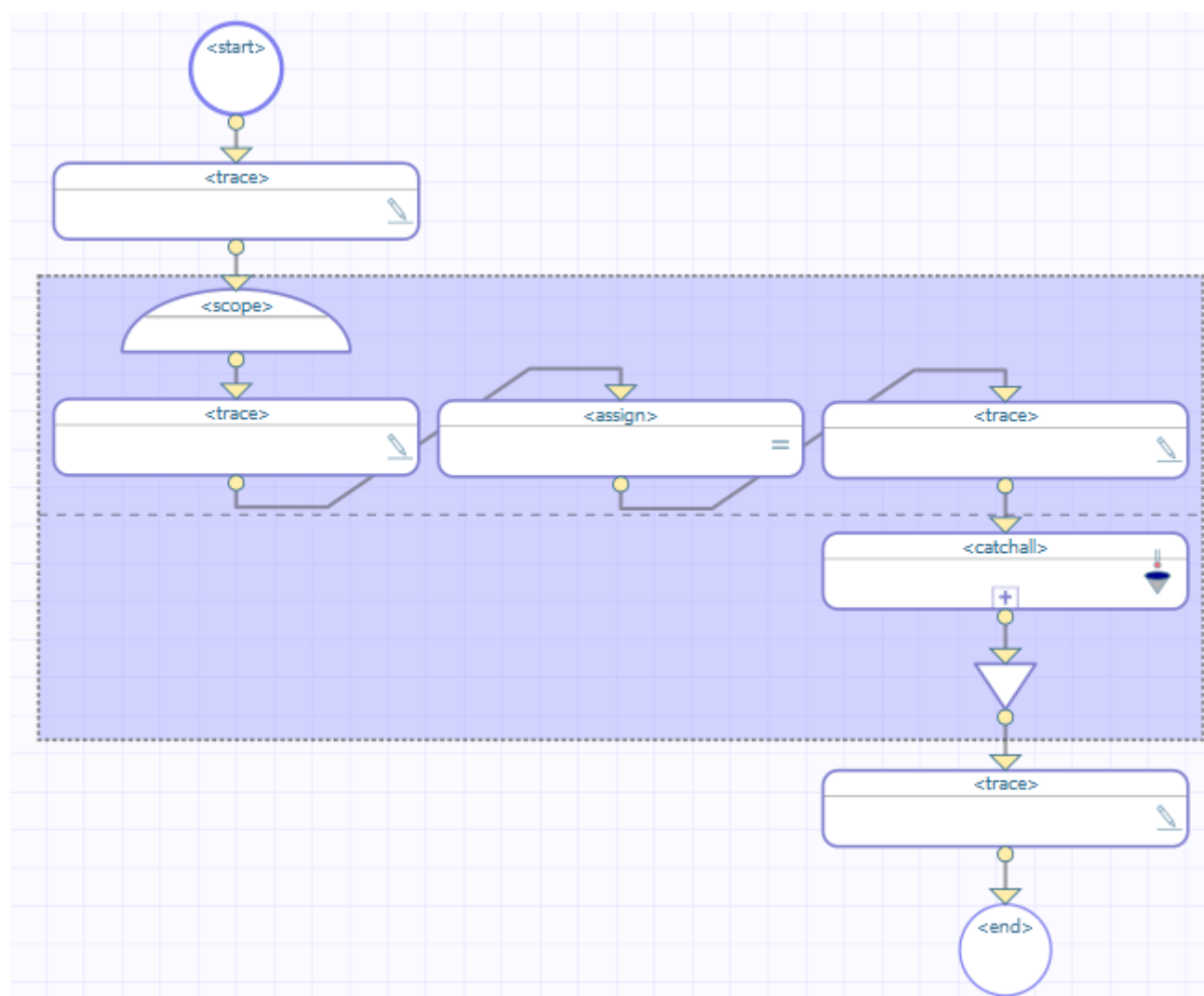
Class Member

```
XData BPL
{
  <process language='objectscript'
    request='Test.Scope.Request'
    response='Test.Scope.Response' >
    <sequence>
      <trace value='before assign' />
      <assign property="SomeProperty" value="1/0" />
      <trace value='after assign' />
    </sequence>
  </process>
}
```

8.2 System Error with Catchall

To enable error handling, BPL defines an element called `<scope>`. A scope is a wrapper for a set of activities. This scope may contain one or more activities, one or more fault handlers, and zero or more compensation handlers. The fault handlers are intended to catch any errors that activities within the `<scope>` produce. The fault handlers may invoke compensation handlers to compensate for those errors.

The following example provides a `<scope>` with a `<faulthandlers>` block that includes a `<catchall>`. Because the `<scope>` includes a `<faulthandlers>` element, the rectangle includes a horizontal dashed line across the middle; the area below this line displays the contents of the `<faulthandlers>`.

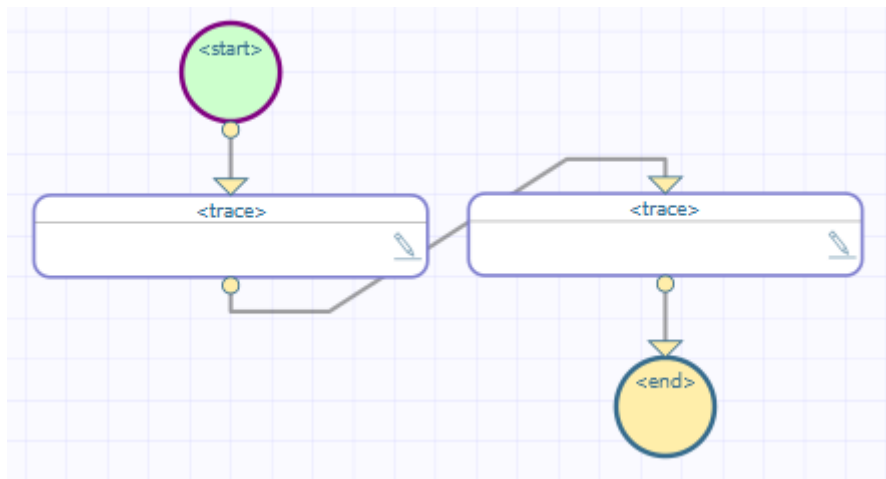


This BPL business process does the following:

1. The first `<trace>` element generates the message before `scope`.
2. The `<scope>` element starts the scope.
3. The second `<trace>` element generates the message before `assign`.
4. The `<assign>` element tries to evaluate the expression `1/0`. This attempt produces a divide-by-zero system error.
5. Control now goes to the `<faulthandlers>` defined within the `<scope>`. The `<scope>` rectangle includes a horizontal dashed line across the middle; the area below this dashed line displays the contents of the `<faulthandlers>` element. In this case, there is no `<catch>`, but there is a `<catchall>` element, so control goes there.

Note that InterSystems IRIS® skips the `<trace>` element message immediately after the `<assign>` element.

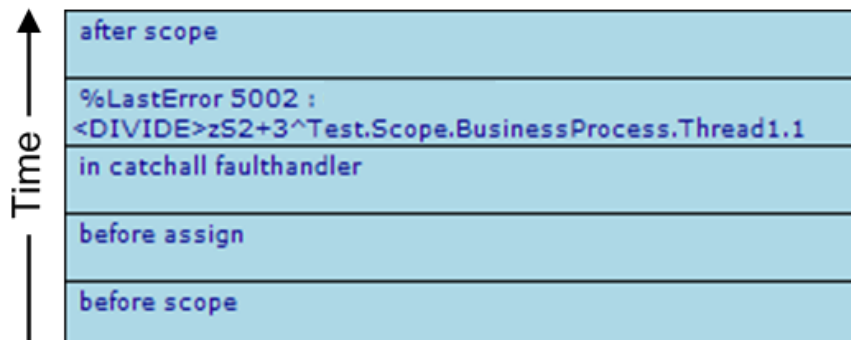
If we drill down into `<catchall>`, we see this:



6. Within `<catchall>`, a `<trace>` element generates the message in `catchall` fault handler.
7. Within `<catchall>`, another `<trace>` element generates a message that explores the nature of the error using `System.Status` methods and the special variables `%Context` and `%LastError`. See the details in [Event Log Entries](#).
8. The `<scope>` ends.
9. The last `<trace>` element generates the message after `scope`.

8.2.1 Event Log Entries

The corresponding Event Log entries look like this.



If an unexpected system error occurs, and a `<faulthandlers>` block is present inside a `<scope>`, the BPL business process does *not* automatically place entries in the Event Log as shown in the [System Error with No Fault Handling](#) example. Rather, the `<faulthandlers>` block determines what the business process will do. In the current example, it outputs a `<trace>` message that contains information about the error. The Event Log entry showing the actual error is produced by the following statement within the `<catchall>` block:

XML

```
<trace value=
  '%LastError ' _
  '$System.Status.GetErrorCodes(..%Context.%LastError)_
  ' : ' _
  '$System.Status.GetOneStatusText(..%Context.%LastError)'
/>
```

The BPL context variable `%LastError` always contains a `%Status` value. If the error was an unexpected system error such as `<UNDEF>` this `%Status` value is created from the error “ObjectScript error” which has code 5002, and the text of the

`$ZERROR` special variable. To get the corresponding error code and text out of `%LastError`, use the `$System.Status` methods `GetErrorCodes` and `GetOneStatusText`, then concatenate them into a `<trace>` string, as shown above.

8.2.2 XData for This BPL

This BPL is defined by the following XData block:

Class Member

```
XData BPL
{
<process language='objectscript'
  request='Test.Scope.Request'
  response='Test.Scope.Response' >
  <sequence>
    <trace value='before scope' />
    <scope>
      <trace value='before assign' />
      <assign property='SomeProperty' value='1/0' />
      <trace value='after assign' />
      <faulthandlers>
        <catchall>
          <trace value='in catchall faulthandler' />
          <trace value=
            '%LastError' _
            '$System.Status.GetErrorCodes(..%Context.%LastError)_
            ' : '_
            '$System.Status.GetOneStatusText(..%Context.%LastError)'
          />
        </catchall>
      </faulthandlers>
    </scope>
    <trace value='after scope' />
  </sequence>
</process>
}
```

8.3 Thrown Fault with Catchall

When a `<throw>` statement executes, its *fault* value is an expression that evaluates to a string. Faults are not objects, as in other object-oriented languages such as Java; they are string values. When you specify a fault string it *needs* the extra set of quotes to contain it, as shown below:

XML

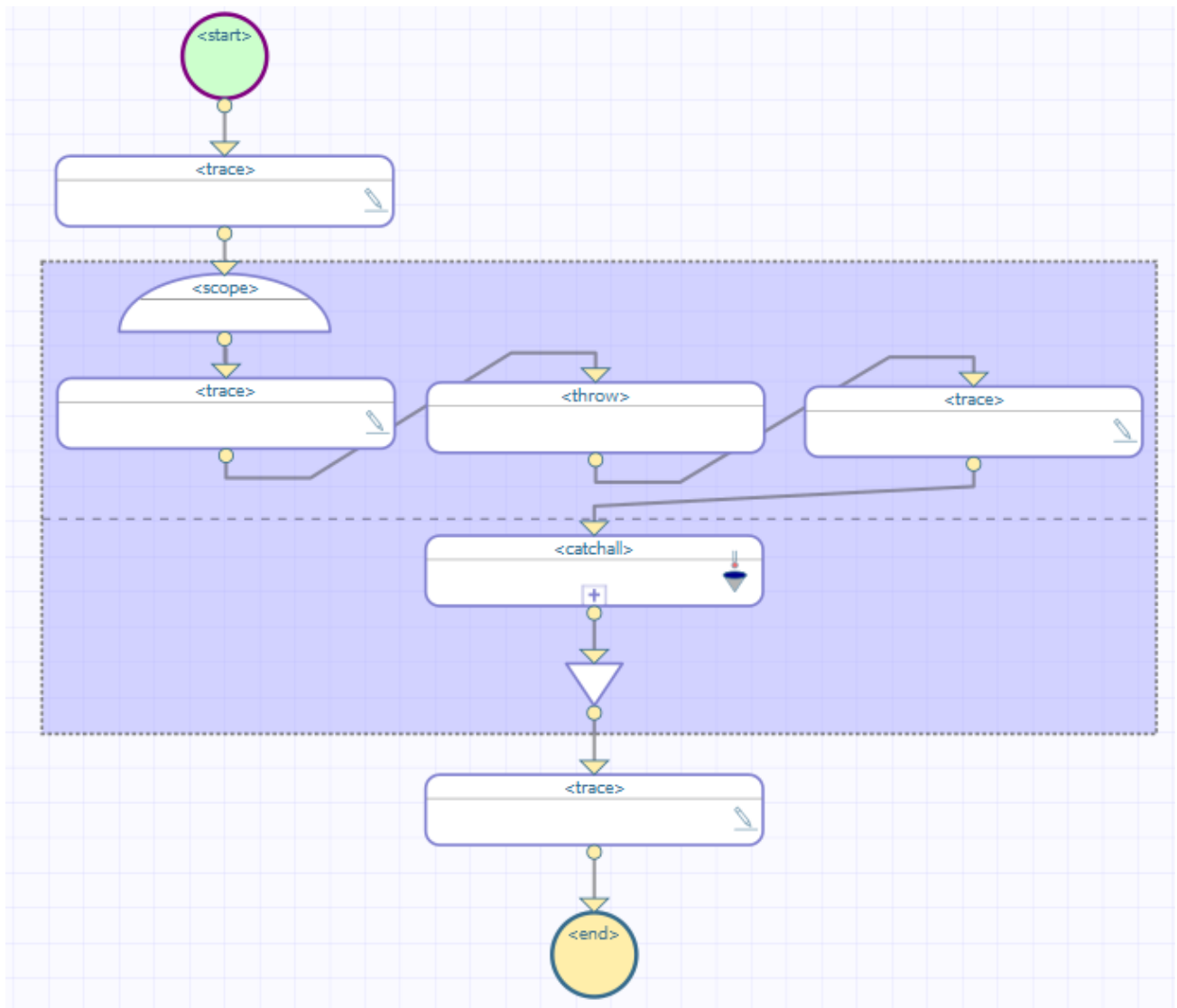
```
<throw fault='"thrown"' />
```

When a `<throw>` statement executes, control immediately goes to the `<faulthandlers>` block inside the same `<scope>`, skipping all intervening statements after the `<throw>`. Inside the `<faulthandlers>` block, the program attempts to find a `<catch>` block whose *value* attribute matches the *fault* string expression in the `<throw>` statement. This comparison is case-sensitive.

If there is a `<catch>` block that matches the fault, the program executes the code within this `<catch>` block and then exits the `<scope>`. The program resumes execution at the next statement following the closing `</scope>` element.

If a fault is thrown, and the corresponding `<faulthandlers>` block contains *no* `<catch>` block that matches the fault string, control goes from the `<throw>` statement to the `<catchall>` block inside `<faulthandlers>`. After executing the contents of the `<catchall>` block, the program exits the `<scope>`. The program resumes execution at the next statement following the closing `</scope>` element. It is good programming practice to ensure that there is always a `<catchall>` block inside every `<faulthandlers>` block, to ensure that the program catches any unanticipated errors.

Suppose you have the following BPL. For reasons of space, the `<start>` and `<end>` elements are not shown.

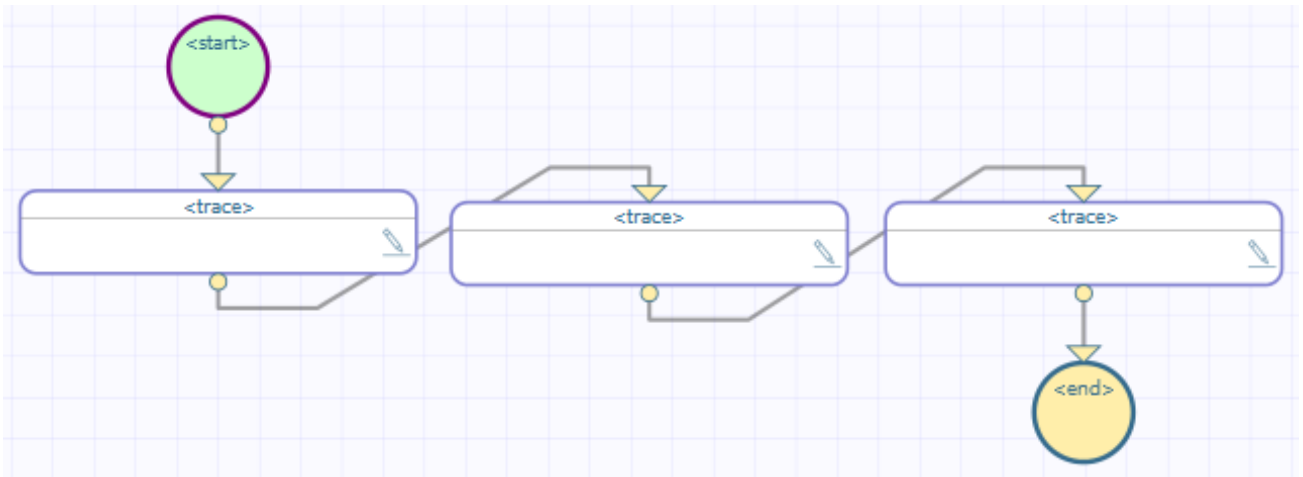


This BPL business process does the following:

1. The first `<trace>` element generates the message before `scope`.
2. The `<scope>` element starts the scope.
3. The second `<trace>` element generates the message before `assign`.
4. The `<throw>` element throws a specific, named fault ("MyFault").
5. Control now goes to the `<faulthandlers>` defined within the `<scope>`. The `<scope>` rectangle includes a horizontal dashed line across the middle; the area below this dashed line displays the contents of the `<faulthandlers>` element. In this case, there is no `<catch>` but there is a `<catchall>` element, so control goes there.

Note that InterSystems IRIS skips the third `<trace>` element.

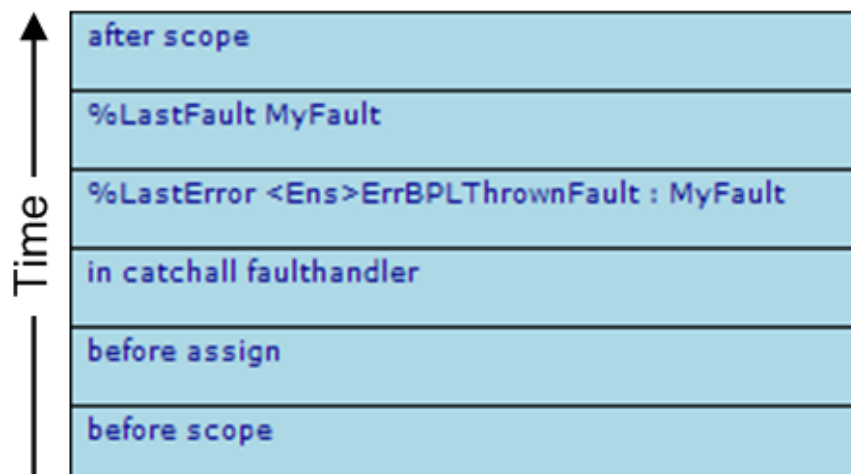
If we drill down into `<catchall>`, we see this:



6. Within `<catchall>`, the first `<trace>` element generates the message in `catchall` `faulthandler`.
7. Within `<catchall>`, the second `<trace>` element generates the message that provides information on the fault using `$System.Status` methods and the special variables `%Context` and `%LastError`. The `%LastError` value as the result of a thrown fault is different from its value as the result of a system error:
 - **GetErrorCodes** returns `<Ens>ErrBPLThrownFault`
 - **GetOneStatusText** returns text derived from the *fault* expression in the `<throw>` statement
8. Within `<catchall>`, the third `<trace>` element generates a message that provides information on the fault using the BPL context variable `%LastFault`. It contains the text derived from the *fault* expression from the `<throw>` statement.
9. The `<scope>` ends.
10. The last `<trace>` element generates the message after `scope`.

8.3.1 Event Log Entries

The corresponding Event Log entries look like this:



8.3.2 XData for This BPL

This BPL is defined by the following XData block:

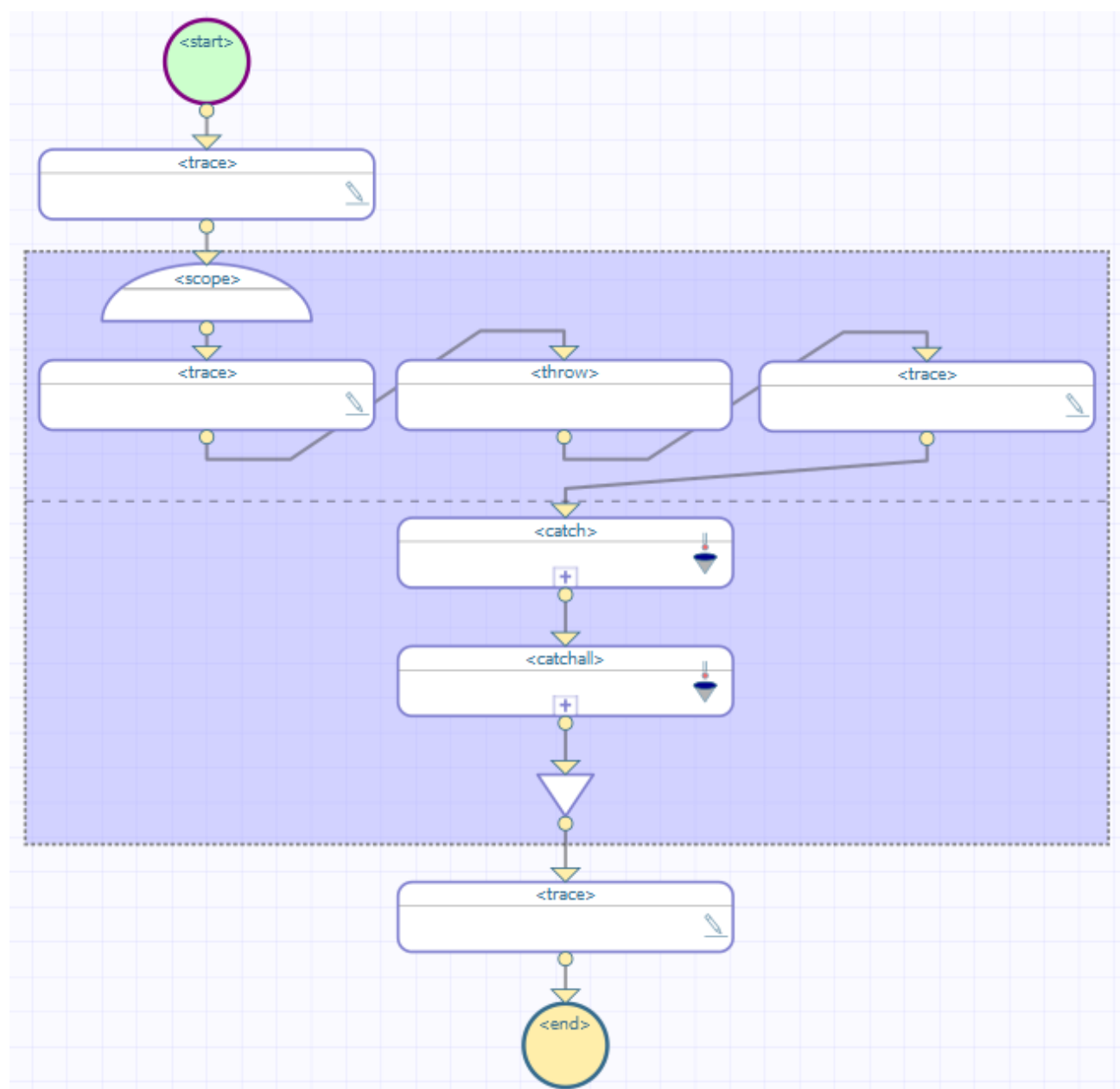
Class Member

```
XData BPL
{
<process language='objectscript'
    request='Test.Scope.Request'
    response='Test.Scope.Response' >
    <sequence>
        <trace value='before scope' />
        <scope>
            <trace value='before assign' />
            <throw fault='MyFault' />
            <trace value='after assign' />
            <faulthandlers>
                <catchall>
                    <trace value='in catchall fault handler' />
                    <trace value=
                        '%LastError' _
                        '$System.Status.GetErrorCodes(..%Context.%LastError)_
                        " : "_
                        '$System.Status.GetOneStatusText(..%Context.%LastError)'
                    />
                    <trace value='"%LastFault" _..%Context.%LastFault' />
                </catchall>
            </faulthandlers>
        </scope>
        <trace value='after scope' />
    </sequence>
</process>
}
```

8.4 Thrown Fault with Catch

A thrown fault may reach a <catchall>, as in the previous example, or it may have a specific <catch>.

Suppose you have the following BPL:

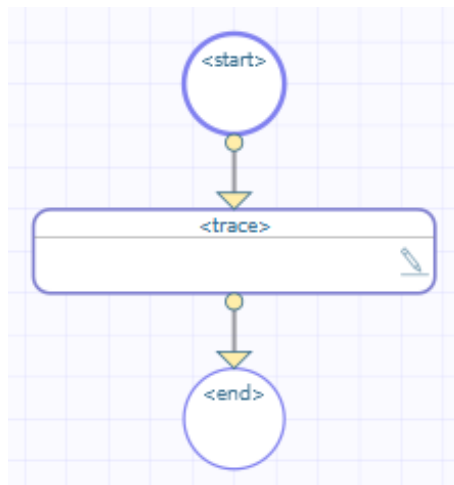


This BPL business process does the following:

1. The first `<trace>` element generates the message before `scope`.
2. The `<scope>` element starts the scope.
3. The second `<trace>` element generates the message before `throw`.
4. The `<throw>` element throws a specific, named fault (`"MyFault"`).
5. Control now goes to the `<faulthandlers>` defined within the `<scope>`. The `<scope>` rectangle includes a horizontal dashed line across the middle; the area below this dashed line displays the contents of the `<faulthandlers>` element. In this case, a `<catch>` element exists whose *fault* value is `"MyFault"`, so control goes there. The `<catchall>` element is ignored.

Note that InterSystems IRIS skips the `<trace>` element message after the `<throw>` element.

If we drill down into `<catch>`, we see this:

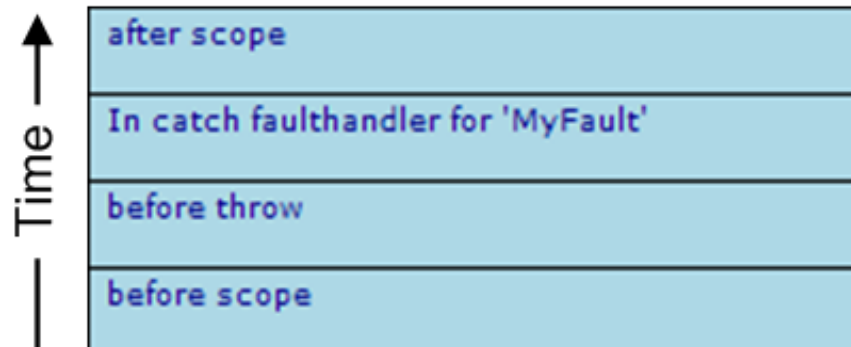


Note: If a `<catchall>` is provided, it must be the last statement in the `<faulthandlers>` block. All `<catch>` blocks must appear before `<catchall>`.

6. Within `<catch>`, the `<trace>` element generates the message in `catch` faulthandler for `'MyFault'`.
7. The `<scope>` ends.
8. The last `<trace>` element generates the message after `scope`.

8.4.1 Event Log Entries

The corresponding Event Log entries look like this:



8.4.2 XData for This BPL

This BPL is defined by the following XData block:

Class Member

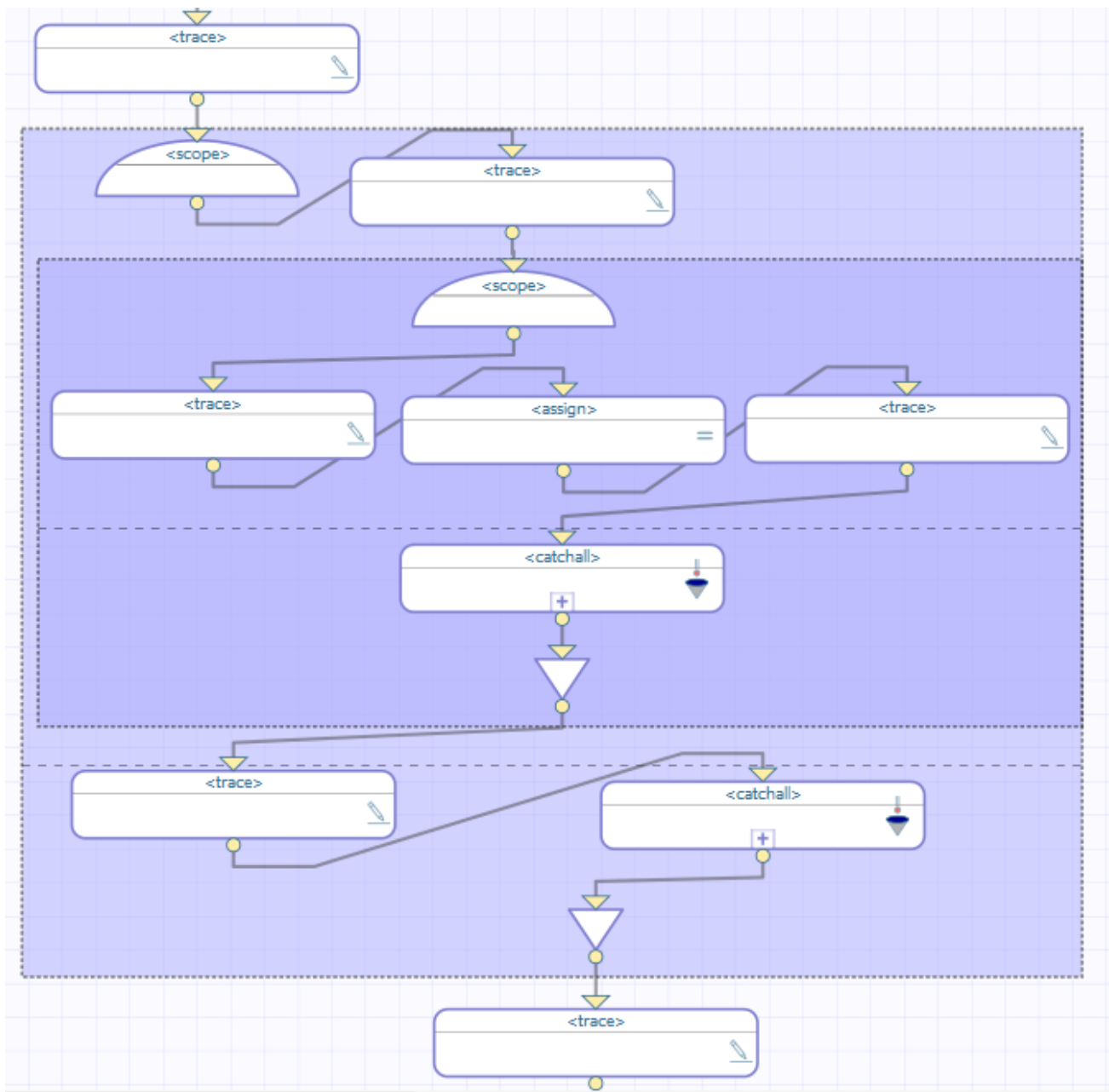
```
XData BPL
{
<process language='objectscript'
  request='Test.Scope.Request'
  response='Test.Scope.Response' >
  <sequence>
    <trace value='before scope' />
    <scope>
      <trace value='before throw' />
      <throw fault='MyFault' />
      <trace value='after throw' />
    <faulthandlers>
      <catch fault='MyFault' />
    </faulthandlers>
  </scope>
  </sequence>
</process>
```

```
    <trace value='In catch fault handler for &apos;MyFault&apos;' />
  </catch>
  <catchall>
    <trace value='in catchall fault handler' />
    <trace value=
      '%LastError' _
      '$System.Status.GetErrorCodes(..%Context.%LastError)_
      " : "_
      '$System.Status.GetOneStatusText(..%Context.%LastError)'
    />
    <trace value='%LastFault' '..%Context.%LastFault' />
  </catchall>
</faulthandlers>
</scope>
<trace value='after scope' />
</sequence>
</process>
}
```

8.5 Nested Scopes, Inner Fault Handler Has Catchall

It is possible to nest `<scope>` elements. An error or fault that occurs within the inner scope may be caught within the inner scope, or the inner scope may ignore the error and allow it to be caught by the `<faulthandlers>` block in the outer scope. The next several topics illustrate how BPL handles errors and faults that occur within an inner scope, when two or more scopes are nested.

Suppose you have the following BPL (shown here without the `<start>` and `<end>` elements):

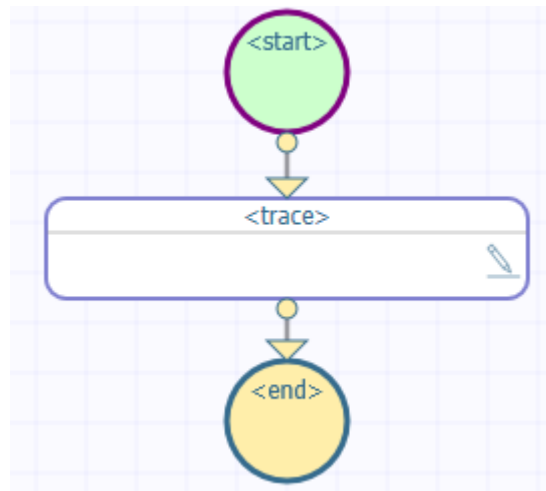


This BPL business process does the following:

1. The first `<trace>` element generates the message before outer scope.
2. The first `<scope>` element starts the outer scope.
3. The second `<trace>` element generates the message in outer scope, before inner scope.
4. The second `<scope>` element starts the inner scope.
5. The next `<trace>` element generates the message in inner scope, before assign.
6. The `<assign>` element tries to evaluate the expression `1/0`. This attempt produces a divide-by-zero system error.
7. Control now goes to the `<faulthandlers>` defined within the inner `<scope>`. This `<scope>` rectangle includes a horizontal dashed line across the middle; the area below this dashed line displays the contents of the `<faulthandlers>` element. In this case, there is no `<catch>` but there is a `<catchall>`, so control goes there.

Note that InterSystems IRIS skips the `<trace>` element immediately after the `<assign>` element.

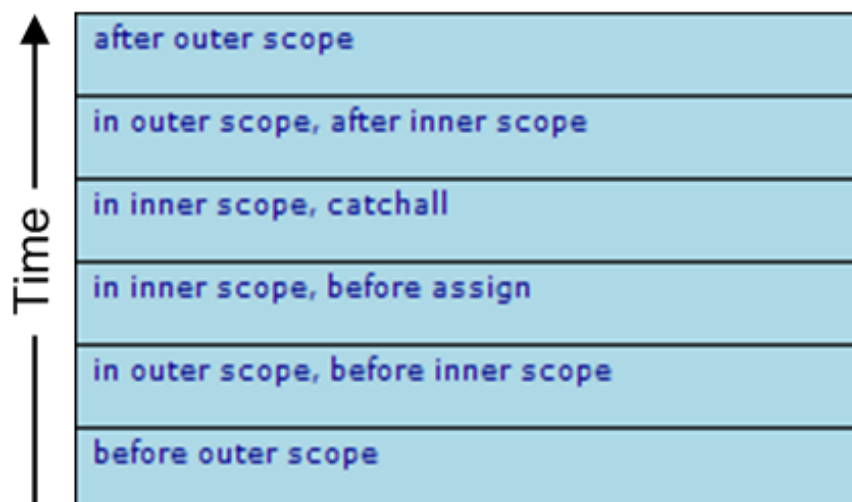
If we drill into this `<catchall>`, we see this:



8. Within this `<catchall>`, the `<trace>` element generates the message in `inner scope, catchall`.
9. The inner `<scope>` ends.
10. The next `<trace>` element generates the message in `outer scope, after inner scope`.
11. The outer `<scope>` rectangle includes a horizontal dashed line across the middle; the area below this dashed line displays the contents of the `<faulthandlers>` element that contains a `<catchall>`. Because there is no fault, this `<catchall>` is ignored.
12. The outer `<scope>` ends.
13. The last `<trace>` element generates the message after `outer scope`.

8.5.1 Event Log Entries

The corresponding Event Log entries look like this:



8.5.2 XData for This BPL

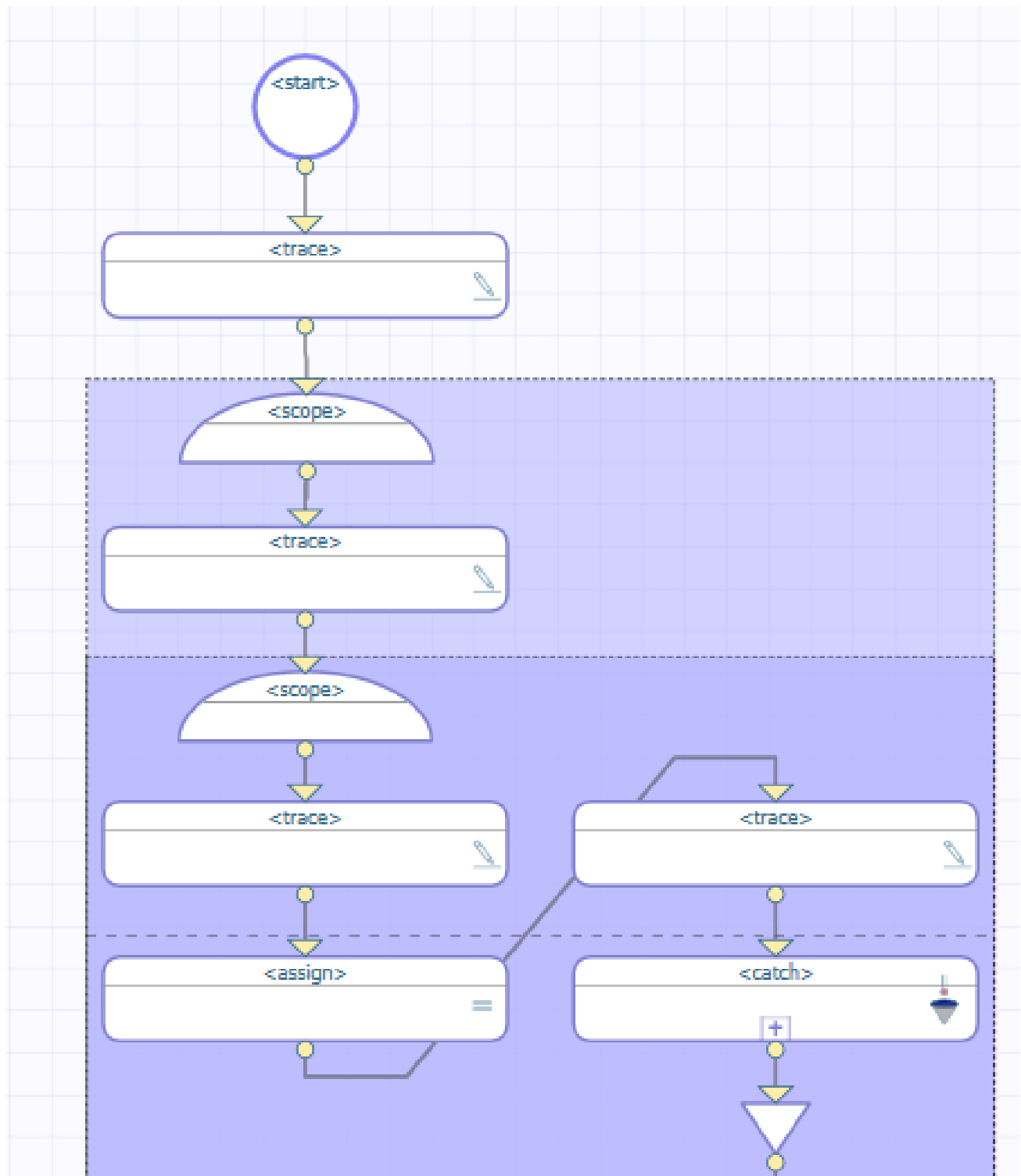
This BPL is defined by the following XData block:

Class Member

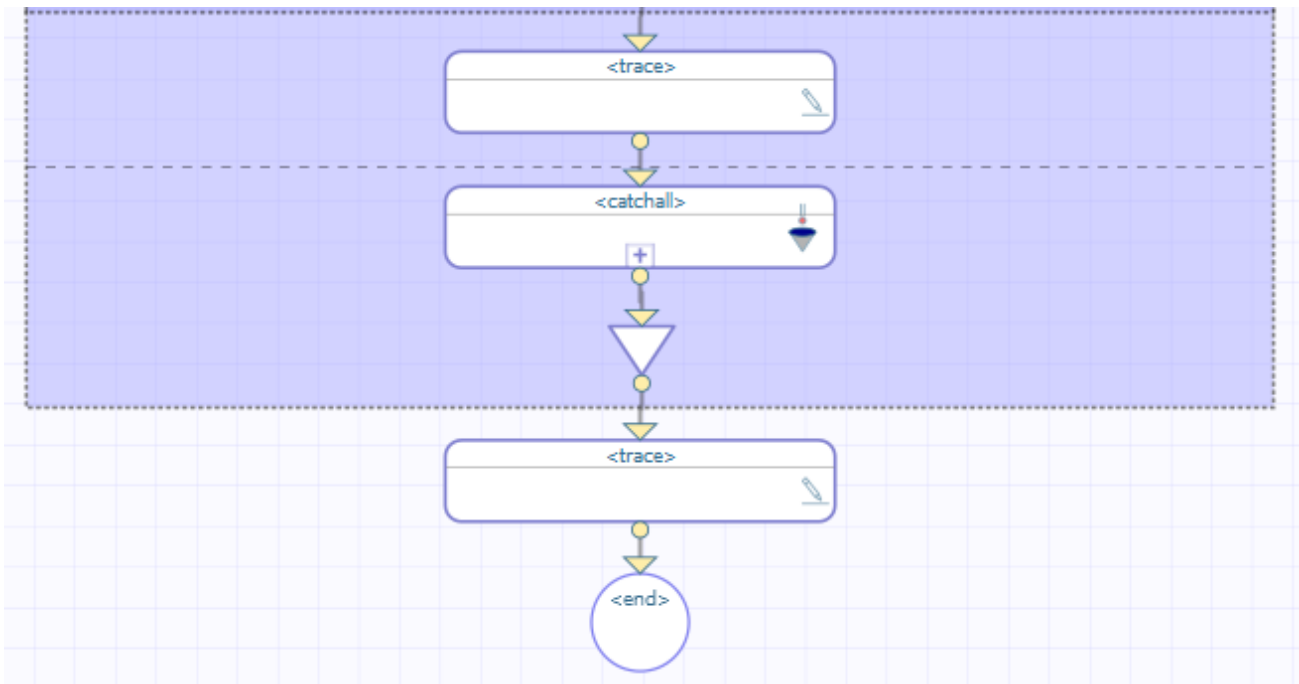
```
XData BPL
{
<process language='objectscript'
    request='Test.Scope.Request'
    response='Test.Scope.Response' >
    <sequence>
        <trace value='before outer scope'/'>
        <scope>
            <trace value='in outer scope, before inner scope'/'>
            <scope>
                <trace value='in inner scope, before assign'/'>
                <assign property='SomeProperty' value='1/0'/'>
                <trace value='in inner scope, after assign'/'>
                <faulthandlers>
                    <catchall>
                        <trace value='in inner scope, catchall'/'>
                    </catchall>
                </faulthandlers>
            </scope>
        <trace value='in outer scope, after inner scope'/'>
        <faulthandlers>
            <catchall>
                <trace value='in outer scope, catchall'/'>
            </catchall>
        </faulthandlers>
    </scope>
    <trace value='after outer scope'/'>
    </sequence>
</process>
}
```

8.6 Nested Scopes, Outer Fault Handler Has Catchall

Suppose you have the following BPL (partially shown):



The rest of this BPL is as follows:



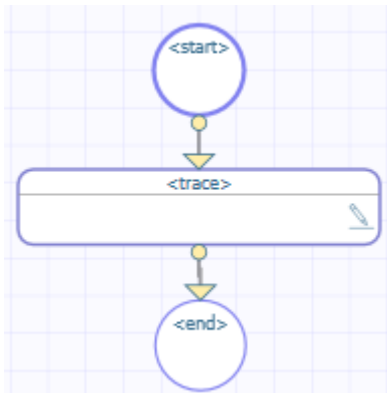
This BPL business process does the following:

1. The first `<trace>` element generates the message before outer scope.
2. The first `<scope>` element starts the outer scope.
3. The next `<trace>` element generates the message in outer scope, before inner scope.
4. The second `<scope>` element starts the inner scope.
5. The next `<trace>` element generates the message in inner scope, before assign.
6. The `<assign>` element tries to evaluate the expression `1/0`. This attempt produces a divide-by-zero system error.
7. Control now goes to the `<faulthandlers>` defined within the inner `<scope>`. This `<scope>` rectangle includes a horizontal dashed line across the middle; the area below this dashed line displays the contents of the `<faulthandlers>` element. In this case, a `<catch>` exists, but its *fault* value does not match the thrown fault. There is no `<catchall>` in the inner scope.

Note that InterSystems IRIS skips the `<trace>` element that is immediately after `<assign>`.

8. Control now goes to the `<faulthandlers>` block in the outer `<scope>`. No `<catch>` matches the fault, but there *is* a `<catchall>` block. Control goes to this `<catchall>`.

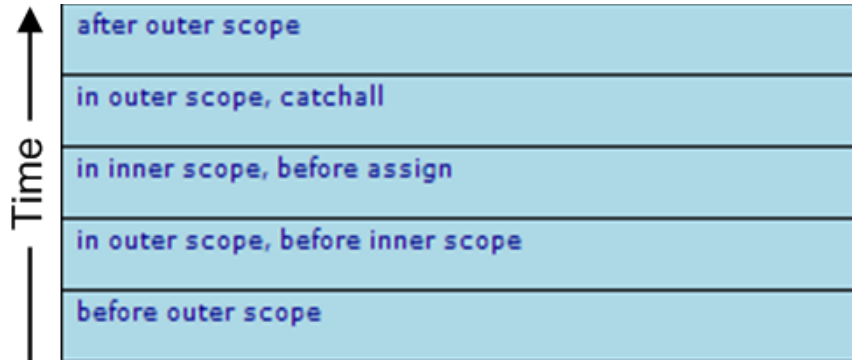
If we drill into this `<catchall>`, we see this:



9. Within this <catchall>, the <trace> element generates the message in outer scope, catchall.
10. The outer <scope> ends.
11. The last <trace> element generates the message after outer scope.

8.6.1 Event Log Entries

The corresponding Event Log entries look like this:



8.6.2 XData for This BPL

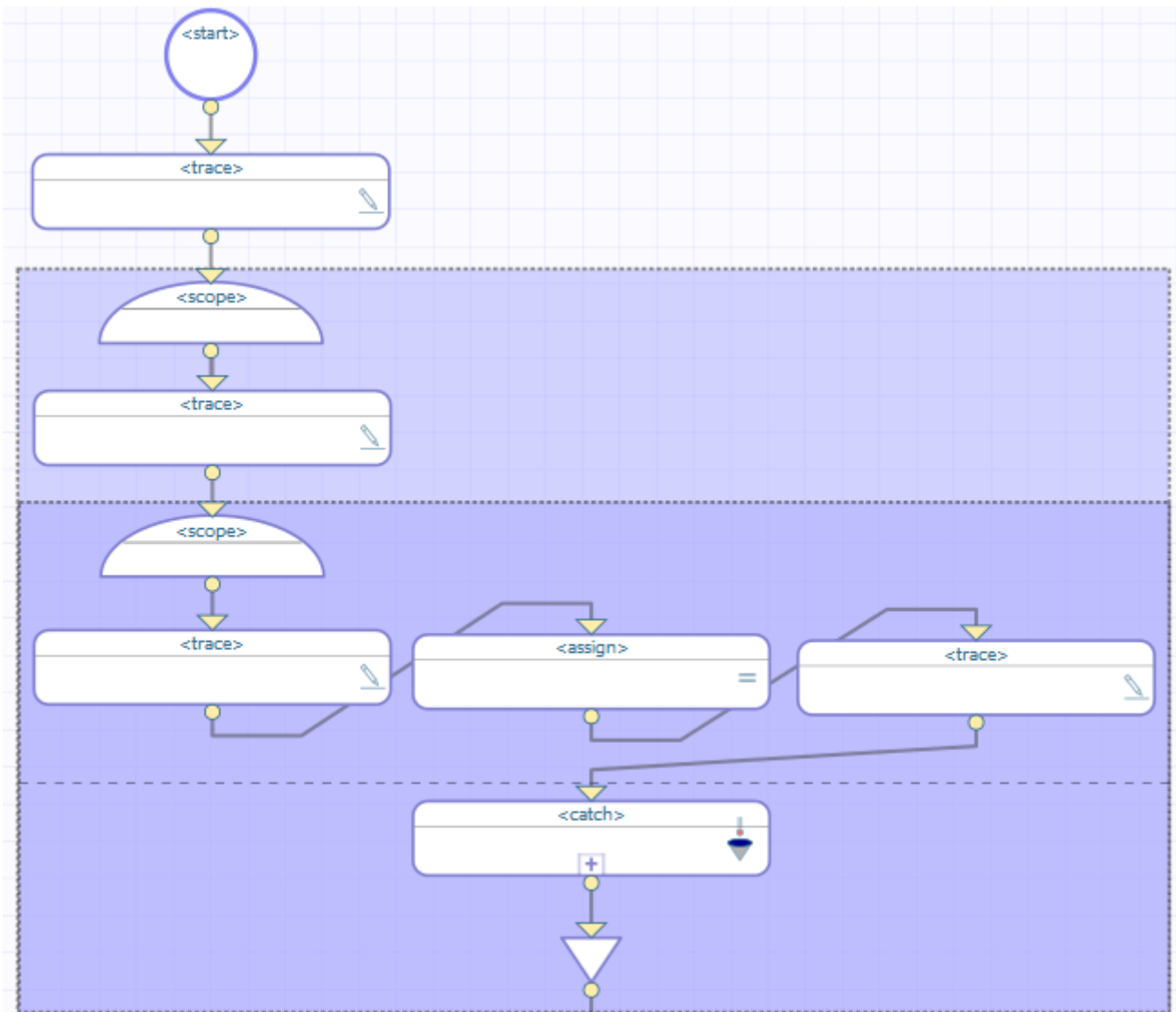
This BPL is defined by the following XData block:

Class Member

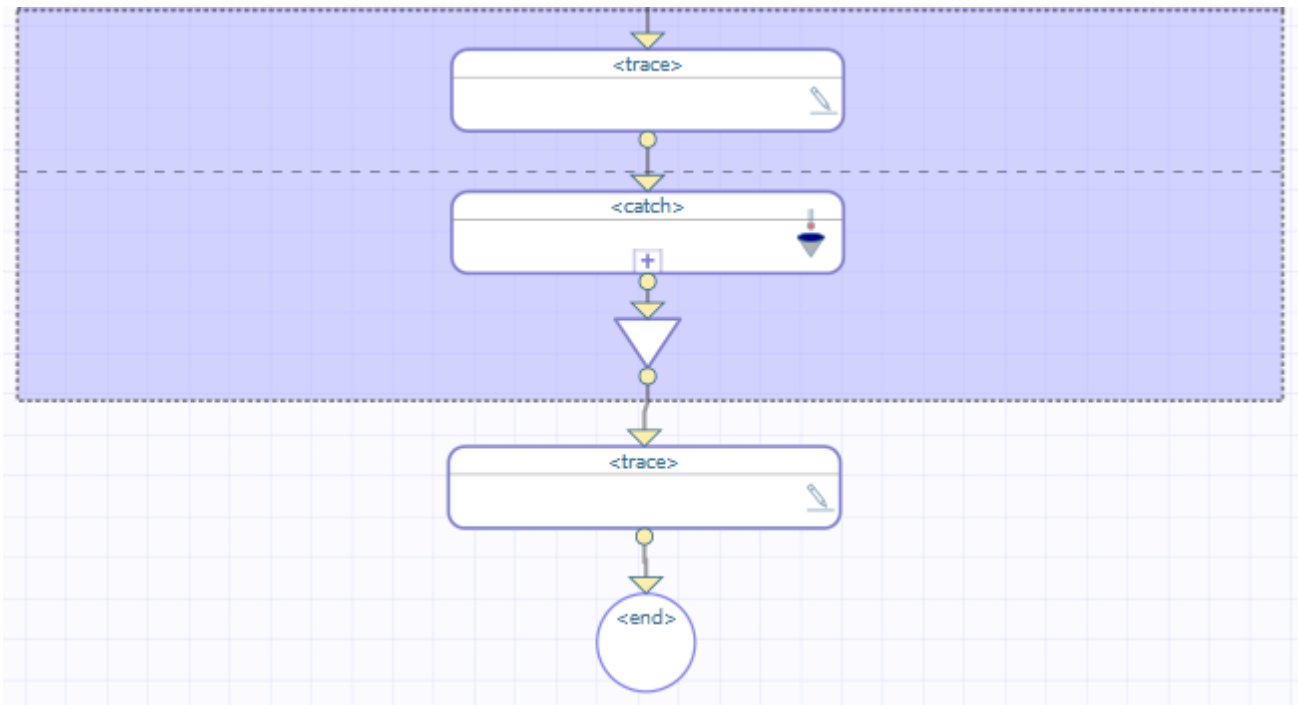
```
XData BPL
{
<process language='objectscript'
  request='Test.Scope.Request'
  response='Test.Scope.Response' >
  <sequence>
    <trace value='"before outer scope"'/>
    <scope>
      <trace value='"in outer scope, before inner scope"'/>
      <scope>
        <trace value='"in inner scope, before assign"'/>
        <assign property="SomeProperty" value="1/0"/>
        <trace value='"in inner scope, after assign"'/>
        <faulthandlers>
          <catch fault='"MismatchedFault"'>
            <trace value=
              '"In catch fault handler for &apos;MismatchedFault&apos;"'/>
          </catch>
        </faulthandlers>
      </scope>
      <trace value='"in outer scope, after inner scope"'/>
      <faulthandlers>
        <catchall>
          <trace value='"in outer scope, catchall"'/>
        </catchall>
      </faulthandlers>
    </scope>
    <trace value='"after outer scope"'/>
  </sequence>
</process>
}
```

8.7 Nested Scopes, No Match in Either Scope

Suppose you have the following BPL (partially shown):



The rest of this BPL is as follows:

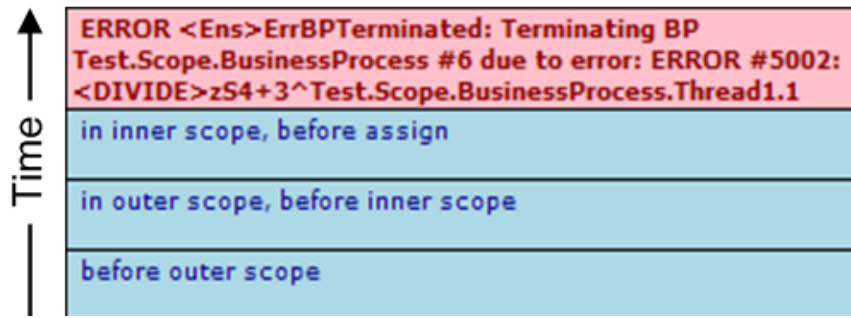


This BPL business process does the following:

1. The first `<trace>` element generates the message before `outer scope`.
2. The first `<scope>` element starts the outer scope.
3. The next `<trace>` element generates the message in `outer scope`, before `inner scope`.
4. The second `<scope>` element starts the inner scope.
5. The next `<trace>` element generates the message in `inner scope`, before `assign`.
6. The `<assign>` element tries to evaluate the expression `1/0`. This attempt produces a divide-by-zero system error.
7. Control now goes to the `<faulthandlers>` block in the inner `<scope>`. The `<scope>` rectangle includes a horizontal dashed line across the middle; the area below this dashed line displays the contents of the `<faulthandlers>` element. In this case, a `<catch>` exists, but its *fault* value does not match the thrown fault. There is no `<catchall>` in the inner scope.
8. Control now goes to the `<faulthandlers>` block in the outer `<scope>`. No `<catch>` matches the fault, and there is no `<catchall>` block.
9. The BPL immediately stops, sending a message to the Event Log.

8.7.1 Event Log Entries

The corresponding Event Log entries look like this.



There is an important difference between this Event Log and the one in the [System Error with No Fault Handling](#) example. The two examples have this in common: Each fails to provide adequate fault handling for the case when the divide-by-zero error occurs.

The difference is that the [System Error with No Fault Handling](#) example has no `<scope>` and no `<faulthandlers>` block. Under these circumstances, InterSystems IRIS automatically outputs the system error to the Event Log, as shown in the first example.

The current example is different because each `<scope>` *does* include a `<faulthandlers>` block. Under these circumstances, InterSystems IRIS does not automatically output the system error to the Event Log, as it did in the [System Error with No Fault Handling](#) example. It is up to the BPL business process developer to decide to output `<trace>` messages to the Event Log in case of an unexpected error. In the current example, no `<faulthandlers>` block catches the fault, so the only information that is traced regarding the system error is contained in the automatic message about business process termination (item 4 above).

The system error message does appear in the ObjectScript shell:

```
ERROR #5002: ObjectScript error: <DIVIDE>zS4+3^Test.Scope.BusinessProcess.Thread1.1
```

8.7.2 XData for This BPL

This BPL is defined by the following XData block:

Class Member

```
XData BPL
{
<process language='objectscript'
  request='Test.Scope.Request'
  response='Test.Scope.Response' >
  <sequence>
    <trace value='"before outer scope"'/>
    <scope>
      <trace value='"in outer scope, before inner scope"'/>
      <scope>
        <trace value='"in inner scope, before assign"'/>
        <assign property="SomeProperty" value="1/0"/>
        <trace value='"in inner scope,after assign"'/>
        <faulthandlers>
          <catch fault='"MismatchedFault"'>
            <trace value=
              '"In catch fault handler for &apos;MismatchedFault&apos;"'/>
          </catch>
        </faulthandlers>
      </scope>
      <trace value='"in outer scope, after inner scope"'/>
    </scope>
    <faulthandlers>
      <catch fault='"MismatchedFault"'>
        <trace value=
          '"In catch fault handler for &apos;MismatchedFault&apos;"'/>
      </catch>
    </faulthandlers>
  </sequence>
  <trace value='"after outer scope"'/>
}
```

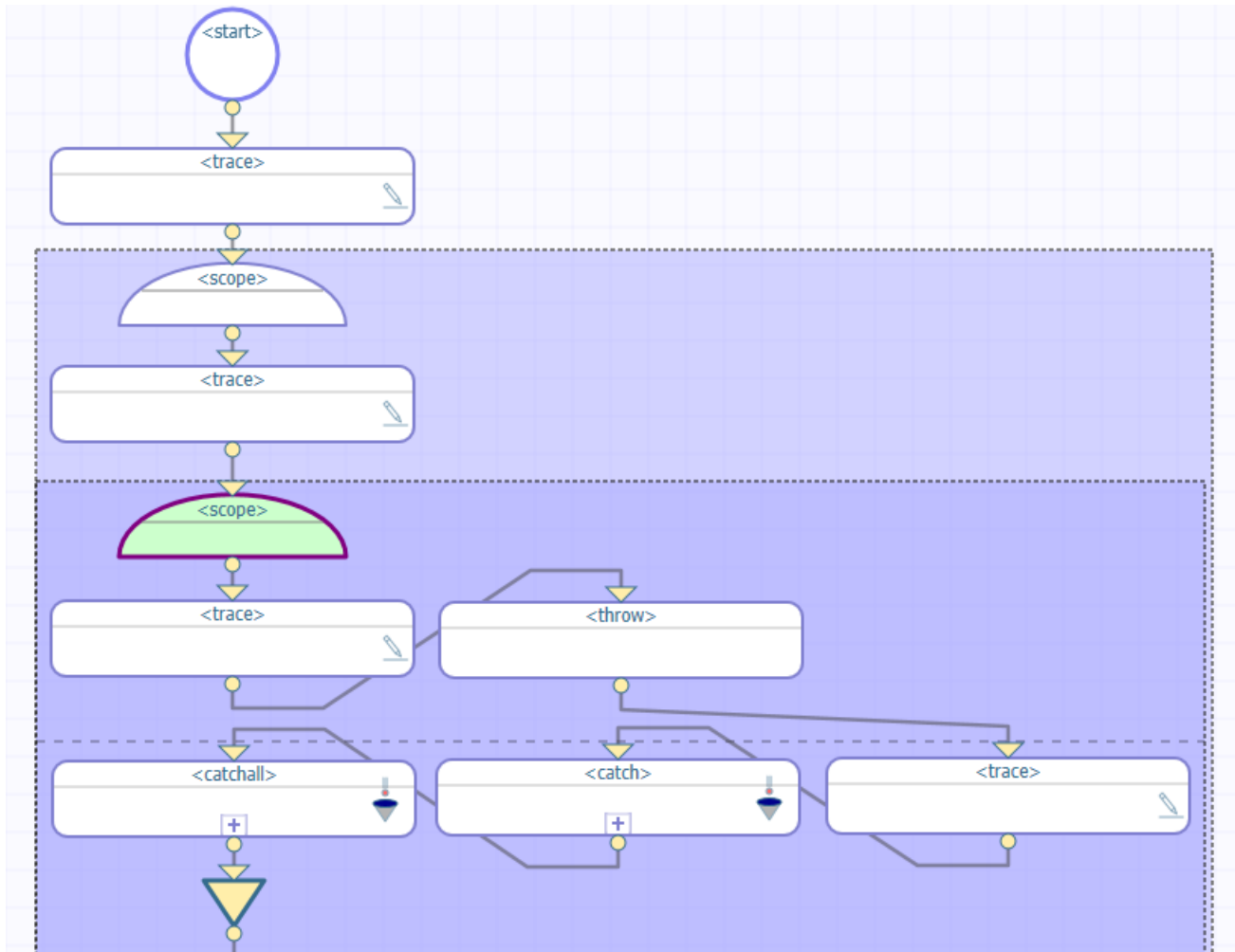
```

    </sequence>
  </process>
}

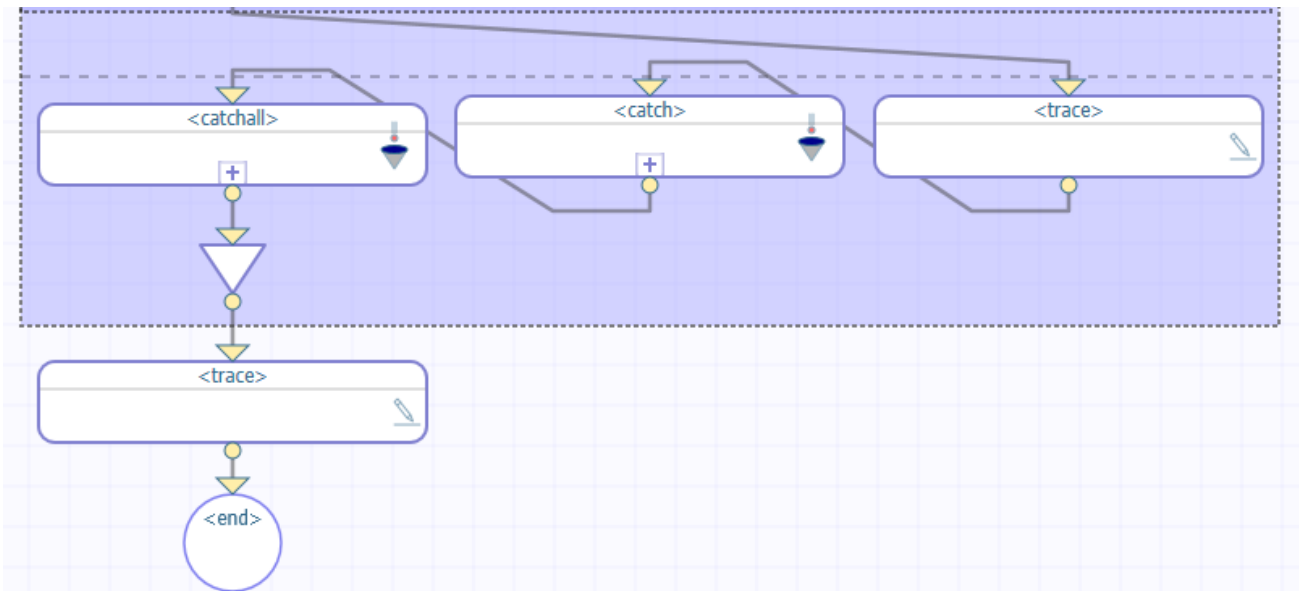
```

8.8 Nested Scopes, Outer Fault Handler Has Catch

Suppose you have the following BPL (partially shown):



The rest of this BPL is as follows:

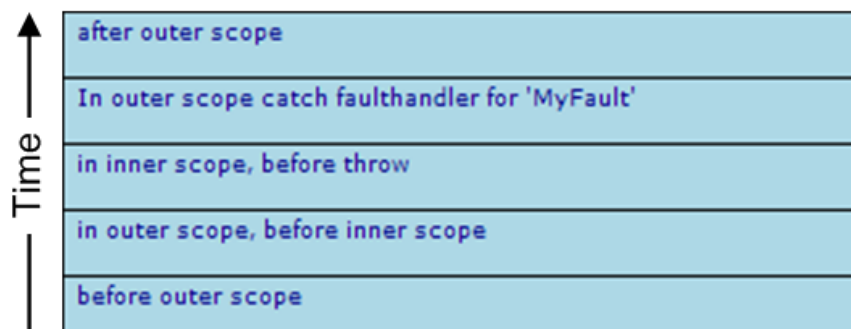


This BPL business process does the following:

1. The first `<trace>` element generates the message before outer scope.
2. The first `<scope>` element starts the outer scope.
3. The next `<trace>` element generates the message in outer scope, before inner scope.
4. The second `<scope>` element starts the inner scope.
5. The next `<trace>` element generates the message in inner scope, before throw.
6. The `<throw>` element throws a specific, named fault ("MyFault").
7. Control now goes to the `<faulthandlers>` defined within the inner `<scope>`. A `<catch>` exists, but its *fault* value is "MismatchedFault". There is no `<catchall>` in the inner scope.
8. Control goes to the `<faulthandlers>` block in the outer `<scope>`. It contains a `<catch>` whose *fault* value is "MyFault".
9. The next `<trace>` element generates the message in outer scope catch fault handler for 'MyFault'.
10. The second `<scope>` ends.
11. The last `<trace>` element generates the message after outer scope.

8.8.1 Event Log Entries

The corresponding Event Log entries look like this:



8.8.2 XData for This BPL

This BPL is defined by the following XData block:

Class Member

```
XData BPL
{
<process language='objectscript'
  request='Test.Scope.Request'
  response='Test.Scope.Response' >
  <sequence>
    <trace value="before outer scope"'/>
    <scope>
      <trace value="in outer scope, before inner scope"'/>
      <scope>
        <trace value="in inner scope, before throw"'/>
        <throw fault='MyFault''/>
        <trace value="in inner scope, after throw"'/>
        <faulthandlers>
          <catch fault='MismatchedFault'>
            <trace value=
'"In inner scope catch fault handler for &apos;MismatchedFault&apos;"'/>
            </catch>
          </faulthandlers>
        </scope>
        <trace value="in outer scope, after inner scope"'/>
        <faulthandlers>
          <catch fault='MyFault'>
            <trace value=
'"In outer scope catch fault handler for &apos;MyFault&apos;"'/>
            </catch>
          </faulthandlers>
        </scope>
        <trace value="after outer scope"'/>
      </sequence>
    </process>
  }
```

8.9 Thrown Fault with Compensation Handler

In business process management, it is often necessary to reverse some segment of logic. This convention is known as “compensation.” The ruling principle is that if the business process does something, it must be able to undo it. That is, if a failure occurs, the business process must be able to compensate by undoing the action that failed. You need to be able to unroll all of the actions from that failure point back to the beginning, as if the problem action never occurred. BPL enables this with a mechanism called a compensation handler.

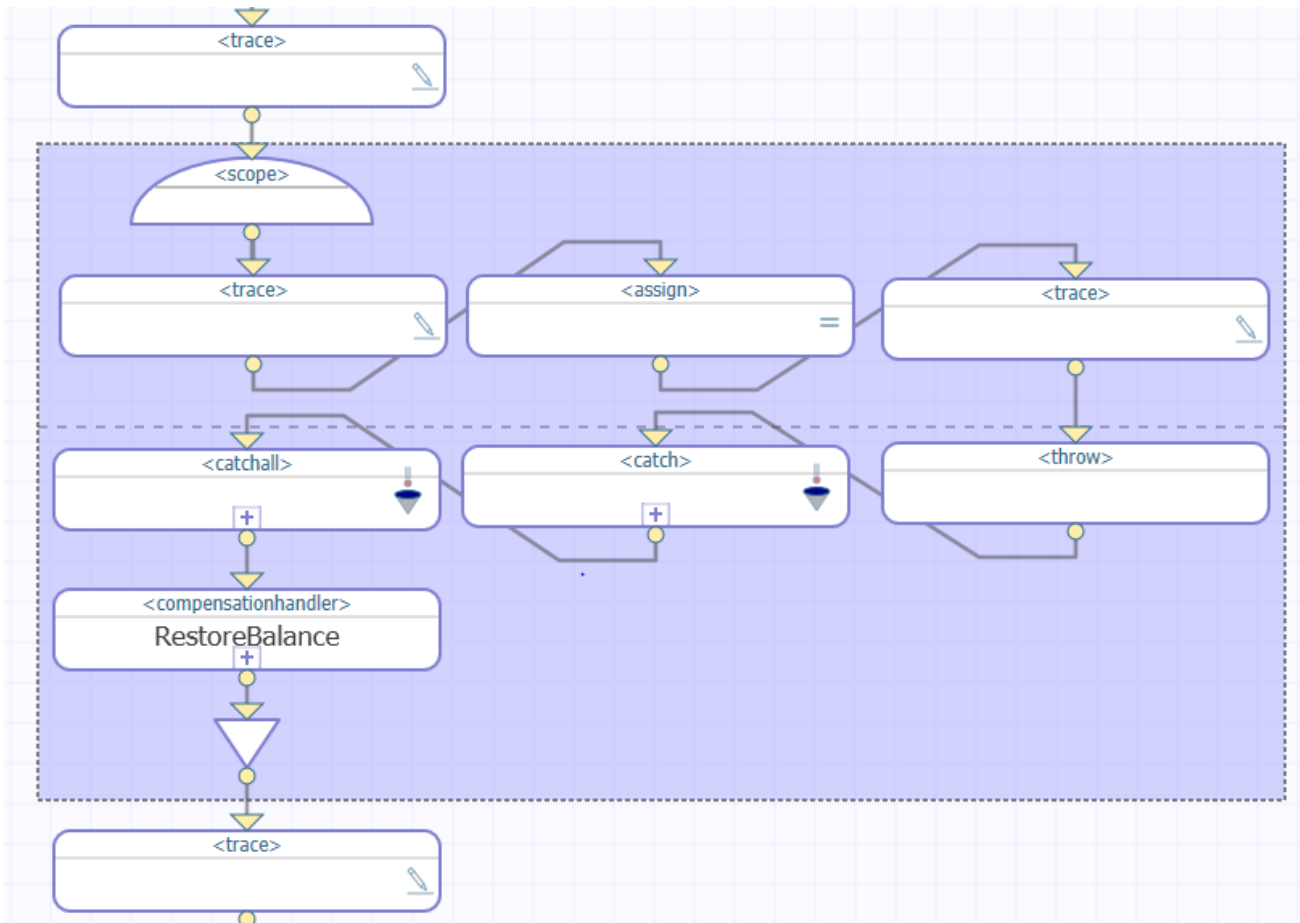
BPL `<compensationhandler>` blocks are somewhat like subroutines, but they do not provide a generalized subroutine mechanism. You can “call” them, but *only* from `<faulthandler>` blocks, and *only* within the same `<scope>` as the `<compensationhandler>` block. The `<compensate>` element invokes a `<compensationhandler>` block by specifying its name as a target. Extra quotes are *not* needed for this syntax:

XML

```
<compensate target="general"'/>
```

Compensation handlers are only useful if you *can* undo the actions already performed. For example, if you transfer money into the wrong account, you can transfer it back again, but there are some actions that cannot be neatly undone. You must plan compensation handlers accordingly, and also organize them according to how far you want to roll things back.

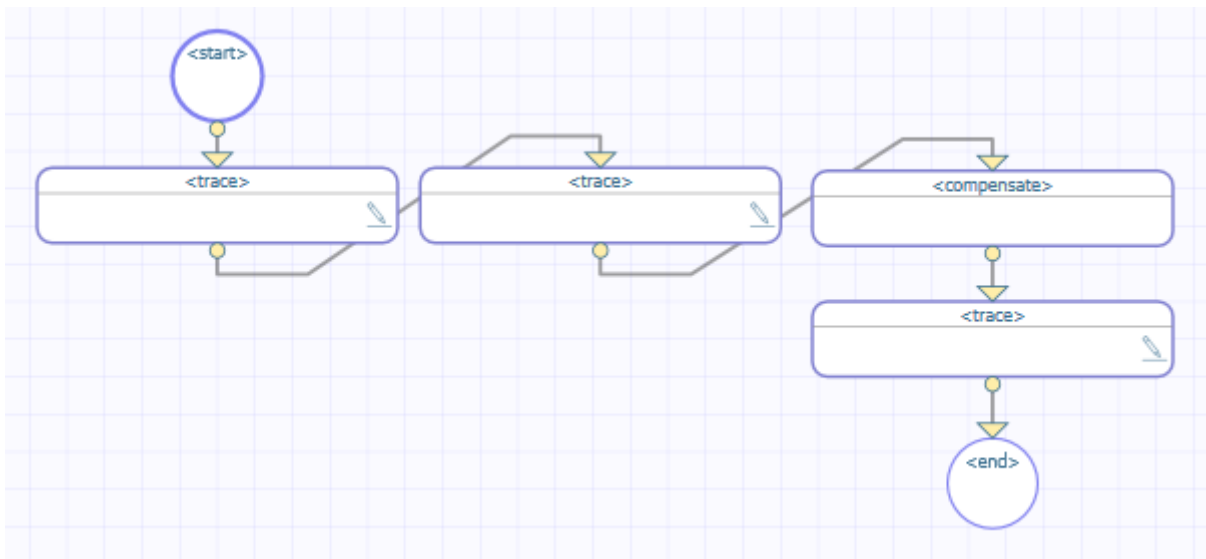
Suppose you have the following BPL:



This BPL business process does the following:

1. The Context tab (not shown) defines a property called MyBalance and sets its value to 100.
2. The first `<trace>` element generates the message before `scope balance is`, followed by the value of MyBalance.
3. The `<scope>` element starts the scope.
4. The next `<trace>` element generates the message before `debit`.
5. The `<assign>` element decrements MyBalance by 1.
6. The next `<trace>` element generates the message after `debit`.
7. The `<throw>` element throws a specific, named fault ("BuyersRegret").
8. Control now goes to the `<faulthandlers>`. A `<catch>` exists whose *fault* value is "BuyersRegret", so control goes there.

If we drill down into this `<catch>` element, we see the following:



9. Within this `<catch>`, the first `<trace>` element generates the message in `catch` fault handler for `'BuyersRegret'`.
 10. Within this `<catch>`, the second `<trace>` element generates the message before `restore balance is`, followed by the current value of `MyBalance`.
 11. The `<compensate>` element is used. For this element, *target* is a `<compensationhandler>` whose *name* is `RestoreBalance`. Within this `<compensationhandler>` block:
 - A `<trace>` statement outputs the message “Restoring Balance”
 - An `<assign>` statement increments `MyBalance` by 1.
- Note:** It is not possible to reverse the order of `<compensationhandlers>` and `<faulthandlers>`. If both blocks are provided, `<compensationhandlers>` must appear first and `<faulthandlers>` second.
12. The next `<trace>` element generates the message after `restore balance is`, followed by the current value of `MyBalance`.
 13. The `<scope>` ends.
 14. The last `<trace>` element generates the message after `scope balance is`, followed by the current value of `MyBalance`.

8.9.1 Event Log Entries

The corresponding Event Log entries look like this:



8.9.2 XData for This BPL

This BPL is defined by the following XData block:

Class Member

```
XData BPL
{
<process language='objectscript'
    request='Test.Scope.Request'
    response='Test.Scope.Response' >
    <context>
        <property name="MyBalance" type="%Library.Integer" initialexpression='100' />
    </context>
    <sequence>
        <trace value='before scope balance is "_context.MyBalance"' />
        <scope>
            <trace value='before debit' />
            <assign property='context.MyBalance' value='context.MyBalance-1' />
            <trace value='after debit' />
            <throw fault='BuyersRegret' />
            <compensationhandlers>
                <compensationhandler name="RestoreBalance">
                    <trace value='Restoring Balance' />
                    <assign property='context.MyBalance' value='context.MyBalance+1' />
                </compensationhandler>
            </compensationhandlers>
            <faulthandlers>
                <catch fault='BuyersRegret'>
                    <trace value='In catch fault handler for "BuyersRegret"' />
                    <trace value='before restore balance is "_context.MyBalance"' />
                    <compensate target="RestoreBalance" />
                    <trace value='after restore balance is "_context.MyBalance"' />
                </catch>
                <catchall>
                    <trace value='in catchall fault handler' />
                    <trace value=
                        '%LastError' "_
                        $System.Status.GetErrorCodes(..%Context.%LastError)_
                        " : "_
                        $System.Status.GetOneStatusText(..%Context.%LastError)'
                    />
                    <trace value='"%LastFault" "_..%Context.%LastFault"' />
                </catchall>
            </faulthandlers>
        </scope>
        <trace value='after scope balance is "_context.MyBalance"' />
    </sequence>
</process>
}
```


9

BPL Business Process Example

In the following [business process](#), three different banks can be consulted for prime rate and credit approval information.

Class Definition

```
/// Loan Approval Business Process for Bank Soprano.
/// Bank Soprano simulates a bank with great service but
/// somewhat high interest rates.
Class Demo.Loan.BankSoprano Extends Ens.BusinessProcessBPL
{
  XData BPL
  {
    <process request="Demo.Loan.Msg.Application"
      response="Demo.Loan.Msg.Approval">

      <context>
        <property name="CreditRating" type="%Integer"/>
        <property name="PrimeRate" type="%Numeric"/>
      </context>

      <sequence>

        <trace value='"received application for "_request.Name'"/>

        <assign name='Init Response'
          property="response.BankName"
          value='"BankSoprano"'>
          <annotation>
            <![CDATA[Initialize the response object.]]>
          </annotation>
        </assign>

        <call name="PrimeRate"
          target="Demo.Loan.WebOperations"
          async="1">
          <annotation>
            <![CDATA[Send an asynchronous request for the Prime Rate.]]>
          </annotation>
          <request type="Demo.Loan.Msg.PrimeRateRequest"/>
          <response type="Demo.Loan.Msg.PrimeRateResponse">
            <assign property="context.PrimeRate"
              value="callresponse.PrimeRate"/>
          </response>
        </call>

        <call name="CreditRating"
          target="Demo.Loan.WebOperations"
          async="1">
          <annotation>
            <![CDATA[Send an asynchronous request for the Credit Rating.]]>
          </annotation>
          <request type="Demo.Loan.Msg.CreditRatingRequest">
            <assign property="callrequest.TaxID" value='request.TaxID'/>
          </request>
          <response type="Demo.Loan.Msg.CreditRatingResponse">
            <assign property="context.CreditRating"
              value="callresponse.CreditRating"/>
          </response>
        </call>
```

```

<sync name='Wait'
  calls="PrimeRate,CreditRating"
  type="all"
  timeout="10">
  <annotation>
    <![CDATA[Wait for the response from the async requests.
              Wait for up to 10 seconds.]]>
  </annotation>
</sync>

<switch name='Approved?'>

  <case name='No PrimeRate'
    condition='context.PrimeRate=""'>
    <assign name='Not Approved'
      property="response.IsApproved"
      value="0"/>
  </case>

  <case name='No Credit'
    condition='context.CreditRating=""'>
    <assign name='Not Approved'
      property="response.IsApproved"
      value="0"/>
  </case>

  <default name='Approved' >
    <assign name='Approved'
      property="response.IsApproved"
      value="1"/>
    <assign name='InterestRate'
      property="response.InterestRate"
      value="context.PrimeRate+10+(99*(1-(context.CreditRating/100)))">
    <annotation>
      <![CDATA[Copy InterestRate into response object.]]>
    </annotation>
    </assign>
  </default>

</switch>

<delay
  name='Delay'
  duration="2+($zcrd(request.Name,4)#5)">
  <annotation>
    <![CDATA[Wait for a random duration.]]>
  </annotation>
</delay>

<trace value='"application is "
  _$(response.IsApproved:"approved for "_response.InterestRate_"%"',
  1:"denied")'"/>

</sequence>
</process>
}
}

```