Web Services and XML

IN THIS CHAPTER

• XML as a Wire Representation 76
• Querying XML Elements Using XPath 79
• Essential XML 80
• URIs and XML Namespaces 84
• XML Schemas 87
• Identifying XML Elements Using Xlink 99
• XML Transformations 101
• .NET's XML Architecture 107
XML is critical to Web Services. It’s a fundamental technology; without it, you wouldn’t have Web Services—at least, not as they’re implemented today.

This is true for two main reasons. First, XML is loosely coupled. That is, XML enables you to loosely bind the client and server, making it easier to change one or both, as well as add versioned information after the initial release. Second, XML is easily interpreted, making it highly interoperable. XML is simply text, so if you can interpret text on your computer system, you can interpret XML. You’ll find XML in use in nearly every contemporary computer system available today, and perhaps even in some venerable systems that you used yesteryear. There is a great deal more to it than that, but, in a nutshell, XML is just text in a file or network packet.

**NOTE**

My observation that XML is merely text is inarguably true. However, XML is becoming increasingly Unicode-aware. Most of the XML documents that you’ll see in use today are encoded using either UTF-8 (single-byte Unicode) or UTF-16 (traditional 2-byte Unicode). Therefore, your system needs Unicode capability to truly handle these contemporary XML documents. .NET provides you with this capability, even if you’re running on a 16-bit version of Windows, such as Windows 98.

The goal of this chapter is to introduce you to the fundamentals of XML and to dig deeper into XML to show how it plays a role in Web Service implementation. If you’re new to XML, Appendix D, “.NET Web Service Resources,” points to alternative sources of information that you can examine on your own. If you do know something about XML, you might want to scan this chapter for information that’s new to you and skip the parts that you know well. With luck, you’ll find something interesting and new, and you’ll expand your understanding of XML, with both Web Services and .NET’s handling of XML in general.

The true goal of this chapter is to make sure that you understand how XML fits into Web Service processing and to show you how to work with XML using the .NET class hierarchy. You might find that you’ll need to modify the XML that your Web Service uses for some reason. If so, you’ll use what you’ve learned from this chapter. We’ll start with a discussion of XML as a wire representation, which should explain why XML is so important to Web Services.

**XML as a Wire Representation**

Whenever you transmit information over a network, that information is ultimately transformed from a binary representation in your computer’s memory into another representation that was designed for network use. Perhaps the network representation is a highly efficient binary one,
or perhaps its purpose is more general. In any case, you’ll probably find that the information’s binary format in memory is very different than its network form. The form that it takes for network transmission is called its wire representation. This is the form that the data takes when it’s transmitted over the network, no matter what the network medium is.

Wire representations, and protocols in general, are often designed to meet specific design criteria, a few of which are listed here:

- Compactness
- Protocol efficiency
- Coupling
- Scalability
- Interoperability

Compactness refers to how terse the network packet becomes while still conveying the same information. Small is usually best. Protocol efficiency is related to compactness—you rate efficiency by examining the overhead required to send the payload. The more overhead you require, the less efficient the protocol is.

A protocol’s coupling, loose or tight, tells you how flexible consumers of the protocol will be if you change things. Loosely coupled protocols are quite flexible and easily adapt to change, while tightly coupled protocols will most likely require significant modifications to both the server and existing clients. Tightly coupled protocols, for example, are those that require (or force) such things as the same in-memory representation or the same processor type to avoid endian issues (byte ordering in multibyte values). Loosely coupled protocols avoid this altogether by abstracting the information to a degree that makes the byte order irrelevant.

Converting an integer represented in big or little endian looks the same when represented as a string. The byte order conversion is made by the software handling the protocol, not by the protocol itself.

Scalability addresses the protocol’s capability to work with a large number of potential recipients. Some protocols are limited to a handful of consumers, while others handle millions of users easily.

Finally, interoperability speaks to the protocol’s acceptance on a variety of computing platforms. Will you have to issue network packets to one specific operating system or platform, or is the protocol a bit more general-purpose, enabling you to send information to a wider variety of systems?

Protocols, including XML, generally lie within a continuum of these characteristics. Highly efficient protocols tend to not scale well. Interoperable protocols tend to scale well but are often not as efficient as proprietary protocols. No single protocol does it all, and network
engineers often make design decisions based upon these and many other criteria. Which protocol you choose depends upon where the protocol falls in the continuum.

XML is both loosely coupled and highly interoperable. In fact, XML is so interoperable that it is nearly ubiquitous. You can send XML to anyone on the planet—not only will that person receive the information, but he’ll also be able to interpret and make use of it in nearly every case.

With XML, you pay for loose coupling and interoperability with protocol efficiency and compactness. XML is actually rather inefficient, although you can tailor this by judiciously choosing your element names (fewer characters in element names often yields more efficient XML, even though terseness of the XML was not a design goal of the inventors). Nor is XML particularly compact. XML is text, and, as with any text document, you might be either conservative with your expressiveness or creative yet a bit more verbose. In either case, you are still left with a loosely coupled and interoperable mechanism for encoding information to be transmitted over the wire.

**XML and Loose Coupling**

If XML isn’t terribly efficient, why use it? Because XML is simply text formatted in a specific way. You could make the same argument about HTTP, SMTP, POP3, NNTP, and a host of other such Internet protocols, but they’ve also proven to be successful. XML works, and it works well—when you transmit information using XML, you’re sending text rather than a proprietary protocol that has arbitrary design limitations. DCOM, for example, uses a fairly efficient wire representation but requires the object’s server to keep track of the client. To do this, the DCOM client issues “I’m still here” messages to the server every two minutes. After three periods without one of these ping messages, the server chops off the object’s head and reclaims the resources (memory and such). But imagine making changes to the DCOM wire representation and then trying to field that worldwide… .

Because XML is nothing more than text, with no claims made to object status or association, changes to your XML are rather easily accepted. You still send and receive text; it’s just that the element names might have changed or the document layout might have been slightly altered.

**XML and Interoperability**

The fact that XML is so loosely coupled has undoubtedly been a contributing factor to its wide acceptance throughout the Internet. Nearly every major computing platform available today has some capability to accept and interpret XML. XML is text—and, if your computer can handle text, your computer can handle XML. Palm PCs, cellular telephones, desktop and laptop PCs, and the largest of mainframes all have XML processing capability.
If this is the case, you should expect to be able to send anyone an XML document and know that the recipient can make use of the information. Because this is so, SOAP architects chose XML as the encoding mechanism for the SOAP protocol. XML is a rich and expressive technology that readily lends itself to method parameter serialization. Best of all, everyone understands what has been serialized, regardless of computing platform, and can make use of it.

As you probably already know, it’s one thing to create the XML document. It’s quite another to interpret the contents using an automated system. You’ll require some mechanism to reach into the XML to extract the portions of interest. That mechanism is called XPath. Here we’ll move away from “Why is XML good?” toward “How do I use XML?”

**Querying XML Elements Using XPath**

The previous section “XML as a Wire Representation” provides you with a few reasons why we’ve seen explosive growth of XML in the past few years, and it told why XML was selected to be a major component of the SOAP protocol. Another factor is the many peripheral XML technologies that make using XML even more attractive. Certainly one of these technologies is XPath, which is the language that you use to query an XML document for specific information.

If you work with XML, you’re probably somewhat familiar with XPath. If not, here is the basic idea. XPath serves the same purpose as the Structured Query Language (SQL) for database access. With SQL, you establish a search expression and provide that to the database. The database, in turn, searches its tables for the data that you requested and provides you with the results. XPath also establishes search patterns, but instead of accessing a database, you apply the XPath query against an XML document that is contained within an XML processor.

XML processors are software components that, when provided an XML document, expose a programmable interface that enables you to work with the XML document in an automated manner. Typically XML processors expose the XML Document Object Model (DOM) or perhaps the Simple API for XML (SAX). The DOM enables you to deal with the XML document as if it were composed of tree nodes. SAX, on the other hand, offers a stream-based approach to accessing the XML information. You take what you need from the stream and discard the rest. (.NET implements another model that you’ll examine in the later section “.NET’s XML Architecture.”)

Because this book is focused on Web Services, we won’t go into much detail regarding DOM or SAX, which are general XML topics that merit their own book. If you need a brush-up, or if XML is new to you, I’ll explain the basics here. For more information, see the references included in Appendix D.
Essential XML

The engine that drives the Web Service is the Simple Object Access Protocol, or SOAP. SOAP’s power is centered on the intrinsic behavior of XML and related XML technologies. To fully appreciate SOAP and its capabilities, it’s important to have a good understanding of XML. It is beyond the scope of this book to teach you all you might need to know to use XML effectively. However, it would be unreasonable to assume that you understand all the emerging technologies associated with XML because several of these are still ongoing efforts. Therefore, this section briefly reviews XML to provide the basis for discussion of the newer aspects to XML.

Documents, Elements, and Attributes

An XML document is really just a collection of data consisting of both physical and logical structure. Physically, the document consists of textual information. It contains entities that can reference other entities that are located elsewhere in memory, on a hard disk, or, more importantly, on the Web. The logical structure of an XML document includes processing instructions, declarations, comments, and elements. XML documents contain ordinary text (as specified by ISO/IEC 10646) that represent markup or character data.

The general form of an XML document looks something like this:

```xml
<?xml version="1.0" ?>
<Car Year="2002">
  <Make>Chevrolet</Make>
  <Model>Corvette</Model>
  <Color>Gunmetal</Color>
</Car>
```

You know that this is an XML document because of the XML declaration on the first line that tells you that this document conforms to XML version 1.0. In this example, `<Car/>` is the root element or document element of this XML document, `<Make/>` is just one of the child elements contained within `<Car/>`, and `Year` is an attribute of the `<Car/>` element.

Any text document is considered a well-formed XML document if it conforms to the constraints set forth in the XML specification. For example, one very important constraint is the limitation to one and only one root element in a document. An XML document is considered valid if it is well formed, if it has an associated Document Type Definition (DTD) or XML Schema, and if the given instance document complies with this definition. The DTD or XML Schema documents act as a template that the associated XML document must precisely match. If not, there is a problem with the formatting of the instance document, and the entire document is considered not valid.
It is common to find *processing instructions* embedded within the document, but they are not considered part of the document’s content. They are used to communicate information to application-level code without changing the meaning of the XML document’s content.

The following notation denotes the syntax of *processing instructions*:

```xml
<?target declaration ?>
```

The processing instruction contains a target followed by one or more instructions, where the target name specifies the application to which the processing instruction is applied. A common target name found in XML documents is the reserved target `xml`. This enables the XML document to communicate instructions to XML parsers. The most common (but optional) processing instruction used in XML documents is the XML declaration itself.

Recall that an *element* consists of a *start tag*, an *end tag*, and a *value*. But what if you have no value? Do you omit the entire XML element? You could, but it’s also useful information to know that an element *could* be there but, in this particular case, you have no value. Rather than using a start and end tag when you have no value to include, you can combine the tags to form an *empty-element tag*:

```xml
<Car Year="2002">
  <Make>Chevrolet</Make>
  <Model>Corvette</Model>
  <Color>Gunmetal</Color>
  <VehicleID/>
</Car>
```

**NOTE**

XML is sensitive to case and whitespace when creating element names. Whitespace is never legal in tag names, and tag names that are spelled the same but differ in alphabetical case represent *different* XML elements.

A parent element can theoretically contain an infinite number of child elements, and the same child element can appear multiple times under its parent element as siblings:

```xml
<Car>
  <Name>Corvette</Name>
  <Name>Speedy</Name>
  <Color>Red</Color>
  <!-- ...etc... -->
</Car>
```
Also, the same element name can appear under different parent elements. In this case, the element `<Name>` appears under the `<Car>` element as well as the `<SoundSystem>` element:

```xml
<Car>
  <Name>Corvette</Name>
  <SoundSystem>
    <Name>Bose</Name>
  </SoundSystem>
</Car>
```

You are not allowed to overlap tags within an XML document. The following document would not be considered a well-formed XML document because the `<Color>` element starts before the `<Name>` element ends:

```xml
<Car>
  <Color>3721-<Name>Red</Name>Corvette</Name>
</Car>
```

This example is trying to specify that the color of the car is 3721-Red and the name of the car is RedCorvette. However, this does not meet the XML specification constraints. Instead, it should be rewritten as follows:

```xml
<Car>
  <Color>3721-Red</Color><Name>RedCorvette</Name>
</Car>
```

Elements can be embedded within values of other elements:

```xml
<Car>
  <Name>Corvette</Name>
  <Base>MSRP<Price>39,475</Price>with options</Base>
</Car>
```

In this case, the `<Price>` element was embedded between the first part of the `MSRP` value and the last part of the `with options` value. This XML document encoding style is very much discouraged in general practice, however. The most common way to logically view an XML document is as a tree, and having elements embedded within values muddies this model.

*Attributes* provide more specific information about a particular element. Choosing between using an attribute and using an element can sometimes be a difficult decision. In a lot of cases, either form will work. One approach is to use attributes to denote element classifications based on the problem domain. Or, consider the attribute as a way to insert metadata that tailors the element in some way. Another aspect to using attributes deals with ease of access to data. If you always want to obtain the car’s color every time you encounter a `Car` element, then `Color` might be a good candidate for an attribute.
Entity References and CDATA

It is not uncommon for character data to contain characters that are used in XML constructs. The following XML does not conform to the XML specification:

```xml
<Car Year="2002 "The Sleekest Vette Yet"">
  <Name>Corvette</Name>
</Car>
```

The additional quotes in "The Sleekest Vette Yet" corrupt the syntax of the root element. Proper use of entity references allows you to instruct parsers to treat data as character data. The preceding example should be changed to this:

```xml
<Car Year="2000 &quot;The New Millennium&quot;"> 
  <Name>Corvette</Name>
</Car>
```

Here, the quotes have been replaced by their entity reference and will now be correctly parsed.

The double quote is but one of five characters that must be replaced with their entity references. The other four include the apostrophe (single quote), the ampersand, and the “less than” and “greater than” brackets. This makes sense because the quote characters are used to encapsulate attributes, the ampersand denotes an entity reference, and the brackets form XML element tags. If you use these characters within general text element values, you confuse the XML parser.

CDATA is an alternate form of markup that is better for larger quantities of text to be explicitly described as character data:

```xml
<Car Year="2002">
  <Model>Corvette</Model>
  <Description><![CDATA["I bet it's fast!"]]></Description>
</Car>
```

Rather than using entity references for each individual character, you can specify that an entire block of text should be treated as character data. This is a nice benefit because you do not need to replace each special character with its entity reference, which can be costly in terms of string manipulations. You simply wrap the text data with the CDATA tag.
There are two limitations to CDATA, however. CDATA sections cannot be nested within one another, and you cannot use CDATA within attribute values. CDATA sections cannot be nested because, by XML specification, the end tag of the enclosing CDATA section is considered to be the end tag for the CDATA section. If there were more text after this tag, the XML parser would become confused and would return to you an error when the document was parsed. For attributes, it is not legal XML syntax to specify XML elements within attribute values, and CDATA falls within that ruling. If you happen to have textual data that contains some of the XML special characters, you should use their respective entity reference values instead.

**URLs and XML Namespaces**

URI, or *uniform resource identifier*, is a generic term used to identify some particular entity or object in the Web world using its string representation. This is the most fundamental addressing scheme of the Web. A perfect use for this uniqueness deals with naming XML elements and attributes so that they don’t conflict with one another. This section reviews the characteristics of URIs and namespaces, and describes how they are related when used with your XML documents.

**URLs and URNs**

Different types of Web resources require different forms of URIs. Specifically, uniform resource locators (URLs) and uniform resource names (URNs) are both forms of a URI (see Figure 3.1). Each has its own syntax designed to fulfill a purpose.

**Figure 3.1**

The relationship between URIs, URNs, and URLs.

URLs are a form of URI used to locate a particular resource in the Web world. Basic URL syntax (see RFC 1738) is dependent on the *scheme* to which it applies, but it follows this format:

```
<scheme>:<scheme-specific syntax>
```

One form of a URL is used to specify a Web page located on a Web server, similar to this:

http://www.endurasoft.com/educenter.htm
In this case, the scheme indicates that the HTTP protocol is to be used to retrieve the HTML text for a particular Web page. By changing just one of the values in this URL, you are specifying a completely different location, resource, or both on the Web.

URNs are another form of URI that provides persistence as well as location-independence. In a nutshell, a URN uniquely describes some resource that will always be available. The following is an example of a URN:

```
urn:foo-bar:foobar.1
```

The exact syntax of URNs is denoted in RFC 2141, but the following is a summary:

```
urn:<Namespace Identifier>::<Namespace Specific String>
```

1. The text “urn:” (uppercase or lowercase) is included.
2. The Namespace Identifier consists of letters, numbers, and hyphens (uppercase or lowercase).
3. The Namespace Specific String consists of letters, numbers, parentheses, commas, periods, hyphens, and other such characters (uppercase or lowercase).

Efforts are underway to provide Internet protocols for resolving URNs. This would work similar to the way DNS (or other name service) resolves hostnames.

**XML Namespaces**

In the .NET Framework, namespaces are used routinely to uniquely identify groupings of logically related classes that might have names that coincide with classes within other framework class groupings. XML also uses namespaces, but for a slightly different purpose. Because you are free to create your own XML elements, chances are good that you will happen to select a tag name that someone else has also used. With everyone declaring their own XML element names and attributes, you can expect ambiguous results when trying to combine this data. For example, one system might use the `<Title/>` element when describing a book, while another system might also use `<Title/>` to describe automobile ownership. As long as the XML data from the two systems is never combined within the same XML document, everything is fine. However, if the data ever merges, there is no way to distinguish between the two semantic meanings.

An XML namespace, as identified by a URI reference, qualifies element and attribute names within an XML document. This not only avoids name collisions, but it also enables vocabularies to be reused. You can think of an XML vocabulary as a published set of element and attribute names common to a particular user domain. SOAP is one such vocabulary, but there are many others.
The following example contains XML that has no associated namespace:

```xml
<Product>
    <ProductName Type='1'>Widget</ProductName>
</Product>
```

To reference a namespace, you must first declare one by creating a namespace declaration using this form:

```xml
xmlns:<Namespace Prefix> = <URI>
```

In the namespace declaration, you specify a namespace prefix and the URI of the namespace. The prefix is attached to the local names of elements and attributes so that they are associated with the correct namespace, like so:

```xml
<pns:Product xmlns:pns="http://www.endurasoft.com/prodns">
    <pns:ProductName pns:Type='1'>Widget</pns:ProductName>
</pns:Product>
```

In this case, `pns` is defined to be the prefix and is used to associate `Product`, `ProductName`, and `Type` with the `http://www.endurasoft.com/prodns` URI. In reality, the URI doesn’t necessarily point to anything—it’s purpose is to provide uniqueness to the namespace.

XML namespaces also provide the concept of a default namespace (denoted in XML as `xmlns`). This enables you to establish a namespace for an element and all its children, thus avoiding the need to use a prefix on each element name.

**NOTE**

Default namespaces do not apply to attribute names. Instead, attributes must be explicitly prefixed to the desired namespace.

Initially, an XML document has no assigned default namespace, so any elements that are not qualified with a prefix will be locally scoped.

Now consider the following example:

```xml
<pns:Product xmlns:pns="http://www.endurasoft.com/prodns">
    <pns:ProductName pns:Type='1'>Widget</pns:ProductName>
    <ProductLoc xmlns="http://www.endurasoft.com/prodlocns">
        <Building>310</Building>
        <Floor>2</Floor>
        <Room>118</Room>
    </ProductLoc>
    <Cost>495.00</Cost>
</pns:Product>
```
On the first line, the `pns` prefix is created to reference a product URI. This same prefix is used to qualify the `Product` and `ProductName` elements. On the third line, a default namespace is created that references a completely different URI than did the first line. All elements contained within the `ProductLoc` element are scoped to the default namespace and, therefore, require no prefix. However, because `<Cost/>` element is not contained by the `ProductLoc` element and doesn’t have a prefix, it is considered locally scoped to the document.

.NET uses namespaces. If you examine the XML used to transfer information between the Web Service and the client, you’d see these namespaces in action. They’re simply there to identify a certain XML element as belonging to a logically related group of XML elements that form a vocabulary. But how is the vocabulary itself specified? This is the primary use of the XML Schema.

### XML Schemas

As you know, all XML documents must be well formed. For example, tags cannot overlap. They must specify some sort of hierarchy. But often the benefits of producing a well-formed document aren’t enough. Saying that the XML elements cannot overlap is not as useful as saying that the XML elements cannot overlap and must follow a specific order or use certain tag names. The XML Schema specification identifies an XML vocabulary that you can use to create other XML vocabularies. In doing so, you tell consumers of your schema how your XML should be constructed to be considered valid by your design.

### Understanding XML Schemas

Many times as you develop XML documents, you often need to place constraints on the way data is represented in the document. You might be concerned that a particular set of XML elements follows a specific order, or you might want to identify an XML element as containing text that actually represents a specific datatype, such as a floating point.

To place constraints on data, you must build Document Type Definitions (DTDs) or XML Schemas to provide data about the data, also known as metadata. DTDs were an early XML constraint mechanism. And although DTDs are beneficial to many XML applications, they do not have the characteristics necessary for describing constructs such as inheritance or complex datatypes. To overcome these limitations, a working group was formed to produce XML Schemas based on an original draft from Microsoft. The XML Schema specification is divided into two parts. The first part, *XML Schema Part 1: Structures*, proposes a way to structure and constrain document content. The second part, *XML Schema Part 2: Data Types*, provides a way to describe both primitive and complex datatypes within a document.
The XML Schema specification establishes a means by which the XML Schema language describes the structure and content of XML documents. A desirable feature of XML Schemas is the fact that they are represented in XML, so standard XML parsers can be used to navigate them.

At this point, you are already familiar with XML and many of the terms used to identify the concepts behind XML. The XML Schema draft defines several new terms that help describe the semantics of using and understanding schemas.

**Instances and Schema**

An XML instance document refers to the document element, including elements, attributes, and content contained within the document, that conforms to an XML Schema. Instances in the more general sense may refer to any element (including its attributes and content) that conforms to an XML Schema. An instance that conforms to a schema is considered to be schema-valid.

Schemas can be independent XML documents, or they can be embedded inside other XML with references to the schema. Schemas take this form:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/TR/xmlschema-1/">
  <!--type definitions, element declarations, etc. -->
</xsd:schema>
```

**Definitions and Declarations**

A great advantage of schemas is that they enable you to create simple or complex types for applying classifications to elements. As in most programming languages, this is called type definition and is shown as follows:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/TR/xmlschema-1/">
  <xsd:complexType name="Person">
    <xsd:element name="FirstName" type="xsd:string" />
    <xsd:attribute name="Age" type="xsd:integer" />
  </xsd:complexType>
</xsd:schema>
```

The preceding example also shows an element declaration (for the element `<FirstName/>`) and an attribute declaration (for the attribute `Age`) that are local to a particular type named `Person`.

Beyond participating in the type definition, elements also may be declared as top-level elements of a particular type, as shown in the following example, where `BaseballPlayer` is a type of `Person`:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/TR/xmlschema-1/">
  <!-- ...Type definition... -->

  <xsd:element name="BaseballPlayer" type="Person" />
</xsd:schema>
```
Attributes, however, can be of only simple types, as defined in XML Schema Part 2: Datatypes, such as string, boolean, and float.

**Target Namespace**
Because element and attribute declarations are used to validate instances, it is necessary for them to match the namespace characteristic of a particular instance. This implies that declarations have an association with a target namespace URI or no namespace at all, depending on whether the instance has a qualified name. For a schema to specify a target namespace, it must use the targetNamespace attribute, as follows:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/TR/xmlschema-1/">
  <targetNamespace>SomeNamespaceURI</targetNamespace>
  <xsd:element name="ElementInNS" type="xsd:string" />
  <xsd:complexType name="TypeInNS">
    <xsd:element name="LocalElementInNS" type="xsd:integer" />
    <xsd:attribute name="LocalAttrInNS" type="xsd:string" />
  </xsd:complexType>
</xsd:schema>
```

As you can see, all global and local elements are associated with SomeNamespaceURI. Lack of the targetNamespace attribute designates that no namespace is associated.

**Datatypes and Schema Constraints**
Datatypes consist of a value space, lexical space, and facets. The value space is the datatype’s permitted set of values, and it can have various properties associated with it. A set of valid literals for a datatype makes up the lexical space of that datatype. Finally, a facet is a single dimension of a concept that enables you to distinguish among different datatypes. Two kinds of facets are used to describe datatypes, fundamental and constraining.

Fundamental facets enable you to describe the order, bounds, cardinality, exactness, and numeric properties of a given datatype’s value space.

Constraining facets enable you to describe the constraints on a datatype’s value space. Possible constraints include minimum and maximum length, pattern matching, upper and lower bounds, and enumeration of valid values.

The following is the fragment of a simple type definition:

```xml
<xsd:simpleType name="HourType">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="1" />
    <xsd:maxInclusive value="12" />
  </xsd:restriction>
</xsd:simpleType>
```
In this case, `HourType` is defined to be of the built-in `integer` datatype and additionally is constrained to values between 1 and 12. This new type can then be used in other type definitions as in the following `Hour` attribute:

```xml
<xsd:complexType name="Time">
    <xsd:attribute name="Hour" type="HourType"/>
    <xsd:attribute name="Minute" type="MinuteType"/>
</xsd:complexType>
```

The instance for this type might look something like this:

```xml
<Time Hour="7" Minute="30"/>
```

That was also an example of a `complex type` definition. The complex type definition combines one or more simple types to form something new. Here is another complex type example:

```xml
<xsd:element name="cars" type="CarsType"/>
<xsd:complexType name="CarsType">
    <xsd:element name="car" type="CarType" minoccurs="0" maxoccurs="unbounded"/>
</xsd:complexType>
<xsd:complexType name="CarType">
    <xsd:element name="make" type="xsd:string"/>
    <xsd:element name="model" type="xsd:string"/>
</xsd:complexType>
```

This type can be represented by an instance as follows:

```xml
<cars xmlns:xsi="http://www.w3.org/TR/xmlschema-1/
    xsi:noNamespaceSchemaLocation="CarSchema.xsd">
    <car>
        <make>Cheverolet</make>
        <model>Corvette</model>
    </car>
</cars>
```

**minOccurs and maxOccurs**

Elements and attributes enable you to specify the minimum and maximum number of times that they may appear in the instance. The following example shows how you can force an attribute to appear one and only one time:

```xml
<xsd:element name="Book">
    <attribute name="Author" type="A" minOccurs="1" maxOccurs="1"/>
    <attribute name="Title" type="T" minOccurs="1" maxOccurs="1"/>
</xsd:element>
```
The `maxOccurs` attribute can also be set to `unbounded` to denote that the element or attribute can appear *many* times. You also can prevent a value from appearing by setting the `maxOccurs` attribute equal to 0.

**Deriving Type Definitions**

Similar to the way object-oriented programming languages work, schemas enable you to derive types from other types in a controlled way. When defining a new type, you may choose to *extend* or *restrict* the other type definition.

When extending another type definition, you can introduce additional elements and attributes, as shown in the following example:

```xml
<xsd:complexType name="Book">
    <xsd:element name="Title" type="xsd:string" />
    <xsd:element name="Author" type="xsd:string" />
</xsd:complexType>

<xsd:complexType name="ElectronicBook">
    <xsd:complexContent>
        <xsd:extension base="Book">
            <xsd:sequence>
                <element name="URL" type="xsd:string" />
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Sometimes an instance wants to explicitly indicate its type. To do this, the instance can use the XML Schema instance namespace definition of `xsi:type`, as follows:

```xml
<Car xsi:type="SportsCar">
    <Driver>Me</Driver>
</Car>
```

Although this was not an exhaustive coverage of schemas, at least you now should realize the following:

- You can constrain your XML documents and their content using XML Schema.
- The XML Schema specification provides you with a set of built-in datatypes.
- You can use the built-in datatypes or create your own datatypes.
- Schemas may be standalone documents or may be combined within other XML documents.

This information is here because XML Schemas are important to .NET. Let’s see why.
.NET Web Services and XML Schemas

If you happen to glance at the SOAP specification, you'll find a section that describes how to encode method parameters, Section 5. This section was necessary when the SOAP specification was introduced because there was no way to otherwise describe the SOAP XML. If you couldn’t somehow validate the incoming SOAP packets, you could not extract the method’s parameter data and actually invoke the method on your local systems.

Section 5 is becoming far less important today because of the Web Service Description Language (WSDL), as you’ll see in detail in Chapter 5, “Web Service Description and Discovery.” WSDL serves as an interface description document that you can use to determine what XML information the Web Service will accept. In other words, you can change the XML formatting for your Web Service by changing the way you describe the Web Service in its WSDL file. As it happens, there is a schema embedded within the WSDL file.

This actually makes a lot of sense. If you think about it, handing someone an arbitrary XML document and expecting that person to figure out by inspection just what you’re asking him to do is a very complex undertaking, if that person even has enough information to make an informed decision. On the other hand, if you hand that same person a document that outlines the datatypes of your method parameters and the order in which they can be found, you’ve given that person enough information to decipher the XML instance documents that you intend to transmit.

For this reason, you’ll find an XML schema embedded within the WSDL document that describes your Web Service. Essentially, with your WSDL document, you’re telling the other side what datatypes you expect and how you want them ordered, as well as how they should appear within the SOAP packet. Web Services are now significantly more flexible.

Now that you know how XML documents are formed and how they are validated, how do you get the values associated with the XML elements back out of the XML document? This is the job of XPath.

We will discuss XPath in some detail, mainly because many people who have worked with XML to some degree still might require a little XPath brush-up. And it’s XPath that makes your work easier if you need to reach into a SOAP packet and modify what you find, as you might do with a .NET SoapExtension. Why? Because of interoperability, if nothing else, but you might have other reasons as well, depending upon your individual system requirements. For example, you might want to retrieve a SOAP parameter value and encrypt it.

Let’s take a more detailed look at XPath to see what it can offer when you’re tweaking XML.
XPath Drilldown

Imagine that you have this XML document you created for yourself to remind you how to access a couple of your favorite Web Services:

```
<?xml version="1.0"?>
<Servers>
  <Server name="Gumby">
    <WebService wsdl="?wsdl">
      <Family>Calculators</Family>
      <EndpointURL>http://www.myurl.com/calc</EndpointURL>
    </WebService>
  </Server>
  <Server name="Pokey">
    <WebService wsdl=".wsdl">
      <Family>Time</Family>
      <EndpointURL>http://www.myurl.com/time</EndpointURL>
    </WebService>
  </Server>
</Servers>
```

This totally fictitious XML document describes two imaginary Web Service servers. The first provides some sort of calculator service, based upon the service's family, and the second gives the time. The calculator service sends its WSDL by adding `?WSDL` to the endpoint URL, which is how .NET works. The second does the same by concatenating `.WSDL`, which is how the SOAP Toolkit works. Of course, only two servers are shown in this case. You could have hundreds or more, so the corresponding XML document could grow to be quite large. How will you find the one particular server's information that is of interest to you?

Now let's say that you want to retrieve all the servers that are part of the Time family. You could use this XPath query string:

```
/Servers/Server/WebService[ ./Family="Time"]
```

The result of this query is a nodeset, which is a set of XML elements that match the query. The query itself can be called a location step, which can be broken into three parts:

- The axis
- The node test
- The predicate

Let's take a closer look at each.
The XPath Axis

The axis is optional—in fact, this particular location step has no axis identified. You use the axis to move through the XML document in some other manner than from the top down. This is because the default location step is \texttt{child::}, so the XPath query returns, by default, a node-set containing children of the current \textit{context node}. The context node is the current XML element that you happen to be examining, which, in this case, is the root or document element. Other possible axes include \texttt{ancestor::}, \texttt{parent::}, \texttt{following::}, and a myriad of other possible values, all shown in Table 3.1.

\begin{table}[h]
\centering
\caption{XPath Axis Values}
\begin{tabular}{ll}
\hline
\textbf{Axis} & \textbf{Purpose} \\
\hline
ancestor & Ancestors of the context node, including the root node if the context node is not already the root node \\
ancestor-or-self & Same as ancestor, but includes the context node \\
attribute & Attributes of the context node (context node should be an element) \\
child & All (immediate) children of the context node \\
descendant & All children of the context node, regardless of depth \\
descendant-or-self & Same as the descendant, but includes the context node \\
following & All nodes following the context node, in document order \\
following-or-sibling & Only sibling nodes following the context node, in document order \\
namespace & Namespace of the context node (the context node should be an element) \\
parent & Immediate parent of the context node, if any (that is, not the root node) \\
preceding & Similar to the following, except returns preceding nodes in document order \\
preceding-sibling & Same as preceding, but for sibling nodes only \\
self & The context node itself \\
\hline
\end{tabular}
\end{table}

You probably noticed the term \textit{document order} mixed into Table 3.1. Document order refers not to the ordering and hierarchy of XML elements, but instead to the literal order in which the element is found in the document, whether it is a parent, sibling, or whatever. Essentially, when you access nodes in document order, you’re flattening any hierarchy that might be present.
The XPath Node Test
The node test is effectively a road map that shows the element names in progression, from the start of the document to the particular element in question. It’s literally a path from the document element (the root XML element) to the data that you’re testing for inclusion into the result nodeset. XPath, as usually implemented, is often more efficient if you specify the complete element path. However, you could have written the example location step as this:

//WebService[./Family="Time"]

The initial double slash, //, tells the XPath processor to start at the document element, search recursively for the <WebService/> element, and, after finding it, execute the predicate.

The XPath Predicate
The predicate, sometimes referred to as the filter, is a Boolean test that you apply to make a final decision about the particular XML element that XPath is examining. If the predicate returns a true result, the XML element is added to the result nodeset. If not, the element is discarded from the nodeset. Essentially, you’re fine-tuning an XML element filter. For example, given the two servers shown in the example XML document, the initial nodeset returned from the axis yields both servers, as does the result of the node test. That is, both <Server/> elements have children <WebService/> elements. It’s the predicate that distinguishes them, in this case, because only the second server, Pokey, exposes a Time family Web Service.

The node test, being a pathway into the XML document, is sensitive to both the alphabetical case and the namespace of the particular XML element shown in the path. That is, imagine that you mistyped the example location step in this manner:

/servers/server/webservice[./family="Time"]

The resulting nodeset would be empty, whereas before it contained the element for the Pokey server. Notice in the second XPath query that all the text is lowercase, which is why it would fail.

To help with XPath query generation, we wrote the application that you see in Figure 3.2. The XPathExerciser is a utility that enables you to load an XML document, display its contents so that you can see what your queries should produce, type in an XPath location step, and display the resulting nodeset using a tree control.

You’ll examine the source code for the XPathExerciser when we discuss .NET’s XML handling capabilities, starting with the upcoming section “.NET and XPath.” This is a tool that you truly will use because you can never be too expert at recording XPath expressions.
**FIGURE 3.2**
The XPathExerciser user interface.

**XPath Operators**

XPath is a language all its own. Like any programming language, XPath has a set of operators. The operators represent intrinsic capabilities that XPath can perform upon request—you see these listed in Table 3.2.

**TABLE 3.2** XPath Intrinsic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>Child operator, which selects child nodes or specifies the root node</td>
</tr>
<tr>
<td>//</td>
<td>Recursive descent, which looks for a specified element at any depth</td>
</tr>
<tr>
<td>.</td>
<td>Current context node (akin to C++ this or VB me)</td>
</tr>
<tr>
<td>..</td>
<td>Shorthand notation for parent of current context node (akin to moving up a file directory)</td>
</tr>
<tr>
<td>*</td>
<td>Wildcard, which selects all elements regardless of their element name</td>
</tr>
<tr>
<td>:</td>
<td>Namespace operator (same use as in XML proper)</td>
</tr>
<tr>
<td>@</td>
<td>Attribute operator, which prefixes an attribute name</td>
</tr>
<tr>
<td>@</td>
<td>Attribute wildcard (when used alone), which is semantically equivalent to *</td>
</tr>
<tr>
<td>+</td>
<td>Addition indicator</td>
</tr>
</tbody>
</table>
TABLE 3.2  Continued

<table>
<thead>
<tr>
<th>Operator</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Subtraction indicator</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication indicator</td>
</tr>
<tr>
<td>div</td>
<td>Floating-point division indicator</td>
</tr>
<tr>
<td>mod</td>
<td>Modulo (remainder from a truncating division operation)</td>
</tr>
<tr>
<td>()</td>
<td>Precedence operator</td>
</tr>
<tr>
<td>[]</td>
<td>Operator that applies a filter (akin to a Boolean test)</td>
</tr>
<tr>
<td>[]</td>
<td>Set subscript operator (akin to an array index specification)</td>
</tr>
</tbody>
</table>

Not too much in Table 3.2 should be too surprising. The square brackets, [ and ], indicate either an array or a filter pattern depending upon how you use them. The single period, ., indicates the current context node, much like the same operator does in a Windows file path. The same is true for the dual period, .. The @ indicates an attribute. Otherwise, you have operators that you would expect to see, such as the wildcard operator, *, and mathematical operations.

Returning to the previous example, you could locate all the SOAP Toolkit Web Services stored in the XML document using this XPath location step:

```
/Servers/Server/WebService[./@wsdl="*.wsdl"]
```

You could accomplish the same task with these two location steps:

```
/WebService[./@wsdl="*.wsdl"]
```
```
/Servers/*/*/wsdl="*.wsdl"
```

If some of the servers had no wsdl attribute but others did, you could test merely for the presence of the attribute, like so:

```
/Servers/Server/WebService[./@wsdl]
```

In this case, the nodeset would contain both servers shown in the example XML document, but this is only because both servers have a wsdl attribute. If one server had no wsdl attribute, the predicate would fail for that particular node, and that XML element would be removed from the result nodeset.

**XPath Intrinsic Functions**

In addition to operators, XPath has an entire suite of intrinsic functions that it exposes to help with your queries. There are a lot of these, so the more commonly used functions are distilled in Table 3.3.
Table 3.3 Commonly Used XPath Intrinsic Functions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ceiling()</td>
<td>Is the smallest integer not less than the argument</td>
</tr>
<tr>
<td>count(nodeset)</td>
<td>Gives the number of nodes in the nodeset argument</td>
</tr>
<tr>
<td>contains(string,string)</td>
<td>Returns true if the first argument contains the second</td>
</tr>
<tr>
<td>false()</td>
<td>Always returns a Boolean false</td>
</tr>
<tr>
<td>floor()</td>
<td>Is the largest integer not greater than the argument</td>
</tr>
<tr>
<td>last()</td>
<td>Gives the context size (number of nodes in context node set)</td>
</tr>
<tr>
<td>local-name()</td>
<td>Returns the local name of the first node (document order)</td>
</tr>
<tr>
<td>name()</td>
<td>Returns the QName of the first node (document order)</td>
</tr>
<tr>
<td>node()</td>
<td>Returns true for any type of node</td>
</tr>
<tr>
<td>not()</td>
<td>Indicates a logical negation</td>
</tr>
<tr>
<td>number(object)</td>
<td>Converts object to a number</td>
</tr>
<tr>
<td>position()</td>
<td>Gives the index number of the node within the parent</td>
</tr>
<tr>
<td>starts-with(string,string)</td>
<td>Returns true if the first argument starts with the second</td>
</tr>
<tr>
<td>string(object)</td>
<td>Turns object into a string</td>
</tr>
<tr>
<td>sum(nodeset)</td>
<td>Converts nodeset to numerical values and adds them</td>
</tr>
<tr>
<td>true()</td>
<td>Always returns a Boolean true</td>
</tr>
</tbody>
</table>

For a complete list of XPath intrinsic functions, you should refer to a good XPath reference. You’ll probably find that these functions will handle most of your XPath needs, however.

Using the Web Service server example, you could identify the first server, or an arbitrary server, using this location step:

/Servers/Server[position()="1"]

Similarly, you find the last server like so:

/Servers/Server[last()]

If you want all the servers but the last one, you query the document in this way:

/Servers/Server[not(position()=last())]

Many people want to locate XML information within a document based upon string values or string searches. Say, for example, that you want all the servers that have names starting with the letter G:

/Servers/Server[starts-with(string(./@name),"G")]

For a complete list of XPath intrinsic functions, you should refer to a good XPath reference. You’ll probably find that these functions will handle most of your XPath needs, however.

Using the Web Service server example, you could identify the first server, or an arbitrary server, using this location step:

/Servers/Server[position()="1"]

Similarly, you find the last server like so:

/Servers/Server[last()]

If you want all the servers but the last one, you query the document in this way:

/Servers/Server[not(position()=last())]

Many people want to locate XML information within a document based upon string values or string searches. Say, for example, that you want all the servers that have names starting with the letter G:

/Servers/Server[starts-with(string(./@name),"G")]

For a complete list of XPath intrinsic functions, you should refer to a good XPath reference. You’ll probably find that these functions will handle most of your XPath needs, however.
Of course, this will return the Gumby server’s XML information.

As you can see, you can produce a wide variety of queries, especially if you combine an axis with a node test and a filter. This becomes important later if you need to crack open a SOAP packet to examine and modify the contents by hand.

SOAP uses another XML technology called XPointer. Let’s now turn to that technology and see what it offers the Web Service.

**Identifying XML Elements Using XLink**

As you gain experience with XML, you’ll find that hierarchical relationships between data elements lend themselves to XML serialization rather naturally. XML is quite happy with a parent/child or sibling/sibling relationship between data elements. But what do you do if the data isn’t hierarchical by nature?

A classic example of this is the linked list. The nodes in a linked list, by definition, have no hierarchical relationship. You could argue in favor of a sibling/sibling relationship, but recall that XML doesn’t specify ordering of elements. To XML, this document arrangement

```xml
<element1/>
<element2/>
```

is semantically the same as this arrangement:

```xml
<element2/>
<element1/>
```

XML doesn’t distinguish between the two unless you consider their document order—and this ordering is considered tenuous, at best. Because we’re talking about a linked list, which has a definite ordering in memory, we need to be a bit more concrete. In this case, XPLink is employed.

XLink uses two attributes, `href` and `id`, to identify the semantic links between elements. Actually, XLink is a great deal more than what is discussed here. We’ll discuss XLink only with respect to its use in the SOAP protocol, as you’ll see in Chapter 4, “.NET Web Services and SOAP.” You could use XLink to link elements in completely different XML documents, for example, but you needn’t go that far for Web Service purposes.

Let’s look at XLink by example. Imagine that you have the linked list that you see in Figure 3.3.

You’ll actually revisit the linked list in the next chapter when we tie XLink to SOAP, as well as in Chapter 6, “Web Services in ASP.NET,” where you’ll send a linked list to a Web Service for processing. Here, though, it’s important to note that you have four nodes, with the usual components—a head node, a tail node, and intermediate nodes. The tail node has its “next” link set to `null`, or nothing, to indicate that there is no more data.
In code, this linked list is made up of nodes:

```java
class Node
{
    int iValue;
    Node pNext;
}
```

In XML, we create XML elements representing the list nodes and link them so that the current node “points” to the next node using the `href` attribute:

```xml
<nod head="true">
    <iValue>24</iValue>
    <pNext href="#node2" />
</node>
<nod root="true" id="node2">
    <iValue>609</iValue>
    <pNext href="#node3" />
</node>
<nod root="true" id="node3">
    <iValue>32</iValue>
    <pNext href="#node4" />
</node>
```

**Figure 3.3**

*Conceptual image of a linked list.*
Each corresponding “next” node has an id attribute that is identical to the href of the previous node. Note that the href attribute precedes the actual id value with a pound sign, #. This indicates that the link is contained within this document instead of some external reference.

A head attribute also was created for this example. This identifies the node as the starting point—without it, you wouldn’t necessarily know which node was the starting point. It’s true that you could look at all the <pNext/> elements and figure it out, but adding the attribute makes for faster processing. SOAP has a similar concept that you’ll read about in the next chapter.

To tie off the list, we added the xsi:null attribute. This comes directly from XML schema, where you can identify a value as null or nothing using this construct. Of course, you must tie off the list—that is required as part of the basic linked list abstract datatype. To do otherwise would invite your code to randomly walk through memory until you hurt yourself. The same requirement applies to the XML list serialization—hence, the attribute.

As the XML list representation indicates, XLink is a necessary technology for serializing Web Services. After all, if you think about it, the methods that you send into the remote Web Service method are roughly equivalent to a linked list. They’re not linked in the traditional sense, but they do have some association, if only because they belong to the same remote method. SOAP looks at your method parameters and, in some cases, will serialize them using XLink.

XML Transformations

The final XML-based technology mentioned here handles how you transform XML from one vocabulary into another. That is, you can change one XML document format into another very easily using the eXtensible Stylesheet Language with Transforms (XSLT). Actually, we’ve already looked at a major component of XSLT—XPath. What we’ll do here, briefly, is combine XPath with the XML style sheet vocabulary to change XML from one form into another.

Why would we want to do this in a Web Service book? After all, .NET does all the soapy stuff, right? Well, that’s true, in a way. But you might have noticed that three versions of SOAP are available (1.0, 1.1, and 1.2). The SOAP specification states that you should return a version mismatch error if you receive a SOAP packet indicating that the packet is formed using a version of SOAP that you don’t handle. But you might want to work with Web Services or clients that expect another version. If you accept a SOAP packet that .NET cannot handle (an old SOAP version, for example), you can easily intercept the packet using a .NET SoapExtension and then can transform it into something .NET can handle. We’re simply showing you how to take SOAP in one form and convert it to another, in case you ever need to do so.
Before diving in, another caveat should be mentioned: XPath and XSL/XSLT are huge topics that merit books of their own, and many are available. What we intend to present here is just enough to get you working with the technology. For an in-depth view, definitely pick up a good reference.

**XSLT Drilldown**

XSLT combines XPath expressions with *templates*. A template combines a nodeset returned from a specific XPath location step and produces output based upon the contents of the template. More often than not, the template uses the results obtained from the nodeset as input.

Let’s say that you were given this XML document:

```xml
<?xml version="1.0"?>
<Parts>
  <Part vehicle="Corvette" manufacturer="GM">
    <Number>3972178</Number>
    <Desc>Special Hi-Perf Camshaft</Desc>
  </Part>
</Parts>
```

**NOTE**

This is by far not the most common use of XSL and XSLT. In the vast majority of the cases we’ve seen, people are using XSL to turn XML documents into an HTML representation for display purposes. You’ll probably never have to actually transform XML from one version of SOAP to another, especially given Microsoft’s aggressive acceptance of the SOAP standard. But this transformation, no matter how unlikely, is still a valid use of the XSLT technology. This use simply relates XML and XSLT to SOAP and Web Services. However, you might find that if you use Web Services often and long enough, you’ll want to access a Web Service that will require you to modify your SOAP packets for one reason or another, and XSLT is an excellent alternative for doing so.

**NOTE**

As I write this, the latest SOAP working draft specification has just been released. Therefore, my prediction that you might use XSL to transform SOAP 1.1 packets into SOAP 1.2 packets isn’t necessarily so far-fetched. Luckily, the differences in the protocol versions aren’t significant enough, for the most part, to truly merit XSL transformation. However, two clichés come to mind—"You never know" and "Never say never"….
This document is clearly parts-centric. What if you want an XML document that is vehicle-centric? Would you need to create that from scratch? Well, assuming that all (or some) of the data that you require is already contained in the parts XML document, you wouldn’t—you could use XSLT to transform it from the parts-centric view to the vehicle-centric view rather easily.

Let’s do just that. The vehicle XML document needs to look like this:

```xml
<?xml version="1.0"?>
<Vehicles>
  <Vehicle>
    <Make/>
    <Model/>
    <Parts>
      <Part>
        <Number/>
        <Description/>
        <ModelYears/>
      </Part>
      <Parts>
        <Additional parts elements here>
      </Vehicle>
    </Vehicles>
</Vehicles>
```

The XSL style sheet to accomplish this is shown in Listing 3.1.

**Listing 3.1** Example XSL Style Sheet

```xml
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
  <xsl:output method="xml"/>

  <xsl:template match="/"
      <Vehicles>
        <xsl:apply-templates select="Parts"/>
      </Vehicles>
```
**LISTING 3.1  Continued**

```xml
<xsl:stylesheet>

<xsl:template match="Parts">
  <Vehicle>
    <Make><xsl:value-of select="Part/@manufacturer"/></Make>
    <Model><xsl:value-of select="Part/@vehicle"/></Model>
    <Parts>
      <xsl:for-each select="Part">
        <Part>
          <Number><xsl:value-of select="Number"/></Number>
          <Description>
            <xsl:value-of select="Desc"/>
            <xsl:text>: </xsl:text>
            <xsl:value-of select="Comment"/>
          </Description>
          <ModelYears>
            <xsl:value-of select="MinYear"/>
            <xsl:text> to </xsl:text>
            <xsl:value-of select="MaxYear"/>
          </ModelYears>
        </Part>
      </xsl:for-each>
    </Parts>
  </Vehicle>
</xsl:template>
</xsl:stylesheet>
```

If you run the XML file and style sheet through Internet Explorer, you might not get the output that you expect. Internet Explorer will execute the transformation, but it will not display the result as XML. If you're interested in viewing the actual transformed output, it's best to run the XML document and style sheet through a program designed to execute the style sheet and save the resulting output to a file for later recall. You'll get an application that does just this when you get to .NET in the upcoming section “.NET and XSL.”

XSL is an XML vocabulary, which is to say that your XSL style sheets will be XML documents in their own right. Many elements are associated with XSL; Table 3.4 gives you the more commonly used XSL instructions.
### Table 3.4  Commonly Used XSL Vocabulary Elements

<table>
<thead>
<tr>
<th>Operator</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;apply-templates/&gt;</td>
<td>Tells the XSL processor to apply all appropriate templates to the nodeset</td>
</tr>
<tr>
<td>&lt;call-template/&gt;</td>
<td>Invokes a template by name</td>
</tr>
<tr>
<td>&lt;choose/&gt;</td>
<td>Uses multiple conditional testing (use with when and otherwise)</td>
</tr>
<tr>
<td>&lt;for-each/&gt;</td>
<td>Applies a template repeatedly to all nodes in the nodeset</td>
</tr>
<tr>
<td>&lt;if/&gt;</td>
<td>Is a conditional construct (has no else clause—use &lt;choose/&gt; instead)</td>
</tr>
<tr>
<td>&lt;number/&gt;</td>
<td>Inserts a formatted integer into the output document</td>
</tr>
<tr>
<td>&lt;otherwise/&gt;</td>
<td>Is a default conditional expression used in conjunction with &lt;choose/&gt;</td>
</tr>
<tr>
<td>&lt;output/&gt;</td>
<td>Specifies serialization of the result tree</td>
</tr>
<tr>
<td>&lt;sort/&gt;</td>
<td>Sorts output nodes (used to rearrange output document)</td>
</tr>
<tr>
<td>&lt;stylesheet/&gt;</td>
<td>Is the XSL style sheet document (root) element</td>
</tr>
<tr>
<td>&lt;template/&gt;</td>
<td>Is the template tag (identifies and encapsulates template)</td>
</tr>
<tr>
<td>&lt;text/&gt;</td>
<td>Inserts literal text into the output document</td>
</tr>
<tr>
<td>&lt;value-of/&gt;</td>
<td>Copies the node of the input document to the output document</td>
</tr>
<tr>
<td>&lt;when/&gt;</td>
<td>Is an optional conditional expression used in conjunction with &lt;choose/&gt;</td>
</tr>
<tr>
<td>&lt;when/&gt;</td>
<td>Is an optional conditional expression used in conjunction with &lt;choose/&gt;</td>
</tr>
</tbody>
</table>

Here you see the XSL document element, `<xsl:stylesheet/>`, as well as the XSL workhorse, `<xsl:template/>`. The style sheet element encapsulates the style sheet itself, while the template element identifies what could be referred to as XSL subroutines. In fact, they’re not subroutines, but you could view them that way and not be terribly incorrect. Although we didn’t show the XSL elements related to variables, XSL does have the capability of passing variables and parameters between templates. Because they’re so powerful, let’s look at XSL templates in a bit more detail.
XSL Templates

Listing 3.1 provided two templates, one for the document element and one for the `<Parts/>` element. We could have created more templates, one for each `<Part/>` node and children, but instead we elected to dig more deeply into the original XML document using `<xsl:for-each/>`. This simply made the template a bit easier to read.

The template itself is the cookie cutter that XSL will use to create the newly formatted document. For each template that you include within your style sheet, you’ll receive formatted output. The key is to set up the templates so that they operate on the nodeset returned from the match attribute. If you know that many XML nodes will match the template, but you want to constrain the use of some of those nodes, you can also apply the mode attribute to the template:

```
<xsl:template match="/">
  ...
  <xsl:apply-templates select="Parts" mode="Engine" />
  <xsl:apply-templates select="Parts" mode="Body" />
  ...
</xsl:template>

<xsl:template match="Parts" mode="Engine">
  (Something to do with engine parts...)
</xsl:template>

<xsl:template match="Parts" mode="Body">
  (Something to do with body parts...)
</xsl:template>
```

Essentially, mode enables you to process original document nodes many times, with each processing iteration producing a different result.

Notice that we also used the XSL instruction `<xsl:apply-templates/>`. This tells XSL to go through its template collection and, for any of the input XML nodes that match the templates, produce the output. The key is to create XPath expressions that draw from the XML document the precise nodeset that you want to work with.

The template is the key to transforming the original XML document. After you establish the XPath expression that will pull the set of nodes of interest, you insert the new XML document structure within the template. For each XML node XSL finds that matches the template, it spits out a copy of the resulting template into the new XML document.

At this point, you’ve reviewed enough XML to move into .NET. Although it’s entirely possible to write .NET-based Web Services all day long and never need to access their XML nature, it’s also very likely that you’ll find a minor change to some part of your SOAP packet to be useful.
In that case, you’ll find yourself using some or all of the technologies described here. If you’re comfortable with what has been discussed so far, let’s now talk about XML within .NET.

**.NET’s XML Architecture**

.NET’s core XML architecture (at least, as of Beta 2) is as you see it in Figure 3.4. This is slightly different than it was for Beta 1, in which the XPathNavigator in Figure 3.4 was called XmlNavigator. The basic layout is somewhat the same, however.

![XML Architecture Diagram](image)

**Figure 3.4**

*.NET’s core XML architecture.*

The main XML class that you’ll use here is XPathNavigator. With this class, you can move through the XML document, using movement commands such as `MoveToNext()` and `MoveToFirstChild()`. Or, as the name suggests, you can provide the XPathNavigator an XPath query, and it will return to you an iterator that represents the nodeset returned from the query. Although it isn’t discussed here, you have XML DOM Level 2 support using XmlDocument. We won’t use XmlDocument here because we’ll be primarily interested in XPath queries, and XPathDocument is optimized for quick searches. It might be helpful to know that XmlDocument exists if you’re used to working with the XML DOM. However, we will be interested in using other .NET XML classes, starting with the .NET class that enables you to read XML data from a stream.

**Reading XML Data**

If you have an XML file on disk, XPathNavigator will accept the file’s filename and path, and will happily load the file for processing. .NET, however, is stream-based, so it might not be
surprising to find that .NET has an XML class designed for reading XML streams. This class is XmlReader, and if you have a stream that you know represents XML data, you can wrap that stream with a reader and it will manage movement through the stream in an XML-friendly fashion.

To see XmlReader in action, let’s simulate the file read that XPathNavigator performs when loading XML data. Imagine that you have an XML file called Activities.xml. To load this file into .NET, and to demonstrate XmlReader, you would use this code:

```csharp
FileStream fstmActivities =
    new FileStream("Activities.xml", FileMode.Open);
XmlTextReader xrdActivities = new XmlTextReader(fstmActivities);
```

Note that XmlReader is an abstract class; you don’t use it directly. Instead, you use a class derived from XmlReader, such as XmlTextReader. And although you could load the file directly into XmlTextReader, the stream-based constructor was used in this brief example simply because that’s how you’ll accept the SOAP information from .NET when processing a SoapExtension.

XmlReader gives you a fast, forward-only XML data stream. XmlTextReader is ideal for quickly determining whether your XML document is well-formed and allowing you to progress through the document. You’ll see an example of XmlTextReader in action in the next section, “Writing XML Data.”

## Writing XML Data

If there is a .NET XmlReader, then there probably should be an equivalent writer. In fact, there is—not surprisingly, its class name is XmlWriter. And, like XmlReader, XmlWriter is an abstract class that must be implemented in a derived class. For XmlWriter, the only class provided by .NET for this is XmlTextWriter.

XmlTextWriter also accepts a stream in its constructor, and it is to this stream that XmlTextWriter will write the XML that is created or otherwise slated for output. Listing 3.2 shows both XmlTextReader and XmlTextWriter in action.

### Listing 3.2  Reading and Writing XML Data Using .NET Framework Classes

```csharp
using System;
using System.IO;
using System.Text;
using System.Xml;

namespace RdrWtr
{
```
Listing 3.2 Continued

/// <summary>
/// Summary description for Class1.
/// </summary>
class Class1
{
    static void Main(string[] args)
    {
        try
        {
            // Create a file stream
            FileStream fstmXmlOut = new FileStream("MyXML.xml",
                FileMode.Create);

            // Create an encoding
            UTF8Encoding objEncoding = new UTF8Encoding();

            // Create an XML text writer
            XmlTextWriter objXmlWriter =
                new XmlTextWriter(fstmXmlOut, objEncoding);

            // Create some XML
            objXmlWriter.WriteStartDocument();
            objXmlWriter.WriteStartElement("m",
                "Employees",
                "http://www.myurl.com");
            objXmlWriter.WriteAttributeString("xmlns",
                    "m",
                    null,
                    "http://www.myurl.com");

            // Write an employee element
            objXmlWriter.WriteStartElement("m",
                "Employee",
                "http://www.myurl.com");
            objXmlWriter.WriteStartElement("m",
                "id",
                "http://www.myurl.com");
            objXmlWriter.WriteString("175-A15");
            objXmlWriter.WriteEndElement(); // id
            objXmlWriter.WriteStartElement("m",
                "Name",
                "http://www.myurl.com");
            objXmlWriter.WriteString("Kenn Scribner");
            objXmlWriter.WriteEndElement(); // Name
        }
    }
}
Listing 3.2 Continued

```csharp
objXmlWriter.WriteStartElement("m",
    "Title",
    "http://www.myurl.com");
objXmlWriter.WriteString("Code Gecko");
objXmlWriter.WriteEndElement(); // Title
objXmlWriter.WriteEndElement(); // Employee

// Write another employee element
objXmlWriter.WriteStartElement("m",
    "Employee",
    "http://www.myurl.com");
objXmlWriter.WriteStartAttribute("m",
    "id",
    "http://www.myurl.com");
objXmlWriter.WriteString("129-B68");
objXmlWriter.WriteEndAttribute();
objXmlWriter.WriteStartElement("m",
    "Name",
    "http://www.myurl.com");
objXmlWriter.WriteString("Mark Stiver");
objXmlWriter.WriteEndElement(); // Name
objXmlWriter.WriteStartElement("m",
    "Title",
    "http://www.myurl.com");
objXmlWriter.WriteString("Code Godzilla");
objXmlWriter.WriteEndElement(); // Title
objXmlWriter.WriteEndElement(); // Employee

// Finish off the document
objXmlWriter.WriteEndElement(); // Employees

// Flush it to the file
objXmlWriter.Flush();
objXmlWriter.Close();
fstXmlOut.Close();

// Create a file stream
FileStream fstXmlIn = new FileStream("MyXML.xml",
    FileMode.Open);

// Create an XML text writer
XmlTextReader objXmlReader = new XmlTextReader(fstXmlIn);

while (objXmlReader.Read())
```
LISTING 3.2  Continued

```csharp
{
    switch (objXmlReader.NodeType)
    {
        case XmlNodeType.XmlDeclaration:
            Console.WriteLine("<?xml version="1.0"?>");
            break;

        case XmlNodeType.Element:
            Pad(objXmlReader.Depth);
            Console.Write("<{0}", objXmlReader.Name);
            if (objXmlReader.HasAttributes)
            {
                bool bAttr =
                    objXmlReader.MoveToFirstAttribute();
                while (bAttr)
                {
                    Console.Write(" {0}="{1}"",
                        objXmlReader.Name,
                        objXmlReader.Value);
                    bAttr = objXmlReader.MoveToNextAttribute();
                } // while
            } // if
            Console.WriteLine(">");
            break;

        case XmlNodeType.Text:
            Pad(objXmlReader.Depth);
            Console.WriteLine(objXmlReader.Value);
            break;

        case XmlNodeType.EndElement:
            Pad(objXmlReader.Depth);
            Console.WriteLine("</{0}>", objXmlReader.Name);
            break;

        default:
            break;
    } // switch
} // while

// Close the file
objXmlReader.Close();
fstmXmlIn.Close();
} // try
```
Listing 3.2  Continued

```csharp
    catch (Exception ex)
    {
        Console.WriteLine("Exception: {0}\n",ex.Message);
    } // catch

    static void Pad(int iDepth)
    {
        for ( int i = 0; i < iDepth; i++ )
        {
            Console.Write("   ");
        } // for
    } // for

As you see in Listing 3.2, we provide XmlTextReader and XmlTextWriter with streams instead of asking them to open the files directly. This is again because you’ll be given streams of XML when dealing with SOAP from within .NET in Chapter 6.

Creating XML elements with XmlTextWriter is a simple matter of deciding what type of element to create and then writing the element information to the stream using the XmlTextWriter method that is most appropriate. We used the combination of WriteStartElement(), WriteString(), and WriteEndElement(), but you could have also used WriteElementString() or created entirely different sorts of elements, like those dedicated to a DTD or comment.

After we created the XML file, we read it back into memory and displayed it on the console screen using XmlTextReader. In this case, to format the output in more familiar terms, we determine what type of element XmlTextReader is currently indicating and spit out formatted text accordingly. We treated the start of an element differently than we treated the content or ending element tag, for example.

It’s nice having XmlTextReader and XmlTextWriter, but they work at a rather low level when it comes to dealing with XML. Next you’ll see how to move up the abstraction hierarchy and work with XML at a slightly higher level.

Navigating XML with .NET

When people who know .NET think of navigating an XML document, they probably think of XPathDocument and XPathNavigator before XmlReader and XmlWriter. This is because these two XPath classes provide you with the capability to deal with the XML document in a couple different ways. First, you can access the XML information using a pull model. That is,
obtain XML information when you request it. Second, you have the option of providing an XPath location step to extract a nodeset, which you access using XPathNodeIterator. In most cases, you’ll probably use a combination of the two.

**Pulling XML Element Information with .NET**

If you are familiar with XML and XML technologies, you’ve undoubtedly heard of SAX, the Simple API for XML. We even mentioned it earlier in the chapter. SAX is a stream-based approach to reading XML information. You create callback functions that the SAX parser will invoke when it senses information that you’re interested in accepting. That is, if you create a callback function for the start of an element tag, SAX will invoke your callback function whenever it reads a starting tag. This is a *push model* because SAX is shoving XML your way as it encounters the information within the document.

.NET, however, works the other way. .NET provides you with a water fountain instead of a fire hose, and you can sip from the fountain when you want. As you sip (read XML element information), more information is readied for your next request. You pull in more information as you’re ready.

The .NET XPath classes expose a window into which you examine the XML. Whenever you’re working with these classes, it’s important to remember that you are working with only a single XML element at any one time. To work with another XML element, you must move the “window.” If you’re new to .NET and .NET’s XML handling capabilities, this is probably the hardest concept to comprehend. This is most likely because of the nature of the classes—by appearance, you perform an element-based action upon the entire XML document, which at first seems odd.

For example, say that you have an instance of XPathNavigator, which contains an XML document. At first, it seems odd to execute code written like this:

```csharp
' objXPN is an instance of XPathNavigator that contains
' an XML document
Dim strElementValue As String = objXPN.Value
```

This seems odd because it isn’t apparent that when you ask for the “value” of the XPathNavigator, you’re really asking for the value associated with the XML element that the XPathNavigator object is currently referencing (the XML element in the “window”). If you “pull” more data, by executing XPathNavigator’s MoveToNext() method, for example, the XML element sitting in the window will change and you’ll retrieve the value for that element rather than the initial element.

When you’re comfortable with the pull model, however, it makes a lot of sense. With SAX, whenever one of your callback methods is invoked, you must deal with that piece of data then and there. If you want that information cached in any way, you implement the caching. With
XPathNavigator, though, the data waits for you. When you're ready for more data, just pull in more data. We'll demonstrate this more graphically in the next section, "NET and XPath," where you'll learn about retrieving a nodeset using XPath only to recursively examine the nodeset as the results are displayed in a .NET Windows Forms tree view.

.NET and XPath

You've already done the hard work if you've created and debugged the XPath query that you intend to apply to an XML document contained within XPathDocument. To execute the query, you create an instance of the associated XPathNavigator and execute its Select() method. Select() accepts as input the XPath location step as a string and returns in response an instance of XPathIterator. You then use XPathIterator to rummage through the nodeset.

The code that you'll need to perform the XPath query is simply this:

```vbnet
Dim objNodeItr As XPathNodeIterator = m_objXPathNav.Select("XPath query text")
```

Of course, a bit more is involved to set up the objects, but making the Select() call is about all you really need. Given the iterator, you can access all the nodes in the nodeset, extracting data from each as necessary.

To demonstrate this, we created a utility, called XPathExerciser, to write and test XPath queries. A portion of the Windows Forms application is shown in Listing 3.3.

**Listing 3.3  XPath Queries Using XPathNavigator**

```vbnet
Private Sub cmdQuery_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles cmdQuery.Click
    ' Execute the query
    Try
        ' Check for text
        If txtQuery.Text.Length >= 1 Then
            ' Perform the query
            Dim objNodeItr As XPathNodeIterator = m_objXPathNav.Select(txtQuery.Text)
            If Not objNodeItr Is Nothing Then
                ' Iterate through the nodeset and display in the
tree control
                FillNodeSetTree(objNodeItr)
            Else
                ' No nodes to add...
                tvwNodeSet.Nodes.Add("No nodes returned fromXPath query...")
            End If
        End If
    End If
End If
```
Listing 3.3  Continued

Catch ex As Exception
  ' Show error
  MsgBox(ex.Message, _
    MessageBoxButtons.Critical, _
    "XPath Exerciser Error")
  tvwNodeSet.Nodes.Clear()
  tvwNodeSet.Nodes.Add(New TreeNode("***Error executing XPath query...", _
    6, 6))
End Try
End Sub

Private Sub FillNodeSetTree(ByVal objNodeItr As XPathNodeIterator)
  ' Clear the tree
  tvwNodeSet.Nodes.Clear()

  ' Cut off screen update
  tvwNodeSet.BeginUpdate()

  ' Create the root node
  Dim node As TreeNode = New TreeNode("(Context node)", 6, 6)
  tvwNodeSet.Nodes.Add(node)

  ' Advance through the nodeset
  While objNodeItr.MoveNext
    AddTreeNode(objNodeItr.Current, node)
  End While

  ' Update screen
  tvwNodeSet.EndUpdate()
End Sub

Private Sub AddTreeNode(ByVal objCurrXMLNode As XPathNavigator, _
                        ByVal nodParent As TreeNode)
  Try
    ' Create the new node
    Dim node As TreeNode = New TreeNode("(Context node)", 6, 6)
    tvwNodeSet.Nodes.Add(node)

    ' Advance through the nodeset
    While objNodeItr.MoveNext
      AddTreeNode(objNodeItr.Current, node)
    End While

    ' Update screen
    tvwNodeSet.EndUpdate()
  End Try
End Sub
Exit Sub

Case XPathNodeType.Attribute
Exit Sub

Case XPathNodeType.Root
    node = New TreeNode("{root}", 1, 1)
End Select

' Add ourselves to the tree
    nodParent.Nodes.Add(node)

' Look for attributes
    Dim objNodeClone As XPathNavigator
    If objCurrXMLNode.HasAttributes Then
        objNodeClone = objCurrXMLNode.Clone
        objNodeClone.MoveToFirstAttribute()
        AddTreeNode(objNodeClone, node)
        While objNodeClone.MoveNextAttribute
            AddTreeNode(objNodeClone, node)
        End While
    End If

' Find children
    objNodeClone = objCurrXMLNode.Clone
    If objNodeClone.HasChildren Then
        objNodeClone.MoveToFirstChild()
        AddTreeNode(objNodeClone, node)
        While objNodeClone.MoveNext()
            AddTreeNode(objNodeClone, node)
        End While
    End If

Catch ex As Exception
    ' Show error
    MsgBox(ex.Message, _
            MsgBoxStyle.Critical, _
            "XPath Exerciser Error")
Listing 3.3  Continued

```vbnet
    tvwNodeSet.Nodes.Clear()

    tvwNodeSet.Nodes.Add(New TreeNode("***Error executing XPath query...", 6, 6))

    End Try
End Sub
```

XPathExerciser enables you to select an XML file against which you want to apply XPath queries. XPathExerciser then displays the file in the Web Browser ActiveX control after you’ve browsed for your particular XML file. Then, with the file in hand, you type in your XPath query and click the Query button. If things go well, the tree control displays the resulting nodeset. If this operation sounds familiar, it should. As you might remember, the XPathExerciser user interface was presented in Figure 3.2.

What is important to see from Listing 3.3 is the combination of XPath and pull methods, such as MoveToFirstChild() and MoveToNext(). Note also that, given an instance of XPathNavigator, you can recursively read the XML element information and build a tree view.

![NOTE](image)

This book was written with the second beta version of Visual Studio .NET and the .NET Framework. As of this second beta, an issue exists with ActiveX and COM interoperability: If the source files were created on the Windows XP operating system (originally called Whistler) and later were transferred to an older version of Windows, the source files will be modified by Visual Studio .NET and the reference to the ActiveX control will be dropped. When Visual Studio .NET drops the ActiveX control, it also drops any sibling controls (controls at the same user interface scope, such as when grouped on a form or group box). Therefore, we have included two versions of XPathExerciser, one for XP and one for other versions of Windows. The executable itself executes without error in either case, and you can mix and match executables at will. Just the source and project files present problems.

.NET and XLink

You might be wondering if special support for XLink is built into the .NET Framework. There is, but it requires a bit of additional overhead.
XPathNavigator has a method, MoveToId(), that is designed to work with id attributes. The additional overhead mentioned is that the id attribute must be explicitly defined in the XML DTD or schema that is associated with your XML document. If you have no DTD or schema, MoveToId() won’t work for you. However, you still may use XPath to search for all elements with an id attribute, or even an id attribute containing a specific identifying value.

.NET and XSL

The final general-purpose XML technology we mentioned is XSL, and .NET supports XSL through its XSLTransform class. If you load an XML document into an instance of XPathDocument and then load an XSL style sheet into XSLTransform, you can run the style sheet in this manner:

objXSL.Transform(objXMLDoc,null,objWriter);

Here, objXSL is the XSLTransform instance, objXMLDoc is the XPathDocument, and objWriter is the XmlTextWriter that you’ll use to persist the changes.

To provide a useful demonstration of transformations within the .NET Framework, we created the XSLEXerciser application, whose user interface you see in Figure 3.5. If you browse for an XML file and an associated style sheet, you can click the XForm button to execute the transform. If you’ve elected to save the results, the new XML document will be recorded within the file that you specified. You can also view the output either as text or as XML contained within the Web Browser control.

![XSL Exerciser Demo Application](image)

**Figure 3.5**
The XSLEXerciser user interface.

Listing 3.5 shows you how to handle the transform action itself. The code that you see is executed when you click the XForm button. You don’t need to load any files until you actually
want to perform the transformation, so if you need to adjust the style sheet, as often happens, you can do so easily and retransform the document.

**Listing 3.4 XPath Queries Using XPathNavigator**

```csharp
private void cmdXForm_Click(object sender, System.EventArgs e)
{
    try
    {
        // Open the original XML document
        m_objXMLDoc = new XPathDocument(m_strXMLPath);

        // Open the stylesheet
        XslTransform objXSL = new XslTransform();
        objXSL.Load(m_strXSLPath);

        // Create a stream to contain the results
        m_stmOutput = new MemoryStream();

        // Create and overlay an XML writer
        ASCIIEncoding objEncoding = new ASCIIEncoding();
        XmlTextWriter objWriter = new XmlTextWriter(m_stmOutput, objEncoding);

        // Do the transform
        objXSL.Transform(m_xpdXMLDoc, null, objWriter);

        // Save output as a string
        byte[] bytes = m_stmOutput.ToArray();
        m_strXMLOutput = objEncoding.GetString(bytes);

        // Check to see if we need to save the file...
        if (optSave.Checked && m_strOutPath.Length > 0)
        {
            FileStream fstmXMLOut = new FileStream(m_strOutPath, FileMode.Create);
            m_stmOutput.Position = 0;
            TextReader objTRFrom = new StreamReader(m_stmOutput);
            TextWriter pbjTWTo = new StreamWriter(fstmXMLOut);
            twTo.WriteLine(trFrom.ReadToEnd());
            twTo.Flush();
            fstmXMLOut.Close();
        } // if

        // Close the stream
        m_stmOutput.Close();
    } // try
} // cmdXForm_Click
```
LISTING 3.4  Continued

    // Enable the UI
    cmdView.Enabled = true;

    // Show results
    lblResults.ForeColor = SystemColors.ControlText;
    lblResults.Text = "Transformation successful, now click View";
} // try
catch (Exception ex)
{
    // Show results
    lblResults.ForeColor = Color.Red;
    lblResults.Text = "Transformation Unsuccessful";
    MessageBox.Show(ex.Message,
                    "XSLExerciser Error",
                    MessageBoxButtons.OK,
                    MessageBoxIcon.Error);
} // catch

A lot of this code you saw previously. One thing that we do differently is to create a memory-based stream, into which we shove the transformed XML:

    // Create a stream to contain the results
    m_stmOutput = new MemoryStream();

We do this because we then have the transformed XML in one location, ready to display or store into a file.

No single book has everything that you’ll need to work with a technology as diverse as XML. However, a few books are well worth examining to gain a bit more detail regarding this fascinating and pliant tool. Unfortunately, as of this printing, no .NET-specific XML books are available, but hopefully this chapter has given you the basics that you’ll need to move forward into the .NET world.

Summary

This chapter began with a discussion of XML and how it suits Web Services well as a wire protocol. For example, XML is loosely coupled and lends itself to a high degree of interoperability.
We then moved into some of the XML technologies that are necessary to help you work with Web Services and, more specifically, the SOAP protocol, if you have to dip into the SOAP packets to adjust their contents. For example, we looked at how XPath helps you dig out information from within the XML document. You establish a location step and hand that to the XPath processor, which returns to you a nodeset if any of the XML elements match your query. We looked at how the XPath axes and intrinsic functions add power to your searching capability.

We then moved to XLink, which enables you to link XML elements that don’t have natural hierarchical relationships. Two attributes, `href` and `id`, enable you to link the XLink elements. The `href` attribute “points” to the element with a matching `id` attribute.

The final general XML technology considered was XSL, which you can use to transform XML from one form to another. This might be necessary to transform SOAP from one version to another; the latest SOAP specification (1.2) was released just as this book was written. The main programmable feature that XSL presents is the template, in which an XPath query extracts a nodeset that is then transformed according to the contents of the template.

We finally turned to .NET and .NET’s handling of XML. You learned about all the .NET XML classes necessary to deal with the XML technologies discussed in the first half of the chapter. `XmlReader` and `XmlWriter`, for example, read and write XML to and from streams. `XPathDocument` and the related `XPathNavigator` enable you to traverse the XML content using either XPath queries or pull-model methods such as `MoveToFirstAttribute()`.

This chapter also provided several useful .NET applications that not only help you learn how the .NET XML classes are used but also serve as useful utilities by themselves. Hopefully you might find XPathExerciser and XSLExerciser are useful beyond the covers of this book.

Now it’s time to move beyond simple XML and see how it’s used in conjunction with the SOAP protocol. Chapter 4 discusses the SOAP protocol and tells how information is converted from memory into XML to be transmitted over the network.