CHAPTER

3

XPath, XPointer, and XLink

IN THIS CHAPTER

Meet a Few of XML's Relatives

In Chapter 2, "XML for ASP.NET Basics," you saw that the XML language provides a structure for marking up data in a uniform, extensible, and descriptive manner that is readable both by humans and computers. Although this capability is certainly a welcome addition to the programming world, XML's true power and flexibility can be realized only as it is combined with other languages and technologies. An XML document is quite useless on its own without a means for manipulating the data it is describing. To understand this further, imagine a SQL Server database that allowed for the creation of databases and tables but had no mechanism for querying the data. Although the data itself would be efficiently organized into relational tables, this structure wouldn't be very useful if you couldn't use Transact SQL (T-SQL) to run queries against the data. Assuming that an XML document is your SQL Server database, you obviously need some type of mechanism that allows you to access the data so that the XML document can be useful in application development.

Several languages have been developed to allow XML documents to be queried and even linked in some interesting ways. These languages include XPath, XPointer, and and XLink. All three can be used in concert to access data located not only within a particular XML document but also within external documents located anywhere in the world. Much of the functionality contained within the XLink and XPointer languages is still being finalized by the W3C. As a result, Microsoft's .NET XML parser implementation (along with many other popular parsers) does not yet implement XLink or XPointer capabilities. However, this won't stop us from discussing some of the exciting capabilities that XLink and XPointer will offer in the near future.

You may be wondering why a new manner of linking documents needs to be developed? After all, doesn't the HTML language provide us with sufficient linking capabilities? The answer is both "yes" and "no." Although most HTML documents employ the `<a>` tag to link different documents, this functionality is extremely limited.

For example, individuals clicking a link may not want to view the entire document that they are linking to on the other end. If they are clicking a link to obtain weather information, they may want only the current temperature for the city they live in. Current HTML linking capabilities require that the entire document be loaded rather than simply the section the individual wants to view. XPath, XPointer, and XLink exist to remedy this problem and improve on HTML's current linking capabilities.

The information in this chapter will be presented in the following order:

- XPath—The SQL of XML?
- XPointer—Accessing XML Document Fragments
- XLink—Resource Relationship Management

XPath—The SQL of XML

When the W3C committees first began working on an XML query language (`http://www.w3.org/TR/xpath`), they realized that the XPointer and XSLT groups were attempting to accomplish a similar task: access specific sections of an XML document. As a result, the XPointer, XSLT, and XPath groups worked together to specify a query language to be used with XML. The current XPath specification (version 1.0) became a W3C recommendation in November of 1999 and is a result of this joint effort.

If you've ever used T-SQL to query a SQL Server database, you know how useful it is for obtaining recordsets. Depending on the type of ASP.NET development you do, you are more than likely used to seeing T-SQL (or ANSI SQL) syntax similar to the following example embedded within ASP.NET pages or stored procedures:

```
SELECT * FROM AUTHORS WHERE au_id = '172-32-1176'
```

Although syntactically different, XPath provides the same access to XML document data as T-SQL does to SQL Server data. The XPath language is used by XSLT (covered in Chapter 7, "Transforming XML with XSLT and ASP.NET") and XPointer to access data found within specific XML document elements, attributes, and so on.

So what does XPath syntax look like? You may be surprised to learn that its syntax does not resemble XML's syntax at all. In some ways it resembles a syntax that you are more than likely used to working with: the Windows file system.

The Windows file system emulates a hierarchical structure with the drive letter representing the root of all other files and folders on that drive. An XML document is structured hierarchically with the root element (also called the root node) containing different child elements (or nodes). These child elements may themselves contain other child elements.

To access a folder on one of your drives (`C:` for example), your computer follows a path similar to the following:

```
c:\temp\subFolder1\subFolder2\xmlFile.xml
```

This path can easily be converted into an XML document:

```
<?xml version="1.0"?>
<C>
    <temp>
        <subFolder1>
            <subFolder2>xmlFile.xml</subFolder2>
        </subFolder1>
    </temp>
</C>
```

TIP

As a point of review, you cannot simply add `C:` as the root element name. Remember in Chapter 2 that we discussed the concept of qualified namespaces. Putting the colon in the root name makes the parser think that `C` was declared as a qualified namespace somewhere in the document. Because it can't find the declaration (or the element name after the colon), the parser will raise an error.

If you wanted to access `subFolder2` in the preceding XML document using XPath syntax, you would simply type the following:

```
C/temp/subFolder1/subFolder2
```

You could also type a shorthand version. This version of the query searches for all `subFolder2` nodes located anywhere within the XML document:

```
//subFolder2
```

You're now an XPath expert! Well, the word "expert" may be a little strong at this point, but your experience in working with the Windows file system should help you in understanding how XPath works. XPath is, of course, much more powerful than this simple example. Aside from its basic syntax, the XPath language also includes its own set of functions to provide additional utilities such as formatting strings and numbers or performing mathematical calculations.

Before going into more detail about XPath, it's important that you realize that it is not designed as a standalone language. Going back to the Windows file system comparison, without having access to a DOS command prompt, File Manager, or Windows Explorer, typing the path to a file wouldn't get you very far and you definitely wouldn't find the file or folder you were looking for. The syntax you saw earlier to access a file is functional only when used in conjunction with another language or program (such as a command prompt). XPath was designed in the same manner. It is useful only when used with another language such as XPointer or XSLT.

XPath Basics

The XPath specification defines a term named "Location Steps" that sums up how the XPath language actually works. To get to a particular node's location, you have to list the steps to get there using an XPath expression or pattern. Going back to the first XPath sample, we defined the starting point as `C` and listed each step that should be followed to walk through the element nodes to get to the destination data. Each step of the process was separated by a `/` character.

Let’s clarify this a little more through giving a simple real-life example. Getting to a node within a document is very similar to explaining your family lineage. If a friend asked you about a portion of your family line going back to your great grandfather, you may give the following information using XPath syntax:

```
Great_Grandfather/Grandmother/Mother/Self
```

You can see from this sample that each step in your family lineage is separated by the `/` character.

Another important term used in the XPath specification is that of the context node. The context node is the node being evaluated at each step of an XPath expression. In your family lineage example, the context node is first your `Great_Grandfather`. The context changes to the `Grandmother` node and then moves to the `Mother` node. Finally, the context node is `Self`. This same logic applies to nodes being evaluated at each step of an XPath expression.

The context node becomes important as evaluations are being done on it. If someone asks you how many children your ______ had, it would be difficult to answer the question unless the blank was filled in (unless you can read minds of course). You need to know where to start (the context) in order to reply. If, however, someone asks you how many children your great grandfather had, you can now provide an answer because you have a context from which to start.

The path to a particular node within an XML document is constructed from three pieces of information:

- An Axis Type
- A Node Test
- Predicates (filtering)

The following sections explain each of these items in turn. After that discussion, we’ll discuss XPath functions and get into some examples so that you can see more XPath expressions at work.

Axis Types

The family lineage XPath statement we looked at earlier took advantage of some XPath abbreviations to make things look a little nicer. Without using abbreviations, the same statement would look like this:

```
Great_Grandfather/child::Grandmother/child::Mother/child::Self
```

Don’t worry about the `::` characters. We’re not going to define any C++ interfaces or anything like that. The `child::` syntax is one type of axis defined by the W3C that can be used when you’re trying to locate a node within a document. Each axis is followed by two colons (`::`) when used within an XPath expression. Axes are somewhat intimidating at first glance but are very useful when you know a few basics about how they work. Table 3.1 contains all the available axes as defined in the XPath specification.

Table 3.1 XPath Axes

Axis	*Axis Description*
`child`	Includes all children of the context node.
`descendant`	Includes all descendants of the context node. This includes children, grandchildren, and so on.
`parent`	Includes the parent of the context node.
`ancestor`	Includes all ancestors of the context node.
`following-sibling`	Includes all siblings that follow the context node in document order.
`preceding-sibling`	Includes all siblings that precede the context node in document order.
`following`	Includes all nodes that follow the context node in document order. This does not include ancestors.
`preceding`	Includes all nodes that precede the context node in document order. This does not include ancestors.
`attribute`	Includes all attributes of the context node.
`namespace`	Includes all namespaces within the scope of the context node. If no namespaces are declared, it is empty.
`self`	Includes the context node only.
`descendent-or-self`	Includes all descendants of the context node as well as the context node itself.
`ancestor-or-self`	Includes all ancestors of the context node as well as the context node itself.

Although many of the axes, such as `self`, `child`, and `parent` are self-explanatory, a few are not as obvious at first. Let's take a look at a few axes examples based on Listing 3.1.

Listing 3.1 Working with Axes in XPath

```
<?xml version="1.0"?>
<familyLine xmlns:me="http://www.someURL.com">
    <greatGrandfather fname="Jonathon" lname="Doe">
        <grandMother fname="Betty" lname="Doe">
            <father fname="James" lname="Doe">
```

Listing 3.1 continued

```
               <me:self fname="John" lname="Doe"/>
               <sister fname="Mary" lname="Doe">
                   <child fname="Ted" lname="Doe"/>
               </sister>
                <brother fname="Billy" lname="Doe"/>
            </father>
        </grandMother>
    </greatGrandfather>
</familyLine >
```

Assuming that the context node is `<father>`, the `descendant` axis to this node would, as the axis description says, select all descendants, which in this case would be the `<self>`, `<sister>`, `<brother>`, and `<child>` nodes. Which nodes would be included in the `descendant-or-self` axis if the `<father>` node is still the context node? This axis would contain the `<father>`, `<self>`, `<sister>`, `<brother>`, and `<child>` nodes. Keep in mind that although this example uses the element named `self`, this has nothing to do with the word self as contained in the `descendant-or-self` axis. `self` as used in the axis name simply refers to the context node, which is `<father>` in this example.

The following are a few more examples that simply list the axis name and nodes contained within it. Assume that the `<self>` node is now the context node:

Axis	*Included Nodes*
`following-sibling`	`<sister>`, `<brother>`
`preceding-sibling`	none
`ancestor`	`<father>`, `<grandMother>`, `<greatGrandfather>`, `<familyLine>`
`ancestor-or-self`	`<self>`, `<father>`, `<grandMother>`, `<greatGrandfather>`, `<familyLine>`
`attribute`	`fname` and `lname`
`namespace`	`me` qualified namespace
`following`	`<sister>`, `<brother>`, `<child>`
`preceding`	none

Node Tests

Node tests let you specify what type of node to locate relative to the context node. Each of the previously discussed axes have their own principal node type. For those axes that can contain elements, the principle node type is element. Because the attribute axes can contain only attributes, its principle node type is. . . you guessed it, attribute.

The previous examples have performed node tests to check for element nodes only. However, you could have done other types of node tests if you had wanted to get to a different location in the document other than an element. The other node tests you could have used include

- The * wildcard character, which includes all items in the current axis. The wildcard works in a manner similar to the SQL * wildcard character used in `SELECT` statements.
- `text()`, which includes all text nodes in the current axis.
- `node()`, which includes all nodes in the current axis.
- `comment()`, which includes all comments in the current axis.
- `processing-instruction()`, which includes all processing-instructions in the current axis.

Although you'll have the opportunity to see plenty of examples of node tests in action, the following are a few simple examples with explanations to get you started:

axis::node test	*Explanation*
`attribute::*`	All attributes of the context node are selected.
`child::text()`	Matches the context node children's text nodes.
`descendant-or-`	Matches all `sibling` elements that are `self::sibling` descendants of the context node. The context node itself is also included if it is a sibling element.

Predicates

Using axes and node tests can help in reaching a specific location within an XML document. But what if the context node contains four children that are elements and you want to select only the third child element? You can use the child axis and element node test to get to the four children, but neither the axes nor node tests help you to select the third element only.

Predicates, otherwise known as `WHERE` clauses, help solve this dilemma for you. A predicate helps to filter out unwanted nodes by taking advantage of built-in functions. This process is very similar to using the RowFilter property of the ADO.NET DataView class. Using ADO.NET, filtering out records that don't have a specific `ID` (as defined in `intID` below) is as easy as typing the following code:

```
dataView.RowFilter = "ID = " + intID;
```

Using XPath, filtering out nodes that don't meet specific requirements is as easy as typing

```
child::sibling[position()=3]
```

This XPath statement uses a predicate to select the third sibling child of the context node.

You can see that a predicate is started and ended by using the `[` and `]` character tokens. XPath has several built-in functions, which you'll learn more about in a moment, that can be used within the predicate statement. The preceding example used the `position()` function. Using one or more of these functions in combination with comparative operators, the processor performs the necessary checks on the current node-set to see which nodes should remain and should be excluded. If you're wondering what a node-set is, think of it as another form of an ADO.NET DataSet. The difference is that the node-set contains a collection of XML nodes, whereas the DataSet contains records returned from a data source.

The predicate statement as a whole produces a Boolean value after it is evaluated. Either the predicate statement is true and the node being evaluated is included in the node-set, or it is false and the node is excluded from the node-set. This Boolean value is derived by comparing the result of a function call to a value using comparative operators, including

- `=`, `!=`
- `<`, `<=`, `>`, `>=`
- `-`, `+`, `*`
- `and`, `or`
- `div`, `mod`
- `|` (union operator)

Your first look at predicates showed how to get at the third sibling child node of the context node. If your goal were to get the third and fourth child sibling nodes, you could use one of the following statements:

```
child::sibling[position() >= 3]
child::sibling[position() > 2]
child::*[position() = 3 or position() = 4]
```

Later in the chapter, you'll be presented with more predicate statements.

The Location Step Framework

Putting the axis, node test, and predicate information types together results in the following location step syntax framework:

```
<axis>::<node-test> [<predicate expression>]
```

All location steps in an XPath expression will follow this syntax. We can break the syntax into pieces: the XPath statement must first specify the type of axis to traverse. The `child` axis is the default if no axis is specified. After it is on the proper axis, the node test is performed to check whether a particular node exists. In cases where a specific node must be accessed, a predicate can be added to the XPath expression, which acts as a filter.

Samples of using this syntax are shown later in the chapter, along with abbreviated forms that can be used to shorten the location step framework. Before these are shown, however, it's important that you understand what native functions exist in the XPath recommendation. Understanding these functions will allow you to fully leverage predicates.

XPath Functions

Many functions are available other than the `position()` function shown earlier. The XPath recommendation categorizes each of the predicate functions based on its purpose and functionality. These categories include node-set functions, string functions, number functions, and Boolean functions. We'll first discuss the node-set functions.

XPath Node-Set Functions

The `position()` function that you have been working with thus far is an example of a node-set function. Node-set functions have the obvious function of returning information about one or more nodes. Table 3.2 shows the node-set functions listed in the W3C XPath recommendation.

Table 3.2 Node-Set Functions

Function Name	*Purpose*
`last()`	Returns the index number assigned to the last node in the current context. The example selects the last `customer` child element. Example: `child::customer[last()]`
`position()`	Evaluates the position of the current node being evaluated within a given context and returns a number. Numbering starts with 1. Example: `child::customer[position() = 2]`
`count(node-set)`	This function takes a node-set as a parameter and returns the number of nodes within the node-set. It is similar to `Count(*)` in SQL. The example selects a father node containing four sibling nodes. Example: `descendant::father[count(sibling) = 4]`
`id(object)`	Selects nodes based on a unique ID attribute and returns a node-set. It expects an object and if the object is a node-set, the node-set is converted to a string value. Any IDs that match this string value will be returned. This function is similar to the `key()` function (covered in Chapter 7), but is more powerful because it can accept a node-set rather than a single string value. It is designed to work with a valid document in which attributes have been declared as type `ID`. The example selects all `customer` child nodes with a unique ID of 1234. The selection will also look for IDREF and IDREFS attribute node types that have a value of 1234 (ID, IDREF, and IDREFS are discussed in Chapter 4, "Understanding DTDs and XML Schemas"). Example: `child::customer[id("1234")]`

Table 3.2 continued

Function Name	*Purpose*
`local-name(node)`	Returns the node name located after the colon if a qualified namespace is associated with the node (applying it to `family:father` would return `father`) or simply the node's name if no namespace is associated. The function can accept a node (that is, `local-name(node)`) or can simply evaluate the current node being evaluated (`local-name()`). The example selects all child nodes of the context node with a local-name of `father`. Example: `child::*[local-name() = "father"]`
`name(node)`	Works like the `local-name()` function, except that it returns the actual node name as it appears in the original XML document, including the namespace prefix if any exists. Example: `child::*[name() = "family:father"]`
`namespace-uri(node)`	This function returns the URI of any namespace associated with the node being evaluated. If no namespace URI exists, an empty string is returned. The example selects all child nodes of the context node that have a namespace URI equal to `http://www.someURL.com`. Example: `child::*[namespace-uri() = "http://www.someURL.com"]`

You'll notice that there is no specific node-set function to extract a namespace prefix (the `me` in `me:local-name` for instance) should one exist on a node. The `local-name()` function allows you to get the node's name, the `name()` function gets the prefix and local-name together, and the `namespace-uri()` function returns only the namespace URI. This certainly appears to present a problem because knowing the prefix associated with a particular node may be a big part of your applications, depending on the number of contributors to an XML document. Fortunately, this problem can be solved by taking advantage of node-set functions used in conjunction with XPath string functions.

XPath String Functions

String functions in XPath provide some of the basic functionality that you are used to working with in ASP.NET documents using VB.NET or C#. Functions are available to convert values to a string, concatenate strings, return substrings, check string lengths, and more. Table 3.3 contains a complete listing of all string functions in the XPath recommendation.

Table 3.3 String Functions

Function Name	*Purpose*
`string(value)`	This function converts a value to a string. If a node-set is passed in, the string value of the first node in the set is converted. This function is not normally needed because string conversions are handled on-the-fly when a string value is required. The example converts the number 101 to “101”. Example: `string(101)`
`concat(string1, string2, string3, ...)`	This function concatenates (combines) all strings supplied as arguments. The example returns “Phoenix, Arizona.” Example: `concat("Phoenix",", ","Arizona")`
`starts-with(string1, string2)`	This function returns true if `string1` starts with `string2`, and returns false otherwise. The comparison is case sensitive. The example will return `true`. If `string2`’s value had been “a”, `false` would be returned. Example: `starts-with("Arizona","A")`
`contains(string1, string2)`	This function is similar to the `InStr()` function in VB.NET or `IndexOf()` function C#, although it doesn’t return a positional value. It returns `true` if `string1` contains `string2` in it. The comparison is case sensitive. The example returns `true`. Example: `contains("Sibling","Sib")`
`substring-before (string1,string2)`	This function is similar to the `Left()` function in VB.NET, although it doesn’t accept a length argument. It returns the portion of `string1` that precedes `string2`. If `string2` doesn’t exist in `string1`, empty strings are returned. The example returns the “child” string. Example: `substring-before("child,sibling",",")` This function can also be used in conjunction with the `name()` function to get a node’s namespace prefix, which solves the problem mentioned in the previous section. Assuming the `name()` function returns `me:myself`, this example will return `me`: Example: `substring-before(name(myself),":")`

Table 3.3 continued

Function Name	*Purpose*
`substring-after (string1,string2)`	This function is similar to the `Right()` function in VB.NET, although it doesn't accept a length argument. It acts like the previous function except it returns the portion of `string1` that occurs after `string2`. The example returns the "sibling" string. Example: `substring-after("child,sibling",",")`
`substring(string, number,number)`	This function is similar to the `Mid()` function in VB.NET or the `Substring()` function in `C#`. It returns the portion of the input string starting at the character index specified in the second argument with a length equal to the third argument. If no length is specified, the returned string will contain the input string starting from the second argument to the end of the string. The first character starts with `1` rather than `0`, like `C#`. The example returns the "ling" string. Example: `substring("sibling",4)`
`string-length(string)`	This function returns the number of characters in the supplied string argument. Passing in an empty string results in a value of 0 being returned. The example returns a value of 7. Example: `string-length("Sibling")`
`normalize-space(string)`	This function is similar to the `Trim()` function in VB, although it also replaces internal sections of whitespace with single spaces. The example returns "How are you?" Example: `normalize-space("  How are you? ")`
`translate(string, from,to)`	This function performs character substitutions. Characters that exist in the first two arguments are converted to the corresponding characters found in the third argument. This function is best understood by seeing an example. The example shown below will return "HOW ARE YOU?" Example: `translate("how are you?","aehouwy", "AEHOUWY")` The next example will return "I AM fiNE" Example: `translate("i am fine","amne","AMNE")`

Table 3.3 continued

Function Name	*Purpose*
`translate(string, from, to)` (continued)	Characters can be replaced (just like VB.NET's `Replace()` function) by supplying the second argument with the character to replace and the third argument with the character that it should be replaced with. For example, the following code will return "how are you?" Example: `translate("how-are-you?"," -"," ")`

XPath Number Functions

Table 3.4 lists all number functions contained within the XPath recommendation. In situations where a function argument cannot be converted to a number, `NaN` ("not a number") will be returned.

Table 3.4 Number Functions

Function Name	*Purpose*
`number(value)`	This function converts the supplied argument to a number. If no argument is supplied, the function defaults to a node-set that contains only the context node. This function is not normally needed because number conversions are handled on-the-fly when a number value is required. If a node-set is supplied, it is first converted to a string that is then applied to the `number()` function. The following example returns `0`. Example: `number('0')` This next example returns a 1. Example: `number(boolean('true'))` You'll learn more about the `boolean()` function in the next section.
`sum(node-set)`	This function returns a number containing the sum of all numeric values of nodes contained within the node-set argument. Assuming that the numeric values of the child nodes found in the following XPath statement include (1,2,3,4), the example would return a calculated value of `10`. Example: `sum(/root/parcelNumbers/*)`
`floor(number)`	This function returns the supplied argument rounded down to the nearest integer. For example, if `10.4` were passed in, the return value would be `10`. Example: `floor(10.4)`

Table 3.4 continued

Function Name	*Purpose*
`ceiling(number)`	This function is similar to the `floor()` function but rounds the supplied argument up to the nearest integer. For example, if `10.4` were passed in, the return value would be `11`. Example: `ceiling(10.4)`
`round(number)`	This function returns a number rounded to the nearest integer. For example, `10.4` rounds to `10`, whereas `10.5` rounds to `11`. Example: `round(10.4)`

XPath Boolean Functions

The XPath specification contains a total of five Boolean functions. Although all these are listed in Table 3.5, you will more than likely only use two of the five functions on a constant basis when working with XPath expressions.

Table 3.5 Boolean Functions

Function Name	*Purpose*
`boolean(value)`	This function returns a Boolean value (`true` or `false`) depending on the value passed into the function. The conversion of a value to Boolean normally occurs automatically as needed. However, the `boolean()` function is useful to explicitly convert a value to a Boolean when necessary. For example, if you have a node named customerExists that contains the text "true," you can convert this value from a string to a Boolean by using the following syntax: Example: `boolean(/root/customerName/customerExists)` To convert a `0 (false)` or `1 (true)` to a Boolean, use the following syntax: Example: `boolean(1)`
`not(boolean)`	This function returns the opposite of what is passed into it. Thus, if a true is passed in, a false will be returned and vice versa. The example returns a true. Example: `not(false)`
`true()`	This function returns `true`. This can be used when a constant Boolean value is needed in an XPath expression.

Table 3.5 Boolean Functions

Function Name	*Purpose*
`false()`	This function returns `false`. This can be used when a constant Boolean value is needed in an XPath expression.
`lang(string)`	This function returns `true` or `false` based on whether the context node's language (as determined by the value of the `xml:lang` attribute) equates with the supplied string argument. Assuming that an `xml:lang` attribute is equal to "en" on a particular node, the following example returns true if evaluated with that node as the context node. Example: `lang("en")`

XPath Abbreviation Examples

Having covered all the various functions available in the XPath specification, you're probably ready to move from the learning phase to the "let me see this in action" phase. Before jumping into some detailed XPath examples, you'll be interested to know that many of the location step types mentioned earlier in the chapter have an abbreviated form that can save you time and make your XPath expressions much easier to read. A few of the key abbreviations need to be touched on briefly to prepare you for the samples. The following are some important XPath expression abbreviations:

- The `@` symbol is an abbreviation for the `attribute::` axis: `attribute::name` can be written as `@name`.
- The context node can be represented by a period character: `self::*` can be written as `.`
- A few of the previously mentioned functions, such as `position()`, can be left out of the predicate expression: `street[position()=3]` can also be written as `street[3]`.
- The root node can be accessed by typing `/`, and access to all children of the root node can be gained by typing `//*` or `descendant::*`.
- An XPath axis such as `child::` can normally be left out of an expression. `child::street` can be written as simply `street`. Understand that leaving out the `axis syntax` doesn't mean that it isn't there any longer. Every location step moves along a defined axis. When no axis is specified, the expression is assumed to follow the child axis.
- The abbreviated code to access a parent node is similar to accessing a parent folder on the Windows file system. A parent node can be accessed by writing `parent::*` or `..`

The following are a few examples shown first with the non-abbreviated syntax and then followed by the abbreviated syntax. Some of the examples include more than two XPath expressions to help you see other ways of accomplishing the same task. The examples are based on the following XML document:

```
<?xml version="1.0"?>
<golfers>
    <golfer skill="excellent" handicap="4" clubs="Taylor Made" id="1111">
        <name>
            <firstName>Heedy</firstName>
            <lastName>Wahlin</lastName>
        </name>
        <favoriteCourses>
            <course city="Pinetop" state="AZ" name="Pinetop Lakes CC"/>
            <course city="Phoenix" state="AZ" name="Ocotillo"/>
            <course city="Snowflake" state="AZ" name="Silver Creek"/>
        </favoriteCourses>
    </golfer>
    <golfer skill="moderate" handicap="8" clubs="Taylor Made" id="2222">
        <name>
            <firstName>Dan</firstName>
            <lastName>Wahlin</lastName>
        </name>
        <favoriteCourses>
            <course city="Pinetop" state="AZ" name="Pinetop Lakes CC"/>
            <course city="Pinetop" state="AZ" name="White Mountain CC"/>
            <course city="Springville" state="UT" name="Hobble Creek"/>
        </favoriteCourses>
    </golfer>
</golfers>
```

XPath Examples

Select the `id` attribute of the first `golfer` element:

```
/golfers/descendant::golfer[position()=1]/attribute::id
```

```
/golfers/golfer[1]/@id
```

Select all descendants of the `golfer` element. This includes the name and `favoriteCourses` elements:

```
/golfers/descendant::golfer/*
//golfer/*
/golfers/golfer/*
```

Select all attribute descendants of the `golfers` root element:

```
//attribute::*
//@*
```

Select the `name` attribute of the third `course` element associated with the second `golfer` element:

```
/golfers/child::golfer[position()=2]/child::favoriteCourses/
➥child::course[position()=3]/attribute::name
/golfers/golfer[2]/favoriteCourses/course[3]/@name
//golfer[2]/favoriteCourses/course[3]/@name
```

Select the first `golfer` element and all children. This selects the `name` and `favoriteCourses` elements:

```
/golfers/child::golfer[position()=1]/child::*
/golfers/golfer[1]/*
```

Select the second `golfer` element's `course` element having a `name` attribute equal to the first `golfer` element's first `course` element `name` attribute (a little confusing unless you break it down into steps):

```
/golfers/child::golfer[position()=2]/child::favoriteCourses/child::course/
➥attribute::name[. =
/golfers/child::golfer[position()=1]/child::favoriteCourses/
➥child::course[position()=1]/attribute::name]
/golfers/golfer[2]/favoriteCourses/course/@name[. = /golfers/golfer[1]/
➥favoriteCourses/course[1]/@name]
```

Select the last `course` element's `name` attribute that is located under the second `golfer` element:

```
/golfers/child::golfer[position()=2]/favoriteCourses/course[last()]/@name
//golfer[2]//course[last()]/@name
/golfers/golfer[2]/favoriteCourses/course[last()]/@name
```

Select the `city` attribute of all `course` elements that have a `name` attribute starting with the letter "P":

```
/golfers/child::golfer/child::favoriteCourses/child::course[starts-with
➥(@name,'P')]/attribute::city
/golfers/golfer/favoriteCourses/course[starts-with(@name,'P')]/@city
```

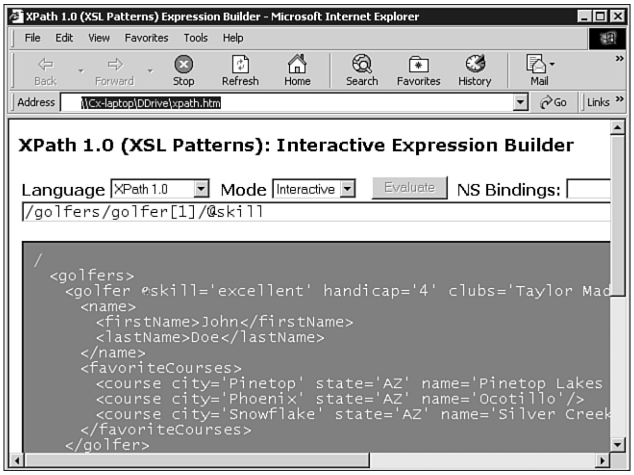
Select the first `golfer` element's `skill` attribute and capitalize it:

```
translate(/golfers/child::golfer[position()=1]/attribute::skill,
➥'abcdefghijklmnopqrstuvwxyz','ABCDEFGHIJKLMNOPQRSTUVWXYZ')
translate(/golfers/golfer[1]/@skill,'abcdefghijklmnopqrstuvwxyz',
➥'ABCDEFGHIJKLMNOPQRSTUVWXYZ')
```

Count the number of `golfer` elements:

```
count(/golfers/child::golfer)
count(//golfer)
```

After seeing the previous XPath statements, you may be wondering how you can test them to see if they actually work. You may also want to practice creating some of your own XPath statements. Fortunately, an excellent browser-based tool for testing XPath statements was created by Aaron Skonnard of DevelopMentor. The tool can be downloaded from `http://staff.develop.com/aarons/xmllinks.htm`. Figure 3.1 shows an example of the tool in action.


XPath 1.0 (XSL Patterns) Expression Builder - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address
Go
Links

\\Cx-laptop\DDrive\xpath.htm

XPath 1.0 (XSL Patterns): Interactive Expression Builder

Language XPath 1.0 Mode Interactive Evaluate NS Bindings:

```
/golfers/golfer[1]/@skill
```

```
/
  <golfers>
    <golfer @skill='excellent' handicap='4' clubs='Taylor Mad
      <name>
        <firstName>John</firstName>
        <lastName>Doe</lastName>
      </name>
      <favoriteCourses>
        <course city='Pinetop' state='AZ' name='Pinetop Lakes
        <course city='Phoenix' state='AZ' name='Ocotillo'/>
        <course city='Snowflake' state='AZ' name='Silver Creek
      </favoriteCourses>
    </golfer>
```

FIGURE 3.1
Using the XPath Interactive Expression Builder.

XPointer—Accessing XML Document Fragments

Earlier in the chapter it was mentioned that the XPath recommendation was jointly developed by several W3C working groups. One of these contributing groups is also responsible for the XPointer language (`http://www.w3.org/TR/xptr`). Their participation was beneficial not only to the XPath language, but also to their own because XPointer leverages XPath's capabilities.

XPointer stands for XML Pointer. When combined with XLink and XPath, the XPointer language opens up many exciting possibilities for information retrieval from dispersed XML documents. The W3C XPointer Candidate Recommendation makes the following statement about XPointer's purpose:

XPointer supports addressing into the internal structures of XML documents. It allows for examination of a document's hierarchical structure and choice of its internal parts based on various properties, such as element types, attribute values, character content, and relative position. In particular, it provides for specific reference to elements, character strings, and other parts of XML documents, whether or not they bear an explicit ID attribute.

Many of the concepts mentioned in this statement strongly resemble functionality already available in the XPath language, which may lead you to believe that much is duplicated between the two. However, this is not the case. XPointer simply extends XPath's functional utility, as you'll see in the following sections. Your current XPath knowledge will be applicable to many XPointer concepts.

XPointer Basics

At this point you may be wondering what the difference is between XPath and XPointer. Because both seem to allow access to specific items found within an XML document, can XPointer be used for something different? Obviously the answer to this question is "yes," given all the work that has gone into creating the language. XPointer differs from XPath because it provides a mechanism for accessing specific content, referred to as document fragments, located in local or remote documents. This content isn't limited just to nodes. XPointer even allows individual characters in a text string to be selected.

Imagine having an XML document containing research data that is composed of three sections, including an introduction, body, and conclusion. The body and conclusion portions of the document do not exist within this XML document. They are located externally within two XML documents that contain information other than simply body or summary text.

Using XPath alone, there is no mechanism for gaining access to the body and summary text in these remote documents. XPath is designed to locate specific nodes within a local XML document. This is where XPointer comes into play. Using portions of XPath syntax, XPointer allows the body and summary text sections to be gathered from the remote documents and placed into the current XML document. All this can be done without having to load the entire remote document (or documents in this case). Comparing this functionality to HTML, combining these languages allows you to link to a specific area of a document and "grab" its content without having to load the entire page and then scroll to the content as is done with HTML anchors.

The capability to access data down to the individual character level opens up a whole new world for data sharing and exchange. Before getting too much further into the details, let's cover a few key XPointer definitions:

Term	*Definition*
sub-resource	The portion of the XML document that XPointer identifies. Although the entire XML document would be considered the resource, the portion being accessed by XPointer would be the subresource.

point	The W3C defines a point simply as "A position within XML information." A point may represent a node within an XML document or simply a specific location within a node's XML character data. The combination of a container node and index value define a point's location. A point may be a position before any character or may precede or follow a node. A point can never exist on a character, though, only before or after it.
	As an example, the "T" character in the following document fragment has a container of type element and has an index of `0`. The "e" would have an index of `1` and the "d" character an index of `2`:
	`<golfer>`
	`<firstName>Ted</firstName>`
	`</golfer>`

character point	A point with a container node that has no child nodes. Examples include a point within text, comments, processing instructions, and so on.
node point	A point with a container node that has child nodes.
range	A selection of XML content located between two end points. When you highlight a section of text in a program (such as Word) using the cursor, the selected area could be considered a range because it has starting and ending points.
collapsed range	A range that has equal start and end points. For example, if we start a point at character three and end a point at character three, the ranged is considered to be collapsed.
location	Similar to a node in XPath. However, because XPointers can begin in one paragraph and end in another and do not have to specify nodes in particular, a location can be a node, point, or range.
location-set	This is similar to a node-set created by an XPath expression. However, aside from including nodes, a location-set can also include points and/or ranges.
singleton	This term is reserved for situations where one range, string range, or single node is located by an XPointer. Multiple areas of XML content identified by an XPointer are not considered to be singletons.
fragment identifier	An XPointer identifier (expression) that has an escaped URI. Escaped refers to replacing characters such as `<` with their URI escape value, which would be `&lt;` in this case.

Points and ranges are used when working with XPointer expressions. Points determine where to start and end a search for XML content within a document, and a range contains the XML content located between two points. The syntax to create XPointer expressions using two points is as follows:

```
http://www.someURL.com/someDOC.xml#xpointer(<point1> to <point2>)
```

This syntax would link to the range specified between `point1` and `point2` and load the entire document. You'll notice that this is similar to how the HTML anchor currently works, although XPointer allows for much more flexibility, as you'll see in the next few sections. After parsers support this feature, you'll be able to pick out a section of XML within a remote document with pinpoint accuracy.

How XPointer Extends XPath

To avoid duplicating functionality already available in the XPath language, XPointer leverages XPath where possible. However, it does extend XPath's functionality in the following ways:

- XPointer adds the point and range location types to the existing XPath node-set. As a result, an XPointer node test can include a test for a point or range in a location-set as well as the other node-set tests found in XPath.
- XPointer builds on the concept of an XPath node-set by adding the locator-set concept, which includes points and ranges in addition to the items included in an XPath node-set.
- The context in which XPath expressions are evaluated are governed by XPointer rules.
- XPointer adds several new functions, including `string-range()`, `range-to()`, `range()`, `range-inside()`, `here()`, `origin()`, `start-point()`, `end-point()`.

XPointer Functions

The additional functions available in the XPointer language along with their corresponding descriptions are listed in Table 3.6. One interesting point about these functions is that they can be called after a `/` character in a location step (that is, `node/function()`). This is different from the XPath language where calling a function after the `/` character would normally result in the parser raising an error (exceptions to this exist, such as the `count() and sum()` functions to name just two).

Table 3.6 XPointer Functions

Function Name	*Purpose*
`range-to(expression)`	This function returns a range for each location in the context. The example returns the range from an element with an `id` of `1111` to another element with an id of 2222.

Table 3.6 continued

Function Name	*Purpose*
	Example: `xpointer(id("1111")/range-to(id("2222")))` This next example returns a range from the first `golfer` element to the third `golfer` element: Example: `xpointer(golfer[1]/range-to(golfer[3]))`
`string-range(location-set, string,number,number)`	This function searches the string-value of a location for a substring that matches the string argument and returns range locations for any occurrences. The first argument accepts a location-set (`//golfer`), and the second argument accepts the string to search for. The third and fourth number arguments are optional but function much like the number arguments found in a C# `Substring()` function. The third argument tells where to start the returned range, and the fourth argument specifies the number of characters in the range. The example would return a range starting with the fourth occurrence of every "XML" string found in the `title` element. The range will start just after the "L" character (four characters in) and contain the five characters that come after the fourth (represented by `[4]`) "XML" string: Example: `string-range(//title,"XML",4,5)[4]`
`range(location-set)`	This function returns ranges that match up with the locations specified in the location-set argument. The example would return ranges for all folder elements that have an attribute named path starting with the letter "D." The range starts just before the `folder` node's beginning tag and ends just after its ending tag: Example: `range(//folder[starts-with(@path,"D")])`
`range-inside(location-set)`	This function returns ranges that match up with the content specified in the location-set argument. This is similar to the `innerText` property found in DHTML, which also returns the text between a start and end tag. It differs from the `range()` function in that it includes only a node's content, not the node and its content. The example would return ranges for all folder elements that have an attribute named path starting with the letter "D." The range starts just after the folder node's beginning tag and ends just before its ending tag:

Table 3.6 continued

Function Name	*Purpose*
`range-inside(location-set)` (continued)	Example: `range-inside(//folder[starts-with(@path,"D")])`
`start-point(location-set)`	This function returns a location-set containing the starting point for each range within the location-set argument. Here's how the `start-point()` function handles different argument types: If start point is passed in, that point is returned unchanged. When a range is passed in, the starting point of the range is returned. When the argument is an element, some text, a comment, or a processing instructions, the argument is converted to a range and the starting point of that range is returned. If the argument is an attribute or namespace, a syntax error is signaled by the function. Example: `start-point(//folder)`
`end-point(location-set)`	This function returns a location-set containing the ending point for each range within the location-set argument. The details on this function are the same as the start-point function, except that an ending point is always returned rather than a starting point. Example: `end-point(//folder)`
`here()`	This function returns a location-set containing the location of the XPointer that is being evaluated. In simpler terms, the `here()` function returns the element containing the XPointer itself, either as a text element or an attribute. The example allows the XPointer expression to point to locations relative to the location of the XPointer expression itself. Example: `#xpointer(here()/books)`
`origin()`	This function is useful only when used within the context of XLink (discussed next) to link XML documents. It returns a location-set containing the location of the originating link.

XPointer Errors

Errors in XPointer can be handled differently than errors in XPath. In situations where an XPath expression finds no matching locations within the XML document, an empty node-set is returned and no error is raised. However, XPointer must be stricter with error handling because it allows access to remote documents. XPointer is required to raise an error if a fragment identifier (expression) returns no results. The following is a summary of the three errors that can be associated with XPointer and their associated description:

Error	*Description*
Syntax Error	This error occurs when an XPointer expression (also called a fragment identifier) contains incorrect syntax.
Resource Error	This error occurs when a resource document does not contain well-formed XML. The fragment identifier (XPointer expression) may be correct but be unable to create a point or range because of resource document problems.
Subresource Error	This error occurs when an XPointer expression is syntactically correct, the resource contains well-formed XML, but no locations are returned. As mentioned, this type of error does not occur in XPath, because empty node-sets are acceptable there.

XPointer Examples

As the XPointer language is moved to the W3C recommended status, more and more examples of using it with supporting parsers will become available. Until then, here are a few examples of using XPointer. The examples are prefixed by a remote resource name of www.someURL.com:

Identifying Fragments Using Bare Names

Select a node with an `id` of `chapter1` that is found in the document named `remoteDocument.xml`. This example shows how to use “bare names,” which is a shorthand method that saves you from having to use the `id()` function. You’ll notice that this is very similar to using an HTML anchor:

```
http://www.someURL.com/remoteDocument.xml#chapter1
```

This can also be written as:

```
http://www.someURL.com/remoteDocument.xml#xpointer(id("chapter1"))
```

Identifying Fragments Using Child Sequences

XPointer allows nodes to be accessed by providing child sequence numbers as shown next:

```
http://www.someURL.com/remoteDocument.xml#/1/3/4
```

The `#/1/3/4` syntax tells XPointer to go to `remoteDocument.xml` and access the first element found there. After finding this element, find its third child element. Now choose this element's fourth child element.

Identifying Fragments Using XPointer Expressions

If every element of every XML document contained an `id` type attribute (as defined in the DTD), accessing XML content would be much easier. However, because many XML documents don't have `id` attributes, a more detailed way of accessing XML content must be used. This is where XPointer expressions are useful. When bare name and child sequence methods of accessing XML content are not specific enough, an XPointer expression can be created. This expression will select the range between the first and fourth `golfer` nodes:

```
http://www.someURL.com/remoteDocument.xml#xpointer(/golfers/golfer[1]/
➥range-to(/golfers/golfer[4]))
```

The following example will pull the content between the first and third `sibling` nodes:

```
http://www.someURL.com/remoteDocument.xml#xpointer(//father/sibling[1] to
➥//father/sibling[3])
```

The next example will return all ranges within the `/golfers/golfer/name` context that contain a substring equal to "Joe":

```
http://www.someURL.com/remoteDocument.xml#xpointer(string-range(/golfers/
➥golfer/name,"Joe"))
```

Finally, the following document will first try to find a `golfer` element's fourth `name` node. If that location cannot be found, it will try to find the last `name` node:

```
http://www.someURL.com/remoteDocument.xml#xpointer(//golfer/name[4])
➥xpointer(//golfer/name[last()])
```

XLink—Resource Relationship Management

The XLink specification (`http://www.w3.org/TR/xlink/`) adds several additional pieces of functionality that make it easier to access external resources and maintain complex linking relationships. At press time, the XLink specification was in "Proposed Recommendation" status, so the majority of the concepts covered in this section are likely to remain unchanged. After the specification reaches "recommended" status at the W3C, you should start to see parsers or applications on the market that support XLink.

You may recall a short discussion we had concerning HTML linking limitations at the beginning of this chapter. Although HTML links work quite well in a browser, some of the limitations found in the HTML link include the following:

- HTML allows links only between two resources.
- HTML links have to be embedded in the originating link's file. This can cause maintenance headaches.
- HTML anchors can help locate specific information within a document but require that the entire page be loaded. This wastes bandwidth.
- HTML links are unidirectional. They can point only in one direction.
- The HTML link has few attributes for customizing linking capabilities based on different linking needs.
- Updating a target document with anchors is not possible without having "write" access to that document.
- HTML links require specific element tags to work (`<a>` for example). This makes it difficult to add linking capabilities to other elements.

A few of these limitations are addressed by the XPointer language because it allows specific content fragments to be accessed in remote XML documents. Although this is a great step forward, the old HTML link can be improved on in many other ways. XLink strives to make these improvements a reality by adding new linking capabilities to documents that comply with the XML version 1.0 recommendation.

XLink Basics

The XLink specification doesn't actually mention linking different documents. Rather, it defines a structure that consists of links between "resources." These resources could be XML documents, media files (images, movies, or music), database information, or any other type of resource that needs to be referenced through a link.

Adding more functionality to any type of established item, such as the HTML link, is oftentimes associated with an overall increase in complexity. This is not necessarily the case with XLink. After you understand its keywords, elements, attributes, and functionality, you'll see that it looks just like regular XML.

Although the XLink language allows you to define links the same way as you do in HTML, it does bring with it more advanced attribute and element definitions that help in linking two, three, or more resources in a uniform manner. Understanding the various attributes and definitions that come with the XLink language is important to leverage its linking power.

Before learning about the attributes and elements that exist in XPath, let’s take a look at some examples of what XPath syntax looks like. First let’s write out the syntax for what is coined a “simple link.” A simple link functions the same as a link found within an HTML page:

```
<golfer xmlns:xlink="http://www.w3.org/1999/xlink/">
    <name xlink:type="simple" xlink:href="http://www.someURL.com/row.aspx">
       John Doe
    </name>
</golfer>
```

The following is an example of an extended link containing four different resources. This type of linking is not possible using HTML’s anchor tag:

```
<xlink:extended
  xmlns:xlink="http://www.w3.org/1999/xlink/"
  role="foursome"
  title="Golfing foursome"
>
  <xlink:locator href="dad.xml"
    role="golfer1"
    title="First golfer in foursome"
  />
  <xlink:locator href="aaronhorne.xml"
    role="golfer2"
    title="Second golfer in foursome"
  />
  <xlink:locator href="toddwahlin.xml"
    role="golfer3"
    title="Third golfer in foursome"
  />
  <xlink:locator href="michellehorne.xml"
    role="golfer4"
    title="Fourth golfer in foursome"
  />
  <xlink:arc from="golfer1"
    to="golfer2"
    actuate="onRequest"
    show="embed"
  />
  <xlink:arc from="golfer3"
    to="golfer4"
    actuate="onRequest"
    show="embed"
  />
</xlink:extended>
```

I'll defer an in-depth discussion of the different elements and attributes being used here until later in the chapter. You can see, however, that after you understand a few basic concepts concerning new elements and attributes defined in the XLink specification, implementing the XLink language isn't much different than working with normal XML documents.

XLink Keyword Definitions

Table 3.7 contains many XLink keywords that will become useful as you learn more about creating links.

Table 3.7 Keywords in XLink

Term	*Description*
Arc	An arc specifies how links are traversed. In the case where multiple resources are being linked, the arc can specify which direction to follow between links with the to and from attributes. In the case of a simple link, the arc is not necessary because it's obvious which direction the link flows.
Ending-resource	In cases where two resources are specified to have a linking relationship, this is the resource identified in the `to` attribute. Example: `<xlink:arc from="resource1" to="resource2"/>`
Extended link	An extended link associates many local and/or remote resources together. The link is considered inline if any of the resources are local and considered out-of-line if none are local.
Inline link	Defined by the W3C as "A link where some content in the `linking element` serves, by virtue of its presence inside the linking element, as a `participating resource.`" Use of the `<a>` tag in HTML is considered an inline link.
Locator	Data used to identify a resource in a link. Identification occurs by specifying a URI or other type of identity.
Linkbase	A file external of any resource that contains an extended link. This type of file is similar to an include file in ASP.NET because it helps maintain specific information referenced by many resources in one location.
Multidirectional link	A multidirectional link can be started by any of the resources participating in the link. As such, the HTML link would be excluded from this definition because it can initiate only a one-way link (hitting the Back button doesn't count).

Table 3.7 continued

Term	*Description*
Out-of-line link	When a link starts from a resource other than the document containing the link, the link is considered out-of-line. This type of link includes only resources that are remote. An out-of-line link exists when a document not under your control (you do not have write permissions to it) is included in a link. This type of link is necessary for remote links to initiate other links.
Simple link	A simple link is very similar to a normal HTML link in that it specifies a one-way link between two resources. Because one of these two documents is always local, simple links are always considered to be inline links.
Traversal	A traversal occurs when a resource identified in a link is accessed. Traversal can be started by a user or from an application.

Now that you're familiar with a few of XLink's definitions and keywords, let's take a look at some attributes that are used in building simple and extended links.

XLink Attributes

The XLink language specifies several attributes that add various linking capabilities. A listing of these attributes with their corresponding description can be found in Table 3.8.

Table 3.8 XLink Attributes

Attribute	*Description*
`xlink:arcrole`	The `arcrole` attribute is a semantic attribute used to describe the meaning of a resource (or a property of a resource) based on the context the link is viewed in. It must contain a URI reference. The W3C explains the `arcrole` attribute in the following manner: *A resource might generically represent a "person," but in the context of a particular arc it might have the role of "mother" and in the context of a different arc it might have the role of "daughter."*
`xlink:href`	Specifies the location to a remote resource. The XLink specification defines this as containing a URI reference.

Table 3.8 continued

Attribute	*Description*
`xlink:label`	This traversal attribute is used to provide a way for `arc` elements to identify a `resource` or `locator` element based on its label. It can be used only on the `resource` or `locator` elements. This is similar to using the `id=". . . "` attribute on HTML elements when working with DHTML.
`xlink:type`	This determines the link type to be used. Possible values include `simple`, `extended`, `locator`, `arc`, `resource`, `title`, `none`. You'll learn more about what the `extended`, `locator`, `arc`, and `title` values accomplish. As far as the `resource` value, it can be used as an attribute value to define link participants that are available locally (rather than remotely). Content enclosed within the resource type element would be considered "local."
`xlink:role`	This attribute must contain a URI reference and is used to indicate a property that a resource has.
`xlink:title`	This attribute contains content that is intended to be read by a human. The `title` attribute helps give information about the link.
`xlink:show`	This determines how content identified by a link will be shown. The possible values are `replace`, `embed`, `new`, `other`, and `none`. The `replace` value functions in the same manner as linking between two pages in HTML, whereas the `embed` value would be similar to the function the `<img>` tag element performs.
`xlink:actuate`	The `actuate` attribute specifies how the link should be traversed from a timing perspective. Rather than always being triggered by human interaction, the link can be set to be triggered by an application. Possible values include `onRequest`, `onLoad`, `other`, and `none`.
`xlink:to`	This attribute is used only with an `arc` element in an extended link to specify the end point of a link. Its value is equal to the `label` attribute on a `resource` or `locator` type element.
`xlink:from`	This attribute is used only with an `arc` element in an extended link to specify the start point of a link. Its value is equal to the `label` attribute on a `resource` or `locator` type element.

The XLink Simple Link

The HTML link that you're already familiar with is actually an example of a simple link in the XLink language. It provides a way for a single document to link to another document. This type of link is unidirectional because the initiating page knows about the destination page, but the destination page is not actually participating in the link. The destination page simply displays its content in the browser. It's true that you could hit the browser's Back button, but the destination document still isn't actually participating in the link. Because the simple link includes a local resource (one in which you created the link), it is always considered to be an inline link. Though limited in flexibility, the simple link has helped place the Web into the limelight and has made it an essential resource for businesses and individuals alike.

Adding a simple link to an XML document that is part of an XLink-aware application is as easy as adding a few attributes to an existing XML element. Note that the XLink qualified namespace must be added to the XML document:

```
<?xml version="1.0"?>
<root xmlns:xlink="http://www.w3.org/1999/xlink>
    <book>
        <chapter xlink:type="simple" xlink:href="http://www.thisBook.com">
            Chapter 1
        </chapter>
    </book>
</root>
```

The `xlink:type` attribute is specified as "simple" and the `xlink:href` lists the link URI. This method of declaring a link is more efficient than the current HTML linking implementation. To understand why, imagine trying to add linking capability to a `<p>` tag in HTML without wrapping it with the `<a>` and `</a>` tags (or using scripting). Using attributes allows a link to be added to any element quite easily.

Although you learned a little about XLink attributes earlier, let's take a look at two interesting attributes that should prove to be very useful in applications that implement XLink in the future. The `actuate` attribute can take on values of `onLoad`, `onRequest`, `other`, and `none`. The `onRequest` value means that the link is traversed when the user requests it. The `onLoad` value is used to traverse a link immediately after a document completes loading. When the `show` attribute is equal to `embed` and `actuate` is equal to `onLoad`, a useful way of embedding content from outside resources can be accomplished. When combined with XPointer, these technologies could be used to create the research XML document discussed earlier that linked to introduction, body, and summary sections located in remote documents.

XLink Extended Links

Extended links perform the obvious task of "extending" the capabilities of a simple link by allowing multiple resource links to be managed rather than just one. Like the simple link, extended links can be created by using XLink attributes placed on existing XML elements. The W3C gives the following definition:

An extended link is a link that associates an arbitrary number of resources. The participating resources may be any combination of remote and local.

The XLink specification also mentions that if all the resources are remote (not editable), the link is considered to be out-of-line. If any of the identified resources are local (editable), then the link is considered to be inline.

Figure 3.2 shows a simple representation of an extended link.

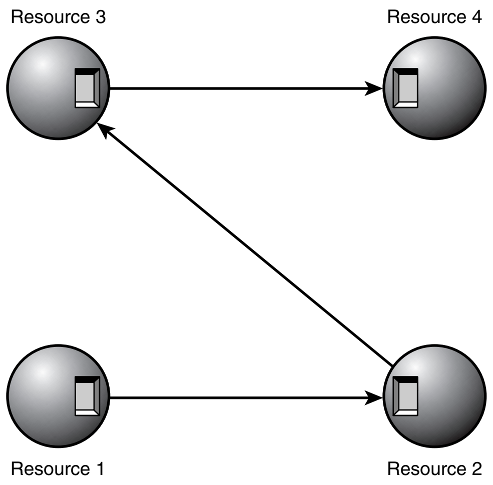


FIGURE 3.2
Extended link relationships.

At first glance, this figure may look very similar to simple links between resources, as in HTML. However, the extended link includes all these resources in a relationship and then specifies how the links should be traversed. Listing 3.2 shows how these links could be written using extended link XPath syntax. It's important to know that the `extendLink` and `loc` element names were chosen to make the sample easier to understand. These element names are not part of the XLink specification. In fact, any element name could be used.

Listing 3.2 An Extended Link

```
<extendedLink xlink:type="extended"
   xmlns:xlink="http://www.w3.org/1999/xlink"
   role="http://www.someURL.com/resources/roles"
   title="Resource Manager">
   <loc xlink:type="locator" href="resource1.xml" label="resource1"
      title="resource1 contents"
   />
   <loc xlink:type="locator" href="resource2.xml" label="resource2"
      title="resource2 contents"
   />
   <loc xlink:type="locator" href="resource3.xml" label="resource3"
      title="resource3 contents"
   />
   <loc xlink:type="locator" href="resource4.xml" label="resource4"
      title="resource4 contents"
    />
</extendedLink>
```

The `extended` element type shown in Listing 3.2 acts as a container for four different resource links. It contains some of the same attributes you saw in the `simple` element type shown earlier, and it also declares the XLink qualified namespace. Inside this container are four locator elements that specify the URI to each resource. Going back to Figure 3.2, the small boxes located on each resource would represent that resource's locator.

Although the code shown in Listing 3.2 specifies the locators for each resource and groups each resource in an extended container, it does not say how links between the resources should be traversed. Should Resource 1 link to Resource 3? Should Resource 4 link to Resource 3? The listing doesn't mention anything about the way the resources should be linked.

To accommodate the need for specifying how links should be traversed, the XLink language introduces the `arc` concept. An arc can be thought of as a traversal path that specifies how to link `from` one resource and `to` another. Listing 3.3 adds on to Listing 3.2's code by adding information about link traversals through the use of `arc` element types.

Listing 3.3 An Extended Link with Arcs

```
<extendedLink xlink:type="extended"
   xmlns:xlink="http://www.w3.org/1999/xlink"
   role="http://www.someURL.com/resources/roles"
   title="Resource Manager">
   <loc xlink:type="locator" href="resource1.xml" label="resource1"
      title="resource1 contents"
   />
```

Listing 3.3 continued

```
    <loc xlink:type="locator" href="resource2.xml" label="resource2"
       title="resource2 contents"
    />
    <loc xlink:type="locator" href="resource3.xml" label="resource3"
       title="resource3 contents"
    />
    <loc xlink:type="locator" href="resource4.xml" label="resource4"
       title="resource4 contents"
    />
    <go xlink:type="arc" from="resource1" to="resource2" show="replace"
       actuate="onRequest"
    />
    <go xlink:type="arc" from="resource2" to="resource3" show="replace"
       actuate="onRequest"
    />
    <go xlink:type="arc" from="resource3" to="resource4" show="replace"
       actuate="onRequest"
    />
</extendedLink>
```

By specifying the `arc` path, you now know the locations of four resources as well as how each link between the resources should be traversed. The arc elements listed previously tell the extended link that `resource1` links to `resource2`, which links to `resource3`, which links to `resource4`. If you had wanted `resource4` to link back to `resource1`, you could have applied another XLink `type` attribute to an element and specified the additional arc path using the `from` and `to` attributes.

Any variety of `arc` paths can be specified as long as all paths are unique. All this is done without having to actually add a physical link inside of `resource3` that goes to `resource4`, as is the case when you link to a resource that you have no control over. Exactly how will `resource3` link to `resource4` if the link isn't physically embedded in `resource3`? To answer this question completely, we will have to wait for the first XLink-capable applications to appear on the scene and analyze how this feature is implemented.

One interesting point made in the XLink specification is that if a `from` or `to` attribute is missing in an arc type element, an arc will be created in the appropriate direction to all locator type elements with `label` attributes located within the extended link grouping. To clarify this less intuitive concept, the W3C provides the following example:

```
<extendedlink xlink:type="extended" xmlns:xlink="http://www.w3.org/1999/xlink">
  <loc xlink:type="locator" xlink:href="..." xlink:label="parent"
➥xlink:title="p1"/>
```

```
  <loc xlink:type="locator" xlink:href="..." xlink:label="parent"
➥xlink:title="p2"/>
  <loc xlink:type="locator" xlink:href="..." xlink:label="child"
➥xlink:title="c1"/>
  <loc xlink:type="locator" xlink:href="..." xlink:label="child"
➥xlink:title="c2"/>
  <loc xlink:type="locator" xlink:href="..." xlink:label="child"
➥xlink:title="c3"/>
  <go xlink:type="arc" xlink:to="child" />
</extendedlink>
```

Based on each locator type's title attribute, the following arcs would be assumed because there are missing `from` and `to` attributes in the `arc` type element:

> `p1-c1`, `p1-c2`, `p1-c3`, `p2-c1`, `p2-c2`, `p2-c3`, `c1-c1`, `c1-c2`, `c1-c3`, `c2-c1`, `c2-c2`, `c2-c3`, `c3-c1`, `c3-c2`, and `c3-c3`

As you can see, this rule causes a type of cross-join to occur that involves each locator type element. Although all possible arcs are pursued, the uniqueness of each arc is maintained.

The extended link automatically allows links between resources to be organized and grouped. For this extended link to be useful, it would need to be embedded in one of the resources. This is called an inline extended link because the resource that contains the embedded link information would be editable, in contrast to a third-party read-only resource. After the extended link is inserted, the relationships between the different resources specified in the `locator` type elements can be managed.

Linkbases and External Linksets

In a perfect world, links within an HTML document have the capability to be changed without having to edit the actual document itself. HTML alone does not allow for this, although you could, of course, maintain all links for an ASP.NET page in a database or XML document if you desired. The XLink authors knew that maintaining links in HTML is difficult and they built in the capability for resource links to be separated from the actual resources. A file referred to in the XLink specification as a "Linkbase" can be in a location that is separate from the actual resources. The contents of this file would contain one or more extended links. Because the extended link isn't actually found within a resource, it is called an "external linkset." Figure 3.3 depicts this.

As Figure 3.3 shows, each resource is tied together by the external linkset. This linkset contains locator information about each resource and arc statements that specify how the links should be traversed, just as you saw earlier. Separating the actual link information from the resources is useful because it allows you to manage a list of links externally making overall maintenance much easier. To add a new link, you would simply add the resource's locator to the external linkset document and provide an arc. No other documents need updating!

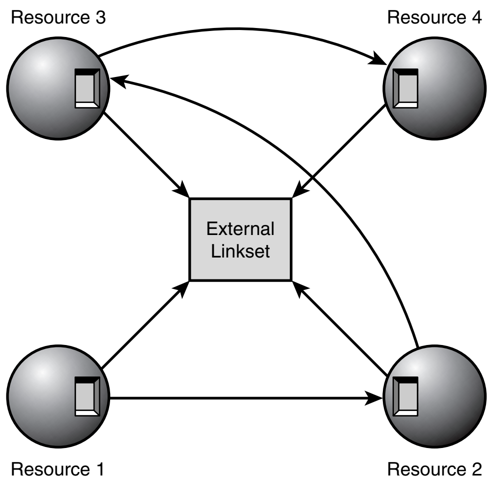


FIGURE 3.3
An out-of-line extended link.

For an external linkset document to be used, a resource needs to know where the external linkset document is located. This is accomplished by placing code similar to the following into an inline resource:

```
<basesloaded xlink:type="extended" xmlns:xlink="http://www.w3.org/1999/xlink">
  <startrsrc xlink:type="locator" xlink:label="spec" xlink:href="spec.xml"/>
  <linkbase xlink:type="locator" xlink:label="linkbase" xlink:href=
➥"linkbase.xml"/>
  <load xlink:type="arc" xlink:from="spec" xlink:to="linkbase"
    xlink:actuate="onLoad"
    xlink:arcrole="http://www.w3.org/1999/xlink/properties/linkbase"
  />
</basesloaded>
```

Maintenance of links suddenly becomes much easier through the use of linkbases and external linksets.

Putting XLink Together

Determining when to use the different XLink attributes can be somewhat confusing. Fortunately, the W3C created the following information table that specifies when attributes are required (R) and when they are optional (O). To see which attribute can be used with a given element type, look down each column. If a cell is empty, you can't use the attribute with the element type. Refer to the table when you can't remember much because it's late at night and you're feverishly working to finish up an XLink project!

	simple	extended	locator	arc	resource	title
type	R	R	R	R	R	R
href	O		R			
role	O	O	O	O	O	
arcrole	O			O		
title	O	O	O	O	O	
show	O			O		
actuate	O			O		
from				O		
to				O		
label			O		O	

Summary

In this chapter we covered a lot of technologies related to XML. Although the XPointer and XLink languages have yet to be implemented fully, they are promising technologies that will make developing with XML even more powerful. In Chapters 6 and 7, you will see how useful the XPath language can be to access data contained within an XML document using the DOM or XSLT. However, before jumping into those chapters, Chapter 4 takes a look at XML DTDs and schemas.