

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Chapter 7. Introduction to BPEL

Business Process Execution Language (BPEL) is an XML-based language for creating a *process*, which is a set of logical steps (called *activities*) that guide a workflow like the following one:

1. Accept a request for an insurance quote.
2. If the submitted details are appropriate, calculate the quote and include it in the response.
3. Otherwise, say "No" and include a justification.

A BPEL process fulfills a workflow primarily by accessing one service after another. Each of those services is called a *partner service*.

Each BPEL activity is equivalent to a function call in a programming language or to a box in a flowchart. The *receive* activity waits for an inbound message, for example, and the *invoke* activity transmits an outbound message. Activities are categorized as either *basic* or *structured*. Basic activities (such as *receive* and *invoke*) do discrete tasks. Structured activities (such as *if* and *while*) specify an order or condition that affects the circumstance for running a set of other, embedded activities, which may be basic, structured, or both.

The running time for a BPEL process can be far longer than in other kinds of software. An insurance-claims process, for example, might wait to receive additional data from a specific customer for as long as the claim is active, even for years.

Listing 7.1 shows an excerpt from the BPEL process ProcessQuote.

Listing 7.1. Excerpt from BPEL process ProcessQuote

```
<process name= "ProcessQuote">

  <!-- omitted namespaces, as well as
    imports of WSDL and XSD definitions -->
  <partnerLinks>
    <partnerLink name="ProcessQuote"
      partnerLinkType="ProcessQuotePLT" myRole="ProcessQuoteRole" />
    <partnerLink name="mainframeQuoteMgr"
      partnerLinkType="PartnerLinkPLT" partnerRole="partnerRole" />
  </partnerLinks>

  <variables>
    <variable name="quoteRequest"
      messageType="placeQuoteRequestMsg" />
    <variable name="highlightQuote"
      messageType="placeQuoteResponseMsg" />
    <variable name="buildQuoteReq"
      messageType="buildQuoteRequestMsg" />
    <variable name="newHighlightQuote"
      messageType="buildQuoteResponseMsg" />
  </variables>

  <sequence>
    <receive name="processQuoteRequest" createInstance="yes"
      operation="placeQuote" partnerLink="ProcessQuote"
      portType="ProcessQuote" variable="quoteRequest">
    </receive>
```

```

<assign name="AssignQuoteReq">
  <copy>
    <from variable="quoteRequest" part="placeQuoteParameters">
      <query>/quoteInformation</query>
    </from>
    <to variable="buildQuoteReq" part="buildQuoteParameters">
      <query>/customerQuoteInfo</query>
    </to>
  </copy>
</assign>

<invoke name="CalculateQuote" inputVariable="buildQuoteReq"
  operation="buildQuote" outputVariable="newHighlightQuote"
  partnerLink="mainframeQuoteMgr" portType="QuoteManagement" />
<assign name="AssignQuoteRes">
  <copy>
    <from variable="newHighlightQuote" part="buildQuoteResult">
      <query>/quote</query>
    </from>
    <to variable="highlightQuote" part="placeQuoteResult" >
      <query>/quote</query>
    </to>
  </copy>
</assign>
<reply name="processQuoteResponse"
  operation="placeQuote" partnerLink="ProcessQuote"
  portType="ProcessQuote" variable="highlightQuote" />
</sequence>
</process>

```

This excerpt

1. creates *partner links*, which give details on the relationship between the BPEL process and each partner service
2. assigns *variables*, which are memory areas that are each described by a Web Services Description Language (WSDL) message but could have been described by an XML Schema element or type
3. receives a quote request
4. uses XPath syntax to copy data from the received message to a variable that is used for invoking another service
5. invokes the other service, which calculates and returns a quote
6. copies the quote details to another variable and in this way formats the response message
7. replies to the invoker, which may have been a Web application or a service

We'll not describe every detail, just yet.

In general, activities may

- run in a preset sequence
- run in a loop
- run on condition that a Boolean expression evaluates to *true*
- run immediately or wait for some period of time, even years
- run in response to an event that occurs after the process starts (specifically, in response to an inbound message, a calendar date and clock time, or the passing of time)

- run in an order that differs for different instances of the same process

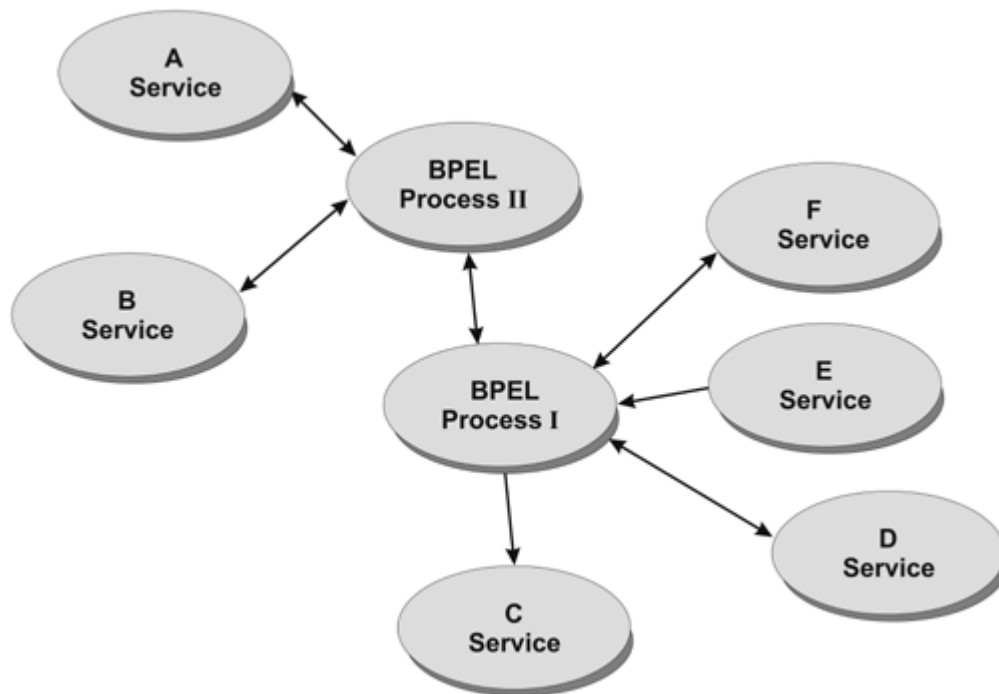
That last option implies *concurrency*, which is a kind of processing in which several activities are issued more or less at the same time. A process may enable the receipt of messages from different services, for example, without requiring that one receipt precede another. Or a process may invoke several services without requiring that one invocation precede another. The general rule is that a service may fulfill several *parallel streams* (often called *branches*), each of which is a series of activities, with different streams running at once.

BPEL also has mechanisms for

- *fault handling*, as needed to respond to business errors (such as a request for an excessive credit line) and to technical errors (such as a network failure)
- *compensation*, as needed to reverse an action (such as a product purchase) that succeeded but was later found to be undesirable
- *correlation*, as needed to direct a message to the correct instance of a BPEL process

Two kinds of BPEL processes are possible. A *BPEL executable process* is itself a Web service and acts as the hub in a service orchestration. The software that runs an executable process is called a *BPEL engine*. One executable process can invoke another, with the effect of connecting one orchestration to another, as [Figure 7.1](#) illustrates.

Figure 7.1. Connected orchestrations



A *BPEL abstract process* is similar to a BPEL executable process but includes a subset of the information. The abstract process is a description of business logic and can be used for the following purposes:

- to specify the behavior of one business unit or partner in a collaboration with others.
- to provide a design guide for programmers in your company and in partnering companies.
- to be an input to commercial software that creates skeletal code for a service implementation. The output may be in Java or some other language.

A BPEL abstract process can include all the content of a BPEL executable process, but omits implementation details in most cases.

When you create a BPEL process, you use two additional kinds of language. A *query language* lets you search for values that are embedded in an XML structure such as a message. An *expression language* lets you calculate numeric and date-time values, manipulate strings, and make comparisons. By default, BPEL uses XPath 1.0 in all cases. If your BPEL engine also lets you use an alternative language (such as Java) to create expressions, you can use XPath or the alternative at different points in the same process.

BPEL does not directly support file or database access and was not designed for extensive computation or string handling. However, the function `doXslTransform` lets you transform data from one XML structure to another. You

might use the function to reorganize data received from one service, making the data appropriate for transmission to a second service. You also might use `doXsltTransform` to reorganize data for internal use, either for easy comparison (for example, to place product details into a single table after receiving data from multiple suppliers) or for easy calculation (for example, to sum prices after receiving details about different input materials).

The next sections provide an overview of BPEL executable processes, and a subsequent chapter highlights event handlers and selected activities. Our comments reflect the WS-BPEL 2.0 specification, which was written under the guidance of the Organization for the Advancement of Structured Information Standards (OASIS). The specification is at http://www.oasis-open.org/committees/documents.php?wg_abbrev=wsbpel.

The following proposals for extending BPEL are under review by several companies:

- *BPEL4People* proposes a way to model human interactions as services: <http://www.ibm.com/developerworks/webservices/library/specification/ws-bpel4people>.
- *BPEL Extensions for Sub-Processes* proposes use of modular code in BPEL processes: <http://www.ibm.com/developerworks/webservices/library/specification/ws-bpelsubproc>.
- *BPELJ* proposes a use of Java in BPEL processes, as well as access of Java Enterprise Edition code from BPEL processes: <http://www.ibm.com/developerworks/library/specification/ws-bpelj>.

Use of WSDL

A BPEL process can use a WSDL definition as the source of the data types used in variable declarations. (Any missing data types must be provided by a separate XML Schema.) As explained in the next sections, the process also references three WSDL extensions that are specific to BPEL: *partner link type*, *property*, and *property alias*.

- Each partner link type is the basis of one or more partner links, which each describe the business relationship of the process and a partner service.
- Each property gives a name to an important subset of business data.
- Each property alias *maps* a property to a type of message; that is, the alias identifies where the property data is located in a particular type of message.

Although a WSDL definition may include binding and location detail for a given service or requester, the BPEL process never accesses that detail, which is called an *endpoint reference*. Instead, the endpoint reference is specified by one of three sources: by the administrator of the BPEL server at configuration time; by an SOA product at run time (as described in our review of Service Component Architecture); or by the BPEL process at run time (as described in our review of the BPEL [assign](#) activity).

PartnerLinkType

A partner link type specifies the roles that are enacted during a runtime conversation between the BPEL process and a partner service. The simplest partner link type declares a single role and relates that role to a WSDL port type:

```
<partnerLinkType name="ProcessQuotePLT">
  <role name="ProcessQuoteRole">
    <portType name="ProcessQuote" />
  </role>
</partnerLinkType>
```

A single role is appropriate when only one port type is involved — that is, when every communication between the BPEL process and the partner service starts from the same direction. (In a synchronous communication, the same port-type operation describes the request and response messages.)

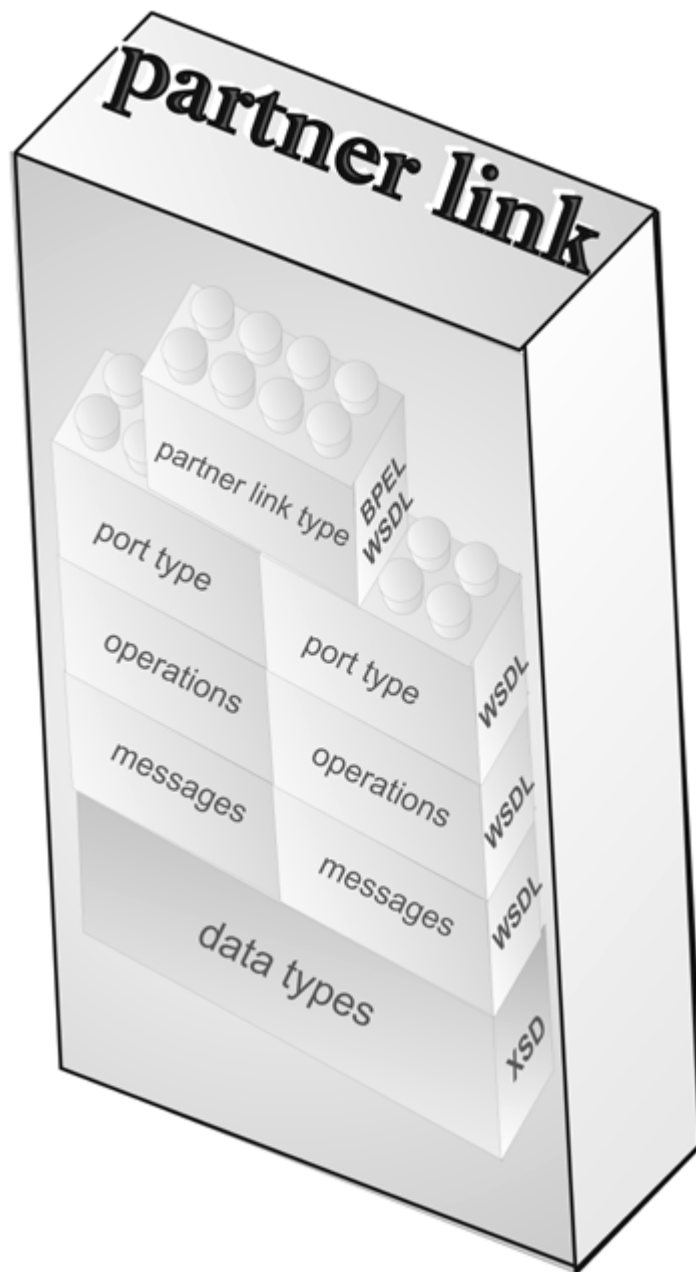
A partner link type specifies two roles when two port types are involved — that is, when a callback is in use, as explained in [Chapter 2](#). Here's an example of a partner link type that defines two roles:

```
<partnerLinkType name="MotorVehicleRecordsLT">
  <role name="motorVehicleRecordsService"
    portType="MotorVehicleRecords" />
  <role name="motorVehicleRecordsCallback"
    portType="MotorVehicleRecordsCallback" />
</partnerLinkType>
```

When you create a partner link type, you're not concerned with whether the BPEL process or the partner service fulfills a specific role. If the partner service is also a BPEL process, the two processes can use the same partner link type to describe the relationship between them. When you write a process, however, you use a partner link type to create a *partner link* (Figure 7.2), which is a kind of variable that (in a sense) "chooses a side" by indicating

- the role of the process you're creating (in other words, which port type describes the operations provided by the process); or
- the role of the partner service (in other words, which port type describes the operations provided by that service); or
- both, but only if the partner link type declares two roles.

Figure 7.2. Partner link



The BPEL engine uses the partner link to help manage the conversation between the two partners.

Properties and Property Aliases

In [Chapter 2](#), we mentioned use of identifiers (purchase-order numbers, invoice numbers, and so on) that help direct a message to the correct service instance at run time. Those identifiers *correlate* the processing done by one instance (which receives a purchase order) with the processing done by other instances (which submit an invoice, acknowledge receipt of payment, and so on).

When you work with BPEL, you can define WSDL-based constructs called properties and property aliases to perform the correlation function. The availability of these constructs provides several benefits.

First, as you code a BPEL process, you can use the same name (such as CustomerID) to access the same data in differently structured variables — for example, in a variable that holds a purchase order and in a variable that holds an invoice. This repeated use of the same name lets you focus on the business meaning of the data.

Second, properties make maintenance is easier. If an inbound message includes a customer ID, and a change occurs in the position of that ID, you don't necessarily change the logic in the BPEL process. If you've used properties well, you can simply change the property alias that associates the name CustomerID with a specific position in the variable.

Third, you can define a BPEL *correlation set*, which is essentially a list of constants (for example, a customer ID followed by an invoice number). Those constants help direct an inbound message to the correct instance of the BPEL process. They also allow automatic validation of positions in an outbound message. If the first element in an outbound message is expected to include a particular customer ID and does not, an error occurs at run time.

Let's consider an example.

[Listing 7.2](#) shows a WSDL definition that describes the message `placeQuoteRequestMsg`.

Listing 7.2. Message `placeQuoteRequestMsg` WSDL definition

```
<wsdl:message name="placeQuoteRequestMsg">
  <wsdl:part name="placeQuoteParameters"
    element="placeQuote" />
</wsdl:message>

<xsd:element name="placeQuote">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element
        name="quoteInformation"
        type="CustomerQuoteInformation"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="placeQuote">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element
        name="quoteInformation"
        type="CustomerQuoteInformation"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:complexType name="CustomerQuoteInformation">
  <xsd:sequence>
    <xsd:element name="applicant" type="NameAddress"
      minOccurs="0" />
    <xsd:element name="dependents" type="NameAddress"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="auto" type="Auto"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="applicantEmailAddr"
      type="xsd:string" minOccurs="0" />
    <xsd:element name="desiredPolicyInfo"
      type="PolicyQuoteTerms"
      minOccurs="0" />
  </xsd:sequence>
</xsd:complexType>
```

As shown, the message (`placeQuoteRequestMsg`) references an XSD element (`placeQuote`), which in turn references a type (`CustomerQuoteInformation`). The type has a field for an email address (`applicantEmailAddr`).

In another WSDL definition, you declare a property (`emailAddr`):

```
<property name="emailAddr" type="xsd:string" />
```

The property is of the same type (`xsd:string`) as the email address.

In this WSDL definition, you also declare a property alias:

```
<propertyAlias
  propertyName="emailAddr"
  messageType="placeQuoteRequestMsg"
  part="placeQuoteParameters" >
  <query>
    /quoteInformation/applicantEmailAddr
  </query>
</propertyAlias>
```

The alias declaration ensures that you can use the property name to refer to the appropriate field in a BPEL variable such as this one:

```
<variable name="quoteRequest"
  messageType="placeQuoteRequestMsg" />
```

The important point about this variable declaration is the `messageType` element, which indicates that the structure of the variable reflects the same message definition as the one referenced in the `propertyAlias` element. By the way, the variable is declared in a BPEL process, not in WSDL.

The `propertyAlias` element's `messageType` and `part` attributes refer by name to the `message` and `part` elements in the WSDL message definition. The `query` element identifies a search string, which is in a query language (XPath 1.0, by default). At run time, the BPEL engine uses the search string (hereafter called a *query*) to access the email address that resides in an appropriately structured variable.

You could have included other property aliases for the same property, in each case referring to a different WSDL message definition.

Assume that in the BPEL process, you declare these two variables:

```
<variable name="quoteRequest"
  messageType="placeQuoteRequestMsg" />
<variable name="theAddress" type="xsd:string" />
```

The process later receives a message into `quoteRequest`:

```
<receive name="processQuoteRequest"
  createInstance="yes"
  operation="placeQuote"
```

```
    partnerLink="ProcessQuote"  
    portType="ProcessQuote"  
    variable="quoteRequest">  
</receive>
```

You can access the email address from `quoteRequest` by invoking the function `getVariableProperty`, which is in a BPEL-specific namespace and takes two arguments (the variable name and the property name):

```
theAddress = bpel:getVariableProperty  
            ("quoteRequest", "emailAddr")
```

You can benefit from the convenience of properties even when your BPEL process accesses a variable that contains data other than a message. For example, after retrieving values from different online stores, you might

- retain a list of product names and, for each product, the prices in all stores
- calculate the average price of each product
- use a variable to hold a table, each row of which holds a product name and the average price for the named product

You can refer easily to the product name in every variable that you use in processing. The steps for establishing a property and a set of property aliases are similar to the steps described earlier. In this case, however, each property alias refers to an XML Schema element or type rather than to a WSDL message definition.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

BPEL File Structure

The topmost structure of a BPEL executable process can include the elements shown in [Listing 7.3](#) and also can include a BPEL extension element (`<extensions>`), which is used in relation to other technologies such as Service Component Architecture.

Listing 7.3. BPEL process elements

```
<process>
  <import> </import>
  <partnerLinks> </partnerLinks>
  <messageExchanges> </messageExchanges>
  <variables> </variables>
  <correlationSets> </correlationSets>
  <faultHandlers> </faultHandlers>
  <eventHandlers> </eventHandlers>
  <!-- The previous two elements include activities,
        as does the subsequent content of the process
        element. Activities can be enclosed in scopes,
        as described later. -->
</process>
```

Here are highlights of a process definition:

- Each import provides access to a WSDL definition or an XML Schema.
- Each partner link is a kind of specialized variable that describes the relationship between the BPEL process and a partner service.
- Each *message exchange* is an identifier that is used to avoid an ambiguous case in a complex business scenario — specifically, to pair a BPEL activity that receives a message with the activity that issues a reply.
- Each variable contains business data, whether to hold a message or for other use in the process logic.
- Each correlation set is a listing of properties used to correlate service instances.
- Each *fault handler* is a set of activities that run in response to a *fault*, which is a failure in the process.
- Each *event handler* is a set of activities that run concurrently with other activities, in response either to the passage of time or to receipt of a message.

User name:**Book:** SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Activities in Brief

[Table 7.1](#) lists the basic activities available to a BPEL-process developer. [Table 7.2](#) lists the structured activities.

Table 7.1. BPEL basic activities

Basic activity	Purpose
assign	Copy one or more values to variables and partner links
compensate	Invoke a set of compensation handlers
compensateScope	Invoke a specific compensation handler
empty	Act as a placeholder
exit	End the process immediately
extensionActivity	Do a task allowed by a technology that extends BPEL
invoke	Invoke a partner service
receive	Receive an inbound message
reply	Reply to an inbound message
rethrow	Re-direct an error from one fault handler to another
throw	Direct an error to a fault handler
validate	Validate one or more variables against their data types
wait	Block processing for a time

Table 7.2. BPEL structured activities

Structured activity	Purpose
extensionActivity	Do a task allowed by a technology that extends BPEL
If (else, elseif)	Process activities conditionally
forEach	Process activities a specified number of times, either sequentially or concurrently
flow	Process activities concurrently
Pick (onMessage, onAlarm)	Wait for one of potentially several events to occur
repeatUntil	Process activities in a loop whose subordinate activity always runs at least once
scope	Process activities that can be compensated separately from other activities
sequence	Process activities sequentially
while	Process activities in a loop whose subordinate activity runs zero or more times

Structured activity	Purpose
---------------------	---------

User name:

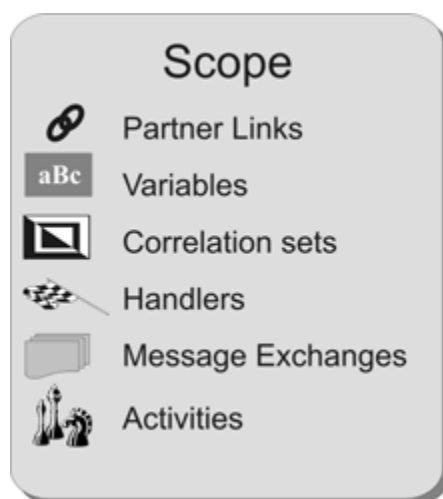
Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Scopes

As suggested in [Figure 7.3](#), the scope is a world of meaning, where the names of some variables and handlers are known and where specific data values and handler effects are available, while the names of other variables and handlers are unknown. A *scope* includes a single activity, which (if structured) can include others. An important characteristic is that a scope can be compensated without interfering with the behavior of activities and handlers in other scopes.

Figure 7.3. Scope

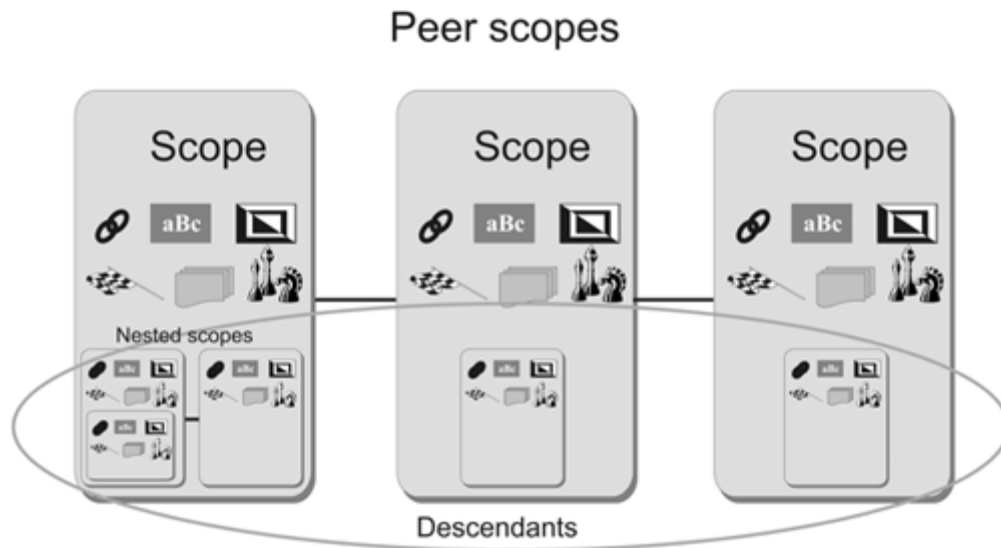


A BPEL process can include many scopes, and they provide a way for you to organize your work. Each scope

- can include one or more scopes, called nested scopes.
- can have peer scopes, which are scopes that are neither nested nor nesting in relation to one another.
- can include (within a nested scope) other nested scopes, to any nesting depth, as suggested in [Figure 7.4](#). All the nested scopes within a scope are said to be the scope's descendants. All superior scopes are said to be ancestor scopes, and an immediately superior scope is said to be a parent.

Figure 7.4. Peer and nesting scopes

[\[View full size image\]](#)



The rules of access for each category of name (for variables, partner links, and so on) are similar to the rules in other languages. Consider the case of variables:

- Variables declared in the topmost level of a process are global to the process.
- Variables declared in a scope are visible both in the scope and in the scope's descendants.
- Variables declared in a scope are not visible to any ancestor or peer scopes.
- A variable declared in a nested scope hides any same-named variable in a superior scope, and the variable in the superior scope is hidden not only from the nested scope but also from descendants of the nested scope.

A scope can include its own partner links, variables, correlation sets, event handlers, fault handlers, and message exchanges. In addition, two other handlers are available in a scope. A compensation handler is a set of activities that compensate for a change made successfully by the scope and that run after being invoked by a parent scope. A termination handler is a set of activities issued when a running scope is being forced to terminate.

The BPEL engine forces termination in any of the following cases:

- if the BPEL process has faulted
- if an ancestor scope has faulted
- if the scope is in a compensation handler, fault handler, or termination handler and the handler has faulted
- if a particular situation is in effect inside a `forEach` activity, as described in [Chapter 8](#)

When a scope is active at run time, the embedded event handlers are available, along with the scope's other activities. A scope

- is active as soon as the BPEL engine issues an activity in the scope
- is active until the BPEL engine gives control to a different scope (other than a descendant) or to the top level of the process
- begins and ends several times if embedded in a loop or event handler

A scope is said to have completed successfully if no faults occur in the scope. A fault may occur in a nested scope, however, and if the fault is handled without affecting the scope that encloses the nested scope, the enclosing scope may complete successfully.

If a fault occurs in a scope, the scope is said to have failed, and compensation is unavailable. Last, if the scope is in a loop, compensation is available only for the scope instances that succeeded.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Message Exchanges

A message exchange is an identifier that helps to define the association between an inbound message and the [reply](#) activity that responds to the inbound message. We provide an example in [Chapter 8](#).

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Variables

A variable is a data area that is based on a WSDL message definition (called a *message type*) or on an XML Schema element or type. In many cases, a variable is complex, not merely an integer or a string.

Listing 7.4 shows a few variable declarations.

Listing 7.4. Sample variable declarations

```
<variables>
  <variable name="quoteRequest"
            messageType="placeQuoteRequestMsg" />

  <variable name="theAddress" type="xsd:string">
    <from variable="quoteRequest"
          property="emailAddr" />
  </from>
</variable>

  <variable name="theAddress02" type="xsd:string">
    <from variable="quoteRequest"
          part="placeQuoteParameters" >
      <query>
        /quoteInformation/applicantEmailAddr
      </query>
    </from>
  </variable>

  <variable name="currentStatus"
            type="xsd:string"
            <from>
              <literal>approved</literal>
            </from>
  </variable>

  <variable name="myRequestID"
            element="placeQuote"/>
</variables>
```

You initialize a variable by including a `from` element in the declaration. A variable is initialized when the scope begins or, for global variables, when the process begins.

The sources of data are the same as the sources in an `assign` activity, which we describe later.

User name:**Book:** SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Variables and XPath

As mentioned in [Chapter 6](#), you prefix a BPEL variable name with a dollar sign (\$) if you need to access data from that variable. If the variable is based directly on an XSD type, use a location path that skips the type name and (as appropriate) uses names declared in the type. For example, assume that the BPEL variable named `applicant` is directly based on the following XSD type:

```
<xsd:complexType name="NameAddress">
  <xsd:sequence>
    <xsd:element name="first" type="xsd:string"
      minOccurs="1" />
    <xsd:element name="last" type="xsd:string"
      minOccurs="1" />
  </xsd:sequence>
</xsd:complexType>
```

Here is the XPath expression for accessing the content of the variable field named `last`.

```
$applicant/last
```

Similarly, if the variable is based on an XSD element, use a location path that skips the element name and (as appropriate) uses names declared in the related XSD type. For example, assume that the BPEL variable named `dmvResponse` is directly based on the following XSD element.

```
<xsd:element name="retrieveLicenseStatusResponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="valid" type="xsd:boolean" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Here is the XPath expression for accessing the content of the variable field named `valid`.

```
$dmvResponse/valid
```

A variation occurs if the variable is based on a WSDL message definition. To derive the variable name used in the

XPath expression, prefix the BPEL variable name with a dollar sign. Then, add a period and the name of a message part.

Assume that the BPEL variable named `dmvResponse` is based on the following WSDL message definition, which refers to the same XSD element as mentioned in the previous example.

```
<wsdl:message name="retrieveLicenseStatusResponseMsg">
  <wsdl:part name="retrieveLicenseStatusResult"
    element="retrieveLicenseStatusResponse" />
</wsdl:message>
```

Here is the XPath expression for accessing the content of the variable field named `valid`.

```
$dmvResponse.retrieveLicenseStatusResult/valid
```

As shown, you use a location path that skips the XSD element name specified in the WSDL message definition.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Partner Links

As noted earlier, the BPEL engine uses a partner link to help manage the conversation between two partners. Each partner link is a kind of variable that is based on a partner link type and contains an endpoint reference. You can use the BPEL [assign](#) activity to copy an endpoint reference to or from a given partner link.

Each activity that accesses a service in any way requires a partner link. Here is a declaration based on a partner link type that defines only one role.

```
<partnerLinks>
  <partnerLink name="ProcessQuote"
               partnerLinkType="ProcessQuotePLT"
               myRole="ProcessQuoteRole" />
</partnerLinks>
```

The attribute `myRole` indicates that the role is enacted by the BPEL process. Specifically, the BPEL process provides the operations described in the port type that is related to the role `ProcessQuoteRole`.

Let's look at a second, similar declaration. Here, the attribute `partnerRole` indicates that the role (related to another port type) is enacted by the partner service.

```
<partnerLinks>
  <partnerLink name="mainframeQuoteMgr"
               partnerLinkType="PartnerLinkPLT"
               partnerRole="handleMainframe" />
</partnerLinks>
```

Here is a declaration based on a partner link type that defines two roles.

```
<partnerLinks>
  <partnerLink name="MotorVehicleRecords"
               partnerLinkType="MotorVehicleRecordsLT"
               myRole="motorVehicleRecordsCallback"
               partnerRole="MotorVehicleRecords">
</partnerLinks>
```

Multiple partner links can be based on the same partner link type. The case arises, for example, when you use the same service interface to interact with multiple vendors.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

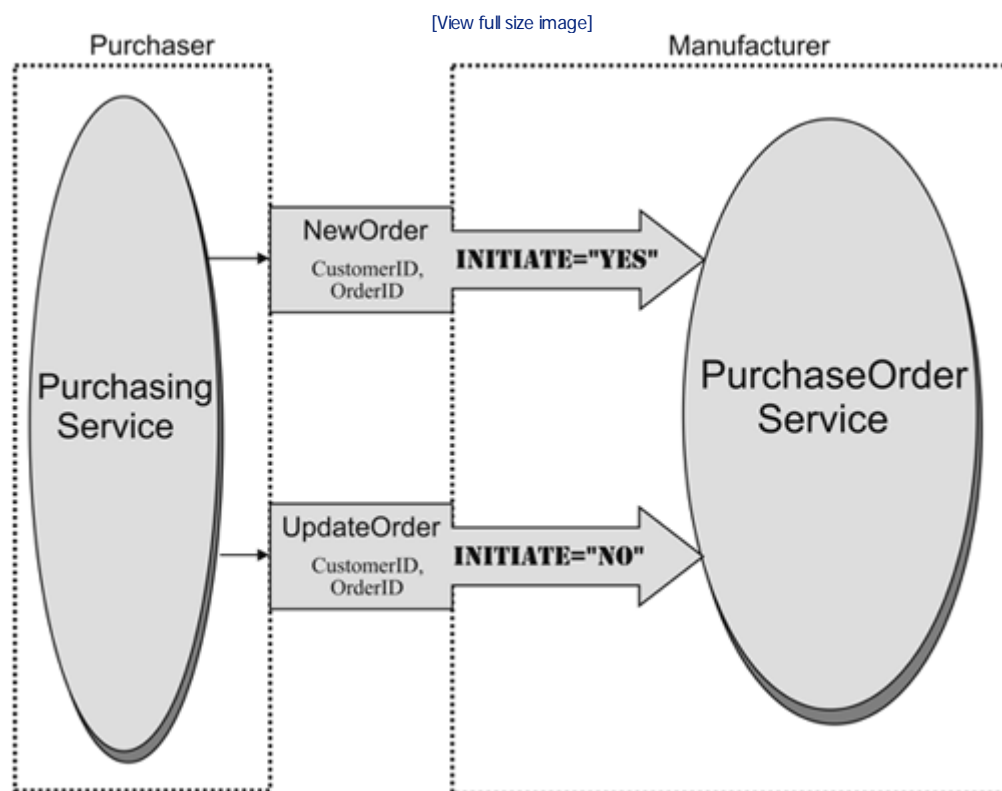
Correlation Sets

An application must maintain data integrity when messages are exchanged between services. A manufacturer that receives orders, for example, must not mix data for one order with the data for another.

A BPEL process addresses this issue with correlation sets, each of which is a list of properties whose values are expected to remain constant throughout a process or throughout a specific scope, even as data is transmitted to and from partner services.

As [Figure 7.5](#) illustrates, the BPEL process `PurchaseOrder` is maintaining a long-term conversation with a purchasing service. The process maintains a correlation set that includes two properties: customer ID and order number. The correlation set is *initiated* (assigned constant values) when a new order arrives. The process keeps those constant values separate from values that are received from any message or are transmitted as part of any message.

Figure 7.5. Correlation in BPEL



You reference a correlation set only in activities that receive or transmit data:

- in each inbound activity that may require the BPEL engine to direct a message to a specific service instance
- in an outbound activity, if you want to verify that the property values at the time of transfer are correct
- in any inbound or outbound activity that initiates the correlation set

In the third case, the values must remain unchanged in later inbound or outbound activities throughout the scope to which the correlation set applies or (outside a scope) throughout the process.

In the PurchaseOrder process example, an inbound activity initiates the correlation set. In an example that involves an outbound activity, the PurchaseOrder process exchanges order-specific data with another service (not shown in the figure). That secondary service checks the availability of parts and invokes the PurchaseOrder service with a callback that may provide only a partial response.

Correlation in this case involves not only the customer ID and order number but also a tracking code. When the BPEL process submits the initial request to the secondary service, the BPEL process initiates the new correlation set. The constant values that are assigned for that second correlation set are maintained separately from the constant values that were assigned for the first.

As suggested by this example, a BPEL process might use a variety of different correlation sets.

Initiation Attributes

Whenever you reference a correlation set in an activity, you indicate whether the activity is initiating the correlation set. In particular, you assign one of the following values to the `initiate` attribute that is specific to the correlation set: *no*, *yes*, or *join*.

The attribute's default value, *no*, means that the activity must not initiate the correlation set. In our example, a request to update an existing order does not assign new constant values. Instead, the BPEL engine uses the values in the inbound message to direct processing to a service instance associated with a correlation set that was already initiated. BPEL throws a fault if the activity determines that the inbound message has unexpected values or that the correlation set was not initiated previously.

If you set the `initiate` value to *no* for an outbound message, the BPEL process verifies that the values being sent are consistent with what was received or transmitted earlier.

An `initiate` value of *yes* means that the activity must initiate the correlation set. In our example, the receipt of a new order (probably in the first `receive` activity) causes initiation. BPEL throws a fault if the activity determines that the correlation set was initiated previously.

The third value possible `initiate` value is *join*. That value is used (for example) if you issue multiple `receive` activities concurrently and the same correlation set is required for each activity. The effect of *join* is that the first-running activity initiates the correlation set, and the BPEL engine directs the other messages to the correct service instance. If the activity determines that the correlation set has unexpected values, BPEL throws a fault.

In an `invoke` activity that invokes a service synchronously, you use the `pattern` attribute in addition to the `initiate` attribute. The need arises because the correlation sets referenced for the outbound message of the `invoke` activity may be different from the correlation sets referenced for the message returned to the BPEL process.

For each correlation set referenced in the synchronous `invoke` activity, you specify one of three values for the `pattern` attribute. A *request* value means that the correlation-set `initiate` attribute refers to the message sent to the partner service. A *response* value means that the correlation-set `initiate` attribute refers to the message returned to the BPEL process. A *request-response* value means that the correlation-set `initiate` attribute refers to both messages.

Effect of Correlation Errors

During synchronous invocation, if the message returned to the BPEL process has unexpected values, the values are received into one or more variables for use in fault processing. A transfer is not completed in other cases of correlation fault, regardless of the activity.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Fault Handling

A fault can come from any of the following sources:

- A partner service returns an error message. In this case, the fault name and data are described in the WSDL definition.
- The SOA runtime product returns an error, as when a partner service is unavailable or a network fails. In this case, the fault name and data are described (at best) in product documentation.
- The BPEL engine reports a runtime problem. Again, check your product documentation. You also can review the BPEL specification, which lists the faults (called *standard faults*) that are thrown by any standards-compliant BPEL engine.
- Your BPEL process issues a `throw` activity in response to a business problem. In this case, your company is defining the fault name and data and should be keeping a record of those definitions so they can be reused in other processes.

The primary purpose of fault handling is to minimize the effect of an error so that the usual work of the business can continue.

After a fault occurs in a given scope, the scope has failed and is not available for compensation by a parent scope. The BPEL engine selects a scope-specific fault handler, terminates other handlers and activities in the current scope, terminates each descendant scope that is still running, and runs the fault handler.

The fault handler can respond in various ways and often issues a `compensate` activity to invoke compensation handlers in nested scopes that completed successfully. If, on completion, the fault handler issues the `rethrow` activity, processing continues in a fault handler in the parent scope. The parent scope has failed and is not available for compensation by its parent scope. If the fault handler does not issue the `rethrow` activity, normal processing resumes either in the parent scope or (if the faulting scope was nested directly in the process) at the top level of the process.

If a fault reaches the top level of the process or originates there, the BPEL engine terminates the process and informs the SOA runtime product of the failure.

Selection at Run Time

A fault handler is composed of a `catch` or optional `catchAll` element, with embedded activities. The outline shown in Listing 7.5 has three fault handlers.

Listing 7.5. Fault handler outline

```
<faultHandlers>
  <catch faultName="oneFault"
    faultVariable="oneVariable"
    faultElement="getDMVRecordElement">
    <empty/>
  </catch>
  <catch faultName="twoFault">
    <empty/>
  </catch>
  <catch
    faultVariable="threeVariable"
    faultMessageType="getDMVRecordRequest">
    <empty/>
  </catch>
  <catchAll>
```

```

    <empty/>
  </catchAll>
</faultHandlers>

```

Each fault handler has selection criteria, which include (optionally) the fault name and the type of data.

If you specify a variable name, the BPEL process implicitly declares a variable that receives data and is local to the fault handler. The declaration is based on either an XML Schema element or a WSDL message type.

The BPEL engine selects the fault handler whose selection criteria most closely mirror the fault. Let's wade into a description of "most closely."

If the fault has a name but no data, the BPEL engine selects the fault handler whose selection criteria match exactly. If no such fault handler is available, the engine selects the `catchAll` handler. If the fault has data, the BPEL engine selects the fault handler whose selection criteria match exactly; or (if necessary) match nearly, with an exact match on fault name as long as no data type is specified in the selection criteria; or (if necessary) match nearly, when no fault name is specified in the selection criteria but an exact match is found on the data type. If no fault handlers conform to those rules, the engine selects the `catchAll` handler.

Here are some results when the fault handlers for a given scope are as shown in the preceding figure:

- If the scope throws a fault named *oneFault* and the fault has no data, the BPEL engine selects the `catchAll` handler.
- If the scope throws a fault named *twoFault*, the BPEL engine selects the fault handler named *twoFault* regardless of whether the fault has data.
- If the scope throws a fault named *unknown* and the fault has data that corresponds with a message of type `getDMVRecordRequest`, the BPEL engine selects the third fault handler.

If any fault handler issues a `rethrow` activity, the data presented to the fault handler of the parent scope is identical to the original data, even if the original fault handler changed the data or did not include a variable to accept the data.

If you don't include a `catchAll` handler in a given scope or at the process level, the BPEL engine provides the following default:

```

<catchAll>
  <sequence>
    <compensate/>
    <rethrow/>
  </sequence>
</catchAll>

```

The Inner Life of a Fault Handler

A fault handler can be quite complex, with internal scopes, and a fault can occur in a fault handler. Here are a few details, which you can skip if your head hurts:

- The top level of a fault handler (for example, the handler called *one*) cannot include a fault, compensation, or termination handler; and no default handlers are available at that level.
- A scope nested directly in the fault handler cannot include a compensation handler. This second restriction is caused by the first, as you have no way to invoke a compensation handler except from a fault, compensation, or termination handler in a parent scope.
- A fault in a fault handler is always thrown to a parent scope, regardless of whether you issue a `rethrow` activity.
- A fault in a fault handler is ultimately thrown to the process — specifically, to the parent scope of the scope that contains the fault handler (in our case, the fault handler called *one*).

Exiting in Response to a Fault

Two attributes of the `process` element affect fault handling.

First, if you set the `exitOnStandardFault` attribute to *yes* rather than to its default *no* value, a BPEL standard fault exits the process without causing invocation of a fault or termination handler. The only standard fault that is unaffected by this attribute is the concurrency-related fault *bpel:joinFailure*.

Second, the attribute `suppressJoinFailure` affects how the process responds to the standard fault *bpel:joinFailure*. The details are later in this chapter, in the section on advanced concurrency.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Compensation Handling

As indicated earlier, compensation for a successfully completed scope occurs under the guidance of a compensation handler, which reverses the scope's effect in response to a business development such as a cancelled purchase.

Imagine that scope in a BPEL process invokes a data-access service to record a customer's purchase. The service commits the changes because a multi-user database should not (and perhaps cannot) stay in an uncommitted state for days. The parent scope, however, runs an event handler for days more, waiting for a cancellation message for as long as the customer is allowed to cancel. If the message comes, the event handler issues a fault, and the fault handler invokes a compensation handler that in turn invokes a service to revise the database.

Invocation of a compensation handler always comes from a parent scope — specifically, from a fault, compensation, or termination handler that issues either of two activities: `compensateScope` or `compensate`.

The `compensateScope` activity invokes the compensation handler of a specific nested scope. Here, a failure in scope A causes invocation of the compensation handler in scope B:

```
<scope name="A">
  <compensationHandler>
    <compensateScope target="B">
  </compensationHandler>
  <scope name="B">
    <compensationHandler>
      <!-- a basic or structured activity is here -->
    </compensationHandler>
  </scope>
</scope>
```

The `compensate` activity invokes the compensation handler in each nested scope, in most cases in an order that is opposite to the order of scope completion. That behavior is guaranteed, however, only when the initiation of one scope depends on the completion of another, as is true when nested scopes run in sequence. Otherwise, the order is specific to a BPEL engine.

The `compensate` activity does not handle descendant scopes other than the immediately nested scopes. The assumption is that an invoked compensation handler will itself invoke the compensation handlers at the next descendant level.

The name of a scope embedded in a loop refers to all the scope instances that run in that loop, and the `compensate` or `compensateScope` activity from the parent scope invokes a *group* of compensation handlers, as appropriate.

A compensation handler has access to all values that were in effect when the scope completed, including the values of variables, correlation sets, and partner links. A problem can arise, however, if scopes are running at the same time as the one being compensated. You can ensure that changes to shared variables in one scope do not affect the values of variables in another. For details on isolating one scope from another, see the description of the `scope` activity in [Appendix C](#).

If you do not specify a compensation handler for a given scope, the BPEL engine uses the following default.

```
<compensationHandler>  
  <compensate/>  
</compensationHandler>
```

A compensation handler (for example, a handler named *inProcess*) can be internally complex, and you can use embedded compensation handlers to ensure (as much as possible) that the activities in handler *inProcess* succeed or fail as a unit, rather than leaving some changes compensated and some uncompensated.

Two rules apply to the internals of a compensation handler. First, the top level cannot include a fault, compensation, or termination handler; and no default handlers are available at that level. Second, a scope nested directly in the compensation handler cannot include a compensation handler.

A business process might need to undo a compensation that completed successfully, but the question arises, "How many reversals are supported?" A compensation that compensates for a previous compensation might need.... You see the issue.

A BPEL process offers no routine way to undo the effects of a compensation handler that completed successfully. A business process ultimately relies on personal intervention.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Termination Handling

As noted earlier, a termination handler is a set of activities that are issued when a running scope is being forced to terminate. If you do not specify a termination handler for a given scope, the BPEL engine uses the following default:

```
<terminationHandler>
  <compensate/>
</terminationHandler>
```

The rules that apply to the internals of a compensation handler apply here as well, with the added restriction that a fault handler in a termination handler cannot rethrow a fault.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

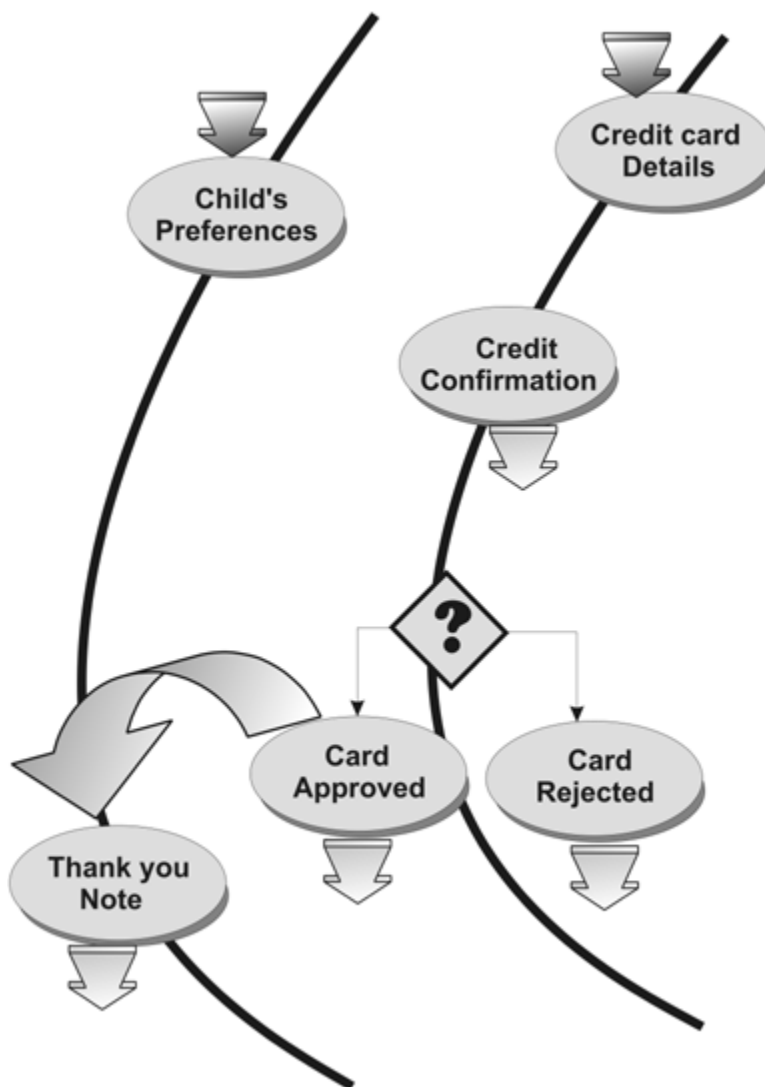
Introduction to Concurrency

We'll explore concurrency by example.

Imagine an online music store that caters to children and requires two kinds of information: preference information from the child, and credit-card details from a parent. You can fulfill those requirements at more or less the same time, in two parallel streams, and we can assume that completion of both streams is required before some other action occurs — for example, before you inform partner companies about the new customer.

In a more complex case, one stream can be internally dependent on another. The process might send a thank-you note to the child, for example, only after the parent's credit is approved, as shown in [Figure 7.6](#).

Figure 7.6. Concurrency example



When activities run concurrently and one activity is forced to wait for another, the activities are said to be

synchronized when the second activity starts.

You may say, "If a thank-you note requires that the process receive preferences and gain credit approval, you don't need a concurrent stream for the note-sending activity at all. The note-sending activity can start later, after you inform the partner companies."

That's true. BPEL provides alternate ways to accomplish the same purpose. If one activity in your business depends on the successful completion of another activity, you can enforce the dependency by writing a BPEL process in either of the following ways:

- by including the two activities in a [sequence](#) activity, where an embedded . . . activity runs subsequent to another embedded activity in an unchanging order
- by fulfilling two tasks:
 - Place both activities in a [flow](#) activity, where an embedded activity runs concurrently with another embedded activity.
 - Establish a *link* between the two activities. Name a link, assign the link as a *source* in the activity that must occur earlier, and assign the link as a *target* in the activity that must occur later.

[Listing 7.6](#) shows an outline of a [flow](#) activity that enforces a dependency but does not fulfill the business requirement. We'll describe the activity and make the necessary changes.

Listing 7.6. Sample flow activity outline

```
<flow>
  <links><link name= "creditApproval"/> </links>

  <sequence name="interactWithChild">
    <receive name="acceptPreferences"/>
    <invoke name="sendNote">
      <targets>
        <target linkName="creditApproval"/>
      </targets>
    </invoke>
  </sequence>

  <sequence name="interactWithParent">
    <receive name="creditCardDetails"/>
    <invoke name="creditConfirmation"/>
    <if name="approved">
      <condition> ... </condition>
      <invoke name="informPartnerCompanies">
        <sources>
          <source linkName="creditApproval"/>
        </sources>
      </invoke>
    <else/>
    </if>
  </sequence>
</flow>
```

In the outline, two activities run concurrently. The [sequence](#) activity named *interactWithChild* accepts the child's preferences and sends a thank-you note. The [sequence](#) activity named *interactWithParent* receives credit-card details, confirms the parent's credit-worthiness, and, if credit is approved, informs the partner companies.

A link called *creditApproval* enforces a dependency, as indicated by the use of the [target](#) and [source](#) elements. The process will not send a note to the child unless the parent-related [invoke](#) activity succeeds. In general terms, a *target activity* depends on one or more *source activities*.

The problem is that the target activity is dependent on informing the partner companies and is not dependent on approving the parent's credit.

"The solution," you may say, "is to move the link source from the [invoke](#) activity to the [if](#) activity."

```
<if name="approved">
```

```

    <condition> ... </condition>
    <sources> <source linkName="creditApproval"/> </sources>
    <invoke name="informPartnerCompanies"/>
<else/>
</if>

```

The change seems sensible but creates a new problem. The source activity is now the `if` activity, which can complete successfully even if the `else` logic (if any) is invoked. Assuming no fault occurs, the process sends a note to the child regardless of whether the parent's credit is approved.

Let's do as follows:

- Introduce the BPEL `empty` activity to the `if` activity, to help us enforce a precise synchronization.
- Use a BPEL `sequence` activity as a container in the `if` activity.

```

<if name="approved">
  <condition> ... </condition>
  <sequence name="oneActivity">
    <empty>
      <sources>
        <source linkName="creditApproval"/>
      </sources>
    </empty>
    <invoke name="informPartnerCompanies"/>
  </sequence>
<else/>
</if>

```

In this case, the empty activity is a container for the link source, and the sequence activity is a syntactic requirement. An if activity can include no more than one activity between the if start tag and the next else or elseif activity, if any.

That example fulfills the business need.

"Wait!" you say as you synchronize your thoughts with what you've read. "Can link sources and targets be in any BPEL activity?"

They can, although a few restrictions apply. It is not valid, for example, to place a link source (or link target) inside a loop while the related link target (or link source) is outside the loop, as in the following outline.

```

<!-- Not Valid! -->
<flow>
  <links><link name="sentFinalContract"/> </links>
  <invoke name="sendIt">
    <sources>
      <source linkName="sentFinalContract"/>
    </sources>
  </invoke>
  <while name="loop">
    <sequence name="oneActivity">
      <empty>
        <targets>
          <target linkName="sentFinalContract"/>
        </targets>
      </empty>
      <invoke name="sendPartyInvitation"/>
    </sequence>
  </while>
</flow>

```

Here is a possible solution:

```
<flow>
  <links><link name="sentFinalContract"/> </links>
  <invoke name="sendIt">
    <sources>
      <source linkName="sentFinalContract"/>
    </sources>
  </invoke>
  <while name="loop">
    <targets>
      <target linkName="sentFinalContract"/>
    </targets>
    <invoke name="sendPartyInvitation"/>
  </while>
</flow>
```

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Advanced Concurrency

BPEL supports complex synchronization scenarios:

- A target activity may depend on several source activities.
- Several target activities may depend on the same source activity.

To see how the ideas fit together, consider the outline shown in [Listing 7.7](#), which is from a service that verifies customer details at Highlight Insurance. Two aspects of the outline are of interest and are explained next: transition conditions, as declared in the `transitionCondition` elements, and join conditions, as declared in the `joinCondition` elements.

Listing 7.7. Advanced concurrency example

```
<flow name="approveAndPurchase">
  <links>
    <link name="CreditLink1"/>
    <link name="DMVLink1"/>
    <link name="DMVLink2"/>
  </links>

  <invoke name="RunCreditCheck">
    <sources>
      <source linkName="CreditLink1">
        <transitionCondition>
          $checkResponse.creditCheckResult/valid
        </transitionCondition>
      </source>
    </sources>
  </invoke>

  <sequence>
    <invoke name="CheckApplicantWithDMV">
      <sources>
        <source linkName="DMVLink1">
          <transitionCondition>
            $dmvResponse.retrieveLicenseStatusResult/valid
          </transitionCondition>
        </source>
      </sources>
    </invoke>

    <invoke name="CheckDependentsWithDMV">
      <sources>
        <source linkName="DMVLink2">
          <transitionCondition>
            $dmvResponse.retrieveLicenseStatusResult/valid
          </transitionCondition>
        </source>
      </sources>
    </invoke>
  </sequence>

  <sequence name="Accepted">
    <targets>
      <target linkName="CreditLink1" />
    </targets>
  </sequence>
</flow>
```

```

        <target linkName="DMVLink1" />
        <target linkName="DMVLink2" />
        <joinCondition>
            $CreditLink1 and $DMVLink1 and $DMVLink2
        </joinCondition>
        <targets>
        <assign name="copyAcceptanceData" />
        <invoke name="sendAcceptanceNote"/>
    </sequence>

    <sequence name="Rejected">
        <targets>
            <target linkName="CreditLink1" />
            <target linkName="DMVLink1" />
            <target linkName="DMVLink2" />
            <joinCondition>
                not($CreditLink1 and $DMVLink1 and $DMVLink2)
            </joinCondition>
        </targets>
        <assign name="copyRejectionData" />
        <invoke name="sendRejectionNote"/>
    </sequence>
</flow>

```

Transition Conditions

A *transition condition* is a Boolean expression that is specific to a link source. When a source activity completes successfully, the BPEL engine evaluates the transition condition for each link source in that activity and assigns the value to each of the related links.

In response to a policy request, for example, Highlight Insurance verifies details with a credit agency and with the DMV, using parallel streams to save time. If *RunCreditCheck* (the `invoke` statement that contacts a credit agency) succeeds (which means, ends without a fault), the link *CreditLink1* is set to *true* or *false*, depending on the credit agency's response. A similar assignment occurs for each of the two DMV-related invocations.

If multiple link sources are present for a single activity, the order of evaluation is the order of the `source` elements. If a transition condition is not present for a given source activity that succeeds, the value of the link is *true*.

When a source activity fails (ending with a fault), the transition condition evaluates to *false*. That rule does not apply, however, in the following cases:

- The source activity is a scope.
- A fault occurs in that scope.
- The scope's fault handler completes without throwing a fault.

In that situation, the BPEL engine evaluates the transition condition as if the scope had succeeded. As always, the default transition condition is *true*.

Join Condition

A *join condition* is an expression that is specific not to a link target but to a target activity. The target activity can run only when two conditions apply:

- The BPEL engine resolves every related link source to *true* or *false*.
- The join condition for the target activity evaluates to *true*.

In our example, if the credit and DMV checks resolve to *true*, the sequence named *Accepted* runs. If any check resolves to *false*, the sequence named *Rejected* runs.

The join condition can include only the value of each link, plus Boolean operators that combine the values. The default join condition combines the link values so that if any of the links are *true*, the target activity runs.

If a join condition evaluates to *false*, processing continues normally or the BPEL engine throws the fault *bpel:joinFailure*. The specific outcome depends on the value of attribute `suppressJoinFailure` in the target activity. If the value of that attribute is *yes*, processing continues normally. The implication is that processing can skip activities for which the join condition was false until an activity is reached that is valid to run. The skipping of target activities during normal processing is called *dead path elimination*.

If a join condition fails and the value of attribute `suppressJoinFailure` is *no*, the BPEL engine throws a fault.

If you do not specify a value for the attribute `suppressJoinFailure`, a value is derived from the nearest enclosing activity or process for which you did specify a value, even if that enclosing activity is outside the `flow` activity. The default value for the process-level `suppressJoinFailure` attribute is *no* so that you don't need to change a default setting to ensure that fault handling occurs.

In the following outline, the value of attribute `suppressJoinFailure` in the `invoke` activity is *yes*.

```
<flow suppressJoinFailure="yes">
  <links><link name="creditApproval"/> </links>
  <sequence>
    <receive/>
    <invoke>
      <targets>
        <target linkName="creditApproval"/>
      </targets>
    </invoke>
  </sequence>
  <sequence>
    .
    .
  </sequence>
</flow>
```

In the next outline, the value of attribute `suppressJoinFailure` in the `invoke` activity is also *yes*.

```
<flow suppressJoinFailure="no">
  <links><link name="creditApproval"/> </links>
  <sequence suppressJoinFailure="yes">
    <receive/>
    <invoke>
      <targets>
        <target linkName="creditApproval"/>
      </targets>
    </invoke>
  </sequence>
  <sequence>
    .
    .
  </sequence>
</flow>
```

If no enclosing activity specifies a value, you can assign a default value at the process level. Here, the value of attribute `suppressJoinFailure` in the `invoke` activity is also *yes*:

```
<process suppressJoinFailure="yes">
  <flow>
    <links><link name="creditApproval"/> </links>
    <sequence>
      <receive/>
      <invoke>
        <targets>
          <target linkName="creditApproval"/>
        </targets>
      </invoke>
    </sequence>
    <sequence>
      .
    </sequence>
  </flow>
```

```
        .  
    </sequence>  
</flow>  
</process>
```