CHAPTER 3

Navigation

3.1 Introduction

Once you've constructed or loaded XML in a query, you need a way to *navigate* over that hierarchical data. In many ways, construction and navigation are the primary operations in any XML query language. XQuery provides a litany of navigation expressions, and this chapter explores them all. Readers who are already familiar with XPath 1.0 may safely skim this chapter. XQuery has some differences from XPath 1.0, but they are minor.

Navigation involves starting from one part of an XML data model and moving to another part of the data model. Navigation can involve local steps, for example, moving from a node to one of its neighbors, or global steps, such as moving from a node to a completely different part of the data model, or even another document.

If you're familiar with relational databases, it may help to reflect that navigating is to XML nodes what cursoring is to relational rowsets. Like using regular expressions to parse strings, using navigation in a query is generally more efficient in space and time than manually traversing an XML structure.

3.2 Paths

Navigation involves starting from one part of an XML document and moving to another part of the document (or a different document). XQuery performs navigation using *paths*. Paths were invented in 1970 for use with the PDP-11 file system. The path concept has been so generally useful that it has found broad application in a variety of systems, including XML query processing.

In XQuery, every path consists of a sequence of *steps* which, conceptually at least, are executed in order from left to right. A step consists of three parts, illustrated in Figure 3.1:

03Brundage_ch03.qxd 1/8/04 1:09 PM Page 64

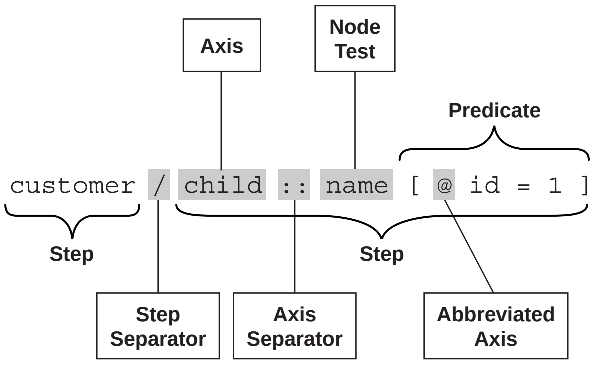

Figure 3.1 Anatomy of a path

- A direction of travel, called the *axis*
- A description of the nodes to select upon arrival, called the *node test*
- Zero or more filters to further narrow that selection (each filter is called a *predicate*)

By allowing some of these parts to be abbreviated or omitted entirely, XQuery keeps paths very concise. Each of these parts is described next, and then Section 3.5 has many examples demonstrating how to use paths to accomplish common tasks.

Each step affects the *evaluation context* for the next step. This context and how it changes with each step are described in Section 3.4, but for now it's enough to know that there is a *current context item* that affects—and is affected by—each step in the path. Except for predicates, navigation steps can be applied only when the current context item is a node (in which case it is often called the *current context node*).

3.2.1 Beginnings

Every path starts somewhere. For the purpose of XQuery navigation, there are effectively three places from which a path can begin:

- The current context node
- The root of the tree in which the current context node resides
- Any other node set, such as a variable or an XML constructor

With each successive step, the path may move to other nodes or alter the context.

The root of the tree in which the current context node resides is selected by a lone forward slash (`/`) or equivalently using the built-in `root()` function. Paths beginning from the root are *absolute*. In contrast, paths starting from the current context node are *relative*. Paths may also start from certain other expressions, such as variables, function calls, or parenthesized expressions (XQuery does not give a name to such paths).

From these humble beginnings, paths may navigate anywhere in the document, or even to other documents, step by step. Listing 3.1 shows a few paths. In a path, individual steps are almost always separated by one forward slash (`/`). (The exception, two forward slashes (`//`), is described in Section 3.2.4.)

Listing 3.1 Absolute, relative, and other paths

```
/AbsolutePath/First/Second
RelativePath[. = "fun"]
$other//x
id("other")[@y > 1]/z
```

Paths with more than one step always result in a (possibly empty) sequence of nodes, sorted in document order. To sort nodes in some other order, you must use a FLWOR expression (see Chapter 6).

3.2.2 Axes

Each step consists of three parts: the axis (optional), the node test, and zero or more predicates. XPath defines a total of thirteen axes, and all but the `namespace` axis appear in XQuery. Of these, the four simplest and most commonly used ones are `child`, `attribute`, `parent`, and `self` (see Table 3.1). The other axes are explained in Section 3.2.4.

Table 3.1 The four basic axes and their abbreviations

Axis name	Abbreviation	Equivalent examples	
`attribute`	`@`	`x/attribute::y`	`x/@y`
`child`		`x/child::y`	`x/y`
`parent`	`..`	`x/parent::node()`	`x/..`
`self`	`.`	`x/self::node()`	`x/.`

The child axis is so common that it is the default axis if no axis name is specified explicitly. The other three common axes all have shorthand abbreviations for convenience. XPath gets much of its succinctness from these shorthand forms. When the non-abbreviated name is used, it is followed by two colons (`::`) to distinguish axis names from XML qualified names (which contain at most one colon).

These four axes behave exactly as their names suggest:

- The `child` axis navigates into the children of the current context node.
- The `attribute` axis navigates into the attributes of the current context node.
- The `self` axis essentially goes nowhere (navigating into the current context node itself).
- The `parent` axis navigates to the parent of the current context node.

For example, `x`, which is short for `child::x`, selects the child elements named `x` from the current context node, while `x/y`, which is short for `child::x/child::y`, first selects the child elements named `x` from the current context node just like the previous example, and then from those selects the child elements named `y`.

3.2.3 Node Tests

Following the axis is the second part of the step, the *node test*. Node tests come in three varieties: names (qualified or unqualified), node kinds, and wildcards.

3.2.3.1 Name Tests

By far the most common node test is the *name test*. A name test selects only those nodes with the same name. Names in XQuery, as in XML, are case-sensitive. For example, the absolute path `/x/y/@z` starts at the root of the current document, navigates to the top-level elements named `x`, navigates to their child elements named `y`, and finally navigates to their attribute nodes named `z`. If you were to execute this XQuery over the XML document in Listing 3.2, it would select the two attributes named `z` and no other nodes.

Name tests can also select names that are in an XML namespace. However, this process is fairly complicated, so this description is deferred until Section 3.6.1.

Listing 3.2 A sample XML document

```
<x thisAttribute="isNotSelected">
  <y z="1"/>
  <y z="2" thisAttribute="alsoIsNotSelected" </y>
</x>
```

3.2.3.2 Node Kind Tests

Name tests are not the only node tests available in navigation steps. In fact, some kinds of XML nodes (for example, text, comment, and document nodes) have no names at all. To select nodes by kind, XQuery uses the same *node kind tests* used by sequence type matching (described in Chapter 2). Listing 3.3 shows two node kind tests.

Listing 3.3 Examples of node kind tests

```
x/comment()                 (: select all comment children of x :)
x/attribute()               (: select all attributes of x :)
attribute(@*, xs:integer)   (: select all integer attributes :)
attribute(y)                (: select all attributes named y :)
attribute(y, xs:integer)    (: select integer attributes named y :)
```

Recall from Chapter 2 that the `node()` node test matches any kind of node, including the document node. The `text()` and `comment()` node kind tests match text nodes and comment nodes, respectively. The `processing-instruction()` node test accepts an optional name argument. When no name is specified, it matches all processing instruction nodes; otherwise, it matches only those with the same name.

The `document-node()` test matches the invisible document node that occurs at the root of any tree loaded from an XML document using `doc()` (or constructed using the document constructor—see Chapter 7). It accepts an optional argument specifying an element node kind test, in which case it matches the document node only if its element content matches that element test.

And finally, the `element()` and `attribute()` node kind tests accept optional name and type arguments. Without these extra arguments, they match all elements and attributes, respectively; with these arguments, they match only elements or attributes that have the specified name and/or type. The name or type can also be `*`, in which case it matches all names or all types, respectively. The name specified in an `attribute()` test must start with an `@` symbol to emphasize that it matches attributes.

3.2.3.3 Wildcards

Sometimes you want to select all nodes whose name is in a particular namespace, or conversely all nodes with the same local name regardless of the namespace. There are two equivalent ways to accomplish this goal. One is to use predicates; in fact, as you will see later, predicates can be used to perform all kinds of tests.

A more succinct way is to use the third kind of node test, the *wildcard*. Wildcard node tests combine aspects of both name and node kind tests; the names matched depend on the wildcard, and the node kind matched depends on the axis. The attribute axis by default selects attribute nodes; all other XQuery axes select elements by default. The default node kind is called the *principal node kind* for the axis.

XQuery supports three wildcard node tests. Two of these come from XPath 1.0: the star (`*`), which matches any name at all, and a qualified star (`prefix:*`) that matches all names in the namespace to which the prefix is bound. XQuery adds a third wildcard node test, `*:local-name`, which matches all names with the given local name and any namespace.

The only difference between the star wildcard `*` and the `node()` node kind test is that `node()` matches every kind of node with any name, while `*` matches only nodes of the principal node kind (with any name).

3.2.4 Other Axes

XQuery supports two more axes from XPath 1.0, called `descendant` and `descendant-or-self`. The `descendant` axis matches all descendants of the current context node. (It is the closure of the child axis under fixed-point recursion.) The `descendant-or-self` axis includes the current context node as well, and so is equivalent to the union of the descendant and self axes.

The `descendant-or-self` axis is so commonly used that it has its own abbreviation, `//`. Some caution should be observed when using it; it's easy to make mistakes when using predicates with `//` (see Chapter 11 for examples).

Additionally, implementations are allowed but not required to support the other six axes from XPath: `ancestor`, `ancestor-or-self`, `following`, `following-sibling`, `preceding`, and `preceding-sibling`. The first two of these are the inverses of descendant and descendant-or-self axes. They select all the ancestors of the current node (ancestor-or-self includes the node itself).

The following and preceding axes select all the nodes in the same document as the current context node that occur before and after it, respectively. There's really no reason to use them in XQuery, because the >> and << node comparison operators allow you to write the same meaning more compactly (see Chapter 5).

Finally, the following-sibling and preceding-sibling axes restrict their selections to the siblings of the current context node (that is, those nodes having the same parent as it).

3.2.5 Predicates

The third and final part of each navigation step consists of zero or more *predicates*. Like the node test, each predicate acts as a filter on the selected nodes, eliminating some from consideration and keeping the rest. For each node selected by the current step, the current context item is set to that node and then the predicate condition is evaluated with that context.

Any XQuery expression may be used inside a predicate; the meaning of the predicate depends on the type of the expression it contains. There are two cases: numeric and boolean predicates.

3.2.5.1 Numeric Predicates

Numeric predicates select nodes by their position in the current context. For example, /x/y[1] selects the first y child element of each x element. As this example demonstrates, predicates bind tightly to the current step. To apply a predicate to the entire results of a path, you must use parentheses. For example, (/x/y)[1] selects the first y element out of all the nodes selected by /x/y.

Because paths can start with other kinds of expressions, such as parenthesized expressions, predicates can be applied to more than just sequences of nodes. For example, the expression ("a", "b", "c")[2] selects the second item in the sequence, the string "b".

Numeric predicates, like the ones in Listing 3.4, filter by position. In general, when a predicate evaluates to a number N, it's as if the predicate were actually the boolean-valued predicate position()=N. For example, the path /x[1] is equivalent to the path /x[position() = 1]. This expansion applies not only to numeric constants, but also to any numeric-typed expression. For example, the path `/x[@y + 1]` is equivalent to the path `/x[position() = @y + 1]`.

Listing 3.4 Numeric predicates filter by position

```
(//Customer)[2]
Fruit[@index + 1]
```

The position is 1-based (the first item in the sequence is at position 1). When the predicate evaluates to a non-integral value, a value less than 1, or a value greater than the length of the sequence, then the predicate will be false for all items in the sequence and the result will be the empty sequence. In other words, it isn't an error to select an index that is out of bounds for the sequence.

3.2.5.2 Boolean Predicates

All other kinds of predicate expressions, such as the ones in Listing 3.5, filter a sequence so that only those items for which the predicate evaluates to true are kept. The predicate is converted to a boolean value by computing the Effective Boolean Value of the expression.

Listing 3.5 All other predicates filter as boolean conditions

```
/x[@a=1 and @b=1]
/x[@a=1]/y[@b < 2]
```

As described in Section 2.6.2, the Effective Boolean Value acts as an existence test on sequences. Consequently, when the predicate is itself a path, the predicate evaluates to true if and only if the node(s) selected by that path exist. For example, `x[y]` matches all `x` elements that have a `y` child element, and `x[not(@y)]` matches all `x` elements that don't have a `y` attribute.

3.2.5.3 Successive and Nested Predicates

Several predicates can be applied to a step, with the effect that each predicate is evaluated with respect to the nodes remaining after the previous predicate.

The order of evaluation of the predicates is always left to right, which matters only when computing positional predicates. For example, the path `x[1][@y=2]` selects the first `x` element (if there is one), and then only if that element has a `y` attribute whose value is 2; while the path `x[@y=2][1]` selects all `x` elements that have a `y` attribute whose value is 2, and then from that set selects the first one. Over the XML `<x y="3"/><x y="2"/>` the first path selects nothing (because the first `x` element has `y="3"`), while the second path selects the second element.

Predicates can also be nested. For example, the path `x[y[@z=1] = 2]` selects all `x` elements where there exists a `y` element with a `z` attribute equal to 1 and the value of the `y` element itself equals 2.

3.3 Navigation Functions

All of the navigation we've considered so far amounts to local steps: from the current context, navigate to some nearby nodes. However, XQuery also defines functions that can navigate more globally to other parts of a document or different documents. These functions are summarized in Table 3.2, and fully documented in Appendix C.

Of these, the `doc()` function is the only one you are likely to commonly use. It takes a single string argument, which is treated as the URI location of an XML document. It then loads that document and returns the correspon-

Table 3.2 Navigation functions

Function	Meaning	XPath 1.0?
`collection()`	A named sequence	No
`doc()`	Navigate to the root of the named XML document	No
`id()`	Navigate to the (unique) element with this ID	Yes
`idref()`	Navigate to the elements that refer to this one	No
`root()`	Navigate to the root of the current document	No

ding XQuery Data Model instance. Certain aspects of this process, such as security permissions and schema validation, vary from one implementation to the next.

If the document cannot be found or is not well-formed, some implementations will raise an error, although they are also allowed to just return the empty sequence. (This is mainly to allow certain XQuery optimizations; for example, `doc("x")[false()]` could be optimized into the empty sequence without attempting to load the document.)

The other functions are much less commonly used, so we defer their description to Appendix C.

3.4 Navigation Context

Every XQuery expression is evaluated within a *context*, and several of the context effects have a bearing on navigation. The context can vary during the evaluation of a path, and some context information can be accessed using functions or other expressions. XPath 1.0 defines six expression context information items, and XQuery adds nine more, all listed in Table 3.3.

Expression context is divided into *static context*, which is available during the compilation of the query, and *evaluation context*, which is available while the expression is being evaluated dynamically. Some context information is global to an entire query, while other context information is local and may vary during compilation or evaluation.

3.4.1 Input Sequence

One value in the evaluation context is the *input sequence*. This sequence can be accessed using the `input()` function, and doesn't change during the execution of a query. This value defines the initial context sequence (for example, used by relative paths at the top of the query) and may be empty.

3.4.2 Focus

Part of the XQuery evaluation context is called the *focus*. Predicates and navigation steps change the focus. This focus consists of three items: the *context item*, the *context position*, and the *context size*.

Table 3.3 XQuery expression context

Context item	Accessed with	Static or dynamic	XPath 1.0?
in-scope namespaces	`get--in-scope-prefixes()`	static	Yes
default element namespace	N/A	static	No
default function namespace	N/A	static	No
in-scope schema definitions	N/A	static	No
in-scope functions	N/A	static	Yes
in-scope collations	N/A	static	No
default collation	`default-collation()`	static	No
base-uri	`base-uri()`	static	No
in-scope variables	`$variable`	both	Yes
context item	`.`	dynamic	Yes
context position	`position()`	dynamic	Yes
context size	`last()`	dynamic	Yes
current date and time	`current-date()` `current-time()` `current-dateTime()`	dynamic	No
implicit timezone	`implicit-timezone()`	dynamic	No
input sequence	`N/A`	dynamic	No

When evaluating a path, the focus changes with each step and predicate. For example, when evaluating the path `x[@y=1]/z`, the step `x` selects a sequence of nodes, which defines the focus for the predicate. The context size is the number of nodes selected by `x`, and then for each node in that sequence, the predicate is evaluated. The node becomes the current context item, and the context position is its position within that sequence. If the predicate evaluates to true, then the node is kept in the result, otherwise it is omitted. The result of this step becomes the focus for the next step `z`.

The current context item can be accessed using the dot (`.`) expression, and in fact `x[@y=1]` is short for `x[./@y = 1]`. Every relative path in a predicate begins at the current context item.

The current context position can be accessed using the function `position()`. For example, when evaluating the path `x[position() > 3]`, the predicate eliminates the first three items in the sequence selected by `x`.

Finally, the context size can be accessed using the function `last()`. For example, the path `x[last()]` selects the last child element named `x`. The efficiency of the `last()` function depends on the implementation. In cases where you are streaming through an XML input, `last()` always requires at least a little buffering to evaluate, and can require a lot of buffering. For example, `x[count(y) < last()]` must first count the number of `x` child elements, and then iterate through each of them testing the condition. (Implementations that preload the XML into memory or a database are less affected by this consideration, because they may already have the sequence length available.)

3.4.3 Variable Declarations

XQuery can also declare and use *variables*. Certain expressions, such as FLWOR and `typeswitch`, introduce new variables into scope. Some implementations also allow externally defined variables to be passed to an XQuery. You'll see examples of both of these later.

There are two aspects to variable context. In the static context are all the *variable declarations*, that is, the names and static types of the variables that are available to the XQuery expression. The evaluation context also contains this information, along with the variable values (and their dynamic types), called the *variable bindings*.

Variables are accessed by name using an expression such as `$variable`. Attempting to use a variable that isn't in the static context (that is, not in scope) causes a compile-time error.

3.4.4 Namespace Declarations

The static context also includes *namespace declarations*, which may be defined in the query prolog or in element constructors. The namespace declarations are just a set of prefix and namespace pairs that allow prefixes to be used to stand in for the namespace names. XQuery allows for two different kinds of default namespaces, one for resolving element and type names, and the other for resolving function names (see Chapter 5 for additional details about the query prolog).

3.4.5 Function Declarations

The static context also includes all functions available to the query. These include the built-in XQuery functions, as well as user-defined functions (see Chapter 4) and possibly other extension functions provided by the implementation (see Chapter 14).

XSLT 1.0 provides a `function-available()` function for determining whether a function is in the static context, but XQuery doesn't have an equivalent.

3.4.6 Collations

Collations are used for string comparisons and sorting; the default collation and possibly other in-scope collations are part of the static context. See Chapter 8 for details.

3.5 Navigation Examples

To illustrate the navigation concepts introduced in this chapter, let's consider a variety of different navigation tasks over the sample XML document, `team.xml`, from Chapter 1. For convenience, it's repeated in Listing 3.6.

This document contains employee information from a fictitious organization. The data consists primarily of `Employee` elements, in which parent/child relationships in the XML correspond to manager/employee relationships in the organization. Just to spice things up a bit, the document also contains a few comments and processing instructions.

Listing 3.6 The team.xml document

```
<?xml version='1.0'?>
<Team name="Project 42" xmlns:a="urn:annotations">
  <Employee id="E6" years="4.3">
    <Name>Chaz Hoover</Name>
    <Title>Architect</Title>
    <Expertise>Puzzles</Expertise>
    <Expertise>Games</Expertise>
    <Employee id="E2" years="6.1" a:assigned-to="Jade Studios">
      <Name>Carl Yates</Name>
      <Title>Dev Lead</Title>
      <Expertise>Video Games</Expertise>
      <Employee id="E4" years="1.2" a:assigned-to="PVR">
        <Name>Panda Serai</Name>
        <Title>Developer</Title>
        <Expertise>Hardware</Expertise>
        <Expertise>Entertainment</Expertise>
      </Employee>
      <Employee id="E5" years="0.6">
        <?Follow-up?>
        <Name>Jason Abedora</Name>
        <Title>Developer</Title>
        <Expertise>Puzzles</Expertise>
      </Employee>
    </Employee>
    <Employee id="E1" years="8.2">
      <!-- new hire 13 May -->
      <Name>Kandy Konrad</Name>
      <Title>QA Lead</Title>
      <Expertise>Movies</Expertise>
      <Expertise>Sports</Expertise>
      <Employee id="E0" years="8.5" a:status="on leave">
        <Name>Wanda Wilson</Name>
        <Title>QA Engineer</Title>
        <Expertise>Home Theater</Expertise>
        <Expertise>Board Games</Expertise>
        <Expertise>Puzzles</Expertise>
      </Employee>
    </Employee>
    <Employee id="E3" years="2.8">
      <Name>Jim Barry</Name>
      <Title>QA Engineer</Title>
      <Expertise>Video Games</Expertise>
    </Employee>
  </Employee>
</Team>
```

Each `Employee` has an `id` attribute that we will assume has been typed as `xs:ID` with a DTD or XML Schema so that it can be looked up by the `id()` lookup function. And finally, the `team.xml` document contains some "annotations" in another namespace (`"urn:annotations"`). These attributes describe additional information about the employees and are used here to demonstrate navigation using qualified names and the other wildcard node tests.

For the first example, the `team.xml` document is loaded using the `doc()` function. For the remaining examples, we will assume that this document is already the input sequence, so that all paths are resolved relative to it without loading it explicitly.

As our first task, consider finding the names of all employees. Because `Employee` elements occur at many different levels in the XML, use the descendant navigation shortcut `//` to match every `Employee` element descendant of the root document node. Finally, select their child elements named `Name`. The result is a list of the names of all employees in the document (returned in document order), as shown in Listing 3.7.

Listing 3.7 Find the names of all employees

```
doc("team.xml")//Employee/Name
=>
<Name>Chaz Hoover</Name>
<Name>Carl Yates</Name>
<Name>Panda Serai</Name>
<Name>Jason Abedora</Name>
<Name>Kandy Konrad</Name>
<Name>Wanda Wilson</Name>
<Name>Jim Barry</Name>
```

Suppose instead we want to select only some of the employees, subject to some condition as in Listing 3.8.

Listing 3.8 Name all employees who have been at the company less than two years

```
//Employee[@years < 2]/Name
=>
<Name>Panda Serai</Name>
<Name>Jason Abedora</Name>
```

This path is the same as the previous one, except that a predicate has been added to the `Employee` step. We want to filter the `Employee` elements so that we select only those whose `years` attribute has a value less than 2. So we use the attribute axis `@` and the less-than comparison operator `<` to compare the years attribute against 2. Then from these filtered employees, their names are selected as in the previous example.

We can also search for attributes in another namespace. For example, we could search for all employees currently assigned, as shown in Listing 3.9.

Listing 3.9 Find all employees currently assigned

```
declare namespace ann = "urn:annotations";
//Employee[@ann:assigned]/Name
=>
<Name>Carl Yates</Name>
<Name>Panda Serai</Name>
```

In this query, we have used the query prolog to declare a namespace prefix, and then used this prefix in the attribute name test `@ann:assigned` to match attributes with the local name equal to `assigned` and namespace equal to `urn:annotations`. Note that the prefix used in the query can be (and in this case is) different from the one used in the original document.

By putting the attribute test in the predicate with no comparison, we test for its existence. The predicate is converted using the Effective Boolean Value rule, which tests whether the sequence is non-empty. Consequently, this XPath finds all employees with an assignment, regardless of what that assignment actually is. Similarly, we could find all employees who lack an assignment by applying the `not()` function, as shown in Listing 3.10.

Listing 3.10 Find all unassigned employees

```
declare namespace ann = "urn:annotations";
//Employee[not(@ann:assigned)]/Name
=>
<Name>Chaz Hoover</Name>
<Name>Jason Abedora</Name>
<Name>Kandy Konrad</Name>
<Name>Wanda Wilson</Name>
<Name>Jim Barry</Name>
```

The query to find all employees with an expertise in puzzles is superficially similar to the previous query. The previous query needed to compare attribute values; this query (see Listing 3.11) needs to compare child element values.

Listing 3.11 Find all employees skilled in puzzles

```
//Employee[Expertise = "Puzzles"]/Name
=>
<Name>Chaz Hoover</Name>
<Name>Jason Abedora</Name>
<Name>Wanda Wilson</Name>
```

This case is made somewhat more difficult by the fact that employees may have more than one expertise. Consequently, we must test whether there exists any child expertise element with the desired value. Fortunately, the general comparison operators like `<` and `=` are defined so that they already do this existence test implicitly (see Chapter 5). Thus, the predicate `Expertise = "Puzzles"` tests whether there exists a child element named `Expertise` whose string value is `"Puzzles"`.

Navigation can also be used to compute other values. For example, Listing 3.12 counts the number of employees in one division.

Listing 3.12 Count the number of people in Chaz Hoover's organization

```
count(//Employee[Name="Chaz Hoover"]/descendant-or-self::Employee)
=>
7
```

The `count()` function computes the number of items in a sequence (see Chapter 5). First we must locate the employee named Chaz Hoover, which can be done using a query like the ones used previously. But then we must count all the employees contained in the sub-tree rooted at Chaz Hoover—in other words, all the descendant `Employee` elements. By using the `descendant-or-self` axis, we have included Chaz Hoover himself in this count. We could exclude him by instead using the `descendant` axis as in the path `count(//Employee[Name="Chaz Hoover"]/descendant::Employee)`.

Instead of counting the entire organization, we could instead count only those employee elements directly under Chaz Hoover, as shown in Listing 3.13.

Listing 3.13 Count the number of Chaz Hoover's direct reports

```
count(//Employee[Name="Chaz Hoover"]/Employee)
=>
3
```

Instead of performing a descendant query with `//`, we have used ordinary child navigation `/` to select only those employees who report directly to Chaz Hoover. This task could also be accomplished in another way, using the parent axis, as shown in Listing 3.14.

Listing 3.14 Use the parent axis to count Chaz Hoover's employees

```
count(//Employee[../Name="Chaz Hoover"])
=>
3
```

Here we first find every `Employee` in the document. Then, we check to see if the parent element has the name Chaz Hoover. We use the `..` abbreviation to navigate to the parent, and then compare its child `Name` element. This query is usually much slower than the previous one, although some implementations can optimize it so that both perform identically.

We can also find other kinds of elements in the tree. For example, we could extract all comment and processing-instruction nodes by using node kind tests, as illustrated in Listing 3.15.

Listing 3.15 Find all comments and processing instructions

```
//comment() | //processing-instruction()
=>
<?Follow up?>
<!-- new hire 13 May -->
```

In this query, we used the union operator `|` to combine the results of both paths. We could have also written `//(comment() | processing-instruction())` to achieve the same effect. (See Chapter 5 for more information about the union operator.)

Notice that the XML declaration `<?xml version='1.0'?>` at the top of the document did not match. Although it looks like a processing instruction, XML doesn't treat it as part of the data model, so XQuery doesn't either. Finally, Listing 3.16 demonstrates looking up elements by their IDs.

Listing 3.16 Find all employees with the same job function as employee E0

```
//Employee[Title = id("E0")/Title]/Name
=>
<Name>Wanda Wilson</Name>
<Name>Jim Barry</Name>
```

Because we wish to find all employee names satisfying some condition, we know that the path will consist of `//Employee/Name` and use a predicate to limit which employees are matched. This predicate should select all employees with the same title as that of employee `E0`. Employee `E0` can be found using the `id()` navigation function: `id("E0")`. Then all that remains is to compare the current employee's title against that of `E0`.

Notice that instead of using `id()`, we could use an absolute path inside the predicate to search from the root of the document to find the employee with id `E0`: `//Employee[Title = //Employee[@id="E0"]/Title]/Name`. This path has the advantage that it doesn't require a DTD or schema to type the `id` attribute. However, it is more complex to write and usually will perform worse than the `id` lookup (which most implementations optimize into an index or table lookup). This path essentially performs a join of the document with itself. Joins like this are often expressed using FLWOR expressions (described in Chapter 6).

3.6 Navigation Complexities

This section discusses the last remaining navigation topics. All of these were either too esoteric or too complex to merit including in the previous sections.

3.6.1 Namespaces

XPath 1.0 doesn't have a way to introduce namespace declarations and doesn't use the prefixes of the data it is navigating. Consequently, any namespace prefixes used in an XPath expression must be defined outside of it. In XQuery, namespaces can be declared in the query prolog (covered in Chapter 5), or using namespace declarations in XML elements (Chapter 7).

Table 3.4 Qualified names versus expanded names

	Prefix	Local	Namespace	Example
Qualified name	Yes	Yes	No	`foo:bar`
Expanded name	No	Yes	Yes	`{urn:baz}bar`

Recall that a *qualified name* consists of two parts, the *prefix* and the *local name*, separated by a colon (`:`). The prefix is bound to a namespace, but is otherwise unimportant for the purposes of navigation. Instead, it is better to think in terms of expanded names. An *expanded name* is the namespace and local name parts of a name (ignoring the prefix). Table 3.4 summarizes the differences between the two.

The second example in Table 3.4 is fabricated; XML and XQuery don't have a syntax for expanded names. Instead, they always associate the namespace with a prefix, and then use a qualified name.

In prose descriptions, expanded names are often written with the namespace part in curly braces (`{}`), like this: `{namespace}local-name`. When there isn't a namespace (because the name was unqualified or had an empty namespace), then the expanded name is written as just `{}local-name`. Again, this syntax isn't used in XML or XQuery, just in descriptions of how they work.

Listing 3.17 XML with namespaces

```
<root xmlns:x="uri1">
  <x:one fish="red"/>
  <two x:fish="blue" xmlns="uri2"/>
</root>
```

For example, in the XML shown in Listing 3.17, there are three elements with qualified names: `root`, `x:one`, and `two`. The expanded names of these elements are `{}root`, `{uri1}one`, and `{uri2}two`, respectively. There are two attributes with qualified names: `fish` and `x:fish`. The first of these has the expanded name `{}fish`, the second has the expanded name `{uri1}fish`.

Expanded names are more verbose than qualified names, which explains why qualified names are used instead. However, most operations—including validation and navigation—operate only on the namespace and local name

03Brundage_ch03.qxd 1/8/04 1:09 PM Page 83

parts of XML names, usually completely ignoring the prefixes that were used in the original XML serialization.

Suppose `doc("sample.xml")` accesses the XML shown in Listing 3.18.

Listing 3.18 sample.xml

```
<this xmlns="urn:default" xmlns:ns1="urn:one">
  <is a="complex">
    <ns1:example ns2:attr="42" xmlns:ns2="urn:two"/>
  </is>
</this>
```

Then you could navigate into it using the XQuery shown in Listing 3.19.

Listing 3.19 Path using namespaces

```
declare namespace x = "urn:default";
declare namespace y = "urn:one";
declare namespace z = "urn:two";
doc("sample.xml")/x:this/x:is/y:example/@z:attr
```

The query prolog introduces three namespace declarations, binding the prefixes `x`, `y`, and `z` to the namespaces `urn:default`, `urn:namespace1`, and `urn:namespace2`, respectively. The path then uses these namespaces to perform its qualified name tests: `x:this`, `x:is`, `y:example`, and `z:attr`. Again, notice that the prefixes in the document and the prefixes in the XQuery are completely unrelated to one another; all that matters are the local name and namespace parts of the names.

XQuery provides two functions that can access the namespaces in scope on a node. The `get-in-scope-prefixes()` function takes one argument, an element node, and returns a list of strings (in any order) that are the namespace prefixes in scope for that element. An empty string is listed for the default namespace declaration, if any.

The `get-namespace-uri-for-prefix()` function can look up the namespace value for a prefix. It takes two arguments, an element node and a string prefix, and returns the string that is the namespace bound to that prefix (use the empty string to look up the default namespace declaration). If there is no namespace bound to that prefix, then this function returns the empty sequence. Both effects are demonstrated by the examples in Listing 3.20.

Listing 3.20 Querying the namespaces in scope

```
declare namespace x = "urn:default";
get-in-scope-prefixes(doc("sample.xml")/x:this)
=>
("ns1", "")

declare namespace x = "urn:default";
get-namespace-uri-for-prefix(doc("sample.xml")/x:this, "ns1")
=>
"urn:one"
```

3.6.2 Node Identity

Navigation has some interesting interactions with node identity. When navigating over constructed XML, it's important to realize that the construction process copies nodes used as content, thereby "losing" their node identity.

For example, in the expression `<x>{doc("y.xml")//y}</x>`, the nodes selected by the path are copied into the `x` element. If you then navigate into that constructed XML, you get different nodes (by identity) than the originals, as demonstrated by Listing 3.21. (See Chapter 7 for more information).

Listing 3.21 Navigating over constructed XML

```
(<x>{doc("y.xml")//y}</x>)//y is doc("y.xml")//y   => false
```

The `doc()` function is special in that whenever the same string value is passed to it, the same node (by identity) is returned. This special behavior prevents you from writing a user-defined function that fully emulates `doc()` using construction, because every time your function is invoked it constructs a new node instance. The difference is demonstrated in Listing 3.22.

Listing 3.22 The `doc()` function can't be completely simulated by your own

```
declare namespace my = "http://www.awprofessional.com";
declare function my:doc($dummy as xs:string) as node() {
  document {
    element root { () }
  }
};

doc("a.xml") is doc("a.xml")       => true (if a.xml exists)
my:doc("a.xml") is my:doc("a.xml") => false
```

3.6.3 Other Context Information

In addition to the context items listed in Section 3.4, XQuery provides several other less commonly used values in the expression context.

The *base uri* property is part of the static context and is used by the `doc()` function when resolving relative URIs. This property can also be accessed using the `base-uri()` function.

The current *XML space* policy is part of the static context and can be changed by the query prolog. It determines how space characters are handled in XML constructors (see Chapter 7).

The static context may also provide schema definitions from imported schemas and a default validation mode and/or validation context. These determine what user-defined type names are available for use in type tests and other type operators. See Chapter 9 for examples.

Finally, the *current date/time* and the *implicit timezone* properties are part of the evaluation context. Despite their names, these don't really provide the current time, but just some point in time determined by the implementation. The value doesn't change during the execution of a query. These values can be accessed using the `current-date()`, `current-time()`, `current-dateTime()`, and `implicit-timezone()` functions (see Appendix C).

3.7 Conclusion

In this chapter, we investigated one of the primary features of XQuery, navigation. We delved into the syntax and meaning of path expressions, and how the evaluation context affects (and is affected by) their evaluation.

The chapter discussed the six required XQuery axes (`attribute`, `child`, `descendant`, `descendant-or-self`, `parent`, and `self`) and the six optional ones (`ancestor`, `ancestor-or-self`, `preceding`, `preceding-sibling`, `following`, and `following-sibling`). In addition, it described the name tests, node kind tests, and wildcards that can be used with axes to select nodes by name, kind, and type.

Predicates can be used to filter an expression by other criteria, including position. Also, XQuery provides several navigation functions, including the important `doc()` function, which loads external XML data.

We also explored how to solve a variety of real-world tasks using path navigation. Because paths are so important to XQuery, many additional examples appear throughout this book, especially in Chapters 10 and 11.

3.8 Further Reading

For more information about XPath 1.0, see the W3C Recommendation at `http://www.w3.org/TR/xpath`. The book *Essential XML Quick Reference: A Programmer's Reference to XML, XPath, XSLT, XML Schema, SOAP, and More* by Aaron Skonnard and Martin Gudginis also a good XPath reference.

A brief history of Unix, including the advent of paths, appears in the article "The Evolution of the Unix Time-sharing System" by Dennis Ritchie. This article has been published in several computer science publications, and is also available online at `http://cm.bell-labs.com/cm/cs/who/dmr/hist.html`.