

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 6. Introduction to XPath

We introduced XML and showed how it made possible the three best-known SOA standards: WSDL, SOAP, and UDDI. We'd like to progress now to *Business Process Execution Language (BPEL)*, which is used to create services that orchestrate real-world business processes.

We can describe how each service (called a *BPEL process*) accepts messages from Web services A and B and sends related messages to Web services X and Y. But first we need to review how the BPEL process queries a message to retrieve specific data and how it calculates and compares values, which may be derived from the queried data.

If we're to do more than transfer data from here to there, we need a lower-level language; by default, BPEL 2.0 relies on *XPath 1.0*.

In addition to its use in BPEL, XPath (which stands for *XML Path Language*) is used in Service Component Architecture (SCA) and Service Data Objects (SDO), to isolate specific values. XPath is also central to XQuery 2.0, which we expect will become a widely used technology for accessing business data. Moreover, XPath plays a key role in XML Stylesheet Language Transformations (XSLT), a language for reorganizing data to accommodate the input requirements of different services, to handle calculations and comparisons more easily, and to allow use of a single XML source from which you derive a variety of outputs.

This chapter describes the first version of XPath and may be sufficient for your work in the language. If you're working with XPath 2.0, you'll need details that are available elsewhere — for example, in Michael Kay's work, *XPath 2.0 Programmer's Reference* (Wrox Press, 2004). Unless otherwise stated, our comments apply in either case.

XPath is a language for *addressing* (that is, accessing) the values in *XML source*, which is either an XML document or a variable based on XML. The language is also used for creating numeric, string, and Boolean expressions. Those expressions can include XML-stored values, as well as literals, operands, function calls, and other XPath expressions.

The defining aspect of XPath is the *location path*, which is the syntax for addressing XML-based values. To get you started with that syntax, this chapter offers examples and informal descriptions. Language specifications are at the following W3C sites: <http://www.w3.org/TR/xpath> (for XPath 1.0) and <http://www.w3.org/TR/xpath20> (for XPath 2.0).

Our explanations don't assume that you're trying our examples or creating your own, but if you wish to gain practical experience, you can set up a Windows 2000/NT/XP environment as described in [Appendix B](#). An alternative for Java programmers is to use the Java API for XML Processing (JAXP), as noted in the following article: <http://www-128.ibm.com/developerworks/library/x-javaxpathapi.html>.

Nodes

The XPath processor reads the XML source and includes the information in a series of data structures called *nodes*, which include only the information necessary for data access. An XPath node doesn't provide detail, for example, on whether an attribute value was embedded in single or double quotation marks.

Seven different kinds of nodes are related to one another in a tree structure that is specific to XPath. The purpose of four of the seven nodes is straightforward. Each *element*, *attribute*, *comment*, and *processing-instruction node* has information that was derived from a corresponding aspect of the XML source. Each *text node* has information on the text value of an XML element. Each *namespace node* has information on the namespaces that are in scope for a given element. Last, the single *root node* (what XPath 2.0 calls the *document node*) has information on the entire XML document.

In the tree structure built from the following example, the children of the root node are, in order, a comment node, an element node, and another comment node.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- here is an insured -->
<Insured></Insured>
```

```
<!-- end of file -->
```

The root node is not the same as the *root element*, which is the ancestor of all other elements in the XML source. The root node is more inclusive; it is the ancestor of the element node that was derived from the root element.

Listing 6.1. Sample XML document

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- CarPolicy applicant -->
<Insured CustomerID="5">
  <CarPolicy PolicyType="Auto">
    <Vehicle Category="Sedan">
      <Make>Honda</Make>
      <Model>Accord</Model>
    </Vehicle>
    <Vehicle Category="Sport" Domestic="True">
      <Make>Ford</Make>
      <Model>Mustang</Model>
    </Vehicle>
  </CarPolicy>
  <CarPolicy PolicyType="Antique">
    <Vehicle Category="Sport">
      <Make>Triumph</Make>
      <Model>Spitfire</Model>
    </Vehicle>
    <Vehicle Category="Coupe" Domestic="True">
      <Make>Buick</Make>
      <Model>Skylark</Model>
    </Vehicle>
    <Vehicle Category="Sport">
      <Make>Porsche</Make>
      <Model>Speedster</Model>
    </Vehicle>
  </CarPolicy>
</Insured>
```

The XPath nodes have no details on the XML declaration. They also lack details on a DOCTYPE declaration, which is present when a validation mechanism called a Document Type Definition (DTD) is in use.

You can access specific data by referencing the nodes in a sequence that leads from the root node to the nodes of interest. Every node has a string value, so you gain access to a unit of business data as soon as you reference (in particular) an element or attribute node. Consider, for example, the XML document shown in Listing 6.1.

Here's a kind of XPath expression (called a *location path*) for accessing the make of the vehicles whose *Category* value is *Coupe*.

```
/Insured/CarPolicy/Vehicle[@Category='Coupe']/Make
```

As we describe this expression in the following paragraphs, we refer to nodes by a type name, as when we say *Vehicle* node.

The initial virgule (/) in the expression indicates that the search for data starts at the root node. The set of characters between one virgule and the next represents a *location step*. Each location step selects nodes based on criteria that you specify.

The first step (*Insured*) brings the search to the node that is subordinate to the root node and has details on the root element. The *Insured* node refers to a single element, but the general rule is important: location steps provide access to a *node set*, which is a group of nodes that (with exceptions) are arranged in *XML-source order* (an order that reflects the sequence of content in the XML source) or is an empty set. An empty set is the outcome when no node conforms to the selection criteria.

We will have more to say about ordering in due time.

In general terms, a location path is an XPath expression that resolves to a node set. An *absolute location path* is one that starts at the root node, and a *relative location path* is one that starts in the middle of a node tree.

The second step in our sample expression (`CarPolicy`) brings us to a node set that has multiple members — specifically, a set of all `CarPolicy` nodes that are children of the `Insured` node.

The third step (`Vehicle[@Category='Coupe']`) continues the path, referencing all `Vehicle` nodes that are children of any `CarPolicy` node that is itself a child of the `Insured` node. The brackets (`[]`) and the syntax internal to them is a *predicate*, which contains a Boolean expression or (as shown later) an abbreviation that is expanded to a Boolean expression. The XPath processor selects only the nodes for which the expression evaluates to *true*.

In this case, the location step means "access all the `Vehicle` nodes, with the further restriction that the string value of the `Category` attribute node is *Coupe*." When you refer to an attribute node in a predicate, you precede the name with the "at sign" (`@`), as shown.

Here's another predicate, outside our example.

```
[@Exterior='white' and @Interior='red']
```

You might read this as, ". . . with the further restriction that the exterior is white and the interior is red."

Continuing with our main example, the fourth step (`Make`) completes the path, referencing the `Make` node for the `Vehicle` nodes whose `Category` attribute value is *Coupe*. In this case, the overall expression resolves to an element node whose string value is *Buick*.

XPath cannot create nodes or add detail to an XML source. You can use XPath in the context of XSLT, however, to create output that is based on, first, an XML source and, second, a set of directions (including XPath expressions) that are supplied in an XML stylesheet. If you use the instructions in [Appendix B](#) to try out XPath expressions, you'll be creating an output with XSLT.

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.

Avoiding Errors

We interrupt this description of a language to give you some practical pointers on avoiding errors.

First, if you work with the environment described in [Appendix B](#), you'll find that the XPath expression causes an error if the syntax is incorrect but does not cause an error if names or values are not in the XML under review. (You receive no error message, for example, if you type [Category](#) instead of [@Category](#).) That lack of errors doesn't extend to every other product that incorporates XPath, however. When you work in BPEL, you'll find that an empty set itself may cause an error if you copy an empty set to a variable.

Second, when you work with XPath in an XML file rather than through a user interface, consider the following points:

- Delimit a string with single quotes (or the characters `'`) if the string is in an expression that is delimited by double quotes. Similarly, delimit a string with double quotes (or the characters `"`) if the string is in an expression that is delimited by single quotes. Use of the same kind of quotation marks for the string and the expression ends the expression prematurely.
- To express comparison operators that include angle brackets, use the following characters:
 - `>` for greater than (`>`)
 - `<` for less than (`<`)
 - `>=` for greater than or equal to (`>=`)
 - `<=` for less than or equal to (`<=`)
- Use `&` if you wish to type an ampersand (`&`).

Last, when you're working on a real-world problem, try to recall a namespace-related issue that we now describe.

If an XML element is in the default namespace, you cannot access the element node by name unless the XPath expression uses a prefix when referring to that name. In the following XML source, for example, [CarPolicy](#) is in namespace `defaultNamespace`.

```
<other:Insured xmlns="defaultNamespace"
               xmlns:other="otherNamespace">
  <CarPolicy type="Antique" />
</other:Insured>
```

The next XPath expression, however, resolves to an empty set because [CarPolicy](#) is not being addressed correctly.

```
/other:Insured/CarPolicy
```

To access the [CarPolicy](#) node, you can use syntax (as shown later) for accessing a node without referencing a node name at all. The solution that applies more often, however, is to ensure that the XPath expression has access to and uses the appropriate namespace.

To demonstrate the solution, we need to show the XPath expression in its native habitat, inside an XML file. Assume,

for example, that the XPath expression resides in the `from` element of a BPEL process, as shown next.

```
<from xmlns:abcde="defaultNamespace"
      xmlns:other="otherNamespace">
  $myVariable/other:Insured/abcde:CarPolicy
</from>
```

The BPEL variable (`$myVariable`) that begins the XPath expression contains XML-based data such as a transmitted message. In this case, the specific variable contains the `other:Insured` node that we just described. The location path that follows the variable identifies how to access the data inside that variable.

Do you see what we've done? The XML element that contains the XPath expression has the information necessary to ensure that the XPath expression works. Specifically, the XPath expression has access to a prefix (`abcde`) that in turn refers to the namespace URI called `defaultNamespace`. The namespace problem is fully solved because `CarPolicy` is being addressed with a prefix that refers to `defaultNamespace`.

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.

Context

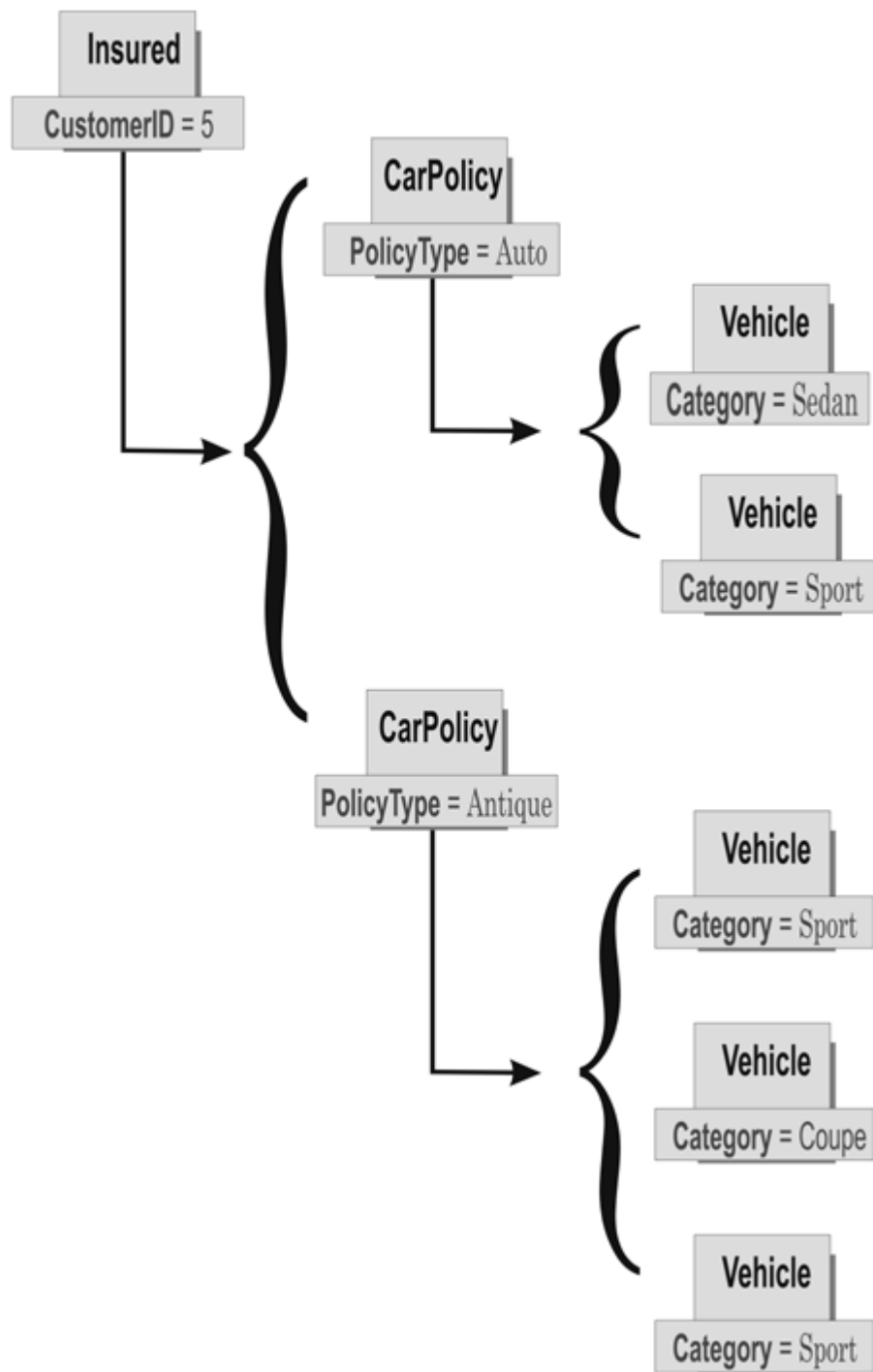
Let's return to the language itself.

The XPath processor evaluates a location path one step at a time, and each node that is selected during a given step provides a *context*, which is information that limits which nodes are available in the next step.

As suggested in [Figure 6.1](#), if we descend the node tree `/Insured/CarPolicy/Vehicle` one generation at a time

- a single `Insured` node provides a context that limits us to selecting (at most) the two `CarPolicy` nodes
- the first of the `CarPolicy` nodes provides a context that limits us to selecting (at most) the first two `Vehicle` nodes
- the second `CarPolicy` node provides a context that limits us to selecting (at most) the last three `Vehicle` nodes

Figure 6.1. Context



The node that provides a context at a given time is called the *context node*.

Perhaps the best way to understand a location path is to consider the location-path characters that act as operators. As we saw earlier, if an expression begins with a virgule, the search starts at the root node. The essential point, however, is that if the virgule is placed between one location step and the next (in `CarPolicy/Vehicle`, for example), the XPath processor selects *in turn* each node (in this case, each `CarPolicy` node) that was retained by the location step at the left of the virgule. The node selected at a given time is the context node. The processor then uses that node when evaluating the location step at the right of the virgule. The virgule completes its operation only when, for each context node, the XPath processor evaluates the right-side location step.

Similarly, the presence of a predicate (in `Vehicle[@Category='Coupe']`, for example) causes the XPath processor to select *in turn* each node (in this case, each `Vehicle` node) that was retained by the syntax that immediately precedes the predicate in the same location step. The selected node is the context node. The processor retains that node only if the predicate evaluates to *true*. The predicate completes its operation only when, for each context node, the XPath processor evaluates the predicate.

In both cases, the node used "in turn" is the context node.

Our definition of context node is the one you need as you explore location paths, but be aware that we're using a subset of the W3C definition, which states that the context node is the node being processed at a given time. The W3C definition and the explanations required to make it meaningful are geared to developers of XPath processors.

The word "context" is included in additional terms. In some cases, *context position* reflects the position of the context node in the sequence of possible context nodes. That position ranges from 1 to the *context size*, which is the number of nodes in the set of nodes that are each used (in turn) as a context node. In a broader sense, *context position* reflects the position of a node in a node set, and *context size* is the number of nodes in that set. We'll show the different uses of these terms as appropriate.

You can use a *positional predicate*, which is a predicate that restricts the addressed nodes based on the context position. For example, the following expression restricts the selection to the second `Vehicle` child within each `CarPolicy` node.

```
/Insured/CarPolicy/Vehicle[position()=2]
```

Those nodes represent *Ford Mustang* and *Buick Skylark*. As described later, however, the only string value that is displayed (if you're trying out these examples) is *Ford Mustang*.

If the comparison operator is an equal sign (=), an abbreviated form of a positional predicate is valid. The last expression can be stated as follows.

```
/Insured/CarPolicy/Vehicle[2]
```

Other operators are valid, too, as listed later. You can select the nodes at any position greater than 1, for example.

```
/Insured/CarPolicy/Vehicle[position()> 1]
```

Here, the returned nodes represent *Ford Mustang*, *Buick Skylark*, and *Porsche Speedster*. However, only *Ford Mustang* is displayed as a string value.

You can verify the number of returned nodes by using the XPath function *count*, which takes a node set as its only argument. The next example returns the value 3.

```
count(/Insured/CarPolicy/Vehicle[position()> 1])
```


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.

Examples

We have more to tell before we present "Location Steps in Summary," a later section that concisely explains what location steps do. For now, let's look at some more examples.

- As mentioned, `/Insured/CarPolicy/Vehicle` makes available all `Vehicle` nodes that are children of any `CarPolicy` child within `Insured`. The following expression resolves to the node whose string value is *Buick*.

```
/Insured/CarPolicy/Vehicle[@Category='Coupe']/Make
```

- In contrast, `/Insured/CarPolicy[1]/Vehicle` is more restrictive, making available all `Vehicle` nodes that are children of the first `CarPolicy` child within `Insured`. The following expression resolves to an empty set.

```
/Insured/CarPolicy[1]/Vehicle[@Category='Coupe']/Make
```

- `/Insured/CarPolicy[last()]` selects (within `Insured`) the last `CarPolicy` node, which is the node whose `PolicyType` attribute has the string value *Antique*.
- `/Insured/CarPolicy[last()]/Vehicle[last()]` selects the third `Vehicle` node within the second `CarPolicy` node. That `Vehicle` node has details on the *Porsche Speedster*.

Let's talk about the returned nodes whose string values are not available to you. The following expression resolves to a set of two `Make` nodes (one for *Ford*, one for *Buick*).

```
/Insured/CarPolicy/Vehicle[2]/Make
```

Only the first `Make` node seems to be available. The reason is that most XPath processors (such as the ones used with BPEL) provide the string value only of the first returned node.

The following expression returns an empty set.

```
/Insured/CarPolicy/Vehicle[2]/Make[2]
```

The set is empty because each of the two `Vehicle` nodes selected by `/Insured/CarPolicy/Vehicle[2]` has only one `Make` child, and the predicate `[2]` refers not to the whole expression but to the `Make` node.

How do you access only the `Make` node that refers to *Buick*? The solution is to use parentheses, which selects a node set that can be filtered by a predicate. In the next example, the XPath processor makes available only the second node in the node set.

```
(/Insured/CarPolicy/Vehicle[2]/Make)[2]
```

Incidentally, a parenthetical expression is valid only at the beginning of a location path in XPath 1.0. The next expression is valid only if you're working with XPath 2.0.

```
<!-- not a valid expression with XPath 1.0 -->
/Insured/CarPolicy/(Vehicle)
```

Now, let's look again at the effect of the `last()` function. The following expression returns, for each `CarPolicy` node, the `Make` child of the second `Vehicle` element:

```
/Insured/CarPolicy/Vehicle[2]/Make[last()]
```

The value of `last()` is `1`, and although the XPath processor returns two nodes (one for *Ford*, one for *Buick*), the only returned string value is *Ford*.

The following expression returns the last `Make` node of the nodes returned from the parenthesized expression:

```
(/Insured/CarPolicy/Vehicle[2]/Make)[last()]
```

The value of `last()` is `2`, and the related string value is *Buick*.

A predicate operates in the context of the previous location steps and the syntax that precedes the predicate in the same location step. Therefore . . . what is the result of the next expression?

```
/Insured/CarPolicy/Vehicle[@Category='Sport'][2]/Make
```

This expression evaluates to a single node whose string value is *Porsche*. The reason is that the predicate `[2]` is operating on each node set that is created by the syntax `Vehicle[@Category='Sport']`:

- Subordinate to the first `CarPolicy` node is a node set that contains all `Vehicle` nodes whose `Category` attribute equals *Sport*. One node is present, and its `Make` child has the string value *Ford*. When applied to this set, the predicate `[2]` yields an empty set.
- Subordinate to the second `CarPolicy` node is a node set that also contains all `Vehicle` nodes whose `Category` attribute equals *Sport*. Two nodes are present, and their `Make` children have the respective string values *Triumph* and *Porsche*. When applied to this set, the predicate `[2]` yields the second node.

Here's a variation:

```
/Insured/CarPolicy/Vehicle[@Category='Sport'][last()]/Make
```

Each of the branches mentioned earlier now contributes one node that is returned by `last()`. The overall expression yields a node set whose string values are *Ford* and *Porsche*, though only the string *Ford* is immediately available.

Order counts, even within a location step. The next example returns the *Make* node for the second *Vehicle* child within each *CarPolicy* node, with the further restriction that the value of attribute *Category* is *Sport*:

```
/Insured/CarPolicy/Vehicle[2][@Category='Sport']/Make
```

The string value of the single returned node is *Ford*.

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.

Parts of a Location Step

A location step includes, at most, three kinds of details: an axis specification, a node test, and predicates.

An *axis specification* includes a range of nodes for subsequent consideration. The most common axis specification is `child`, whose use means that the location step accesses *children* of the context. Our examples used the `child` specification, but `child` is the default and so was not required in our syntax. If you assign an axis specification explicitly, you must follow it with two colons (for example, `child:.`).

By varying the axis specification

- you can indicate how to select nodes in a given location step; for example, you can include all descendant nodes rather than stepping down the tree one generation at a time
- you can indicate in which direction to seek nodes for selection; for example, you can include nodes that are siblings rather than children or descendants

A *node test* is a criterion to compare against each of the nodes included in an axis specification. That test is either a node name to be matched (as in our prior examples) or a preset value.

Predicates are optional and further restrict the set of nodes that are selected.

Axis Specifications

This section lists the thirteen axis specifications. As you read through the list, note the following distinction. The *forward axes* are the axis specifications that assign context-position numbers to nodes in XML-source order. The *reverse axes* are the axis specifications that assign context-position numbers to nodes in reverse XML-source order.

The XML processor always returns nodes in XML-source order, regardless of whether you use a forward or reverse axis. Only the context-position number that is assigned to each node is affected by the axis direction, as we'll show later.

The axis specifications are as follows:

- `ancestor` is a reverse axis that includes the parent of the context node, the parent of the parents, and so on, up the tree.
- `ancestor-or-self` is a reverse axis that includes the context node from the previous location step, along with (as you'd guess) the parent of the context node, the parent of the parents, and so on, up the tree.
- `attribute` is described later.
- `child` is a forward axis that includes the children of the context node and is the default axis specification.
- `descendant` is a forward axis that includes the children of the context node, the children of those children, and so on, down the tree.
- `descendant-or-self` is a forward axis that includes the context node from the previous location step, along with (right!) the children of the context node, the children of the children, and so on, down the tree.
- `following` is a forward axis that includes all nodes subsequent to the context node, including subsequent siblings but excluding children and other descendants.
- `following-sibling` is a forward axis that includes all the next siblings (the first through the last) of the context node.
- `namespace` is described later.
- `parent` is a reverse axis that includes the parent of the context node; however, the context position is 1 at

most.

- `preceding` is a reverse axis and includes all prior nodes, including prior siblings and text nodes, but excluding parents and other ancestors.
- `preceding-sibling` is a reverse axis and includes all previous siblings (the first through the last) of the context node.
- `self` includes the context node. The axis is considered forward, but the context position is 1 at most.

In general, you can restrict the selection of nodes by using attribute values in predicates. Only two axis specifications, however, actually include attribute nodes or (what is similar) namespace nodes.

The `attribute` axis specification includes attribute nodes. This specification selects nodes only if the context node is an element node. In the XPath model:

- The parent of an attribute node is an element node and is accessible through the `parent` or `ancestor` axis specification. The converse relationship is not true, however: an attribute node is not a child of an element node. You can access an attribute node in two ways: by the attribute axis specification when the context node is an element, and by the `self` axis specification when the context node is an attribute.
- A namespace URI in a declaration such as `xmlns = 'www.ibm.com'` or `xmlns:tns = 'www.ibm.com'` is available through the `namespace` axis specification but not through the `attribute` axis specification.
- The order of attribute nodes within an element node can vary from one XPath processor to another. (In general in XML, attributes are unordered.)

The `namespace` axis specification includes namespace nodes. In XPath 2.0, this rarely used specification is *deprecated* (is made a target for future removal from the language). Therefore, you may want to skim or skip the next paragraph.

The `namespace` specification selects nodes only if the context node is an element node. The string value of a namespace node is a namespace URI. In the XPath model:

- Each element node includes a namespace node that represents the XML system namespace. (The XML system namespace allows use of attributes such as `xml:lang`, which identifies a human language.) The string value of the namespace node is <http://www.w3.org/XML/1998/namespace>.
- Each element node also includes a node for each namespace that is in scope for the XML element. Consider the following XML document:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<other:Insured xmlns="defaultNamespace"
  xmlns:other="otherNamespace">
  <CarPolicy/>
</other:Insured>
```

The related `CarPolicy` node includes namespace nodes whose string values are `defaultNamespace` and `otherNamespace`.

- The parent of a namespace node is an `element` node. The converse relationship is not in effect, however: a namespace node is not a child of an `element` node. You can access a namespace node in two ways: by the `namespace` axis specification when the context node is an element, and by the `self` axis specification when the context node is a namespace node.
- The order of namespace nodes within an element node can vary from one XPath processor to another.

You can categorize axis specifications in a different way:

- Six axis specifications provide access vertically, up and down the node tree: `ancestor`, `ancestor-or-self`, `child`, `descendant`, `descendant-or-self`, and `parent`. In relation to BPEL, you'll primarily use `child` and, on occasion, `descendant`.
- Six axis specifications provide access horizontally, across the node tree: `attribute`, `following`, `following-sibling`, `namespace`, `preceding`, and `preceding-sibling`. In relation to BPEL, you'll probably use only `attribute` and on occasion, `following` and `following-sibling`.

The `attribute` and `namespace` specifications do not concern the relationship of one element node to another

but let you access nodes derived from details that are internal to an XML element.

- The `self` specification stands alone and is used on occasion in BPEL.

Node Tests

As mentioned, a *node test* is a criterion to compare against each of the nodes included in an axis specification. Several preset node tests are available:

- `*`
- `comment()`
- `text()`
- `node()`
- `processing-instruction()`

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.

★

The asterisk (*) node test retains all the element nodes and no others in a given axis specification, except in two instances. In the case of the [attribute](#) specification, the asterisk retains all the attribute nodes and no others. In the case of the [namespace](#) specification, the asterisk retains all the namespace nodes and no others.

The asterisk provides a way to include a node without referencing a name. In relation to our main example, the following expression accesses the second [CarPolicy](#) node.

```
/Insured/*[2]
```


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.

comment()

The `comment()` node test retains comment nodes, which are children and siblings of element nodes or are children of the root node. Consider the following XML document.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- here is an Insured -->
<Insured><CarPolicy><!-- Cancelled --></CarPolicy></Insured>
```

The next expression resolves to the contents of the second comment, including the space that precedes and follows the word *Cancelled*.

```
/descendant::comment()[2]
```


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.

text()

The `text()` node test retains text nodes, which are children and siblings of element nodes or are children of the root node. In relation to our main example, the following expression returns a single text node, and the string value is *Buick*.

```
/Insured/CarPolicy/Vehicle[@Category = 'Coupe']/Make/text()
```

If you wish to access multiple text nodes for subsequent string processing, you can select an element node that has descendant element nodes. For example, assume that the `Material` element includes no line break in the next XML document.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Material>Leather<Product>Bucket Seats</Product>
<Quantity>20</Quantity></Material>
```

The XPath expression `/Material` provides the following string value, which represents the content of the three text nodes that are descendants of the `Material` node.

LeatherBucket Seats20

Also, the string value of the root node (which is expressed by a virgule) resolves to the same text nodes as does the root element node (which is subordinate). Either of the following expressions resolves to the string *Bucket Seats*.

```
/descendant::text()[2]
/Material/descendant::text()[2]
```

Last, you may become perplexed about the location of text nodes that include or are composed of white space (carriage return, spaces, tabs). Consider the following `Material` element, for example.

```
<Material>Leather
  <Product>Bucket Seats</Product>
  <Quantity>20</Quantity>
</Material>
```

Three text nodes are children of that element:

- the string *Leather* followed by the white space that is between the end of *Leather* and the left angle bracket of the `Product` start-tag

- the white space between the right angle bracket that ends the `Product` element and the left angle bracket of the `Quantity` start-tag
- the white space between the right angle bracket that ends the `Quantity` element and the left angle bracket of the `Material` end-tag

You can access any of those text nodes by a positional predicate. Here's a location path that returns white space.

```
/Material/text()[2]
```

XPath does not retain any space that precedes the first element node.

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.

node()

The `node()` node test retrieves all nodes in a given axis specification:

- In the `attribute` axis specification, the test retrieves only attribute nodes.
- In the `namespace` axis specification, the test retrieves only namespace nodes.
- Otherwise, the test retrieves all nodes in the axis specification.

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.

processing-instruction()

The `processing-instruction()` node test retrieves a set of processing-instruction (PI) nodes, which are children and siblings of element nodes or are children of the root node. Here's an XML document with a single processing instruction:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Material>Leather
  <?HandleThis how="somehow" ?>
  <Product>Leather Seats</Product>
  <Quantity>20</Quantity>
</Material><?HandleThat how="another way" ?>
```

The next expression returns the contents of the instruction, including the spaces that follow the phrase *how="somehow"*.

```
/Material/processing-instruction('HandleThis')
```

The node test can identify a processing-instruction node by specifying a PI target, such as *HandleThis*, but the test is also valid without specifying a PI target. The next expression returns two nodes.

```
count(/descendant::processing-instruction())
```

The XML declaration statement (which starts with `<?xml`) is not a processing instruction and is not available to an XPath expression.

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.

Predicates

As noted earlier, a predicate restricts the context nodes based on some criterion, as specified in a Boolean expression. A later section suggests some of the flexibility that is available to you. XPath 1.0 will convert any expression to a Boolean expression, but that behavior may or may not yield the result you intend. We'll give you a hint of the issues now.

A predicate can restrict your selection to a context node that has a specified string value. Either of the following, equivalent location paths, for example, return the second `Make` node from our main example.

```
/descendant::Make[self::node()='Ford']
/descendant::Make[.='Ford']
```

However, if you use a string where a Boolean is expected, the value of the expression is always *true*. Any of the following location paths selects all five `Make` nodes.

```
/Insured/descendant::Make['Ford']
/Insured/descendant::Make['false']
/Insured/descendant::Make['true']
```

In each case, the string value of the first returned node is *Honda*.

You can use an XPath function in a predicate. Let's consider two examples.

The `starts-with()` function returns *true* if the second string argument is at the beginning of the first. Each of the following expressions returns the one `Model` node whose string value is *Skylark*.

```
/descendant::Model[starts-with(self::node(),'Sky')]
/descendant::Model[starts-with(.,'Sky')]
```

The `contains()` function returns *true* if the second string argument is within the first. Each of the following expressions also returns the one `Model` node whose string value is *Skylark*.

```
/descendant::Model[contains(self::node(),'lark')]
/descendant::Model[contains(.,'lark')]
/descendant::Model[contains(.,'Sky')]
```

Last, you can use a predicate to test for the presence of an attribute or of an immediately subordinate element. Here's an expression that returns *5*, which is the number of `Vehicle` nodes that are parents of a `Make` node.

```
count(/descendant::Vehicle[Make])
```

The next expression returns 2, which is the number of `Vehicle` nodes that include the `Domestic` attribute.

```
count(/descendant::Vehicle[@Domestic])
```

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.

Abbreviations

The XPath 1.0 location-path abbreviations are as follows:

- You can omit `child::`.
- A short form is available for a predicate that contains only a position value with an equal sign. The predicate `[2]` is a short form of `[position()=2]`. No short form is available for a predicate such as `[position() < 2]`, which has an operator other than an equal sign. Also, if you use a Boolean operator, you must use the long form for the position value. The predicate `[position()=1 or position()=5]` is not expressed as `[1 or 5]`, for example.
- The at sign is used in front of an attribute name in a predicate but also functions as a short form of `attribute::`. In our main example, the following location path selects the string value `5`.

```
/Insured/@CustomerID
```

- The double virgule (`//`) lets you step through many levels of the tree, with the children of the context node potentially included as one of the selected nodes. The formal definition is `/descendant-or-self::node()/.`

You can use the double virgule at the beginning of a location path or in the middle. Each of the next expressions includes all `Vehicle` nodes that are the first children of their parents.

```
//Vehicle[1]
/Insured//Vehicle[1]
```

In our case, each expression makes available the nodes for *Honda Accord* and *Triumph Spitfire*, displaying only the string value *Honda Accord*.

If your XML source includes hundreds of lines and if you use the double virgule at the start of a location path rather than in the middle, the processing time may be too long for your purpose.

The following example (not an abbreviation) may seem similar to `//Vehicle[1]`, but selects only the first `Vehicle` node in the document.

```
/descendant::Vehicle[1]
```

The displayed string value is *Honda Accord*.

- A single period (`.`) is a short form for the context node; specifically, the period is a short form of `self::node()`, as noted earlier.

```
/descendant::Make[.='Ford']
```

- A double period (`..`) is a short form of `parent::node()`, which is the parent of the context node. The following example returns the five `Vehicle` nodes, starting with the one for *Honda Accord*.

```
/descendant::Make/..
```

Examples with Descendants and Siblings

Some axis specifications cause behavior that you might not expect. Let's try examples with the `descendant` specification (which is straightforward) and with the `following-sibling` specification (which is less so).

The string value of the next expression refers to the third of the `Insured` node's `Vehicle` descendants whose `Category` value is *Sport*.

```
/Insured/descendant::Vehicle[@Category='Sport'][3]/Make
```

The string value is *Porsche*.

The string value of the following expression also is *Porsche*, which refers to the fifth *Vehicle* descendant of the *Insured* node.

```
/Insured/descendant::Vehicle[last()]/Make
```

The following expression selects two nodes — specifically, the second *Vehicle* descendant of each *CarPolicy* node. The displayed string value is *Ford*, from the first of the two selected nodes.

```
/Insured/CarPolicy/descendant::Vehicle[last()]/Make
```

We'll now introduce the union (`|`) operator, which selects nodes based on a prior and subsequent location path. The following expression retrieves one *Vehicle* node whose *Category* attribute value is *Sedan* and two whose *Category* attribute is *Sport*.

```
/descendant::CarPolicy[1]/Vehicle[@Category='Sedan']/Make | /descendant::CarPolicy[2]/Vehicle[@Category='Sport']/Make
```

The string value of the first retrieved node is *Honda*.

When you work with child or descendant elements, the position value `[2]` is the second child or descendant; but when you work with siblings, the position value `[2]` refers to the second sibling (forward or backward, depending on the axis specification). In the following expression, five *Vehicle* nodes are used at different times as the context node for the second-to-last location step (`following-sibling::Vehicle`), which selects three unique nodes.

```
/Insured/CarPolicy/Vehicle/following-sibling::Vehicle/Make
```

The string values of the selected *Make* nodes are as follows:

- *Ford* is the only string value that is immediately available. That value is present because the expression selected all subsequent siblings of the first *Vehicle* child of the first *CarPolicy* node.
- *Buick* is available if you use parentheses and the predicate `[2]`, as in previous examples. That value is present because, in relation to the second *CarPolicy* node, the expression selected all subsequent siblings of the first *Vehicle* child.
- *Porsche* is available if you use parentheses and the predicate `[3]`. That value is present because, in relation to the second *CarPolicy* node, the expression selected all subsequent siblings of the first and second *Vehicle* nodes and then removed the duplicate node that has string value *Porsche*. XPath 1.0 expressions never provide a duplicate node, although the option to retrieve duplicate nodes is available in XPath 2.0.

The following expression returns only one node because in only one case is a sibling a *second* sibling, two nodes away from a context node.

```
/Insured/CarPolicy/Vehicle/following-sibling::Vehicle[2]
```

The expression resolves to the string value *Porsche Speedster*, and the string value of the relevant context node is *Triumph Spitfire*.

Consider a variation:

```
/Insured/descendant::Vehicle/following-sibling::Vehicle[2]
```

The effect is precisely the same as that of the previous expression. The sibling relationships of the *Vehicle* nodes are not dependent on the *CarPolicy* context node. The node for *Ford Mustang* is never a sibling of the node for *Triumph Spitfire*, for example, no matter the location path.

Examples with a Reverse Axis

Let's explore the effect of using a reverse axis. As we said earlier, the meaning of reverse axis is that the context-position numbers are assigned in reverse XML-source order.

In the following case, the XPath processor returns the first-level ancestor node of each *Make* node.

```
/descendant::Make/ancestor::*[1]
```

The XPath processor returns five *Vehicle* nodes, each of which is the first ancestor of a *Make* node. The ancestor nodes are returned in XML-source order, not in reverse order. The string value of the first returned node is *Honda Accord*.

In the following case, the XPath processor returns the third-level ancestor node of each *Make* node.

```
/descendant::Make/ancestor::*[3]
```

The third-level ancestor node of every *Make* node is the *Insured* node. The string value is the concatenation of all text nodes in the XML source and contains the make and model of all five cars.

Last, notice the difference between our first ancestor example and the following one, which includes parentheses.

```
(/descendant::Make/ancestor::*)[3]
```

The parenthetical expression returns eight nodes (five *Vehicle* nodes, two *CarPolicy* nodes, and the *Insured* node). The predicate selects the third of those nodes in XML source order; in other words, it selects the first *Vehicle* node, whose string value is *Honda Accord*.

Location Steps in Summary

Now we can give you a big story in a few words. For a given location step, the XPath processor handles one context node at a time and fulfills four steps in relation to each context node:

1. Selects all nodes in the axis specification.
2. Retains each node that passes the node test.
3. Assigns a number to each retained node in a way that reflects the XML-source order (for forward axes) or that reflects the opposite (for reverse axes).
4. Processes each predicate in turn, in a loop that acts as follows:
 - a. Reviews the Boolean expression.
 As described earlier, the predicate can include one or more values to be tested against `position()` and may include a value returned from `last()`. The XPath processor compares the test values against the numbers assigned in step 3. The value returned from `last()` is the highest number that was assigned in step 3.
 - b. Retains all nodes for which the Boolean value in the predicate is *true* and removes the rest.
 - c. Reassigns a number to each node, as needed to respond to the removal of nodes during step 4b.

XPath 1.0 always removes duplicate nodes, and XPath (whether 1.0 or 2.0) returns the nodes in XML-source order.

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.

Other Aspects of XPath 1.0

We've described location paths at length and now sprint through aspects of XPath 1.0 that are more like other languages. For additional details, refer to descriptions in your product documentation, which should include product-specific extensions, if any.

Expressions

XPath 1.0 expressions resolve to

- Boolean values that are directly available by invoking the functions `true()` and `false()`
- strings
- numbers (double-precision, 64-bit floating point)
- node sets

In relation to Booleans, strings, and numbers, the XPath 1.0 processor automatically converts from one type to another in accordance with the XPath functions `string()`, `number()`, and `Boolean()`:

- When a comparison of one operand to another is based on the equal sign or the not-equal sign (`!=`), data-type conversions occur as follows:
 - A comparison that includes a Boolean is a Boolean comparison.
 - A non-Boolean comparison that includes a number is a numeric comparison.
 - A comparison that lacks Booleans and numbers is a string comparison.
- When a comparison of one operand to another uses `<=`, `<`, `>=`, or `>`, the comparisons are numeric.
- A zero is equivalent to a Boolean *false*, and any non-zero number is equivalent to a Boolean *true*.
- A string with any value is equivalent to a Boolean *true*, and a string with no characters is equivalent to a Boolean *false*.
- The characters *NaN* mean *not a number* and may appear in a situation where you've tried to convert a string such as *Hello!* to a number.

In relation to node sets, the following rules are in effect:

- A comparison of two node sets evaluates to *true* if the string value of any node in the first node set is the same as a string value of any node in the second. The following example resolves to *true* because each of the two node sets includes a node whose string value is *Sport*:

```
/descendant::CarPolicy[1]/Vehicle/@Category =
  /descendant::CarPolicy[2]/Vehicle/@Category
```

One implication is that an equality comparison of one node to another is a comparison of string values and is never a comparison to determine whether the nodes are the same.

- A comparison of a node set to a number evaluates to *true* if the string value of a node in the node set can be converted to that number, as in the next example.

```
/Insured/@CustomerID = 5
```

- A comparison of a node set to a string evaluates to *true* if the string value of a node in that node set is equivalent to the string, as in the next example.

```
/Insured/@CustomerID = '5'
```

- A node set evaluates to *true* in a Boolean comparison if the node set includes at least one node. The following evaluates to *true* because the XML source includes at least one *Make* element whose string value is *Porsche* and at least one *Model* element whose string value is *Mustang*.

```
/descendant::Make[.='Porsche'] and /descendant::Model[.='Mustang']
```

As mentioned earlier, a predicate can accept any Boolean expression. The following expression returns any *Vehicle* node whose *Make* child has the string value *Ford* and whose *Model* child has the string value *Mustang*.

```
/descendant::Vehicle[child::Make = 'Ford'][child::Model = 'Mustang']
```

That expression returns the second *Vehicle* node, as does the next expression, which is the same but omits references to the *child* axis specification.

```
/descendant::Vehicle[Make = 'Ford'][Model = 'Mustang']
```

Last, we want to shed light on a confusing aspect of Boolean logic in XPath. Our focus is twofold: the effect of the not-equal operator when it is used to compare a node set and a string, and the effect of the `not()` function, which returns the opposite of the Boolean value that is passed to it.

We offer four examples:

- The first expression evaluates to *true* if one or more *Make* nodes has the string value *Buick*.

```
/descendant::Make = 'Buick'
```

- The second expression evaluates to *true* if no *Make* node has the string value *Buick*.

```
not (/descendant::Make = 'Buick')
```

That is, the example evaluates to *false* if any *Make* node has the string value *Buick*.

- The third expression evaluates to *true* if one or more *Make* nodes has a string value that is not *Buick*:

```
/descendant::Make != 'Buick'
```

That is, the example evaluates to *false* only if every *Make* node has the string value *Buick*.

- The fourth expression evaluates to *true* only if every *Make* node has the string value *Buick*.

```
not(/descendant::Make != 'Buick')
```

The subtle difference in meaning between the second and third expressions causes a big difference between the first and fourth.

Consider the first and third expressions again and ask yourself, "What do the following Boolean expressions have in common?"

```
/descendant::Make[position() < 3] =
  /descendant::Make[position() < 3]

/descendant::Make[position() < 3] !=
  /descendant::Make[position() < 3]
```

Both Boolean expressions evaluate to *true*. The first expression evaluates to *true* because at least one node in the first node set has the same string value as at least one node in the second node set. The second expression evaluates to *true* because at least one node in the first node set has a string value that is different from the string value of at least one node in the second node set.

Aren't you glad you asked?

Numeric and Boolean Operators

Table 6.1 lists the numeric and Boolean XPath operators in order of decreasing precedence. The operators in a given cell are processed in left-to-right order in a given expression.

Table 6.1. XPath numeric and Boolean operators

Operator	Meaning
*	Multiply
div	Divide
mod	Use modular arithmetic, where the output is negative only if the dividend is negative: <ul style="list-style-type: none"> • 7 mod 3 yields 1 • 7 mod 3 yields 1 • -7 mod -3 yields -1 • -7 mod -3 yields -1
+	Plus
-	Minus
<=	Less than or equal to
<	Less than
>=	Greater than or equal to
>	Greater than
=	Equal
!=	Not equal
and	Boolean and

Operator	Meaning
or	Boolean or

Functions

The next sections give a brief overview of most XPath 1.0 functions. For further details, see your product documentation.

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.

Returns a Boolean

Each function in [Table 6.2](#) returns a Boolean.

Table 6.2. XPath 1.0 functions that return a Boolean value

Function	Meaning
contains(string, string)	<p>Indicates whether the first argument includes a string equal to the second argument. Given our main example, each the following expressions resolves to <i>true</i> because the string <i>lark</i> is within the string <i>Skylark</i>.</p> <pre>contains('Skylark','lark') contains(descendant::Model[4],'lark') contains(descendant::Model[position() >= 4],'lark')</pre> <p>The first argument in the third expression returns the nodes whose string values are <i>Skylark</i> and <i>Speedster</i>; however, only the first node is available, and the only tested string value is <i>Skylark</i>.</p> <p>The next expression resolves to <i>false</i> because string operations are case-sensitive.</p> <pre>contains('Skylark','LARK')</pre>
false()	Returns <i>false</i>
not(Boolean)	Returns the opposite value to the value of the argument.
starts-with (string, string)	<p>Indicates whether the first argument starts with a string equal to the second argument. Given our main example, each the following expressions resolves to <i>true</i> because the string <i>Sky</i> is at the beginning of the string <i>Skylark</i>.</p> <pre>starts-with('Skylark','Sky') starts-with(descendant::Model[4],'Sky') starts-with(descendant::Model[position() >= 4],'Sky')</pre> <p>The first argument in the third expression returns the nodes whose string values are <i>Skylark</i> and <i>Speedster</i>; however, only the first node is available, and the only tested string value is <i>Skylark</i></p> <p>The next expression resolves to <i>false</i> because string operations are case-sensitive.</p> <pre>starts-with('Skylark','SKY')</pre>
true()	Returns <i>true</i> .

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.

Returns a String

Each function in [Table 6.3](#) returns a string.

Table 6.3. XPath 1.0 functions that return a string

Function	Meaning
<code>concat(string, string, . . .)</code>	<p>Concatenates any number of string arguments. The following expression resolves to <i>Buick Skylark</i>.</p> <pre>concat('Buick ', 'Sky', 'lark')</pre>
<code>name(node set)</code>	<p>Returns the name of the first node in the node set. Given our main example, the following expression resolves to <i>CarPolicy</i>, which is the name of the first retrieved node</p> <pre>name(/Insured/*)</pre>
<code>normalize-space(string)</code>	<p>Removes leading and trailing spaces and removes all but one space when multiple white-space characters (carriage returns, spaces, tabs) are between other characters. The following expression resolves to <i>Buick Skylark</i></p> <pre>normalize-space(' Buick Skylark ')</pre>
<code>substring(string, number, number)</code>	<p>Returns a substring of the first argument, starting at a specified position (the first number). The substring continues for a specified number of characters (the second number) or (if the second number is omitted) returns the rest of the string.</p> <p>Each of the following expressions resolves to <i>Speed</i></p> <pre>substring('Porsche Speedster', 9, 5) substring('Porsche Speed', 9)</pre>
<code>substring-after(string, string)</code>	<p>Returns a substring of the first argument, starting at one position after the first occurrence of the second string.</p> <p>The following expression resolves to a space, then <i>Speedster</i>.</p> <pre>substring-after('Porsche Speedster', 'e')</pre>
<code>substring-before(string, string)</code>	<p>Returns a substring of the first argument, starting at the beginning of the first argument and ending at one position before the first occurrence of the second string.</p> <p>The following expression resolves to <i>Porsche</i>, then a space.</p> <pre>substring-before('Porsche Speedster', 'S')</pre>
<code>translate(string, string, string)</code>	<p>Returns a variant of the first argument:</p> <ul style="list-style-type: none"> • Changes any character that matches a character listed in the second argument • Substitutes a character listed in the third argument. If no substitution value is listed there, the character is not returned from the function <p>The second and third arguments are a matching array of characters. If the third argument is longer than the second the extra positions have no effect. If the third</p>

Function	Meaning	
	<p>argument is shorter than the second, the missing position removes a character from the returned string.</p> <p>The second example in the next table shows a character removal.</p>	
	Invocation	Returns
	<code>translate('98786', '8', 'X')</code>	<code>9X7X6</code>
	<code>translate('98786', '87', 'X')</code>	<code>9XX6</code>
	<code>translate('98786', '87', 'XPA')</code>	<code>9XPX6</code>

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.

Returns a Number

Each function in [Table 6.4](#) returns a number.

Table 6.4. XPath 1.0 functions that return a number

Function	Meaning	
ceiling(number)	Returns the value of a numeric expression, rounded upward to the greater integer. Here are examples:	
	Invocation	Returns
	<code>ceiling(2)</code>	2
	<code>ceiling(2.01)</code>	3
	<code>ceiling(-2)</code>	2
	<code>ceiling(-2.01)</code>	-2
	<code>ceiling(-2.99)</code>	-2
count(node set)	Returns the number of nodes in a node set. Given our main example, the following expression resolves to 5. <code>count(/descendant::Vehicle[Make])</code>	
floor(number)	Returns the value of a numeric expression, rounded downward to the lesser integer. Here are examples:	
	Invocation	Returns
	<code>floor(2)</code>	2
	<code>floor(2.99)</code>	2
	<code>floor(-2)</code>	-2
	<code>floor(-2.01)</code>	-3
	<code>floor(-2.99)</code>	-3
last()	Returns the context size, as described earlier. In our main example, the value of last() in the following expression is 2. <code>(/Insured/CarPolicy/Vehicle[2]/Make)[last()]</code>	
round(number)	Returns the value of a numeric expression, rounded to the nearest integer. The next examples show the effect of mid points.	
	Invocation	Returns
	<code>round(2.499)</code>	2
	<code>round(2.5)</code>	3

Function	Meaning
	<code>round(-2.5)</code> -2
	<code>round(-2.501)</code> -3
string-length(string)	<p>Returns the number of characters in the argument. Given our example, the following invocation returns 5 (the number of letters in <i>Honda</i>).</p> <pre>string-length(/descendant::Make[1])</pre>
sum(node set)	<p>Returns the sum of the numeric values in each node in a node set. If the node includes non-numeric text, the returned value is <i>NaN</i> (not a number).</p> <p>Consider the following XML source</p> <pre><?xml version="1.0"?> <Options> <Towing>50</Towing> <Rental>20</Rental> </Options></pre> <p>The following expression resolves to 70.</p> <pre>sum(/Options/*)</pre>