



# Business Process and Data Transformation Language Reference

Version 2025.1  
2025-06-03

*Business Process and Data Transformation Language Reference*

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](mailto:support@InterSystems.com)

# Table of Contents

<b>BPL Elements</b> .....	<b>1</b>
Common Attributes and Elements .....	2
<alert> .....	4
<assign> .....	5
<branch> .....	12
<break> .....	14
<call> .....	15
<case> .....	20
<catch> .....	22
<catchall> .....	24
<code> .....	26
<compensate> .....	29
<compensationhandlers> and <compensationhandler> .....	30
<context> .....	32
<continue> .....	34
<default> .....	36
<delay> .....	38
<empty> .....	40
<false> .....	41
<faulthandlers> .....	42
<flow> .....	44
<foreach> .....	46
<if> .....	48
<label> .....	50
<milestone> .....	51
<parameters> and <parameter> .....	52
<process> .....	54
<property> .....	58
<pyFromImport> .....	60
<reply> .....	61
<request> .....	62
<response> .....	63
<rule> .....	65
<sequence> .....	67
<scope> .....	69
<sql> .....	71
<switch> .....	73
<sync> .....	75
<throw> .....	81
<trace> .....	83
<transform> .....	84
<true> .....	86
<until> .....	87
<while> .....	88
<xpath> .....	89
<xslt> .....	91
<b>DTL Elements</b> .....	<b>93</b>

DTL <annotation> .....	94
DTL <assign> .....	95
DTL <break> .....	98
DTL <case> .....	99
DTL <code> .....	100
DTL <comment> .....	102
DTL <default> .....	103
DTL <>false> .....	104
DTL <foreach> .....	105
DTL <group> .....	107
DTL <if> .....	108
DTL <sql> .....	109
DTL <subtransform> .....	110
DTL <switch> .....	112
DTL <trace> .....	113
DTL <transform> .....	114
DTL <>true> .....	116

# BPL Elements

This reference provides detailed information about each BPL element.

# Common Attributes and Elements

---

Describes attributes and elements that are present in most BPL elements.

## Common Attributes

Most BPL elements can contain the following attributes, which are listed here for brevity.

### ***name***

Usually optional. The name of this element. Specify a string of up to 255 characters.

### ***disabled***

Optional. You can temporarily disable the element by setting its *disabled* attribute to 1 (true). To re-enable the element, either remove the *disabled* attribute or set it to 0 (false).

### ***xpos***

Optional. Sets the *x* coordinate of the graphic that represents this element in BPL diagrams. Ignored by the BPL compiler. Specify a positive integer.

### ***ypos***

Optional. The *y* coordinate. Specify a positive integer.

### ***xend***

Optional. If the graphic that represents this element has two icons (start and end), then *xend* sets the *x* coordinate for the ending icon. Ignored by the BPL compiler. Specify a positive integer.

### ***yend***

Optional. The ending *y* coordinate. Specify a positive integer.

## Common Element: <annotation>

Most BPL elements can contain the <annotation> element, which allows you to associate descriptive text with a shape in a BPL diagram. This element is as follows:

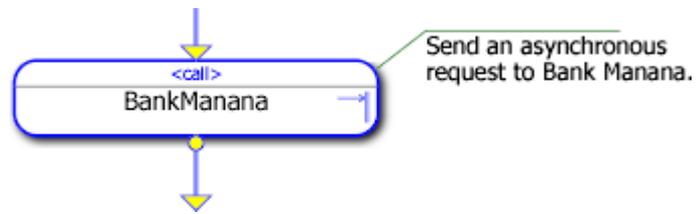
```
<annotation>
  <![CDATA[ Gets the current Account Balance for a customer.]]>
</annotation>
```

The text within the CDATA block appears as a commentary on the associated activity. The following example provides an <annotation> for a <call> activity:

### **XML**

```
<call name="BankManana">
  <annotation>
    <![CDATA[Send an asynchronous
      request to Bank Manana.]]>
  </annotation>
</call>
```

The CDATA block enables you to include line breaks and special characters such as the apostrophe (') without needing to use XML escape sequences. Note the line break between *asynchronous* and *request* in the example above, which the diagram reproduces literally as follows:



The maximum length of the <annotation> string is 32,767 characters, including the CDATA escape characters.

## <alert>

---

Send an alert message to a user device during execution of a business process.

### Syntax

```
<alert value="The system needs service right away." />
```

### Attributes and Elements

#### **value attribute**

Required. The text for the alert message. Specify an expression or a literal string.

#### **LanguageOverride attribute**

Optional. Specifies the scripting language in which any expressions (within in this element) are written.

Can be "python", "objectscript", or "basic" (not documented). Default is the language specified in the containing [<process>](#) element.

#### **name, disabled, xpos, ypos, xend, yend attributes**

See [Common Attributes and Elements](#).

#### **<annotation> element**

See [Common Attributes and Elements](#).

### Description

The [<alert>](#) element sends an alert message to a user device.

The text of the message is always written to the [Event Log](#) as an entry of type Alert. However, the real purpose of the [<alert>](#) element is to contact the user through email or other notification device. The [<alert>](#) element does this by sending the text of the message to a configuration item called Ens.Alert, which has been set up with all the information necessary to contact user devices outside InterSystems IRIS.

**Important:** If no Ens.Alert item has been configured as a member of the production, the [<alert>](#) simply goes to the [Event Log](#).

For details, see [Defining Alert Processors](#).

## <assign>

Assign a value to a property in the business process execution context.

### Syntax

```
<assign property="propertyname" value="expression" />
```

### Attributes and Elements

#### **property attribute**

Required. The target of this assignment. This must be a property in an execution context object, usually *context*, *request*, *response*, *callrequest*, or *callresponse*. For details, see the table in the [Description section](#).

#### **value attribute**

Required. Value of the property. Specify a literal value or an expression that returns a valid value for the property.

#### **LanguageOverride attribute**

Optional. Specifies the scripting language in which any expressions (within in this element) are written.

Can be "python", "objectscript", or "basic" (not documented). Default is the language specified in the containing [<process>](#) element.

#### **action attribute**

Optional. If *property* is a collection (list or array), use *action* to specify the type of assignment to perform on the collection. If not specified, a *set* is performed. Specify a literal string, either *append*, *set*, *clear*, *insert*, or *remove* as described below.

#### **key attribute**

Optional, except in some cases when *property* is a collection (list or array). If so, you must use this key to specify the member of the collection that is the target of this assignment. Specify an expression that evaluates to a key.

#### **name, disabled, xpos, ypos, xend, yend attributes**

See [Common Attributes and Elements](#).

#### **<annotation> element**

See [Common Attributes and Elements](#).

## Description

This section describes the importance of the execution context to BPL business processes, and explains how to use the [<assign>](#) element to set values in the business process execution context.

A business process must have certain state information saved to disk and restored from disk, whenever it suspends or resumes execution. This feature is especially important for long-running business processes, which may take days or weeks to complete. To address this need, InterSystems IRIS provides every BPL business process with a group of objects and variables called 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 correct operation of a BPL business process depends on the appropriate use of these variables.

Every variable in the execution context has a specific name and purpose, and can have its value set using the `<assign>` element. The following table lists the variables in the execution context.

Variable	Purpose
<i>callrequest</i>	The <i>callrequest</i> object contains any properties that are required to build the request message object to be sent by a <code>&lt;call&gt;</code> . Within the corresponding <code>&lt;request&gt;</code> activity, use a sequence of <code>&lt;assign&gt;</code> elements to set the property values in <i>callrequest</i> .
<i>callresponse</i>	Upon completion of a <code>&lt;call&gt;</code> activity, the <i>callresponse</i> object contains the properties of the response message object that was returned to the <code>&lt;call&gt;</code> . Within the corresponding <code>&lt;response&gt;</code> activity, use a sequence of <code>&lt;assign&gt;</code> elements to copy the returned values from properties on <i>callresponse</i> into properties on <i>context</i> or <i>response</i> .
<i>context</i>	The <i>context</i> object is a general-purpose data container for the business process. <i>context</i> has no automatic definition. To define properties of this object, use the <code>&lt;context&gt;</code> element. That done, you may refer to these properties anywhere inside the <code>&lt;process&gt;</code> element using dot syntax, as in: <code>context.Balance</code>
<i>request</i>	The <i>request</i> object contains any properties of the original request message object that caused this business process to be instantiated. You may refer to <i>request</i> properties anywhere inside the <code>&lt;process&gt;</code> element using dot syntax, as in: <code>request.UserID</code>
<i>response</i>	The <i>response</i> object contains any properties that are required to build the final response message object to be returned by the business process. You may refer to <i>response</i> properties anywhere inside the <code>&lt;process&gt;</code> element using dot syntax, as in: <code>response.IsApproved</code> . Use the <code>&lt;assign&gt;</code> element to assign values to these properties.
<i>status</i>	<i>status</i> is a value of type <code>%Status</code> that indicates success or failure. As the BPL business process runs, if at any time <i>status</i> acquires a failure value, InterSystems IRIS immediately terminates the business process and writes a text message to the <a href="#">Event Log</a> indicating the reason for failure. In general, this happens automatically, when unsuccessful values are returned from <code>&lt;call&gt;</code> activities. However, BPL business process code can initiate a sudden, but graceful exit by setting the <i>status</i> value using <code>&lt;assign&gt;</code> or <code>&lt;code&gt;</code> . See the description at the end of this topic.
<i>syncresponses</i>	<i>syncresponses</i> is a collection of response objects, keyed by the names of the <code>&lt;call&gt;</code> activities being synchronized. Only completed calls are represented. You can retrieve a response from <i>syncresponses</i> only after a <code>&lt;sync&gt;</code> and before the end of the current <code>&lt;sequence&gt;</code> . Do so using the syntax <code>syncresponses.GetAt("MyName")</code> where the relevant call was defined as <code>&lt;call name="MyName"&gt;</code>

Variable	Purpose
<i>synctimedout</i>	<p>The <i>synctimedout</i> value is an integer. <i>synctimedout</i> indicates the outcome of a <a href="#">&lt;sync&gt;</a> activity after several calls. You can test the value of <i>synctimedout</i> after the <a href="#">&lt;sync&gt;</a> and before the end of the <a href="#">&lt;sequence&gt;</a> that contains the calls and <a href="#">&lt;sync&gt;</a>. <i>synctimedout</i> has one of three values:</p> <ul style="list-style-type: none"> <li>• If 0, no call timed out. All the calls had time to complete. This is also the value if the <a href="#">&lt;sync&gt;</a> activity had no <i>timeout</i> set.</li> <li>• If 1, at least one call timed out. This means not all <a href="#">&lt;call&gt;</a> activities completed before the timeout.</li> <li>• If 2, at least one call was interrupted before it could complete.</li> </ul> <p>Generally you will test <i>synctimedout</i> for status and then retrieve the responses from completed calls out of the <i>syncresponses</i> collection.</p>

**CAUTION:** Like all other execution context variable names, *status* is a reserved word in BPL. Do not use it with [<assign>](#) except as described above.

The BPL [<assign>](#) element specifies a target and an expression that will be assigned to it. The target may be a property in one of the objects in the business process execution context, or it may be one of the single-valued variables such as *status*. The properties involved in an [<assign>](#) element can be data types, objects, or collections of either. Collection properties are declared by setting the *collection* attribute to “array” or “list” in the corresponding [<property>](#) element.

As described in the above table, the object called *context* serves as a general-purpose context object for the business process. Properties in the *context* object are defined using the [<context>](#) and [<property>](#) elements at the beginning of the [<process>](#) environment. For example:

## XML

```
<process request="Demo.Loan.Msg.Application" response="Demo.Loan.Msg.Approval">
  <context>
    <property name="CreditRating" type="%Integer"/>
    <property name="PrimeRate" type="%Numeric"/>
  </context>
  ...
</process>
```

The above BPL excerpt defines two *context* properties for this business process—*context.CreditRating* and *context.PrimeRate*—but does not assign values to them. An [<assign>](#) element anywhere below this [<context>](#) element and within the [<process>](#) environment can assign a value to any of these properties as needed. For example:

## XML

```
<process request="Demo.Loan.Msg.Application" response="Demo.Loan.Msg.Approval">
  <context>
    <property name="CreditRating" type="%Integer"/>
    <property name="PrimeRate" type="%Numeric"/>
  </context>
  <sequence>
    <call name="PrimeRate" target="Demo.Loan.WebOperations" async="0">
      <request type="Demo.Loan.Msg.PrimeRateRequest">
        </request>
      <response type="Demo.Loan.Msg.PrimeRateResponse">
        <assign property="context.PrimeRate" value="callresponse.PrimeRate"/>
      </response>
    </call>
    ...
  </sequence>
  ...
</process>
```

The above BPL excerpt continues the first one. Note that the `<call>` element in this example is synchronous, and has both a `<request>` and a `<response>` element.

The `<response>` in this case contains an `<assign>` operation that references two properties on objects inside the execution context: `context.PrimeRate` (from the general-purpose context object) and `callresponse.PrimeRate` (from the response object associated with the current `<call>` element, in this case `Demo.Loan.Msg.PrimeRateResponse` as you can see above). The `<assign>` operation receives the value of the `PrimeRate` property returned from the `<call>` and places it in the general-purpose context object.

Inside the `<sequence>` element shown above, and continuing from the `<call>` element just discussed, the example continues as follows:

### XML

```
<call name="CreditRating" target="Demo.Loan.WebOperations" async="0">
  <request type="Demo.Loan.Msg.CreditRatingRequest">
    <assign property="callrequest.SSN" value='request.SSN' />
  </request>
  <response type="Demo.Loan.Msg.CreditRatingResponse">
    <assign property="context.CreditRating" value="callresponse.CreditRating" />
  </response>
</call>
```

The above statements assign the `SSN` property from the primary request (`request.SSN`) to the `SSN` property in the request being made by the current `<call>` element (`callrequest.SSN`). After this assignment is made, the `<call>` element issues the request. It is a synchronous call of type `Demo.Loan.WebOperations`. When a response returns, the `<call>` element gets the value of the `CreditRating` property returned from the `<call>` (`callresponse.CreditRating`) and places it in a property on the general-purpose context object (`context.CreditRating`).

The following statement assigns the integer value 1 to the `IsApproved` property in the primary response object for the business process (`response.IsApproved`). In this example, `IsApproved` is a Boolean value (true or false) according to InterSystems IRIS conventions. That is, an integer value of 1 means true (the applicant was approved), and 0 means false (the applicant was not approved).

### XML

```
<assign name='IsApproved' property="response.IsApproved" value="1">
  <annotation>
    <![CDATA[Copy IsApproved into the response object.]]>
  </annotation>
</assign>
```

The following statement assigns a calculated value—the result of an expression involving two properties in the general-purpose context object—to the `InterestRate` property in the primary response object for the business process (`response.InterestRate`):

### XML

```
<assign name='InterestRate'
  property="response.InterestRate"
  value="context.PrimeRate+1+(2*(1-(context.CreditRating/100)))">
  <annotation>
    <![CDATA[Copy InterestRate into the response object.]]>
  </annotation>
</assign>
```

## Types of `<assign>` Operation

The syntax for the BPL `<assign>` element works as follows:

1. The property attribute identifies an object and property that is the target of the assignment operation.

- 2. The value attribute provides the value for the target property. This may be an expression that is evaluated at runtime to provide a value for the assignment. Expressions within an <assign> element must use the language specified by the <process> element for the business process.
- 3. There are several types of BPL <assign> operation, as specified by the optional action attribute. The allowable values for the action attribute are:

Value	Description
append	Add the target element to the end of the list.
set	(Default) Set the target element to a new value.
insert	Insert a new value into the collection.
remove	Remove the target element from the collection.
clear	Clear the contents of the target collection.

Aside from the default value `set`, most of these variations are intended to handle assignments involving collection properties. The various assignment types are summarized in the following table.

Property Type	action Attribute Value	key Attribute Required	Result
Non-collection	set	No	Property is set to new value
Array	clear	No	Array is cleared
Array	remove	Yes	Element at <i>key</i> is removed
Array	set	Yes	Element at <i>key</i> is set to new value
List	append	No	Element is added to the end of the list
List	clear	No	List is cleared
List	insert	Yes	Element is inserted at position determined by <i>key</i>
List	remove	Yes	Element at <i>key</i> is removed
List	set	Yes	Element at <i>key</i> is replaced

Details about each type of BPL <assign> operation follow.

### The append Operation

The `append` operation adds the target element to the end of a list property.

### The set Operation

The `set` operation sets the value of the specified property to the value of the value attribute. Note that the value attribute contains an expression and can itself refer to an object or property of an object within the execution context:

### XML

```
<assign name='CopyResult' property='context.SSN' value='callresponse.SSN' />
```

If the target property is an array collection, then the value of the key attribute specifies an item in the array, otherwise the key attribute is ignored.

If the target property is a collection and the `value` attribute specifies a collection of the same type, then the collection contents are copied into the target collection:

### XML

```
<assign name='CopyResults' property='context.List' value='callresponse.List' />
```

The default action for the `assign` element is the set operation; if action is not specified, then the `assign` specifies a set operation.

### The clear Operation

This operation applies to collection properties only. The `clear` operation clears the contents of the specified collection property. The `value` and `key` attributes are ignored, but since the BPL schema for the `<assign>` element requires it, a `value` attribute must be present in the statement.

For example, the following will clear the contents of the collection property `List`:

### XML

```
<assign name='ClearResults' property='context.List' action='clear' value='' />
```

### The insert Operation

This applies to list collection properties only. The `insert` operation inserts a value into the specified collection property. If the `key` attribute is present the new value is inserted after the position (an integer) specified by `key` otherwise the new item is inserted at the end.

For example, the following will insert a value into the array collection property `Array` using the key `primary`:

### XML

```
<assign name='Ins' property='context.Array'
  action='insert'
  key='primary'
  value='request.Primary' />
```

### The remove Operation

This applies to collection properties only. The `remove` operation removes an item from the specified collection property. The `value` attribute is ignored, but since the BPL schema for the `<assign>` element requires it, a `value` attribute must be present in the statement.

If the target property is an array collection, then the value of the `key` attribute specifies an item in the array, otherwise the `key` attribute is ignored.

For example, the following will remove the element with key `abc` from the array property `Array`:

### XML

```
<assign name='Remove' property='context.Array' action='remove'
  key='abc' value='' />
```

## Using `<assign>` to Set the status Variable

`status` is a business process execution context variable 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.

You can use an [<assign>](#) element to 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.

*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`

## See Also

- [<call>](#)
- [<context>](#)

## <branch>

---

Conditionally cause an immediate change in the flow of execution.

### Syntax

```
<branch condition="myVar='1'" label="JumpToMe" />
```

### Attributes and Elements

#### **condition** attribute

Required. An expression that, if true, causes the flow of control to jump to the identified [<label>](#).

Specify an expression that evaluates to the integer value 1 (if true) or 0 (if false).

#### **LanguageOverride** attribute

Optional. Specifies the scripting language in which any expressions (within in this element) are written.

Can be "python", "objectscript", or "basic" (not documented). Default is the language specified in the containing [<process>](#) element.

#### **label** attribute

Required. The name of the [<label>](#) to jump to. Specify a string of up to 255 characters.

#### **name, disabled, xpos, ypos, xend, yend** attributes

See [Common Attributes and Elements](#).

#### **<annotation>** element

See [Common Attributes and Elements](#).

### Description

The [<branch>](#) element causes an immediate change in the flow of execution if the value of its *condition* expression is true. Control passes to the [<label>](#) element whose name is specified as the value of the *label* attribute in the [<branch>](#).

In the following BPL example, if the *condition* expression is true, the flow of control shifts directly from the [<branch>](#) with the *label* value TraceSkipped to the [<label>](#) with the *name* value TraceSkipped, while the intervening [<trace>](#) element is ignored:

```
<branch condition="myVar='1'" label="TraceSkipped" />  
<trace value="Ignore me when myVar is 1..." />  
<label name="TraceSkipped" />
```

If the [<branch>](#) *condition* expression is false, control simply passes to the next BPL statement following the [<branch>](#), in this case [<trace>](#).

A destination [<label>](#) must be in the same scope as the [<branch>](#) that references it. Thus:

- Each [<sequence>](#) element within a [<flow>](#) has its own [<label>](#) scope. The BPL execution engine prevents any attempt to [<branch>](#) to a [<label>](#) outside the current [<sequence>](#) container.
- There are similar restrictions on any other BPL container element that controls the flow of execution at runtime. Each container has its own [<label>](#) scope.

In addition to these restrictions, each <label> *name* value must be unique across the entire BPL business process, not just within the current scope.

**CAUTION:** As is true in all programming languages, the BPL branch mechanism must be used with care. The BPL editor does not prevent basic programming mistakes such as infinite loops or invalid branch cases.

## <break>

---

Break out of a loop and exit the loop activity.

### Syntax

```
<break/>
```

### Attributes and Elements

***name, disabled, xpos, ypos, xend, yend* attributes**

See [Common Attributes and Elements](#).

**<annotation> element**

See [Common Attributes and Elements](#).

### Description

BPL syntax permits any element that can contain a sequence of activities—<case>, <default>, <foreach>, <false>, <sequence>, <true>, <until>, or <while>—to contain a <break> element if desired.

The <break> element allows the flow of control to exit a loop immediately without completing any more of the operations inside the containing loop. For example:

#### XML

```
<while condition="0">
  //...do various things...
  <if condition="somecondition">
    <true>
      <break/>
    </true>
  </if>
  //...do various other things...
</while>
```

In the above example, it is the <true> element that contains the <break> element. However, the loop affected by this <break> is actually the containing <while> loop.

The example works as follows: If on some pass through this loop, the <if> element finds “somecondition” to be true (that is, equal to the integer value 1) then the flow of control passes to the <true> element inside the <if>. Upon encountering the <break> element, execution immediately exits the containing <while> loop and proceeds to the next statement following the </while>.

Loop activities that you might want to modify by using a <break> element include [<foreach>](#), [<until>](#), and [<while>](#).

**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.

### See Also

- [<continue>](#)

## <call>

Send a request to a business operation, or to another business process.

### Syntax

```

<call name="Call" target="MyApp.MyOperation" async="1">
  <request type="MyApp.Request">
    ...
  </request>
  <response type="MyApp.Response">
    ...
  </response>
</call>

```

### Attributes and Elements

#### **name attribute**

Required. The name of the <call> element; provide a literal string, or by using the @ [indirection](#) operator to refer to the value of an [execution context variable](#) variable. If you wish to use a <sync> element to retrieve responses from asynchronous calls, refer to them using this *name*.

Specify a string of up to 255 characters.

#### **target attribute**

Required. The configured name of the business operation or business process to which the request is being sent. Provide this value as a literal string, or by using the ObjectScript @ [indirection](#) operator to refer to the value of an [execution context variable](#).

#### **async attribute**

Required. Specifies the type of request to make. If 1 (true), the request is asynchronous. If 0 (false), the request is synchronous. Specify 1 (true) or 0 (false).

#### **timeout attribute**

Optional. Sets a timeout on a synchronous call. The *timeout* value is used only when the *async* attribute of the <call> is set to 0 (false). Specifies the time, in seconds, to wait for the response, as an expression that evaluates to an XML `xsd:dateTime` value.

For example `2023:10:19T10:10`

#### **name, disabled, xpos, ypos, xend, yend attributes**

See [Common Attributes and Elements](#).

#### **<annotation> element**

See [Common Attributes and Elements](#).

#### **<request> element**

Required. Specifies the type (class name) of the request to send.

#### **<response> element**

Optional. Specifies the type (class name) of the response to return. If omitted, no response is returned from this <call>.

## Description

The `<call>` element sends a request (synchronously or asynchronously) to a business operation or business process. The `<call>` element has a required attribute, `async`, that determines how the request is made:

- If `async` is 0 (or False), the request is made synchronously; the business process waits until it receives a response before continuing execution.

**Important:** A `<call>` element with `async= 'False'` and a `<response>` block defined suspends execution of its business process thread until the called operation completes.

A `<call>` element with `async= 'False'` but with *no* `<response>` block defined behaves as if `async= 'True'`. If you want to send a synchronous request but do not require a response, create a non-functioning `<response>` block so that the `<call>` waits for the target host to finish before continuing execution.

- If `async` is 1 (or True), the request is made asynchronously; the business process continues to execute after making the request. The business process can later receive the responses from several asynchronous calls by providing a `<sync>` element that specifies a list of the `<call>` elements for which it is waiting. For details, see `<sync>`.

The `<call>` element has the child elements `<request>` and `<response>` which identify the class of request and response objects to use in making the call. Either element can contain one or more `<assign>` elements. In the `<request>` element, `<assign>` elements are used to fill in the properties of the request object used for the call. The `<response>` element uses `<assign>` elements when it needs to move the properties of the resulting response object to a new location, such as the `context` or `response` variables in the business process execution context.

**Note:** There is detailed information about the business process execution context in documentation of the `<assign>` element. Also see [Business Process Execution Context](#).

In the case of an asynchronous request, the `<assign>` elements within the body of the `<response>` element are executed when the corresponding request is received. There is no guarantee when this will occur, so a business process will typically use the `<sync>` element to wait for an asynchronous response. Note that if a response is not received within the `timeout` period specified by the `<sync>` element, then the assignments defined by the corresponding `<response>` block will not be executed, and the response itself will be marked with a status of Discarded.

If the call is synchronous, an optional timeout can be specified using the `timeout` attribute on the `<call>` element itself. This attribute cannot be used for asynchronous calls. If the `<call>` element has `async` set to 1 (true) then the only way to set a timeout period is to use the `timeout` attribute on the `<sync>` element that is being used to collect the asynchronous response(s).

The following example sends an synchronous `Ens.StringRequest` request to the Get Weather Report business operation:

```
<call name='Get Weather Report' target='Get Weather Report' async='0' >
  <request type='Ens.StringRequest' >
    <assign property="callrequest.StringValue" value="context.Location" action="set" />
  </request>
  <response type='Demo.Service.Msg.WeatherOperationResponse' >
    <assign property="context.OperationReport" value="callresponse" action="set" />
  </response>
</call>
```

The following example uses the `<call>` element to send an asynchronous `MyApp.SalaryRequest` request to the `MyApp.PayrollApp` business operation:

## XML

```
<call name="FindSalary" target="MyApp.PayrollApp" async="1">
  <request type="MyApp.SalaryRequest">
    <assign property="callrequest.Name" value="request.Name" />
    <assign property="callrequest.SSN" value="request.SSN" />
  </request>
  <response type="MyApp.SalaryResponse">
    <assign property="context.Salary" value="callresponse.Salary" />
  </response>
</call>
```

Whenever a <call> element is executed, the BPL engine inserts the *name* of the <call> element into the message header so that it is visible in later Message Browser and Visual Trace displays.

## Use of the <assign> Element

The above example includes <assign> elements that manipulate properties in the variables in the business process execution context such as *context*, *request*, *callrequest*, and *callresponse*. While many details concerning these variables are found in the documentation for the <assign> element, the following table describes the execution context variables as they relate to the <call> activity:

The <call> element can refer to the following variables and their properties. Do not use variables not listed here.

Variable	Purpose
<i>callrequest</i>	A <call> element contains a <request> element that identifies the type of message that will be sent to the target. If this message type has input parameters, the <request> element must provide <assign> elements that assign values to properties in the <i>callrequest</i> object. These properties must match the input parameters for the message type. After the <request> completes, the <i>callrequest</i> object goes out of scope.
<i>callresponse</i>	If the request message type has a corresponding response message type, the <call> element contains a <response> element. When the response arrives, control passes to the <response> element. The output parameters from the response message become properties of the <i>callresponse</i> object. Since <i>callresponse</i> only has meaning inside the <response> element, to preserve these values the <response> element must provide <assign> elements that assign <i>callresponse</i> values to properties of other, more permanent objects in the business process execution context, usually <i>context</i> or <i>response</i> .
<i>context</i>	Throughout the business process, the <i>context</i> object serves as a general-purpose container for any business process data that needs to be persistent.
<i>request</i>	Throughout the business process, the <i>request</i> object contains the original properties that were sent to the business process as parameters of the request that instantiated it.
<i>response</i>	The <i>response</i> object retains its scope throughout the business process. It contains the properties that are expected to be returned to the caller as output parameters of this business process. Whatever is inside the <i>response</i> object, when a business process completes or exits, will be interpreted as the return values of the business process.
<i>status</i>	<i>status</i> is a %Status value that indicates success or failure. When a BPL business process starts up, <i>status</i> is automatically assigned a value indicating success. As the BPL business process runs, if at any time <i>status</i> acquires a failure value, the business process immediately exits and writes the corresponding text message to the <a href="#">Event Log</a> . <i>status</i> automatically receives the returned %Status value returned from any <call> activity, without any special statements in the BPL code. Thus, if any <call> activity fails, the BPL business process immediately exits and writes an <a href="#">Event Log</a> entry.

Variable	Purpose
<i>syncresponses</i>	<i>syncresponses</i> is a collection of response objects, keyed by the names of the <code>&lt;call&gt;</code> activities being synchronized. Only completed calls are represented. You can retrieve a response from <i>syncresponses</i> only after a <code>&lt;sync&gt;</code> and before the end of the current <code>&lt;sequence&gt;</code> . Do so using the syntax <code>syncresponses.GetAt("MyName")</code> where the relevant call was defined as <code>&lt;call name="MyName"&gt;</code>
<i>synctimedout</i>	<p>The <i>synctimedout</i> value is an integer. <i>synctimedout</i> indicates the outcome of a <code>&lt;sync&gt;</code> activity after several calls. You can test the value of <i>synctimedout</i> after the <code>&lt;sync&gt;</code> and before the end of the <code>&lt;sequence&gt;</code> that contains the calls and <code>&lt;sync&gt;</code>. <i>synctimedout</i> has one of three values:</p> <ul style="list-style-type: none"> <li>• If 0, no call timed out. All the calls had time to complete. This is also the value if the <code>&lt;sync&gt;</code> activity had no <i>timeout</i> set.</li> <li>• If 1, at least one call timed out. This means not all <code>&lt;call&gt;</code> activities completed before the timeout.</li> <li>• If 2, at least one call was interrupted before it could complete.</li> </ul> <p>Generally you test <i>synctimedout</i> for status and then retrieve the responses from completed calls out of the <i>syncresponses</i> collection.</p>

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

## Indirection in the name or target Attributes (Accessing Execution Context Variables)

The values of the *name* or *target* attributes are strings. 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 an [execution context variable](#) that contains the appropriate string. This example accesses the value of the `nextCallName` and `nextBusinessHost` properties of the `context` object.

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

## Using Multiple Asynchronous `<calls>` in a Loop, Followed by a `<sync>`

This section describes how to use multiple asynchronous `<calls>` in a loop, followed by a `<sync>`.

When a BPL makes a `<call>` it makes note of the name of the call; in the `<sync>`, you must specify that same name to designate which pending request to wait for. In some scenarios, you have multiple asynchronous calls in a loop, as in this example:

```
<sequence>
  <while condition='...'>
    <call name="A" async="1" />
  </while>
  ...
  <sync calls="A" type="all" timeout="3600"/>
</sequence>
```

Because the BPL tracks which call to wait for by the call name, the sync completes as soon as the *first* response comes in. If you want the sync to wait until all such calls are completed, it is necessary to generate a set of unique call names and then use that list of names. Here is a way to do so:

1. Create a context variable containing a string which changes for each call by adding a numeric iterator (*i* in the example below). Before the call, initialize this variable as in the following example:  

```
set context.callname = "A" _ context.i
```
2. Set the Name of the <call> equal to this variable.
3. Create a string containing all the <call> names, comma-separated, i.e.: "A1 , A2 , A3 , A4 , A5 ". Save that in a separate variable, `context.allCallNames`, in the example below.
4. Set the `calls` attribute of the <sync> equal to the variable containing the list of calls.

```
<sequence>
  <while condition='...'>
    .... code here to set up callname and allCallNames ...
    <call name="@context.callname" async="1" />
  </while>
  ...
  <sync calls="@context.allCallNames" type="all" timeout="3600"/>
</sequence>
```

## See Also

- [<assign>](#)
- [<code>](#)
- [<reply>](#)
- [<sequence>](#)
- [<sync>](#)

## <case>

---

Perform a set of activities when a condition is matched within a [<switch>](#) element.

### Syntax

```
<switch>
  <case>
    ...
  </case>
  ...
  <default>
    ...
  </default>
</switch>
```

### Attributes and Elements

#### **condition** attribute

Required. If this expression evaluates to true, the contents of this [<case>](#) element are executed. If false, this [<case>](#) is ignored.

Specify an expression that evaluates to the integer value 1 (if true) or 0 (if false).

#### **LanguageOverride** attribute

Optional. Specifies the scripting language in which any expressions (within in this element) are written.

Can be "python", "objectscript", or "basic" (not documented). Default is the language specified in the containing [<process>](#) element.

#### **name, disabled, xpos, ypos, xend, yend** attributes

See [Common Attributes and Elements](#).

#### **<annotation>** element

See [Common Attributes and Elements](#).

#### **Other elements**

Optional. [<case>](#) may contain zero or more of the following elements in any combination: [<alert>](#), [<assign>](#), [<branch>](#), [<break>](#), [<call>](#), [<code>](#), [<continue>](#), [<delay>](#), [<empty>](#), [<flow>](#), [<foreach>](#), [<if>](#), [<label>](#), [<milestone>](#), [<reply>](#), [<rule>](#), [<scope>](#), [<sequence>](#), [<sql>](#), [<switch>](#), [<sync>](#), [<throw>](#), [<trace>](#), [<transform>](#), [<until>](#), [<while>](#), [<xpath>](#), or [<xslt>](#).

### Description

A [<case>](#) element is used within [<switch>](#).

A [<switch>](#) element contains a sequence of one or more [<case>](#) elements and an optional [<default>](#) element.

When a [<switch>](#) element is executed, it evaluates each [<case>](#) condition in turn. These conditions are logical expressions in the scripting language of the containing [<process>](#) element. If any expression evaluates to the integer value 1 (true), then the contents of the corresponding [<case>](#) element are executed; otherwise, the expression for the next [<case>](#) element is evaluated.

If no [<case>](#) condition is true, the contents of the [<default>](#) element are executed.

As soon as one of <case> elements is executed, execution control leaves the surrounding <switch> statement. If no <case> condition matches, control leaves the <switch> after the <default> activity executes.

Activities within a <case> element can be any BPL activity, including <assign> elements as in the example below:

## XML

```
<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>
```

## See Also

- [<switch>](#)
- [<default>](#)

## <catch>

---

Catch a fault produced by a <throw> element.

### Syntax

```
<scope>
  <throw fault="MyFault" />
  ...
  <faulthandlers>
    <catch fault="MyFault" />
    ...
  </faulthandlers>
</scope>
```

### Attributes and Elements

#### **fault** attribute

Required. The name of the fault. It can be a literal text string (up to 255 characters) or an expression to be evaluated.

#### **LanguageOverride** attribute

Optional. Specifies the scripting language in which any expressions (within in this element) are written.

Can be "python", "objectscript", or "basic" (not documented). Default is the language specified in the containing <process> element.

#### **name, disabled, xpos, ypos, xend, yend** attributes

See [Common Attributes and Elements](#).

#### **<annotation>** element

See [Common Attributes and Elements](#).

#### **Other elements**

Optional. <catch> may contain zero or more of the following elements in any combination: <alert>, <assign>, <branch>, <break>, <call>, <code>, <compensate>, <continue>, <delay>, <empty>, <foreach>, <if>, <label>, <milestone>, <reply>, <rule>, <scope>, <sequence>, <sql>, <switch>, <sync>, <throw>, <trace>, <transform>, <until>, <while>, <xpath>, or <xslt>.

### Description

When a <throw> statement executes, control immediately shifts 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. When you specify a fault string it *needs* the extra set of quotes to contain it, as shown below:

#### **XML**

```
<catch fault="thrown" />
```

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 shifts 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.

For details, see [Handling Errors in BPL](#).

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

## See Also

- [<catchall>](#)
- [<compensate>](#)
- [<compensationhandlers>](#)
- [<faulthandlers>](#)
- [<scope>](#)
- [<throw>](#)

## <catchall>

Catch a fault or system error that does not match any [<catch>](#).

### Syntax

```
<scope>
  <throw fault="MyFault" />
  ...
  <faulthandlers>
    <catch fault="MyFault">
      ...
    </catch>
    <catch fault="OtherFault">
      ...
    </catch>
  </faulthandlers>
  <catchall>
    ...
  </catchall>
</scope>
```

### Attributes and Elements

#### *name, disabled, xpos, ypos, xend, yend* attributes

See [Common Attributes and Elements](#).

#### <annotation> element

See [Common Attributes and Elements](#).

#### Other elements

Optional. <catchall> may contain zero or more of the following elements in any combination: <alert>, <assign>, <branch>, <break>, <call>, <code>, <compensate>, <continue>, <delay>, <empty>, <foreach>, <if>, <label>, <milestone>, <reply>, <rule>, <scope>, <sequence>, <sql>, <switch>, <sync>, <throw>, <trace>, <transform>, <until>, <while>, <xpath>, or <xslt>.

### Description

A <catchall> element is used within a [<faulthandlers>](#) element.

When a [<throw>](#) statement executes, control immediately shifts 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. If it finds one, 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 shifts 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.

For details, see [Handling Errors in BPL](#).

**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.

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

## See Also

<catch>, <compensate>, <compensationhandlers>, <faulthandlers>, <scope>, and <throw>.

## <code>

---

Execute lines of custom code.

### Syntax

```
<code name='CodeWrittenInBasic'>
  <![CDATA[ 'invoke custom method "MyApp.MyClass".Method(context.Value)  ]]>
</code>
```

### Attributes and Elements

#### **LanguageOverride** attribute

Optional. Specifies the scripting language in which the code in this element is written.

Can be "python", "objectscript", or "basic" (not documented). Default is the language specified in the containing [<process>](#) element.

#### **name, disabled, xpos, ypos, xend, yend** attributes

See [Common Attributes and Elements](#).

#### **<annotation>** element

See [Common Attributes and Elements](#).

### Description

The BPL [<code>](#) element executes one or more lines of user-written code within a BPL business process. You can use the [<code>](#) element to perform special tasks that are difficult to express using the BPL elements. Any properties referenced by the [<code>](#) element must be properties within the business process execution context.

The scripting language for a BPL [<code>](#) element is specified by the language attribute of the containing [<process>](#) element. This should be `objectscript`. For further information, see:

- [Using ObjectScript](#)
- [ObjectScript Reference](#)

Typically a developer wraps the contents of a [<code>](#) element within a CDATA block so that it is not necessary to escape special XML characters such as the apostrophe (') or the ampersand (&). For example:

#### XML

```
<code name="MyCode" language="objectscript">
  <![CDATA[ callrequest.Name = request.FirstName & " " & request.LastName]]>
</code>
```

To ensure you can properly suspend and restore execution of a business process, follow these guidelines when using the [<code>](#) element:

- The execution time should be short; custom code should not tie up the general execution of the business process.
- Do not allocate any system resources (such as taking out locks or opening devices) without releasing them within the same [<code>](#) element.
- If a [<code>](#) element starts a transaction, make sure that the same [<code>](#) element ends the transactions in *all possible scenarios*; otherwise, the transaction can be left open indefinitely. This could prevent other processing or can cause significant downtime.

- Do not rely on variables that are not part of the business process execution context. InterSystems IRIS automatically restores the contents of the execution context whenever a business process is suspended and later resumed; any other variables will be cleaned up.

Also, InterSystems strongly recommends that instead of including multiple lines of code within <code>, you invoke a class method or a routine that contains the needed code. This approach makes it far easier to test and debug your processing.

## Available Variables

The <code> element can refer to the following [execution context variables](#) and their properties. Do not use variables not listed here.

Variable	Purpose
<i>context</i>	The <i>context</i> object is a general-purpose data container for the business process. <i>context</i> has no automatic definition. To define the properties of this object, use the <context> element. That done, you may refer to these properties anywhere inside the <process> element using dot syntax, as in: <code>context.Balance</code>
<i>request</i>	The <i>request</i> object contains any properties of the original request message object that caused this business process to be instantiated. You may refer to <i>request</i> properties anywhere inside the <process> element using dot syntax, as in: <code>request.UserID</code>
<i>response</i>	The <i>response</i> object contains any properties that are required to build the final response message object to be returned by the business process. You may refer to <i>response</i> properties anywhere inside the <process> element using dot syntax, as in: <code>response.IsApproved</code> . Use the <assign> element to assign values to these properties.
<i>status</i>	<i>status</i> is a value of type %Status that indicates success or failure. When a BPL business process starts up, <i>status</i> is automatically assigned a value indicating success. As the BPL business process runs, if at any time <i>status</i> acquires a failure value, InterSystems IRIS immediately terminates the business process and writes the corresponding text message to the <a href="#">Event Log</a> . In general, this happens automatically, when unsuccessful values are returned from <call> activities. However, BPL business process code can initiate a sudden, but graceful exit by setting the <i>status</i> value using <assign> or <code>. See the description at the end of this topic.
<i>process</i>	The <i>process</i> object represents the current instance of the BPL business process object (an instance of the BPL class). This object has one property for each property defined in that class. You can invoke methods of the <i>process</i> object; for example: <code>process.SendRequestSync()</code>

**CAUTION:** Like all other execution context variable names, *status* is a reserved word in BPL. Do not use it in <code> blocks except to cause the <code> block to exit.

## Using <code> to Set the status Variable

*status* is a business process execution context variable 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.

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*.

*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`

## See Also

- [<call>](#)
- [<sql>](#)

## <compensate>

Invoke a <compensationhandler> from <catch> or <catchall>.

### Syntax

```
<scope>
  <throw fault="BuyersRegret" />
  <faulthandlers>
    <catch fault="BuyersRegret" '>
      <compensate target="RestoreBalance"/>
    </catch>
  </faulthandlers>
  <compensationhandlers>
    <compensationhandler name="RestoreBalance">
      <assign property='context.MyBalance' value='context.MyBalance+1' />
    </compensationhandler>
  </compensationhandlers>
</scope>
```

### Attributes and Elements

#### **target** attribute

Required. The name of a <compensationhandler> that provides a sequence of activities to undo previous actions.

Specify a string of up to 255 characters.

#### **<annotation>** element

See [Common Attributes and Elements](#).

### Description

The <compensate> element invokes a <compensationhandler> block by specifying its name as a target:

#### **XML**

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

<compensate> may only appear within <catch> or <catchall>. Its *target* value must match the *name* of a <compensationhandler> within the same BPL business process.

For details, see [Handling Errors in BPL](#).

## <compensationhandlers> and <compensationhandler>

Provide compensation handlers, each of which performs a sequence of activities to undo a previous action.

### Syntax

```
<scope>
  <throw fault="BuyersRegret" />
  <faulthandlers>
    <catch fault="BuyersRegret" />
      <compensate target="RestoreBalance" />
    </catch>
  </faulthandlers>
  <compensationhandlers>
    <compensationhandler name="RestoreBalance">
      <assign property='context.MyBalance' value='context.MyBalance+1' />
    </compensationhandler>
  </compensationhandlers>
</scope>
```

### Elements

#### <compensationhandler>

Zero or more <compensationhandler> elements may appear inside the <compensationhandlers> container. Each <compensationhandler> element contains a specific sequence of BPL activities that undo a previous action.

In turn, a <compensationhandler> has all the [common attributes and elements](#)

### Description

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.

For details, see [Handling Errors in BPL](#).

**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.

### See Also

- [<catch>](#)
- [<catchall>](#)

- <compensate>
- <faulthandlers>
- <scope>
- <throw>

## <context>

Define general-purpose properties in the business process execution context.

### Syntax

```
<context>
  <property name="P1" type="%String" />
  <property name="P2" type="%String" />
  ...
</context>
```

### Elements

#### <property> element

Optional. Zero or more <property> elements may appear. Each defines one property of the business process execution context.

### Description

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. A BPL business process supports the business process life cycle with a group of variables known as the *execution context*.

The execution context variables include the objects called *context*, *request*, *response*, *callrequest*, *callresponse* and *process*; the integer value *synctimeout*; the collection *syncresponses*; and the %Status value *status*. Each variable has a specific purpose, as described in documentation for the <assign>, <call>, <code>, and <sync> elements.

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 must define. Any value that you want to be persistent and available everywhere within the business process should be declared as a property of the *context* object. You can do this by providing <context> and <property> elements at the beginning of the BPL document, as follows. The resulting BPL code is the same whether you use the Business Process Designer or type the code directly into the BPL document:

- When using the Business Process Designer, you can add properties of various types to the *context* object from the **Context** tab to the right of the BPL diagram. Add whatever properties you need by clicking the plus-sign next to **Context properties**. You can also edit or delete a property using the icons next to its name. The appropriate <context> and <property> elements appear in the generated BPL for the business process.
- You can add <context> and <property> elements together at the beginning of the <process> element, as shown in the following example.

### XML

```
<process request="Demo.Loan.Msg.Application" response="Demo.Loan.Msg.Approval">
  <context>
    <property name="BankName" type="%String"
      initialexpression="BankOfMomAndDad" />
    <property name="IsApproved" type="%Boolean"/>
    <property name="InterestRate" type="%Numeric"/>
    <property name="TheResults"
      type="Demo.Loan.Msg.Approval"
      collection="list"/>
    <property name="Iterator" type="%String"/>
    <property name="ThisResult" type="Demo.Loan.Msg.Approval"/>
  </context>
  ...
</process>
```

Each <property> element defines the name and data type for a property. For a list of available data type classes, see Parameters. You may assign an initial value in the <property> element by providing an *initialexpression* attribute. Alternatively, you may assign values during business process execution, using the <assign> element.

## <continue>

---

Jump to the next iteration within a loop, without exiting the loop.

### Syntax

```
<continue/>
```

### Attributes and Elements

***name, disabled, xpos, ypos, xend, yend* attributes**

See [Common Attributes and Elements](#).

**<annotation> element**

See [Common Attributes and Elements](#).

### Description

BPL syntax permits any element that can contain a sequence of activities—<case>, <default>, <foreach>, <false>, <sequence>, <true>, <until>, or <while>—to contain a <continue> element if desired.

The <continue> element allows the flow of control to jump to the next iteration of a loop, without completing the remaining operations inside the current iteration. For example:

#### XML

```
<foreach property="P1" key="K1">
  //...do various things...
  <if condition="somecondition">
    <true>
      <continue/>
    </true>
  </if>
  //...do various other things...
</foreach>
```

In the above example, it is the <true> element that contains the <continue> element. However, the loop affected by this <continue> is actually the containing <foreach> loop.

The example works as follows: If on some pass through this loop, the <if> element finds “somecondition” to be true (that is, equal to the integer value 1) then the flow of control passes to the <true> element inside the <if>. Upon encountering the <continue> element, execution halts the current pass through the <foreach> loop, proceeds to the next item in the collection (if there is a next item), and begins processing that next item from the beginning of the loop.

Loop activities that you might want to modify by using a <continue> element include <foreach>, <until>, and <while>. The effect of <continue> for each type of loop element is to halt the current pass through the loop, jump to the condition test for the loop, and allow that test and the type of loop to determine what to do next: continue looping, or exit the loop, as normal for that type of loop. For example:

Containing Loop	Behavior of <continue>
<a href="#">&lt;foreach&gt;</a>	Test for the next item in the collection. If an item is found, begin processing it from the top of the loop. However, if there are no more items in the collection that match the test condition, exit the loop.
<a href="#">&lt;until&gt;</a>	Jump to the condition test at the bottom of the loop. If the condition is true, exit the loop; if false, execute the statements in the loop.
<a href="#">&lt;while&gt;</a>	Jump to the condition test at the top of the loop. If the condition is true, exit the loop; if false, execute the statements in the loop.

## See Also

- [<break>](#)

## <default>

---

Perform a set of activities when no matching condition can be found within a <switch> element.

### Syntax

```
<switch>
  <case>
    ...
  </case>
  ...
  <default>
    ...
  </default>
</switch>
```

### Attributes and Elements

***name, disabled, xpos, ypos, xend, yend* attributes**

See [Common Attributes and Elements](#).

**<annotation> element**

See [Common Attributes and Elements](#).

#### Other elements

Optional. <default> may contain zero or more of the following elements in any combination: <alert>, <assign>, <branch>, <break>, <call>, <code>, <continue>, <delay>, <empty>, <flow>, <foreach>, <if>, <label>, <milestone>, <reply>, <rule>, <scope>, <sequence>, <sql>, <switch>, <sync>, <throw>, <trace>, <transform>, <until>, <while>, <xpath>, or <xslt>.

### Description

A <default> element is an optional part of a <switch> element. A <switch> element contains a sequence of one or more <case> elements and an optional <default> element.

When present, the <default> element must be the last element in the <switch>. Correspondingly, in the Business Process Designer, the <default> element must be the right-most option in the <switch> part of the diagram.

When a <switch> element is executed, it evaluates each <case> condition in turn. These conditions are logical expressions in the scripting language of the containing <process> element. If any expression evaluates to the integer value 1 (true), then the contents of the corresponding <case> element are executed; otherwise the expression for the next <case> element is evaluated.

If no <case> condition is true, the contents of the <default> element are executed.

Activities within a <default> element can be any BPL activity listed above, including <assign> elements as in the example below:

## XML

```
<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>

---

Delay execution of a business process for a specified duration or until a future time.

### Syntax

```
<delay duration="PT60S" />
```

Or:

```
<delay until="2020-10-19T10:10" />
```

### Attributes and Elements

#### *duration* attribute

Optional. Specifies the duration of the delay as an expression that evaluates to an XML `duration` value.

For example: `PT60S` for 60 seconds or `P1Y2M3DT10H30M` for 1 year, 2 months, 3 days, 10 hours, and 30 minutes. The `<delay>` element ignores fractional seconds. If duration has a value less than one second, it is treated as 0 seconds.

For details on XML duration values, see appropriate entry in the Primitive Datatypes section of the W3C Recommendation XML Schema Part 2: Datatypes Second Edition, which you can view at the following:

- <https://www.w3.org/TR/xmlschema-2/#duration>
- <https://www.w3.org/TR/xmlschema-2/#dateTime>

#### *until* attribute

Optional. Specifies a future time at which the delay will expire, as an expression that evaluates to an XML `dateTime` value.\*

For example `2023:10:19T10:10`

#### *name, disabled, xpos, ypos, xend, yend* attributes

See [Common Attributes and Elements](#).

#### **<annotation>** element

See [Common Attributes and Elements](#).

### Description

The `<delay>` element suspends execution of a business process (or the current thread within a `<flow>`) for either a specified duration or until a specific time. For example:

#### XML

```
<sequence>
  <annotation>
    <![CDATA[ Write the time now, and sixty seconds later.]]>
  </annotation>
  <trace value="The time is: " & Now' />
  <delay duration="PT60S" />
  <trace value="The time is: " & Now' />
</sequence>
```

---

The <delay> element causes the execution of a business process to pause for either a specific duration (specified by the *duration* attribute) or until a specific future time (specified by the *until* attribute). You must provide either the *duration* attribute or the *until* attribute, or no delay will take place.

During the delay period, execution of the current business process thread is suspended and the state of the business process is saved to the database.

The format for values of *duration* and *until* is discussed at length in World Wide Web Consortium documents about XML data types. For details, see the “Primitive Datatypes” section of the W3C Recommendation XML Schema Part 2: Datatypes Second Edition, which you can view at <https://www.w3.org/TR/xmlschema-2/#built-in-primitive-datatypes>. Some *duration* examples are:

- PT60S or PT1M for one minute
- PT219S or PT3M39S for 3 minutes, 39 seconds

Whenever a <delay> element is executed, the BPL engine inserts the *name* of the <delay> element into the message header so that it is visible in later Message Browser and Visual Trace displays.

## <empty>

---

Perform no action.

### Syntax

```
<empty />
```

### Attributes and Elements

***name, disabled, xpos, ypos, xend, yend* attributes**

See [Common Attributes and Elements](#).

**<annotation> element**

See [Common Attributes and Elements](#).

### Description

The <empty> element performs no operation. Its purpose is to serve as a placeholder within a BPL definition or as a place to hold additional annotation without affecting the execution of the business process. For example.

#### XML

```
<empty>  
  <annotation>This is an empty element.  
  </annotation>  
</empty>
```

## <false>

Perform a set of activities when the condition for an <if> element is false.

### Syntax

```
<if condition="0">
  <true>
    ...
  </true>
  <false>
    ...
  </false>
</if>
```

### Attributes and Elements

*name, disabled, xpos, ypos, xend, yend* attributes

See [Common Attributes and Elements](#).

**<annotation>** element

See [Common Attributes and Elements](#).

#### Other elements

Optional. <false> may contain zero or more of the following elements in any combination: <alert>, <assign>, <branch>, <break>, <call>, <code>, <continue>, <delay>, <empty>, <flow>, <foreach>, <if>, <label>, <milestone>, <reply>, <rule>, <scope>, <sequence>, <sql>, <switch>, <sync>, <throw>, <trace>, <transform>, <until>, <while>, <xpath>, or <xslt>.

### Description

A <false> element is used within an <if> to contain elements that need to be executed if the condition is false.

### See Also

- [<if>](#)
- [<true>](#)

## <faulthandlers>

Provide zero or more <catch> and one <catchall> element to catch faults and system errors.

### Syntax

```
<scope>
  <throw fault="MyFault" />
  ...
  <faulthandlers>
    <catch fault="MyFault" >
      ...
    </catch>
    <catch fault="OtherFault" >
      ...
    </catch>
    <catchall>
      ...
    </catchall>
  </faulthandlers>
</scope>
```

### Attributes and Elements

#### *name, disabled, xpos, ypos, xend, yend* attributes

See [Common Attributes and Elements](#).

#### <catch> element

There may be zero or more <catch> elements inside <faulthandlers>. Each catches a specific, named fault produced by a <throw> element.

#### <catchall> element

Catch a fault or system error that does not match any <catch>. If there are no <catch> elements in <faulthandlers>, there must be a <catchall>. Otherwise, <catchall> is optional.

### Description

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 <faulthandlers> element is intended to catch any errors that activities within the <scope> produce. The <catch> and <catchall> elements within <faulthandlers> may provide <compensate> statements that invoke <compensationhandler> elements to compensate for those errors.

When a <scope> provides no <faulthandlers> block, InterSystems IRIS automatically outputs the system error to the [Event Log](#). When a <scope> *does* contain a <faulthandlers> block, the BPL business process must output <trace> messages to the [Event Log](#) for system error messages to appear there. System error messages appear in the ObjectScript shell, in either case.

For details, see [Handling Errors in BPL](#).

**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.

### See Also

- [<catch>](#)
- [<catchall>](#)

- <compensate>
- <compensationhandlers>
- <scope>
- <throw>

## <flow>

---

Perform activities in a non-determinate order.

### Syntax

```
<flow>
  <sequence name="thread1">
    ...
  </sequence>
  <sequence name="thread2">
    ...
  </sequence>
  ...
</flow>
```

### Attributes and Elements

#### ***name, disabled, xpos, ypos, xend, yend* attributes**

See [Common Attributes and Elements](#).

#### **<annotation> element**

See [Common Attributes and Elements](#).

#### **<sequence> element**

Optional. Zero or more <sequence> elements contain whatever activities are needed for the <flow>. If no <sequence> elements are provided, no action is taken by the <flow>.

### Description

The <flow> element specifies that each of the elements it contains are executed in a non-determinate order. A <flow> element contains one or more <sequence> elements, each of which is referred to as a *thread*.

When you are using the Business Process Designer and you add a <flow> element to the business process, a <sequence> element is automatically inserted inside the <flow>, as you can see by examining the generated BPL code.

If you need to temporarily disable one of the <sequence> elements within a <flow>, you can edit the generated BPL code by setting the *disabled* attribute of the corresponding <sequence> element.

The following abbreviated example shows the usage of the <flow> element. In this hand-coded BPL example, the developer has decided to use two parallel sequences inside the flow. Each is executed in a separate thread: *thread1* and *thread2*.

### XML

```
<process>
  <flow>
    <sequence name="thread1">
      <call name="A" />
      <call name="B" />
    </sequence>
    <sequence name="thread2">
      <call name="C" />
      <call name="A" />
    </sequence>
  </flow>
  <call name="E" />
</process>
```

In this example, the <flow> element defines two threads, specified by <sequence> elements *thread1* and *thread2*. The order in which the two threads are executed is indeterminate (though, of course, the <call> elements *within* the <sequence> elements are executed in sequential order).

If possible, the execution of threads is interlaced. For example, if the execution of one thread is suspended (say it is waiting for a response from a asynchronous call), then execution of one of the other threads proceeds (if possible).

Note that, strictly speaking, the threads within a <flow> element do not execute at the same time: this is because only one thread is given access to the business process execution context at a time, to preserve proper concurrency and data consistency.

**Note:** For more information about the business process execution context, see [<assign>](#) , and see [Developing BPL Processes](#).

The <flow> element waits for all of its threads to complete before it allows execution to continue. After both threads in the previous example are completed, execution continues and <call> element E is executed.

A thread within a <flow> element may contain additional, nested <flow> elements.

For information about using <sync> with <flow>, see documentation of the [<sync>](#) element.

## <foreach>

Define a sequence of activities to be executed iteratively.

### Syntax

```
<foreach property="P1" key="K1">
  ...
</foreach>
```

### Attributes and Elements

#### *property* attribute

Required. The collection property (list or array) to iterate over. It must be the name of a valid object and property in the execution context.

#### *key* attribute

Required. The index used to iterate through the collection. It must be a name of a valid object and property in the execution context. It is assigned a value for each element in the collection.

#### *name, disabled, xpos, ypos, xend, yend* attributes

See [Common Attributes and Elements](#).

#### <annotation> element

See [Common Attributes and Elements](#).

#### Other elements

Optional. <foreach> may contain zero or more of the following elements in any combination: <alert>, <assign>, <branch>, <break>, <call>, <code>, <continue>, <delay>, <empty>, <flow>, <foreach>, <if>, <label>, <milestone>, <reply>, <rule>, <scope>, <sequence>, <sql>, <switch>, <sync>, <throw>, <trace>, <transform>, <until>, <while>, <xpath>, or <xslt>.

### Description

The <foreach> element defines a sequence of activities that are executed iteratively, once for every element within a specified collection property. For example:

#### XML

```
<foreach property="callrequest.Location" key="context.K1">
  <assign property="total"
    value="context.total+context.prices.GetAt(context.K1)"/>
</foreach>
```

The <foreach> element can refer to the following variables and their properties. Do not use variables not listed here.

Variable	Purpose
<i>context</i>	The <i>context</i> object is a general-purpose data container for the business process. <i>context</i> has no automatic definition. To define properties of this object, use the <context> element. That done, you may refer to these properties anywhere inside the <process> element using dot syntax, as in: <code>context.Balance</code>

Variable	Purpose
<i>request</i>	The <i>request</i> object contains any properties of the original request message object that caused this business process to be instantiated. You may refer to <i>request</i> properties anywhere inside the <process> element using dot syntax, as in: <code>request.UserID</code>
<i>response</i>	The <i>response</i> object contains any properties that are required to build the final response message object to be returned by the business process. You may refer to <i>response</i> properties anywhere inside the <process> element using dot syntax, as in: <code>response.IsApproved</code> . Use the <assign> element to assign values to these properties.

**Note:** There is more information about the business process execution context in documentation of the <assign> element.

You can fine-tune loop execution by including <break> and <continue> elements within a <foreach> element. See the descriptions of these elements for details.



Evaluate a condition and perform one action if true, another if false.

## Syntax

```
<if condition="1">
  <true>
    ...
  </true>
  <false>
    ...
  </false>
</if>
```

## Attributes and Elements

### **condition** attribute

Required. An expression that, if true, causes the contents of the <true> element to execute. If false, the contents of the <false> element are executed.

Specify an expression that evaluates to 1 (if true) or 0 (if false).

### **LanguageOverride** attribute

Optional. Specifies the scripting language in which any expressions (within in this element) are written.

Can be "python", "objectscript", or "basic" (not documented). Default is the language specified in the containing <process> element.

### **name, disabled, xpos, ypos, xend, yend** attributes

See [Common Attributes and Elements](#).

### **<annotation>** element

See [Common Attributes and Elements](#).

### **<true>** element

Optional. If the condition is true, activities inside the <true> element are executed.

### **<false>** element

Optional. If the condition is false, activities inside the <false> element are executed.

## Description

The <if> element evaluates an expression and, depending on its value, executes one of two sets of activities (one if the expression evaluates to a true value, the other if it evaluates to a false value).

The <if> element may contain a <true> element and a <false> element which define the actions to execute if the expression evaluates to true or false, respectively.

If both <true> and <false> elements are provided, they may appear within the <if> element in any order.

If the condition is true and there is no <true> element, or if the condition is false and there is no <false> element, no activity results from the <if> element.

The following example shows an <if> element used to coordinate the results of a combination of <call> and <sync> elements used together.

## XML

```
<sequence name="thread1">
  <call name="A" />
  <call name="B" />
  <sync calls="A,B" type="all" timeout="10" />
  // Did the synchronization time out before it finished?
  <if condition='synctimedout="1" '>
    <true>
      <trace value="thread1 timeout: Call A or B did not return." />
    </true>
    // If not, then the calls came back, so assign the results.
    <false>
      <assign property="context.TheResultsFromEast"
        value='syncresponses.GetAt("A")'
        action="append"/>
      <assign property="context.TheResultsFromWest"
        value='syncresponses.GetAt("B")'
        action="append"/>
    </false>
  </if>
</sequence>
```

The <if> activity in this example has a condition that tests the execution context variable *synctimedout* against the integer value 1. *synctimedout* can have the value 0, 1, or 2 as described in the documentation for <call>. If the two values are equal, this <if> condition receives the integer value 1 and statements inside the <true> element are executed. Otherwise, statements inside the <false> element are executed.

**Note:** There is more information about the business process execution context in documentation of the <assign> element.

## See Also

- <true>
- <false>

## <label>

---

Provide a destination for a conditional branch operation.

### Syntax

```
<label name="JumpToMe" />
```

### Attributes and Elements

#### **name attribute**

Required. The name of this label. This name must be unique across the entire BPL business process. Specify a string of up to 255 characters.

#### **name, disabled, xpos, ypos, xend, yend attributes**

See [Common Attributes and Elements](#).

#### **<annotation> element**

See [Common Attributes and Elements](#).

### Description

The <label> element provides the destination for a conditional <branch> element.

For details, see the documentation for <branch>.

## <milestone>

Store a message to acknowledge a step achieved by a business process.

### Syntax

```
<milestone value="'The applicant has been notified of the interest rate.'" />
```

### Attributes and Elements

#### *value* attribute

Required. This is the text for the milestone message. It can be a literal text string or an expression to be evaluated.

#### *LanguageOverride* attribute

Optional. Specifies the scripting language in which any expressions (within in this element) are written.

Can be "python", "objectscript", or "basic" (not documented). Default is the language specified in the containing <process> element.

#### *name, disabled, xpos, ypos, xend, yend* attributes

See [Common Attributes and Elements](#).

#### <annotation> element

See [Common Attributes and Elements](#).

### Description

A <milestone> activity writes a message to the InterSystems IRIS database. <milestone> works very much like <trace>, but unlike <trace> messages, <milestone> messages exist only while the associated business process is running. After the business process exits, all messages generated by <milestone> activities are removed.

Often a programmer uses <trace> messages for diagnostic purposes, whereas <milestone> messages can be helpful to track the progress of a correctly operating, long-running business process.

You can retrieve <milestone> messages by examining values in the ^Ens.Milestone global. The global is defined only if your production has issued <milestone> messages. To obtain the value of ^Ens.Milestone:

- Programmatically, use the information in [Using Multidimensional Storage \(Globals\)](#).
- From the Management Portal, navigate to the **System Explorer > Globals** page, ensure that the **Namespaces** option is selected, and click the name of the namespace where your production runs. The **View Globals** option is selected by default.

## <parameters> and <parameter>

Specifies the parameters for another BPL element as a set of name-value pairs.

### Syntax

```
<parameters>
  <parameter name='MAXLEN' value='1024' />
  <parameter name='MINLEN' value='1' />
</parameters>
```

### Elements

#### <parameter> element

Zero or more <parameter> elements may appear inside the <parameters> container. Each <parameter> element defines one parameter.

Each <parameter> element has two attributes, *name* and *value*, as described below.

### Description

The optional <parameters> element is valid only within <property> or <xslt>. <parameters> defines the parameters for its containing BPL element as a set of name-value pairs:

- Within <context>, <parameters> contains the data type parameters for a <property> that you are defining in the business process execution context. There is a detailed explanation of the business process execution context in documentation of the <assign> element.
- Within <xslt>, <parameters> contains any XSLT name-value pairs that you wish to pass to the stylesheet that controls the XSLT transformation.

<parameters> does not support any BPL attributes. It is simply a container for zero or more <parameter> element, one for each parameter. You may provide as many <parameter> elements as you wish, but all must appear within the same <parameters> block. For example:

### XML

```
<context>
  <property name='Test' type='%Integer' initialexpression='342' >
    <parameters>
      <parameter name='MAXVAL' value='1000' />
    </parameters>
  </property>
  <property name='Another' type='%String' initialexpression='Yo' >
    <parameters>
      <parameter name='MAXLEN' value='2' />
      <parameter name='MINLEN' value='1' />
    </parameters>
  </property>
</context>
```

### <parameter> Attributes

#### *name* attribute

Required. The name of this parameter:

- Within <property>, *name* identifies a data type parameter for the property. For valid names, see Parameters.
- Within <xslt>, *name* must be the name of a valid XSLT parameter.

***value* attribute**

Optional. The value to assign to the parameter.

**See Also**

- [<context>](#)
- [<xslt>](#)

## <process>

---

Define a business process.

### Syntax

```
<process request="MyApp.Request" response="MyApp.Response">
  <context>
    ...
  </context>
  <sequence>
    ...
  </sequence>
</process>
```

### Attributes and Elements

#### ***request attribute***

Required. The name of the request class, specifying the type of the initial request to this business process.

#### ***response attribute***

Optional. The name of the response class, specifying the type of the response returned by this business process, if any.

#### ***component attribute***

Optional. Setting this value to 1 (true) designates this <process> as a [reusable component](#).

Specify a Boolean value: 1 (true) or 0 (false). The default is false.

#### ***contextsuperclass attribute***

Optional. Lets you specify the superclass for your business process context. This is useful if you have many different business processes that share the same [execution context variables](#). The idea is that you subclass `Ens.BP.Context` yourself, adding your own properties, then use that class for the *contextsuperclass*. If not specified, `Ens.BP.Context` is the default. Specify a class name.

#### ***height attribute***

Optional. Refers to the graphical representation of the business process in the Business Process Designer. Specify a positive integer.

#### ***includes attribute***

Optional. A comma-delimited list of ObjectScript include file names, so that you can use macros in your <code> segments.

#### ***language attribute***

Optional. Specifies the default language in which any expressions or <code> elements are written.

Can be "python", "objectscript", or "basic" (not documented). Default is "objectscript".

**layout attribute**

Optional. The name of the layout style used in BPL diagrams for this business process. The value `automatic` indicates that the Business Process Designer and BPL Viewer will choose layouts for the diagram elements. The value `manual` overrides the tools to use the exact layout that you specify. Specify a string, either `manual` or `automatic`. If not specified, the default layout is `automatic`.

**version attribute**

Optional. An integer that expresses a version number. Higher values indicate later versions. If an expression, the `version` attribute value must use ObjectScript. Specify a positive integer. May be a literal integer, or an expression that evaluates to an integer.

**width attribute**

Optional. Refers to the graphical representation of the business process in the Business Process Designer. Specify a positive integer.

**<context> element**

Optional. Defines general-purpose properties in the business process execution context. For information about the business process execution context, see [<assign>](#), and see [Developing BPL Processes](#).

**<pyFromImport> element**

Optional. 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.

**<sequence> element**

Optional. Zero or more `<sequence>` elements may appear. Each defines actions that the business process can perform.

## Description

The `<process>` element is the outermost element for a BPL document. All the other BPL elements are contained within a `<process>` element.

A business process consists of an execution context (defined by the `<context>` element) and a sequence of activities (defined by the `<sequence>` element).

The `request` attribute defines the type (class name) for the business process's initial request. The `response` attribute defines the type (class name) for the eventual response from the business process. The `request` attribute is required, but the `response` attribute is optional, since the business process might not return a response.

### 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. A BPL business process supports the business process life cycle with a group of variables known as the *execution context*.

The execution context variables include the objects called `context`, `request`, `response`, `callrequest`, `callresponse` and `process`; the integer value `synctimedout`; the collection `syncresponses`; and the `%Status` value `status`. Each variable has a specific purpose, as described in documentation for the `<assign>`, `<call>`, `<code>`, and `<sync>` elements.

### Example

The following sample business process provides a `<sync>` element to synchronize several `<call>` elements. Further activities within the `<process>` element are replaced by ellipses (...) near the end of the example:

## XML

```

<process request="Demo.Loan.Msg.Application">
<context>
  <property name="BankName" type="%String"/>
  <property name="IsApproved" type="%Boolean"/>
  <property name="InterestRate" type="%Numeric"/>
  <property name="TheResults" type="Demo.Loan.Msg.Approval" collection="list"/>
  <property name="Iterator" type="%String"/>
  <property name="ThisResult" type="Demo.Loan.Msg.Approval"/>
</context>
<sequence>
  <trace value='"received application for "_request.Name'/'>
  <call name="BankUS" target="Demo.Loan.BankUS" async="1">
    <annotation>
      <![CDATA[Send an asynchronous request to Bank US.]]>
    </annotation>
    <request type="Demo.Loan.Msg.Application">
      <assign property="callrequest" value="request"/>
    </request>
    <response type="Demo.Loan.Msg.Approval">
      <assign property="context.TheResults"
        value="callresponse"
        action="append"/>
    </response>
  </call>

  <call name="BankSoprano" target="Demo.Loan.BankSoprano" async="1">
    <annotation>
      <![CDATA[Send an asynchronous request to Bank Soprano.]]>
    </annotation>
    <request type="Demo.Loan.Msg.Application">
      <assign property="callrequest" value="request"/>
    </request>
    <response type="Demo.Loan.Msg.Approval">
      <assign property="context.TheResults"
        value="callresponse"
        action="append"/>
    </response>
  </call>

  <call name="BankManana" target="Demo.Loan.BankManana" async="1">
    <annotation>
      <![CDATA[Send an asynchronous request to Bank Manana.]]>
    </annotation>
    <request type="Demo.Loan.Msg.Application">
      <assign property="callrequest" value="request"/>
    </request>
    <response type="Demo.Loan.Msg.Approval">
      <assign property="context.TheResults"
        value="callresponse"
        action="append"/>
    </response>
  </call>

  <sync name='Wait for Banks'
    calls="BankUS,BankSoprano,BankManana"
    type="all"
    timeout="5">
    <annotation>
      <![CDATA[Wait for responses. Wait up to 5 seconds.]]>
    </annotation>
  </sync>
  <trace value='"sync complete"/'>
  ...
</sequence>
</process>

```

## Replies

The *primary response* from a business process is the response it returns to the request that originally invoked the specific business process instance. Normally, the business process returns its primary response automatically, as soon as it is done executing. However, the `<reply>` element can be used to return the primary response sooner. This can be useful if the response needed by the original caller is ready to be returned, but there is additional work for the business process to perform as a result of the original call.

## Language

The <process> element defines the scripting language used by a business process by providing a value for the language attribute: The value should be "objectscript". Any expressions found in the business process, as well as lines of code within <code> elements, must use the specified language.

## Versioning

Developers can update the *version* number for a BPL business process to indicate that its new functionality is incompatible with previous versions. A higher number indicates later versions. There is no automatic versioning of BPL business processes. A developer manually updates the value of the *version* attribute within the BPL <process> element to highlight the fact that the new code contains changes that are incompatible with previous versions of the same business process. Examples include adding or deleting properties within the business process <context>, or changing the flow of activities within the business process <sequence>.

Prior versions of the same BPL business process that have instances already executing continue to execute their original activities, with their original context. New versions use their own context and their own activities. InterSystems IRIS achieves this by generating new context and thread classes for each version. The version appears as a subpackage in the generated class hierarchy. For example, if you have a class MyBPL, version 3 generates MyBPL.V3.Context and MyBPL.V3.Thread1.

## Layout

By default, when a user opens a BPL diagram in the Business Process Designer, the tool display the diagram using automatic layout arrangements. These automatic choices may or may not be appropriate for a particular drawing. If you suspect that this may be an issue for your diagram, you can disable automatic layout to ensure that your diagram always displays with exactly the layout you want.

The most direct way to control the layout of your diagram is to clear the **Auto arrange** check box on the **Preferences** tab.

You can also click the General tab and choose either Automatic or Manual for the Layout. The manual selection preserves the exact position of each element each time you save the diagram, so that when the diagram is displayed in the Business Process Designer, it does not take on any layout characteristics except any that you specify.

Problems in scrolling through a business process diagram in the Business Process Designer can be fixed by adjusting the height or width attributes of the <process> element. You can do this using the General tab as for the layout attribute.

## <property>

Define a property within the <context> element for a business process.

### Syntax

```
<property name='Test' type='%Integer' initialexpression='342' >
  <parameters>
    <parameter name='MAXVAL' value='1000' />
  </parameters>
</property>
```

### Attributes and Elements

#### **name attribute**

Required. The name of this property. It must be a valid property name.

#### **type attribute**

Optional. The name of the class that specifies the type of this property. It can be a data type class (%String) or a serial or persistent class.

#### **initialexpression attribute**

Optional. This ObjectScript expression is evaluated to provide a default value for the property. Specify an expression that provides a valid value for the property. See the discussion below.

#### **instantiate attribute**

Optional. Acts as a `create` flag for the property. If not specified, the default is 0 (do not create). Specify 1 (create) or 0 (do not create)

#### **collection attribute**

Optional. If present, specifies that this property is a collection of a certain type. Specify a literal string, either `list`, `array`, `binarystream`, or `characterstream`

#### **<parameters>**

An optional <parameters> element may appear. Inside the <parameters> container, zero or more <parameter> elements may appear. Each <parameter> element defines one data type parameter for the property by providing a parameter *name* and *value*. For valid names and values, see Parameters.

### Description

The <property> element defines a property within the business process 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. A BPL business process supports the business process life cycle with a group of variables known as the *execution context*.

The execution context variables include the objects called *context*, *request*, *response*, *callrequest*, *callresponse* and *process*; the integer value *synctimeout*; the collection *syncresponses*; and the %Status value *status*. Each variable has a specific purpose, as described in documentation for the <assign>, <call>, <code>, and <sync> elements.

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 must define. Any value that you want to be persistent and available everywhere within the business process should be declared as a property of the *context* object. You can do

this by providing [<context>](#) and [<property>](#) elements at the beginning of the BPL document. Each [<property>](#) element defines one property of the *context* object.

A [<property>](#) element must provide a name.

For non-collection properties, the *initialexpression* and *instantiate* attributes dictate how the object will be initialized. If the *instantiate* attribute has the integer value 1 (true), then a call to “new” the object will be generated. If an *initialexpression* attribute is specified as well, then the result of this expression will be assigned to the object.

The *instantiate* attribute should be used to initialize properties that can be instantiated, whereas the *initialexpression* attribute should be used to initialize data type classes such as %String. For string values, be sure to provide the string quotes wrapped inside another set of quotes. That is: *initialexpression*=' "hello" ' to set an initial string value of "hello".

If the *collection* attribute is set (*list*, *array*, *binarystream*, or *characterstream*) the property is automatically instantiated as a collection of that type.

The following example shows a set of [<property>](#) elements within the [<context>](#) element at the beginning of a business process:

## XML

```
<process request="Demo.Loan.Msg.Application" response="Demo.Loan.Msg.Approval">
  <context>
    <property name="BankName" type="%String"
      initialexpression="BankOfMomAndDad" />
    <property name="IsApproved" type="%Boolean"/>
    <property name="InterestRate" type="%Numeric"/>
    <property name="TheResults"
      type="Demo.Loan.Msg.Approval"
      collection="list"/>
    <property name="Iterator" type="%String"/>
    <property name="ThisResult" type="Demo.Loan.Msg.Approval"/>
  </context>
  ...
</process>
```

Each [<property>](#) element defines the name and data type for a property. For a list of available data type classes, see [Parameters](#). [<property>](#) may assign an initial value by providing an *initialexpression* attribute. Alternatively, you may assign values during business process execution, using the [<assign>](#) element.

## See Also

- [<parameters>](#)

## <pyFromImport>

---

Specifies optional Python `from / import` statements.

### Syntax

```
<pyFromImport>  
from math import cos  
</pyFromImport>
```

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.

## <reply>

Send a response from a business process before its execution is complete.

### Syntax

```
<reply/>
```

### Attributes and Elements

***name, disabled, xpos, ypos, xend, yend* attributes**

See [Common Attributes and Elements](#).

**<annotation> element**

See [Common Attributes and Elements](#).

### Description

The “primary response” from a business process is the response it returns to the request that originally invoked the specific business process instance. Normally, the business process will return its primary response automatically, as soon as it is done executing. However, the <reply> element can be used to return the primary response sooner. This can be useful if the response needed by the original caller is ready to be returned, but there is additional work for the business process to perform as a result of the original call.

The <reply> element returns the *response* object from the business process execution context, so a business process must use the <assign> element to assign values to properties on the *response* object, prior to making a <reply>.

**Note:** There is more information about the business process execution context in the documentation for [<assign>](#).

### Example

The following example shows the reply action, used to return a response before continuing to execute the rest of the business process:

```
<call name="FindSalary" target="MyApp.PayrollApp" async="1">
  <request type="MyApp.SalaryRequest">
    <assign property="callrequest.Name" value="request.Name" />
    <assign property="callrequest.SSN" value="request.SSN" />
  </request>
  <response type="MyApp.SalaryResponse">
    <assign property="context.Salary" value="callresponse.Salary" />
  </response>
</call>
<assign property="response.Salary" value="context.Salary" />
</reply>
<call name="UpdateSalaryCache" target="MyApp.PayrollApp" async="0">
  <request type="MyApp.SalaryCacheRequest">
    <assign property="callrequest.SSN" value="request.SSN" />
    <assign property="callrequest.Salary" value="context.Salary" />
  </request>
</call>
```

## <request>

---

Prepare a request within a <call> element.

### Syntax

```
<call name="Call" target="MyApp.MyOperation" async="1">
  <request type="MyApp.Request">
    ...
  </request>
  <response type="MyApp.Response">
    ...
  </response>
</call>
```

### Attributes and Elements

#### **type attribute**

Required. The name of the request message class.

#### **name attribute**

Optional. The name of the <request> element. Specify a string of up to 255 characters.

#### **Other elements**

Optional. <request> may contain zero or more of the following elements in any combination: <assign>, <empty>, <milestone>, or <trace>.

### Description

A <request> element is a required child element of <call>. Inside the <call> context, the <request> element specifies the type (class name) of the request to send. The <request> element can also contain one or more <assign> elements. Each of these assigns a value to a property on an object in the business process execution context. For example:

#### **XML**

```
<call name="FindSalary" target="MyApp.PayrollApp" async="1">
  <request type="MyApp.SalaryRequest">
    <assign property="callrequest.Name" value="request.Name" />
    <assign property="callrequest.SSN" value="request.SSN" />
  </request>
  <response type="MyApp.SalaryResponse">
    <assign property="context.Salary" value="callresponse.Salary" />
  </response>
</call>
```

The intention of any <assign> elements found within a <request> element is usually to assign values to properties on the *callrequest* object. This object is the member of the business process execution context that acts as a container for the properties of the request object used for the call. However, properties on the *context*, *request*, and *response* objects can also be set, appended, or otherwise manipulated in an <assign> element inside a <request>.

For further discussion of the business process execution context, see the documentation for <call> and <assign>.

### See Also

- <process>
- <reply>

## <response>

Handle a response received within a <call> element.

### Syntax

```
<call name="Call" target="MyApp.MyOperation" async="1">
  <request type="MyApp.Request">
    ...
  </request>
  <response type="MyApp.Response">
    ...
  </response>
</call>
```

### Attributes and Elements

#### **type** attribute

Required. The name of the response message class.

#### **name** attribute

Optional. The name of the <response> element. Specify a string of up to 255 characters.

#### **Other elements**

Optional. <response> may contain zero or more of the following elements in any combination: <assign>, <empty>, <milestone>, or <trace>.

### Description

A <response> element is an optional child element of <call>. Inside the <call> context, the <response> element specifies the type (class name) of the response to return from the call. The <response> element can also contain one or more <assign> elements. For example:

#### **XML**

```
<call name="FindSalary" target="MyApp.PayrollApp" async="1">
  <request type="MyApp.SalaryRequest">
    <assign property="callrequest.Name" value="request.Name" />
    <assign property="callrequest.SSN" value="request.SSN" />
  </request>
  <response type="MyApp.SalaryResponse">
    <assign property="context.Salary" value="callresponse.Salary" />
  </response>
</call>
```

When a call returns a response to the calling business process, any output parameters from the message type named in the <response> element become properties of the *callresponse* object in the business process execution context. Since *callresponse* only has meaning inside the <response> element, to preserve these values the <response> element must provide <assign> elements that assign *callresponse* values to properties of other, more permanent objects in the business process execution context, usually *context* or *response*.

For further discussion, see the documentation for <call> and <assign>.

While a <request> element is required inside every <call>, a <response> is not. If the <response> element is omitted from a <call> element, no response is returned from the <call>, even if the <request> type is designed to return a response. When the <request> is asynchronous, the <assign> elements within the body of the <response> element are executed only after the call response is received. There is no guarantee when this will occur, so a business process will typically use the <sync> element to wait for an asynchronous response.

If a response is not received within the timeout period specified by the <sync> element, then the assignments defined by the corresponding <response> block will not be executed. The response itself will be marked with a status of Discarded.

## See Also

- [<process>](#)
- [<reply>](#)

## <rule>

Refer to a production business rule class.

### Syntax

```
<rule name="ApproveLoan"
      rule="LoanApproval"
      resultLocation="context.Answer"
      reasonLocation="context.Reason">
</rule>
```

### Attributes and Elements

#### **name attribute**

Required. The name of the <rule> element.

#### **rule attribute**

Required. The name of the business rule to be executed. This must be a valid rule within the namespace; see [Identifying the Rule](#), below. If the rule is not defined or otherwise cannot be found at runtime, the rule will return a default value of "" (an empty string).

#### **ruleContext attribute**

Optional. If defined, this is an expression that identifies the object to pass to the rules engine; see [Identifying the Context](#), below. For example:

```
context.MyObject
```

By default the rule passes the business process execution context to the rules engine.

#### **resultLocation attribute**

Optional. The location in which to store the return value of the rule. Typically this is a property within the business process execution context; that is, context.MyValue.

Specify the name of a valid property and object, usually within the business process execution context.

#### **reasonLocation attribute**

Optional. The location in which to store the reason returned by the rule. The rule reason is a string indicating why a business rule reached its decision. For example, "Rule 1" or "Default". If the business rule is empty (for example, it is a rule set that contains no rules) then the reason given for the decision is `Rule Missing`.

#### **disabled, xpos, ypos, xend, yend attributes**

See [Common Attributes and Elements](#).

#### **<annotation> element**

See [Common Attributes and Elements](#).

### Description

The <rule> element invokes a business rule from a business process. When a <rule> executes, it invokes its associated business rule (named by the *rule* attribute) and gets its response immediately (in the same manner as a <code> or <assign> activity).

## Identifying the Rule

When you use the `<rule>` element in BPL, the value of the *rule* attribute can be either of the following:

- A simple **Rule Name**:

```
MyRule
```

- A full **Package Name** plus **Rule Name** combination:

```
MyClassPackage.Organization.Levels.MyRule
```

If any `<rule>` element identifies a simple **Rule Name**, InterSystems IRIS automatically prepends a **Package Name** that is equal to the full package and class name of the BPL business process that contains that `<rule>` element. That is:

```
BPLFullPackageAndClassName.MyRule
```

This combination must identify a valid rule within the namespace, or the return value of the `<rule>` will be a null string.

## Identifying the Context

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.

## A Simple Example

The following is a BPL excerpt showing the use of the `<rule>` activity with a `<switch>` element to process the results from the rule:

```
<sequence>
  <rule name="ExecuteRule"
        rule="MyRule"
        resultLocation="context.MyResult" />
  <switch>
    <case condition="context.MyResult=1">
      <!-- ...Rule is true... -->
    </case>
    <default>
      <!-- ... Rule is false... -->
    </default>
  </switch>
</sequence>
```

The `<rule>` activity in this example returns a Boolean value (true or false) according to InterSystems IRIS conventions. That is, an integer value of 1 means true; 0 means false. All rules return a single value, as in this example, but the type need not be Boolean. The single value returned from a rule may be *any literal value* such as an integer number, decimal number, or text string.

## Return Values

The result and reason for the result are stored in the variables identified by the *resultLocation* and *reasonLocation* attributes, respectively. Usually, these attributes give the names of properties in the *context* variable. This is the general-purpose, persistent variable that you define at the beginning of the BPL business process using `<context>` and `<property>` elements.

## See Also

- [Developing Business Rules](#)

## <sequence>

Perform activities in sequential order.

### Syntax

```
<sequence>
  . . .
</sequence>
```

### Attributes and Elements

#### ***name, disabled, xpos, ypos, xend, yend* attributes**

See [Common Attributes and Elements](#).

#### **<annotation> element**

See [Common Attributes and Elements](#).

#### **Other elements**

Optional. <sequence> may contain zero or more of the following elements in any combination: <alert>, <assign>, <branch>, <break>, <call>, <code>, <continue>, <delay>, <empty>, <flow>, <foreach>, <if>, <label>, <milestone>, <reply>, <rule>, <scope>, <sequence>, <sql>, <switch>, <sync>, <throw>, <trace>, <transform>, <until>, <while>, <xpath>, or <xslt>.

### Description

A <sequence> element is used within a <process> or a <flow> to contain elements that need to be executed in sequential order.

#### **<sequence> and <process>**

Every BPL document must have at least one <sequence> element within its <process> element that specifies the main sequence of activities for the business process. For example:

#### **XML**

```
<process>
  <sequence>
    <call name="A" />
    <call name="B" />
  </sequence>
</process>
```

When you use the Business Process Designer, as you add activities between the <start> and <end> elements of a new, top-level BPL diagram, everything you add is contained within a single top-level <sequence> that the BPL code generator places inside the <process> element in the generated code. Such a <sequence> is shown in the preceding example. The <call> element A is executed first, followed by the <call> element B.

When you use the Business Process Designer, if you need to temporarily disable the top-level <sequence> within a <process>, you can disable it in [your IDE](#) in the generated BPL code by adding the *disabled* attribute to the corresponding <sequence> element.

## Nested `<sequence>` Elements

A `<sequence>` can contain other sequences. Nested `<sequence>` elements do not start additional execution threads; for that you need the `<flow>` element. However, you can use superfluous nested `<sequence>` elements as a means to group items within a BPL document. For example:

### XML

```
<process>
  <sequence>
    <sequence>
      <call name="A" />
      <call name="B" />
    </sequence>
  </sequence>
</process>
```

Nested `<sequence>` elements have no effect on the code generated for the business process. The BPL diagram, however, displays such nested sequences as a single `<sequence>` icon. You can drill down into the `<sequence>` icon to view the elements contained within.

### `<sequence>` and `<flow>`

When you are using the Business Process Designer and you add a `<flow>` element to the business process, a `<sequence>` element is automatically inserted inside the `<flow>`, as you can see by examining the generated BPL code. You may add additional `<sequence>` elements to the flow; in fact, each branch of the `<flow>` *must* be enclosed within its own `<sequence>` element.

If you need to temporarily disable one of the `<sequence>` elements within a `<flow>`, you can do it in [your IDE](#) in the generated BPL code by adding the *disabled* attribute and setting it to true in the corresponding `<sequence>` element.

## <scope>

Define the error handling mechanisms for a sequence of activities.

### Syntax

```
<scope>
  <throw fault="MyFault" />
  ...
  <faulthandlers>
    <catch fault="MyFault" />
    ...
  </faulthandlers>
</scope>
```

### Attributes and Elements

#### *name, disabled, xpos, ypos, xend, yend* attributes

See [Common Attributes and Elements](#).

#### <annotation> element

See [Common Attributes and Elements](#).

#### Other elements

Optional. <scope> may contain zero or more of the following elements in any combination: <alert>, <assign>, <branch>, <break>, <call>, <code>, <continue>, <delay>, <empty>, <flow>, <foreach>, <if>, <label>, <milestone>, <reply>, <rule>, <scope>, <sequence>, <sql>, <switch>, <sync>, <throw>, <trace>, <transform>, <until>, <while>, <xpath>, or <xslt>.

### Description

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 and 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>:

#### 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 fault handler" />
          <trace value=
            "%LastError" _
            $System.Status.GetErrorCodes(..%Context.%LastError)_
            " : "_
            $System.Status.GetOneStatusText(..%Context.%LastError) '
          />
        </catchall>
      </faulthandlers>
    </scope>
```

```
    <trace value="after scope" />
  </sequence>
</process>
}
```

When a `<scope>` provides no `<faulthandlers>` block, InterSystems IRIS automatically outputs the system error to the [Event Log](#). When a `<scope>` *does* contain a `<faulthandlers>` block, the BPL business process must output `<trace>` messages to the [Event Log](#) for system error messages to appear there. System error messages do appear in the ObjectScript shell, in either case.

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.

For details, see [Handling Errors in BPL](#).

## See Also

- [<catch>](#)
- [<catchall>](#)
- [<compensate>](#)
- [<compensationhandlers>](#)
- [<faulthandlers>](#)
- [<throw>](#)

## <sql>

Execute an embedded SQL SELECT statement.

### Syntax

```

<sql name="LookUp">
  <![CDATA[
    SELECT SSN INTO :context.SSN
    FROM MyApp.PatientTable
    WHERE PatID = :request.PatID  ]]>
</sql>

```

### Attributes and Elements

#### *name, disabled, xpos, ypos, xend, yend* attributes

See [Common Attributes and Elements](#).

#### <annotation> element

See [Common Attributes and Elements](#).

### Description

The <sql> element executes an arbitrary embedded SQL SELECT statement from within the execution of a business process.

The <sql> element is especially powerful for performing lookup operations using tables. For example, suppose the primary request coming into a business process provides a PatId property that indicates a Patient Identity number, and you need to find the matching Social Security number (SSN) before the business process can perform work. If you have available a PatientTable table relating PatId with SSN, you can perform the lookup using the following <sql> element:

#### XML

```

<process>
  <sql name="LookUp"><![CDATA[
    SELECT SSN INTO :context.SSN
    FROM MyApp.PatientTable
    WHERE PatID = :request.PatID
  ]]>
</sql>
</process>

```

Where the execution context variable *context* has an SSN property that is suitable to receive the result of the SQL query. The execution context variable *request* automatically contains the PatId property, as it always contains the properties received in the primary request object.

**Note:** For more information about the business process execution context, see <assign>, and see [Developing BPL Processes](#).

If you maintain a local copy of the PatientTable within the InterSystems IRIS database, the above example is especially efficient, as it can be executed without using any expensive network operations or additional middleware.

To use the <sql> element effectively, keep the following tips in mind:

- Always use the fully qualified name of the table, including both the SQL schema name and table name, as in:

```
MyApp.PatientTable
```

Where *MyApp* is the SQL schema name and *PatientTable* is the table name.

- The contents of the <sql> element must contain a valid embedded SQL SELECT statement.

It is convenient to place the SQL query within a CDATA block so that you do not have to worry about escaping special XML characters.

- If the SQL returns a SQLCODE error, this action will also return an error which can be handled using [BPL error handling](#).
- Any tables listed in the SQL query's FROM clause must either be stored within the local InterSystems IRIS database or linked to an external relational database using the SQL Gateway.
- Within the INTO and WHERE clauses of the SQL query, you can refer to a property of one of the variables in the business process execution context by placing a colon (:) in front of the variable name. For example:

### XML

```
<sql name="LookUp"><![CDATA[
  SELECT Name INTO :response.Name
  FROM MainFrame.EmployeeRecord
  WHERE SSN = :request.SSN AND City = :request.Home.City
]]>
</sql>
```

- Only the first row returned by the query will be used. Make sure that your WHERE clause correctly specifies the desired row.

## <switch>

Evaluate a set of conditions to determine which of several actions to perform.

### Syntax

```
<switch>
  <case>
    ...
  </case>
  ...
  <default>
    ...
  </default>
</switch>
```

### Attributes and Elements

***name, disabled, xpos, ypos, xend, yend* attributes**

See [Common Attributes and Elements](#).

**<annotation> element**

See [Common Attributes and Elements](#).

**<case> element**

Required (at least one). Each <case> element defines a condition that may or may not be true.

**<default> element**

Optional. Specifies the action to take if no <case> condition is satisfied. If present, must appear last in the <switch> element.

### Description

The <switch> element contains a sequence of one or more <case> elements and an optional <default> element.

When a <switch> element is executed, it evaluates each <case> condition in turn. These conditions are logical expressions in the scripting language of the containing <process> element. If any expression evaluates to the integer value 1 (true), then the contents of the corresponding <case> element are executed; otherwise the expression for the next <case> element is evaluated.

If no <case> condition is true, the contents of the <default> element are executed.

As soon as one of <case> elements is executed, execution control leaves the surrounding <switch> statement. If no <case> condition matches, control leaves the <switch> after the <default> activity executes.

If no <case> is true and there is no <default>, no activity results from the <switch> statement.

Activities within a <case> element can be any BPL activity, including <assign> elements as in the example below:

**XML**

```
<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>
```

## <sync>

Wait for a response from one or more asynchronous requests.

### Syntax

```

<sequence>
  <call name="A" async="1" />
  <call name="B" async="1" />
  ...
  <sync calls="A,B" type="all" timeout="3600"/>
</sequence>

```

### Attributes and Elements

#### **calls** attribute

Required. A list of the names of one or more asynchronous [<call>](#) elements that [<sync>](#) will wait for. Specify a comma-separated list of [<call>](#) element names. This value can be provided as a literal string, or by using the ObjectScript [@](#) indirection operator to refer to the value of an [execution context variable](#). See details below.

#### **allowresync** attribute

Optional. If true, the [<sync>](#) element can “poll” repeatedly to detect completion of an asynchronous call. That is, you can [<sync>](#) repeatedly on the same call. This feature is useful when a call may take an indefinite time to complete. The default *allowresync* value is false. Specify 1 (true) or 0 (false).

#### **timeout** attribute

Optional. Specifies the time, in seconds, to wait for the responses, as an expression that evaluates to an XML `xsd:dateTime` value. For example: `2023:10:19T10:10`.

#### **type** attribute

Optional. Specify either `"all"` (the default) or `"any"`

#### **name, disabled, xpos, ypos, xend, yend** attributes

See [Common Attributes and Elements](#).

#### **<annotation>** element

See [Common Attributes and Elements](#).

### Description

A typical business process makes one or more requests to external systems. These requests are usually made asynchronously to compensate for the fact that the external system may be slow to respond or occasionally unavailable. The [<sync>](#) element provides an easy way to wait for a response from one or more asynchronous calls. It is used in conjunction with the [<call>](#) element.

The behavior of the [<sync>](#) element is specified via the `calls`, `timeout`, and `type` attributes. The `type` attribute either has the value `all`, which specifies that the [<sync>](#) should wait for a response from all calls, or `any`, which specifies that it will only wait for the first response it receives (in this case, the remaining responses are discarded in the same manner as an expired timeout).

The following BPL fragment makes a two asynchronous requests, A and B, and then uses the [<sync>](#) element to wait for their responses (for up to one hour):

## XML

```
<sequence>
  <call name="A" async="1" />
  <call name="B" async="1" />
  <sync calls="A,B" type="all" timeout="3600" />
</sequence>
```

Any responses received after the timeout period are marked with a status of Discarded, and are not processed by the business process. If no value is provided for a timeout, the <sync> element continues to wait until all responses are received, regardless of the amount of time that passes. Meanwhile, however, the business process is saved to disk and the job in which it was running is freed up to host other business processes while the <sync> element is waiting.

The following sample BPL <process> issues two calls, then waits for 5 seconds.

## XML

```
<process request="Demo.Loan.Msg.Application">
  <context>
    <property name="BankName" type="%String"/>
    <property name="IsApproved" type="%Boolean"/>
    <property name="InterestRate" type="%Numeric"/>
    <property name="Results" type="Demo.Loan.Msg.Approval" collection="list"/>
    <property name="Iterator" type="%String"/>
    <property name="ThisResult" type="Demo.Loan.Msg.Approval"/>
  </context>
  <sequence>
    <trace value='received application for "_request.Name"/>
    <call name="BankUS" target="Demo.Loan.BankUS" async="1">
      <annotation>
        <![CDATA[Send an asynchronous request to Bank US.]]>
      </annotation>
      <request type="Demo.Loan.Msg.Application">
        <assign property="callrequest" value="request"/>
      </request>
      <response type="Demo.Loan.Msg.Approval">
        <assign property="context.Results"
          value="callresponse"
          action="append"/>
      </response>
    </call>

    <call name="BankSoprano" target="Demo.Loan.BankSoprano" async="1">
      <annotation>
        <![CDATA[Send an asynchronous request to Bank Soprano.]]>
      </annotation>
      <request type="Demo.Loan.Msg.Application">
        <assign property="callrequest" value="request"/>
      </request>
      <response type="Demo.Loan.Msg.Approval">
        <assign property="context.Results"
          value="callresponse"
          action="append"/>
      </response>
    </call>

    <call name="BankManana" target="Demo.Loan.BankManana" async="1">
      <annotation>
        <![CDATA[Send an asynchronous request to Bank Manana.]]>
      </annotation>
      <request type="Demo.Loan.Msg.Application">
        <assign property="callrequest" value="request"/>
      </request>
      <response type="Demo.Loan.Msg.Approval">
        <assign property="context.Results"
          value="callresponse"
          action="append"/>
      </response>
    </call>

    <sync name='Wait for Banks'
      calls="BankUS,BankSoprano,BankManana"
      type="all"
      timeout="5">
      <annotation>
        <![CDATA[Wait for responses from the banks. Wait up to 5 seconds.]]>
      </annotation>
    </sync>
```

```

<trace value='"sync complete"' />
</sequence>
</process>

```

Whenever a <sync> element is executed, the BPL engine inserts the *name* of the <sync> element into the message header so that it is visible in later Message Browser and Visual Trace displays.

### Unique Names for <call> Elements

If you attempt to define a <call> element with the same *name* as another <call> element, the BPL editor displays an error message and requires that you provide a unique name.

### Indirection in the calls Attribute

The value of the *calls* attribute is a string. This string must provide a comma-separated list of <call> element names. The string can be a literal value:

```
calls="BankUS, BankSoprano, BankManana"
```

Or the @ indirection operator can be used to access the value of an [execution context variable](#) that contains the appropriate string:

```
calls="@context.myListOfCalls"
```

### syncresponses

There is an additional mechanism for dealing with responses received as a result of the <call> element.

Whenever the <sync> element is used, it fills a collection with the various responses it receives. This collection is a variable in the business process execution context called *syncresponses*. The execution context also provides a integer variable called *synctimedout*. The two variables *synctimedout* and *syncresponses* work together as follows:

Object	Description
<i>syncresponses</i>	<i>syncresponses</i> is a collection of response objects, keyed by the names of the <call> activities being synchronized. Only completed calls are represented. You can retrieve a response from <i>syncresponses</i> only after a <sync> and before the end of the current <sequence>. Do so using the syntax <code>syncresponses.GetAt("MyName")</code> where the relevant call was defined as <code>&lt;call name="MyName"&gt;</code>
<i>synctimedout</i>	<p>The <i>synctimedout</i> value is an integer. <i>synctimedout</i> indicates the outcome of a &lt;sync&gt; activity after several calls. You can test the value of <i>synctimedout</i> after the &lt;sync&gt; and before the end of the &lt;sequence&gt; that contains the calls and &lt;sync&gt;. <i>synctimedout</i> has one of three values:</p> <ul style="list-style-type: none"> <li>• If 0, no call timed out. All the calls had time to complete. This is also the value if the &lt;sync&gt; activity had no <i>timeout</i> set.</li> <li>• If 1, at least one call timed out. This means not all &lt;call&gt; activities completed before the timeout.</li> <li>• If 2, at least one call was interrupted before it could complete.</li> </ul> <p>Generally you will test <i>synctimedout</i> for status and then retrieve the responses from completed calls out of the <i>syncresponses</i> collection.</p>

As soon as a <sync> activity executes, the *syncresponses* collection is cleared in preparation for new responses. As the calls return, their responses go into the *syncresponses* collection. When the <sync> activity completes, *syncresponses* may contain some or all of the responses that you were waiting for.

For example, suppose you `<sync>` on `Call1` and `Call2` with this syntax:

```
<sync type="all" timeout="60">
```

Suppose that `Call1` returns within 60 seconds, but `Call2` does not. At this point, `syncresponses` contains the response to `Call1`, but not `Call2`. You can test the value of `synctimeout` to determine whether or not to expect the appropriate values to be present in `syncresponses`.

Following a `<sync>` activity, you can access whatever responses have returned by using the name of the `<call>` activity as a key. For a call defined as:

```
<call name="nameOfCall">
```

You would access the response using this syntax:

```
syncresponses.GetAt("nameOfCall")
```

Suppose the following sequence executes:

## XML

```
<sequence>
  <call name="A" async="1" />
  <call name="B" async="1" />
  <call name="C" async="1" />
  <sync calls="A,B,C" type="all" />
</sequence>
```

After the `<sync>` element completes, the `syncresponses` collection will contain references to three response objects, as follows:

- `syncresponses.GetAt("A")` = Response from A (if any)
- `syncresponses.GetAt("B")` = Response from B (if any)
- `syncresponses.GetAt("C")` = Response from C (if any)

If no responses were received, the `syncresponses` collection will be empty.

**Note:** For more information about the business process execution context, see `<assign>`, and see [Developing BPL Processes](#).

## *syncresponses in Multiple Threads*

When you use the `<sync>` element in conjunction with `<flow>`, be aware that there is a separate `syncresponses` collection for each thread, including the primary thread in which the `<process>` itself executes. Therefore, in the course of a business process there may be different `syncresponses` collections that go in and out of scope; each has relevance only in its immediate `<sequence>` and not in any other.

The following example illustrates the use of `synctimeout` and `syncresponses` in three threads, the primary business process thread and two additional threads created by a `<flow>`:

### Class Member

```
XData BPL
{
<process>
  <context>
    <property name="ResultsFromNorth" type="%String"/>
    <property name="ResultsFromSouth" type="%String"/>
    <property name="ResultsFromEast" type="%String"/>
    <property name="ResultsFromWest" type="%String"/>
  </context>
  <sequence>
    // In this context, syncresponses refers to the primary process thread
    <flow>
      // This flow runs two sequences (two threads) in parallel
      <sequence name="thread1">
```

```

// In this context, syncresponses refers to results in thread1
<call name="A" />
<call name="B" />
<sync calls="A,B" type="all" timeout="10" />
// Did the synchronization time out before it finished?
<if condition='synctimedout="1"'>
  <true>
    <trace value='"thread1 timeout: Call A or B did not return.'" />
  </true>
  // If not, then the calls came back, so assign the results.
  <false>
    <assign property="context.ResultsFromEast"
      value='syncresponses.GetAt("A")'
      action="append"/>
    <assign property="context.ResultsFromWest"
      value='syncresponses.GetAt("B")'
      action="append"/>
  </false>
</if>
</sequence>

<sequence name="thread2">
// In this context, syncresponses refers to results in thread2
<call name="C" />
<call name="A" />
<sync calls="C,A" type="all"/>
// Assign the results
<assign property="context.ResultsFromNorth"
  value='syncresponses.GetAt("C")'
  action="append"/>
<assign property="context.ResultsFromSouth"
  value='syncresponses.GetAt("A")'
  action="append"/>
</sequence>
</flow>

// In this context, syncresponses refers to the primary process thread
<call name="E" />
</sequence>
</process>
}

```

The `<if>` activity in this example has a condition that tests `synctimedout` against the integer value 1. `synctimedout` can have the value 0, 1, or 2 as described in the documentation for `<call>`. If the two values are equal, this `<if>` condition receives the integer value 1 and statements inside the `<true>` element are executed. Otherwise, statements inside the `<false>` element are executed.

## allowresync

The BPL business process can make the `<call>` and then `<sync>` on this call multiple times, with or without a timeout. The `<sync>` `allowresync` attribute controls this behavior. If you set `allowresync` to 1 (true) this enables a subsequent `<sync>` on the same `<call>`. You can do this repeatedly until the call completes. A value of 0 (false) for `<sync>` `allowresync` disallows a subsequent `<sync>` on the same call. The default `allowresync` value is 0.

Suppose you have an asynchronous `<call>` A, a long-running activity whose response can be indefinitely delayed. Suppose you `<sync>` on A with a `timeout` of 5. This `<sync>` returns immediately if A is complete, or returns in 5 seconds if A is not complete but the timeout expires. Now, suppose you know that A can take an indefinite amount of time, but generally returns without problems. That is, suppose A usually completes within 5 seconds, but sometimes takes over an hour, and that the delay is acceptable when it occurs. In this case, you will want to check A frequently for completion, in case it does complete in the usual time, but also allow subsequent `<sync>` activities on the same `<call>`, in case it takes longer to complete.

The following would be typical usage:

## XML

```

<sequence>
  <call name="A" async="1" />
  <sync call="A" timeout="5" allowresync="1" />
  <while condition='synctimedout=1'>
    <alert value="Waiting for call A to complete." />
    <sync call="A" timeout="5" allowresync="1" />
  </while>
</sequence>

```

If a timeout is not specified in the <sync>, then it is important to check the *synctimeout* variable before each <sync>. Otherwise the <sync> could be waiting for a call that has already completed.

### **Consecutive <sync> Timeout**

Suppose you have multiple consecutive <sync> elements that refer to the same <call> element, and each <sync> has a *timeout* value. Once the first <sync> has been satisfied, either because the <call> has returned or because the <sync> *timeout* value has expired, the second <sync> element does not wait but instead completes immediately.

### **XML**

```
<sequence>
  <call name="A" async="1" />
  <sync name="Sync1" calls="A" type="all" timeout="60" />
  <sync name="Sync2" calls="A" type="all" timeout="300" />
</sequence>
```

# <throw>

Throw a specific, named fault.

## Syntax

```

<scope>
  <throw fault="MyFault" />
  ...
  <faulthandlers>
    <catch fault="MyFault" />
    ...
  </faulthandlers>
</scope>

```

## Attributes and Elements

### **fault** attribute

Required. The name of the fault. It can be a literal text string (up to 255 characters) or an expression to be evaluated.

### **LanguageOverride** attribute

Optional. Specifies the scripting language in which any expressions (within in this element) are written.

Can be "python", "objectscript", or "basic" (not documented). Default is the language specified in the containing <process> element.

### **name, disabled, xpos, ypos, xend, yend** attributes

See [Common Attributes and Elements](#).

### **<annotation>** element

See [Common Attributes and Elements](#).

## Description

When a <throw> statement executes, control immediately shifts 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. When you specify a fault string it *needs* the extra set of quotes to contain it, as shown below:

### **XML**

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

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 shifts 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.

For details, see [Handling Errors in BPL](#).

## See Also

- [<catch>](#)
- [<catchall>](#)
- [<compensate>](#)
- [<compensationhandlers>](#)
- [<faulthandlers>](#)
- [<scope>](#)

## <trace>

Write a message to the foreground ObjectScript shell.

### Syntax

```
<trace value="'The time is: ' & Now' />
```

### Attributes and Elements

#### **value attribute**

Required. This is the text for the trace message. It can be a literal text string (up to 255 characters) or an expression to be evaluated.

In an ObjectScript expression, you can also use virtual property syntax using the { } convention.

#### **LanguageOverride attribute**

Optional. Specifies the scripting language in which any expressions (within in this element) are written.

Can be "python", "objectscript", or "basic" (not documented). Default is the language specified in the containing <process> element.

#### **name, disabled, xpos, ypos, xend, yend attributes**

See [Common Attributes and Elements](#).

#### **<annotation> element**

See [Common Attributes and Elements](#).

### Description

The <trace> element writes a message to the ObjectScript shell. <trace> messages appear only if the BPL business process that generates them has been configured to **Run in Foreground** mode.

Trace messages may be written to the [Event Log](#) as well as to the console. A system administrator controls this behavior by configuring a production from the **Interoperability > Configure > Production** page in the Management Portal. If a BPL business process has the **Log Trace Events** option checked, it writes trace messages to the [Event Log](#) as well as displaying them at the console. If a trace message is logged, its [Event Log](#) entry type is Trace.

The BPL <trace> element generates trace message with User priority; the result is the same as calling the \$\$\$TRACE utility from ObjectScript.

**Note:** For details, see [Adding Trace Elements](#).

## <transform>

---

Transform one object into another using a data transformation.

### Syntax

```
<transform class="MyApp.SAPtoJDE" target="context.xform" source="request" />
```

### Attributes and Elements

#### **class attribute**

Required. The name of the data transformation class that will perform the data transformation. This value can be provided as a literal string, or by using the ObjectScript @ indirection operator to refer to the value of an [execution context variable](#). See details below. Specify the name of a data transformation class.

#### **target attribute**

Required. The target (output object) for this data transformation. This is one of the objects in the execution context, or a property of one of these objects. Specify the name of a valid property and object in the execution context.

#### **source attribute**

Required. The source (input object) for this data transformation. This is one of the objects visible in the current execution context or a property of one of these objects. Specify the name of a valid property and object in the execution context.

#### **name, disabled, xpos, ypos, xend, yend attributes**

See [Common Attributes and Elements](#).

#### **<annotation> element**

See [Common Attributes and Elements](#).

### Description

The <transform> element lets you invoke a data transformation class from within a business process.

A data transformation class (a subclass of `Ens.DataTransform`) defines a method that takes an instance of an input object and transforms it into an instance of an output object. The transform element invokes this method to automatically transform an object of one type into another using the data transformation class specified by the class attribute.

The source attribute specifies the input object for the transformation. This is an object (or one of its object-valued properties) visible within the business process execution context and should be of the input type expected by the specified data transformation class.

The target attribute specifies the destination of the output object. This is also an object (or one of its object-valued properties) visible within the business process execution context and should be of the output type expected by the specified data transformation class.

#### **Variables in the Execution Context**

The <transform> element can refer to the following variables and their properties. Do not use variables not listed here.

Variable	Purpose
<i>context</i>	The <i>context</i> object is a general-purpose data container for the business process. <i>context</i> has no automatic definition. To define properties of this object, use the <context> element. That done, you may refer to these properties anywhere inside the <process> element using dot syntax, as in: <code>context.Balance</code>
<i>request</i>	The <i>request</i> object contains any properties of the original request message object that caused this business process to be instantiated. You may refer to <i>request</i> properties anywhere inside the <process> element using dot syntax, as in: <code>request.UserID</code>
<i>response</i>	The <i>response</i> object contains any properties that are required to build the final response message object to be returned by the business process. You may refer to <i>response</i> properties anywhere inside the <process> element using dot syntax, as in: <code>response.IsApproved</code> . Use the <assign> element to assign values to these properties.

**Note:** There is more information about the business process execution context in documentation of the <assign> element.

### Value of the class Attribute

While the <transform> element lets you invoke a data transformation class from within a business process, the data transformation class itself must already be defined using the Data Transformation Language (DTL), a subset of BPL. For more information about DTL, see [Data Transformation Language Reference](#).

### Indirection in the class Attribute

The value of the *class* attribute is a string that identifies the package and class name of a DTL data transformation. The string can be a literal value:

```
<transform class="MyApp.SAPtoJDE" target="context.xform" source="request" />
```

Or the @ indirection operator can be used to access the value of an [execution context variable](#) that contains the appropriate string:

```
<call class="@context.nextTransform" target="context.xform" source="request" />
```

## <true>

---

Perform a set of activities when the condition for an <if> element is true.

### Syntax

```
<if condition="1">
  <true>
    ...
  </true>
  <false>
    ...
  </false>
</if>
```

### Attributes and Elements

*name, disabled, xpos, ypos, xend, yend* attributes

See [Common Attributes and Elements](#).

**<annotation>** element

See [Common Attributes and Elements](#).

#### Other elements

Optional. <true> may contain zero or more of the following elements in any combination: <alert>, <assign>, <branch>, <break>, <call>, <code>, <continue>, <delay>, <empty>, <flow>, <foreach>, <if>, <label>, <milestone>, <reply>, <rule>, <scope>, <sequence>, <sql>, <switch>, <sync>, <throw>, <trace>, <transform>, <until>, <while>, <xpath>, or <xslt>.

### Description

A <true> element is used within an <if> to contain elements that need to be executed if the condition is true.

### See Also

- [<if>](#)
- [<false>](#)

## <until>

Perform activities repeatedly *until* a condition is true.

### Syntax

```
<until condition='context.IsApproved="1"'>
  .
  .
  .
</until>
```

### Attributes and Elements

#### **condition** attribute

Required. This expression is evaluated at the end of each pass through the activities in the <until> element. Once true, it stops execution of the <until> element.

#### **LanguageOverride** attribute

Optional. Specifies the scripting language in which any expressions (within in this element) are written.

Can be "python", "objectscript", or "basic" (not documented). Default is the language specified in the containing <process> element.

#### **name, disabled, xpos, ypos, xend, yend** attributes

See [Common Attributes and Elements](#).

#### **<annotation>** element

See [Common Attributes and Elements](#).

#### **Other elements**

Optional. <until> may contain zero or more of the following elements in any combination: <alert>, <assign>, <branch>, <break>, <call>, <code>, <continue>, <delay>, <empty>, <flow>, <foreach>, <if>, <label>, <milestone>, <reply>, <rule>, <scope>, <sequence>, <sql>, <switch>, <sync>, <throw>, <trace>, <transform>, <until>, <while>, <xpath>, or <xslt>.

### Description

The <until> element defines a sequence of activities that are repeatedly executed until a logical expression evaluates to the integer value 1 (true). The expression is re-evaluated *after* each loop through the sequence.

To fine-tune loop execution, include <break> and <continue> elements within an <until> element. See the descriptions of these elements for details.

## <while>

Perform activities repeatedly *as long as* a condition is true.

### Syntax

```
<while condition='context.IsApproved="1"'>
    .
    .
</while>
```

### Attributes and Elements

#### **condition** attribute

Required. This expression is evaluated before each pass through the activities in the <while> element. Once false, it stops execution of the <while> element.

Specify an expression that evaluates the integer value 1 (if true) or 0 (if false).

#### **LanguageOverride** attribute

Optional. Specifies the scripting language in which any expressions (within in this element) are written.

Can be "python", "objectscript", or "basic" (not documented). Default is the language specified in the containing <process> element.

#### **name, disabled, xpos, ypos, xend, yend** attributes

See [Common Attributes and Elements](#).

#### **<annotation>** element

See [Common Attributes and Elements](#).

#### **Other elements**

Optional. <while> may contain zero or more of the following elements in any combination: <alert>, <assign>, <branch>, <break>, <call>, <code>, <continue>, <delay>, <empty>, <flow>, <foreach>, <if>, <label>, <milestone>, <reply>, <rule>, <scope>, <sequence>, <sql>, <switch>, <sync>, <throw>, <trace>, <transform>, <until>, <while>, <xpath>, or <xslt>.

### Description

The <while> element defines a sequence of activities that are repeatedly executed as long as a logical expression evaluates to the integer value 1 (true). The expression is re-evaluated *before* each loop through the sequence.

You can fine-tune loop execution by including <break> and <continue> elements within a <while> element. For example:

#### **XML**

```
<while condition="0">
    //...do various things...
    <if condition="somecondition">
        <true>
            <break/>
        </true>
    </if>
    //...do various other things...
</while>
```

# <xpath>

Evaluate XPath expressions on a target XML document.

## Syntax

```
<xpath name="xpath"
      source="request.MetadataXML"
      property="context.Result" context="/staff/doc"
      expression="name[@last='Marston']"/>
```

Note that the editor uses double quotes around the values of the attributes. Thus if an attribute value needs to include quotes, those must be single quotes as shown here.

## Attributes and Elements

### **source** attribute

Required. An expression that yields a stream containing the XML on which the XPath expressions are to be performed. Typically the *source* attribute will name a context or request property.

### **property** attribute

Required. The property (typically a context property) in which to place the result of the evaluation.

### **context** attribute

Required. The document context.

### **expression** attribute

Required. The XPath expression.

### **prefixmappings** attribute

Optional. Specifies prefix mappings for the document. This is a comma-delimited list of prefix-to-namespace mappings. See details below. Specify a string of up to 255 characters.

### **schemaspec** attribute

Optional. The schema specification. Specify a string of up to 255 characters.

### **name, disabled, xpos, ypos, xend, yend** attributes

See [Common Attributes and Elements](#).

### **<annotation>** element

See [Common Attributes and Elements](#).

## Description

The <xpath> element allows a business process to evaluate XPath expressions on a target XML document.

When the <xpath> element executes, the *source* stream is processed into an XPath document and then the XPath expressions are evaluated in sequence. The BPL runtime engine automatically manages the lifetime of the documents and caches them to allowing processing to be as efficient as possible

Each *prefixmappings* entry is defined as a prefix, a space, and then the URI to which that prefix maps. This is especially useful if the document defines a default namespace with the `xmlns="http://somenamespaceuri"` syntax, but does not supply an explicit prefix mapping. The following *prefixmappings* string would map the `myprefix` prefix to the `http://somenamespaceuri` URI. Note the space character in the string:

```
prefixmappings="myprefix http://somenamespaceuri"
```

The BPL `<xpath>` element is intended to support XPath expressions which yield a scalar value, that is a single piece of text, number, date etc. It is *not* intended to deal with expressions that yield an XPath DOM. This means that if the expression does yield a DOM, the target property will *not* be updated. DOM programming is beyond the scope of BPL. If your business needs such processing, then the XPath should be performed in a code block or a call to a utility class.

## See Also

- [<xslt>](#)

## <xslt>

Execute an embedded XSLT transformation.

### Syntax

```

<xslt name='simon'
      xslurl="https://www.intersystems.com/transform.xsl"
      source="context.a" target="context.b">
  <parameters>
    <parameter name="surname" value="sez"/>
  </parameters>
</xslt>

```

### Attributes and Elements

#### ***xslurl*** attribute

Required. URI of the XSLT definition that controls the transformation. The URI may begin with one of the following strings: “file:” “http:” “url:” or “xdata:” Specify a string of up to 255 characters.

#### ***source*** attribute

Required. Name of the source (stream) object. Specify a string of up to 255 characters.

#### ***target*** attribute

Required. Name of the target (stream) object. Specify a string of up to 255 characters.

#### ***name, disabled, xpos, ypos, xend, yend*** attributes

See [Common Attributes and Elements](#).

#### **<annotation>** element

See [Common Attributes and Elements](#).

#### **<parameters>** element

An optional <parameters> element may appear. Inside the <parameters> container, zero or more <parameter> elements may appear. Each <parameter> element defines an XSLT name-value pair to pass to the stylesheet that controls the XSLT transformation.

#### ***xsltversion*** attribute

Specifies whether the XSLT transformation uses XSLT 1.0 or 2.0. Specify either "1.0" or "2.0"

### Description

The <xslt> element allows you to apply an XSLT transformation during a business process. The <xslt> element transforms an input stream to an output stream via an arbitrary XSLT definition. The XSLT definition may be in an external file, or it may be defined in a class in the same namespace as the BPL business process.

The *source* and *target* stream objects must be declared as properties of the *context* object for the business process. The *context* object is a general-purpose data container for the business process. You may define *context* properties by providing <context> and <property> elements at the beginning of the <process> element. That done, you may refer to these properties anywhere inside the <process> element using dot syntax, as in: `context.MyInputStream` or `context.MyOutputStream`

The *xslurl* string is a URI that identifies the location of the XSLT definition. The *xslurl* value may begin with one of the following strings:

```
file:  
http:  
url:  
xdata:
```

Where *file:*, *http:*, and *url:* have the standard meanings. An *xdata:* string takes this form:

```
xdata://PackageName.ClassName:XDataName
```

Where:

- *PackageName*.*ClassName* identifies a class in the same namespace as the BPL business process.
- *XDataName* is the name of an XData block within that class that contains the XSLT definition for this `<xslt>` statement. This convention allows XSLT definitions to be stored inside InterSystems IRIS classes, as an efficient alternative to storing them outside InterSystems IRIS in the local file system or on the Web.

If the XSLT requires parameters, include them in a `<parameters>` block within the `<xslt>` element.

## See Also

- `<parameters>`
- `<xpath>`

# DTL Elements

This reference provides detailed information about each DTL element.

## DTL <annotation>

---

Provide a descriptive comment for a DTL element.

### Syntax

```
<annotation>
  <![CDATA[ Sends patient data from lab to CRM system. ]]>
</annotation>
```

### Description

The <annotation> element allows you to associate a descriptive comment with a DTL element. An <annotation> must appear as the first child of the element that it is annotating. For example:

### XML

```
<transform targetClass='Demo.DTL.ExampleTarget'
  sourceClass='Demo.DTL.ExampleSource'
  create='new'
  language='objectscript'>
  <annotation>
    <![CDATA[Implement current naming conventions.]]>
  </annotation>
  <trace value='Convert from lowercase to uppercase' />
  <assign property='target.UpperCase'
    value='$ZCONVERT(source.LowerCase,"U")'
    action='set'>
    <annotation>This is a comment for the assign element</annotation>
  </assign>
</transform>
```

The previous example uses CDATA syntax around the annotation text. This convention is optional, but it lets you use line breaks and special characters such as the apostrophe (') without worrying about XML escape sequences. The maximum length of the <annotation> string is 32,767 characters, including the CDATA escape characters.

Also notice that the annotation for the assign element appears as a child immediately following the opening assign tag.

Most elements within DTL support <annotation> as a child element. This allows you to associate a descriptive comment with a DTL element. Unlike BPL, which offers positional attributes for every element, <annotation> is the *only* child element or attribute that most DTL elements have in common. If you use the <annotation> element, it must appear as the first child of the element that it is annotating.

## DTL <assign>

Assign a value to a property within an object.

### Syntax

```
<assign property="propertyname" value="expression" />
```

### Attributes

Attribute	Description	Value
<i>property</i>	Required. The property that is the target of this assignment.	A string.
<i>value</i>	Required. Provides a value for the property.	An ObjectScript expression that provides a valid value for the property.
<i>action</i>	Optional. If <i>value</i> is a collection property (list or array), then use <i>action</i> to specify the type of assignment to perform. The default is a set action.	One of the following values: <i>set</i> , <i>clear</i> , <i>remove</i> , <i>append</i> , <i>insert</i> . See the <a href="#">actions</a> section for details.
<i>key</i>	Optional, except in some cases when <i>value</i> is a collection property (list or array). If so, then use this key to specify the element upon which the assignment will be performed.	A string that is an expression that evaluates to a key.

### Elements

Element	Purpose
<a href="#">&lt;annotation&gt;</a>	Optional. A text string that describes the <assign> element.

### Description

The DTL <assign> element is used from within a DTL <transform> element to specify a target property and an expression whose value will be assigned to it. Generally, this expression involves values from the source object for the data transformation, but they may also be literal values. All properties involved in a DTL <assign> activity must be properties within the source or target object for the data transformation.

The source and target objects are generally production message body objects, as described in [Messages](#). These consist of a message header and a message body object.

Properties in the standard production message body can be data types, objects, or collections of either. Collection properties are declared with either [ `Collection = list` ] or [ `Collection = array` ] in the class definition. You can refer to the properties on the standard production message body using dot syntax as for any object property.

Properties in a virtual document require the unique syntax described in the following topics:

- [Virtual Property Paths](#)
- [Syntax Guide for Virtual Property Paths](#)

## Actions of the <assign> Element

There are several types of DTL <assign> operation, as specified by the optional action attribute. Aside from the default of *set*, these variations are intended to handle assignments involving collection properties within a standard production message body. The following table describes the actions of the <assign> element.

Assign action	Description	Example
set	Sets the value of the specified property to that of the value attribute. Note that the value attribute contains an expression and can itself refer to an object or property of an object.	The following statement sets the value of the target BankName property:  <b>XML</b>  <pre>&lt;assign property='target.BankName' value='process.BankName' action='set' /&gt;</pre>
append	Adds the target element to the end of a list property	
clear	Clears the contents of the specified collection property. The value and key attributes are ignored. (Applies to collection properties only.)	The following statement clears the contents of the collection property List:  <b>XML</b>  <pre>&lt;assign property='target.List' action='clear' /&gt;</pre>
insert	Inserts a value into the specified collection property. If the key attribute is present the new value is inserted after the position (an integer) specified by key; otherwise, the new item is inserted at the end. (Applies to list collection properties only.)	The following statement inserts a value into the array collection property Array using the key primary:  <b>XML</b>  <pre>&lt;assign property='target.Array' action='insert' key='primary' value='source.Primary' /&gt;</pre>
remove	Removes an item from the specified collection property. The value attribute is ignored. (Applies to collection properties only.)	

**Note:** Virtual documents do not use any action value other than *set*.

The **set** action sets the value of the specified property to that of the value attribute. Note that the value attribute contains an expression and can itself refer to an object or property of an object:

### XML

```
<assign property='target.SSN' value='source.SSN' />
```

If the target property is an array collection, then the value of the key attribute specifies an item in the array, otherwise the key attribute is ignored.

If the target property is a collection and the value attribute specifies a collection of the same type, then the collection contents are copied into the target collection:

## XML

```
<assign property='target.List' value='source.List' />
```

The default action for the assign element is the set operation; if action is not specified, then the assign specifies a set operation.

## Objects and Object References

If you <assign> from the top-level source object or any object property of another object as your source, the target receives a cloned copy of the object rather than the object itself. This prevents inadvertent sharing of object references and saves the effort of generating cloned objects yourself. However, if you want to share object references between source and target you must <assign> from the source to an intermediate temporary variable, and then <assign> from that variable to the target.

## Wholesale Copy

To create a target object that is an exact copy of the source, do not use:

```
<assign property='target' value='source' />
```

Instead use the `create='copy'` attribute in the containing <transform> element.

The *create* option may have one of the following values:

- `new`—Create a new object of the target type, before executing the elements within the data transformation. This is the default.
- `copy`—Create a copy of the source object to use as the target object, before executing the elements within the transform.
- `existing`—Use an existing object, provided by the caller of the data transformation, as the target object.

## DTL <break>

---

Terminate a For Each loop or stop processing the data transformation.

### Syntax

```
<break/>
```

### Attributes

None.

### Elements

Element	Purpose
<annotation>	Optional. A text string that describes the <break> element.

### Description

When included in a <foreach> element, the <break> element terminates the For Each loop. If <break> is outside of a For Each loop, the entire data transformation terminates as soon as the break is executed.

## DTL <case>

Execute a block of actions within a <switch> element when the specified condition is met.

### Syntax

```
<switch>
  <case condition="1">
    ...
  </case>
  <default>
    ...
  </default>
</switch>
```

### Attributes

Attribute	Description	Value
<i>condition</i>	Required. An ObjectScript expression that, if true, causes the contents of the <case> element to be executed.	An expression that evaluates to the integer value 1 (if true) or 0 (if false).

### Elements

Element	Purpose
<annotation>	Optional. A text string that describes the <case> element.

### Description

The <switch> element contains one or more <case> elements. The elements within a <case> element are executed if the condition evaluates to true.

## DTL `<code>`

Execute lines of custom code.

### Syntax

```
<code>
  <![CDATA[ target.Name = source.FirstName & " " & source.LastName]]>
</code>
```

### Elements

Element	Purpose
<code>&lt;annotation&gt;</code>	Optional. A text string that describes the <code>&lt;code&gt;</code> element.

### Description

The DTL `<code>` element executes one or more lines of user-written code within a DTL data transformation. You can use the `<code>` element to perform special tasks that are difficult to express using the DTL elements. Any properties referenced by the `<code>` element must be properties within the source or target object for the data transformation.

The scripting language for a DTL `<code>` element is specified by the `language` attribute of the containing `<transform>` element. The value should be `objectscript`. Any expressions found in the data transformation, as well as lines of code within `<code>` elements, must use the specified language.

For further information, see the following items:

- Using ObjectScript
- ObjectScript Reference

Typically a developer wraps the contents of a `<code>` element within a CDATA block to avoid having to worry about escaping special XML characters such as the apostrophe (') or the ampersand (&). For example:

### XML

```
<code>
  <![CDATA[ target.Name = source.FirstName & " " & source.LastName]]>
</code>
```

In order to ensure that execution of a data transformation can be suspended and restored, you should follow these guidelines when using the `<code>` element:

- The execution time should be short; custom code should not tie up the general execution of the data transformation.
- Do not allocate any system resources (such as taking out locks or opening devices) without releasing them within the same `<code>` element.
- If a `<code>` element starts a transaction, make sure that the same `<code>` element ends the transactions in *all possible scenarios*; otherwise, the transaction can be left open indefinitely. This could prevent other processing or can cause significant downtime.

### Available Variables

The variables that are available in a DTL `<code>` element are dependent upon the method used to call the element. Refer to the following table to see the available variables and their properties:

Variable Name	Purpose	Available when DTL is called through:
<i>source</i>	Contains properties of the source message.	All methods
<i>target</i>	Contains properties of the target message.	All methods
<i>process</i>	The <i>process</i> object represents the current instance of the BPL business process object (an instance of the BPL class). This object has one property for each property defined in that class. You can invoke methods of the <i>process</i> object; for example: <code>process.SendRequestSync ( )</code>	BPL business processes
<i>context</i>	The <i>context</i> object is a general-purpose data container for the business process. <i>context</i> has no automatic definition. To define properties of this object, use the <a href="#">&lt;context&gt;</a> element. That done, you may refer to these properties anywhere inside the <a href="#">&lt;process&gt;</a> element using dot syntax, as in: <code>context.Balance</code>	BPL business processes
<i>aux</i>	Contains information from the business rule that called the DTL.	Business rules

## DTL <comment>

---

Add comments to the DTL.

### Syntax

```
<comment>  
  <annotation>  
  ...  
  </annotation>  
</comment>
```

### Attributes

None.

### Elements

Element	Purpose
<annotation>	A text string that contains the comment.

### Description

The contents of the <annotation> element of <comment> appear in the Management Portal to describe the DTL actions.

## DTL <default>

Execute contents if none of the <case> elements in a <switch> element evaluate to true.

### Syntax

```
<switch>
  <case condition="1">
    ...
  </case>
  <default>
    ...
  </default>
</switch>
```

### Attributes

None.

### Elements

Element	Purpose
<a href="#">&lt;annotation&gt;</a>	Optional. A text string that describes the <default> element.

### Description

The <default> element appears at the end of the <switch> element, and is executed if none of the <case> elements evaluate to true.

## DTL <false>

Perform a set of activities when the condition for an <if> element is false.

### Syntax

```
<if condition="0">
  <true>
    ...
  </true>
  <false>
    ...
  </false>
</if>
```

### Attributes

None.

### Elements

Element	Purpose
<annotation>	Optional. A text string that describes the <false> element.
Most activities	Optional. <false> may contain zero or more of the following elements in any combination: <assign>, <code>, <foreach>, <if>, <sql>, <subtransform>, or <trace>.

### Description

A <false> element is used within an <if> to contain elements that need to be executed if the condition is false.

## DTL <foreach>

Define a sequence of activities to be executed iteratively.

### Syntax

```
<foreach property="P1" key="K1">
  .
  .
  .
</foreach>
```

### Attributes

Attribute	Description	Value
<i>property</i>	Required. The collection property (list or array) to iterate over. It must be the name of a valid object and property in the execution context.	A string of one or more characters.
<i>key</i>	Required. The index used to iterate through the collection. It must be a name of a valid object and property in the execution context. It is assigned a value for each element in the collection.	A string of one or more characters.

### Elements

Element	Purpose
<annotation>	Optional. A text string that describes the <foreach> element.
Most activities	Optional. <foreach> may contain zero or more of the following elements in any combination: <assign>, <code>, <foreach>, <if>, <sql>, <subtransform>, or <trace>.

### Description

The <foreach> element defines a sequence of activities that are executed iteratively, once for every element that exists within a specified collection property. If the element is null, the sequence is not executed. The sequence is executed if the element has an empty value, that is, the separators are there but there is no value between them, but is not executed for a null value, that is, the message is terminated before the field is specified.

For example:

#### XML

```
<foreach key='i' property='target.{PID:3()}'>
  <assign property='target.{PID:3(i).4}' value='"001"' action='set' />
</foreach>
```

Or:

## XML

```
<foreach key='key' property='source.{PID:PatientIDInternalID()}'>
  <if condition='source.{PID:PatientIDInternalID(key).identifiertypecode}="PAS"'>
    <true>
      <assign property='target.{PID:PatientIdentifierList(key).identifiertypecode}'
              value='MR'
              action='set' />
    </true>
  </if>
  <if condition='source.{PID:PatientIDInternalID(key).identifiertypecode}="GMS"'>
    <true>
      <assign property='target.{PID:PatientIdentifierList(key).identifiertypecode}'
              value='MC'
              action='set' />
      <assign property='target.{PID:PatientIdentifierList(key).assigningfacility}'
              value='AUSHIC'
              action='set' />
    </true>
  </if>
</foreach>
```

The properties referenced by the `<foreach>` element must be properties in the source or target object for the data transformation.

### ***Nested `<foreach>`***

Nesting of `<foreach>` elements is allowed, but see the next subsection for an alternative.

### ***Shortcuts for `<foreach>`***

When you are working with a document-based message or “virtual document” type, the `<assign>` statement offers a shortcut notation that iterates through every instance of a repeating field within a document structure. This means you do not actually need to set up `<foreach>` loops with 'i' 'j' and 'k' just for the purpose of handling repeating fields. Instead, you can use a much simpler notation with empty parentheses. See [Iterating Through Repeating Fields](#).

### ***Avoiding `<STORE>` Errors with Large Messages***

As you loop over segments in a message or object collections, they are brought into memory. If these objects consume all the memory assigned to the current process, you may get unexpected errors.

To avoid this, remove the objects from memory after you no longer need them. For example, if you are processing many segments in a `<foreach>` loop, you can call the `commitSegmentByPath` method on both the source and target as the last step in the loop. Similarly, for object collections, use the `%UnSwizzleAt` method.

If you cannot make code changes, a temporary workaround is to increase the amount of memory allocated for each process. You can change this by setting the `bbsiz` parameter on the **Advanced Memory Settings** page in the Management Portal. Note that this requires a system restart and should only occur after consulting with your system administrator.

## DTL <group>

---

Organize related elements into a display unit.

### Syntax

```
<group>  
  .  
  .  
</group>
```

### Attributes

None.

### Elements

Element	Purpose
<a href="#">&lt;annotation&gt;</a>	Optional. A text string that describes the <group> element.
All	Any element can be added to a <group> element.

### Description

The <group> element organizes related elements into a logical unit.

## DTL <if>

Evaluate a condition and perform one action if true, another if false.

### Syntax

```
<if condition="1">
  <true>
    ...
  </true>
  <false>
    ...
  </false>
</if>
```

### Attributes

Attribute	Description	Value
<i>condition</i>	Required. An ObjectScript expression that, if true, causes the contents of the <true> element to execute. If false, the contents of the <false> element are executed.	An expression that evaluates to the integer value 1 (if true) or 0 (if false).

### Elements

Element	Purpose
<annotation>	Optional. A text string that describes the <if> element.
<true>	Optional. If the condition is true, activities inside the <true> element are executed.
<false>	Optional. If the condition is false, activities inside the <false> element are executed.

### Description

The <if> element evaluates an expression and, depending on its value, executes one of two sets of activities (one if the expression evaluates to a true value, the other if it evaluates to a false value).

The <if> element may contain a <true> element and a <false> element which define the actions to execute if the expression evaluates to true or false, respectively.

If both <true> and <false> elements are provided, they may appear within the <if> element in any order.

If the condition is true and there is no <true> element, or if the condition is false and there is no <false> element, no activity results from the <if> element.

## DTL <sql>

Execute an embedded SQL SELECT statement within a data transformation.

### Syntax

```
<sql>
  <![CDATA[
    SELECT SSN INTO :context.SSN
    FROM MyApp.PatientTable
    WHERE PatID = :request.PatID ]]>
</sql>
```

### Elements

Element	Purpose
<annotation>	Optional. A text string that describes the <sql> element.

### Description

The DTL <sql> element executes an arbitrary embedded SQL SELECT statement from within a DTL <transform> element.

To use the <sql> element effectively, keep the following tips in mind:

- Always use the fully qualified name of the table, including both the SQL schema name and table name, as in:
 

```
MyApp.PatientTable
```

 Where `MyApp` is the SQL schema name and `PatientTable` is the table name.
- The contents of the <sql> element must contain a valid embedded SQL SELECT statement.
 

It is convenient to place the SQL query within a CDATA block so that you do not have to worry about escaping special XML characters.
- Any tables listed in the SQL query's FROM clause must either be stored within the local InterSystems IRIS database or linked to an external relational database using the SQL Gateway.
- Within the INTO and WHERE clauses of the SQL query, you can refer to a property of the source or target object by placing a : (colon) in front of the variable name. For example:

#### XML

```
<sql><![CDATA[
  SELECT Name INTO :target.Name
  FROM MainFrame.EmployeeRecord
  WHERE SSN = :source.SSN AND City = :source.Home.City
]]>
</sql>
```

- Only the first row returned by the query will be used. Make sure that your WHERE clause correctly specifies the desired row.

## DTL <subtransform>

Invoke another data transformation.

### Syntax

```
<subtransform class='class-name'
  targetObj='target-value'
  sourceObj='source-value' />
```

### Attributes

Attribute	Description	Value
<i>class</i>	<p>Required. Name of the class that contains the data transformation to be invoked. This class must be in the same namespace as the class that invokes it.</p> <p>Often, <i>class</i> is a DTL data transformation defined using a DTL &lt;transform&gt; element, as shown in the examples in this topic.</p> <p>Alternatively, <i>class</i> can identify a custom subclass of <code>Ens.DataTransform</code> that implements the <b>Transform</b> method and does not use DTL.</p>	The full package and class name.
<i>sourceObject</i>	<p>Required. Identifies the property being transformed. This may be an object property or a virtual document property. Generally it is a property of the source object identified by the containing &lt;transform&gt; element's <i>sourceClass</i> and (for virtual documents) <i>sourceDocType</i>. In this case it is referenced using dot syntax as follows:</p> <p><code>source.<i>property</i></code> or <code>source.<i>{propertyPath}</i></code></p>	Property name. For virtual documents and their segments, use virtual property syntax.
<i>targetObject</i>	<p>Required. Identifies the property into which the transformed value will be written. This may be an object property or a virtual document property. Generally it is a property of the target object identified by the containing &lt;transform&gt; element's <i>targetClass</i> and (for virtual documents) <i>targetDocType</i>. In this case it is referenced using dot syntax as follows:</p> <p><code>target.<i>property</i></code> or <code>target.<i>{propertyPath}</i></code></p> <p>In the case of a subtransform with <b>Create</b> as <code>new</code> or <code>copy</code>, it is not necessary to have a pre-existing target object.</p>	Property name. For virtual documents and their segments, use virtual property syntax.

### Elements

Element	Purpose
<annotation>	Optional. A text string that describes the <subtransform> element.

## Description

The <subtransform> element invokes another data transformation. Making a call to <subtransform> allows the containing <transform> element to invoke other data transformations to complete segments of its work. This allows developers greater flexibility in maintaining a suite of reusable DTL transformation code.

Before the <subtransform> element was available, every DTL <transform> stood alone. In order to write multiple DTL transformations that contained an identical sequence of actions, it was necessary to copy and paste the corresponding sections of code from one class into another. Now, each of these DTL classes can replace repeated lines with a <subtransform> element that invokes another class to perform the desired sequence.

The source or target objects for a <subtransform> may be ordinary InterSystems IRIS objects, virtual document *message* objects, or virtual document *segment* objects representing an individual segment within a virtual document message. The <subtransform> is especially important for interface developers working with Electronic Data Interchange (EDI) formats, where each message or document may contain many independent segments that need to be transformed. Having the <subtransform> available means you can create a reusable library of segment transformations that you can call as needed, without duplicating code in the calling transformation.

For virtual documents and their segments, you must use virtual property syntax, such as the { } curly bracket syntax in the following examples. The property path inside the brackets must refer to a particular segment, not to a field within a segment or to a group of segments. For background information, see [Using Virtual Documents in Productions](#); details are available in [Virtual Property Path](#).

## DTL <switch>

Evaluate <case> elements and execute the contents of the first one that evaluates to true.

### Syntax

```
<switch>
  <case condition="1">
    ...
  </case>
  <default>
    ...
  </default>
</switch>
```

### Attributes

None.

### Elements

Element	Purpose
<annotation>	Optional. A text string that describes the <switch> element.
<case>	The first <case> element that evaluates to true is executed.
<default>	Optional. If none of the <case> elements evaluate to true, the contents of the <default> element are executed.

### Description

The <switch> element contains one or more <case> elements along with an optional <default> element. The contents of a <case> element are executed if the condition evaluates to true. Once a <case> element evaluates to true, none of the other <case> elements nor the <default> element are evaluated. The contents of the <default> element are executed if none of the <case> elements evaluate to true.

## DTL <trace>

Write a message to the foreground ObjectScript shell.

### Syntax

```
<trace value="'The time is: ' & Now' />
```

### Attributes

Attribute	Description	Value
<i>value</i>	Required. This is the text for the trace message. It can be a literal text string or an ObjectScript expression to be evaluated.	A string of one or more characters. May be a literal string or an expression.

### Elements

Element	Purpose
<annotation>	Optional. A text string that describes the <trace> element.

### Description

The <trace> element writes a message to the ObjectScript shell. <trace> messages appear only if the business host that invokes the DTL data transformation has been configured to **Run in Foreground** mode.

Trace messages may be written to the [Event Log](#) as well as to the console. A system administrator controls this behavior from the Management Portal Configuration page. If the business host that invokes the DTL data transformation has the **Log Trace Events** option checked, it writes trace messages to the [Event Log](#) as well as displaying them at the console. If a trace message is logged, its [Event Log](#) entry type is Trace.

The DTL <trace> element generates trace message with User priority; the result is the same as calling the \$\$\$TRACE utility from ObjectScript.

**Note:** For details, see [Adding Trace Elements](#).

## DTL <transform>

Transform an object of one type into an object of another type.

### Syntax

```
<transform sourceClass="MyApp.SAPtoJDE"
  targetClass="AlsoMine.JDE" />
```

### Attributes

Attribute	Description	Value
<i>sourceClass</i>	Required. The class name of the input object for the data transformation.	The name of a valid object and property.
<i>targetClass</i>	Required. The class name of the output object for the data transformation.	The name of a valid object and property.
<i>sourceDocType</i>	Optional. When the input object is a virtual document, this string identifies its DocType.	A string.
<i>targetDocType</i>	Optional. When the output object is a virtual document, this string identifies its DocType.	A string.
<i>language</i>	Optional. Should be <code>objectscript</code>	<code>objectscript</code>
<i>create</i>	Optional. The create option desired for the target object. If not specified, the default is <code>new</code> .	This can take one of the following values: <code>new</code> , <code>copy</code> , or <code>existing</code> as detailed in the following description.

### Elements

Element	Purpose
<annotation>	Optional. A text string that describes the <transform> element.
Most activities	Optional. <transform> may contain zero or more of the following elements in any combination: <assign>, <code>, <foreach>, <if>, <sql>, <subtransform>, or <trace>.

### Description

The <transform> element is the outermost element for a DTL document. All the other DTL elements are contained within a <transform> element. Within the <transform>, the two objects have the names *source* and *target*, respectively. For example:

#### XML

```
<transform targetClass='Demo.DTL.ExampleTarget'
  sourceClass='Demo.DTL.ExampleSource'
  create='new'
  language='objectscript'>
  <trace value='"Convert from lowercase to uppercase"' />
  <assign property='target.UpperCase'
    value='$ZCONVERT(source.LowerCase,"U")'
    action='set' />
</transform>
```

## Source and Target Objects

The *sourceClass* and *targetClass* may identify standard production message classes, each of which contains a set of properties. If so, the *sourceDocType* and *targetDocType* attributes are not needed.

Alternatively, the *sourceClass* and *targetClass* may identify virtual documents. In this case the *sourceDocType* and *targetDocType* attributes are needed to tell InterSystems IRIS which message structure to expect in the virtual document.

## Values for the create Option

The *create* option for the target object may have one of the following values:

- *new*—Create a new object of the target type, before executing the elements within the data transformation. This is the default.
- *copy*—Create a copy of the source object to use as the target object, before executing the elements within the transform.
- *existing*—Use an existing object, provided by the caller of the data transformation, as the target object.

## DTL <true>

Perform a set of activities when the condition for an <if> element is true.

### Syntax

```
<if condition="1">
  <true>
    ...
  </true>
  <false>
    ...
  </false>
</if>
```

### Attributes

None.

### Elements

Element	Purpose
<annotation>	Optional. A text string that describes the <true> element.
Most activities	Optional. <true> may contain zero or more of the following elements in any combination: <assign>, <code>, <foreach>, <if>, <sql>, <subtransform>, or <trace>.

### Description

A <true> element is used within an <if> to contain elements that need to be executed if the condition is true.