1

XML Syntax

Extensible Markup Language (XML) is now in widespread use. Many applications on the Internet or residing on individual computers use some form of XML to run or manage the processes of an application. Earlier books about XML commented that XML was to be the "next big thing." Now, it *is* "the big thing." In fact, there really isn't anything bigger.

For this reason, you want to understand XML and its various applications. This book focuses on some of the more common ways to apply XML to the work you are doing today. Whether you need Web services, searching, or application configuration, you can find immediate uses for XML. This book shows you how to apply this markup language to your work.

This first chapter looks at the basics of XML, why it exists, and what makes it so powerful. Finally, this chapter deals with XML namespaces and how to properly apply them to XML instance documents. If you are already pretty familiar with the basics of XML, feel free to skim this chapter before proceeding.

The Purpose of XML

Before you actually get into the basics of XML, you should understand why this markup language is one of the most talked about things in computing today. To do this, look back in time a bit.

During the days of mainframes, information technology might have seemed complicated, but it actually got a heck of a lot more complicated when we moved from the mainframes and started working in a client-server model. Now the users were accessing information remotely instead of sitting at the same machine where the data and logic were actually stored. This caused all sorts of problems — mainly involving how to visually represent data that was stored on larger mainframes to remote clients. Another problem was application-to-application communication. How was one application sitting on one computer going to access data or logic residing on an entirely different computer?

Two problems had to be resolved. One dealt with computer-to-human communications of data and logic; another dealt with application-to-application communications. This is illustrated in Figure 1-1.

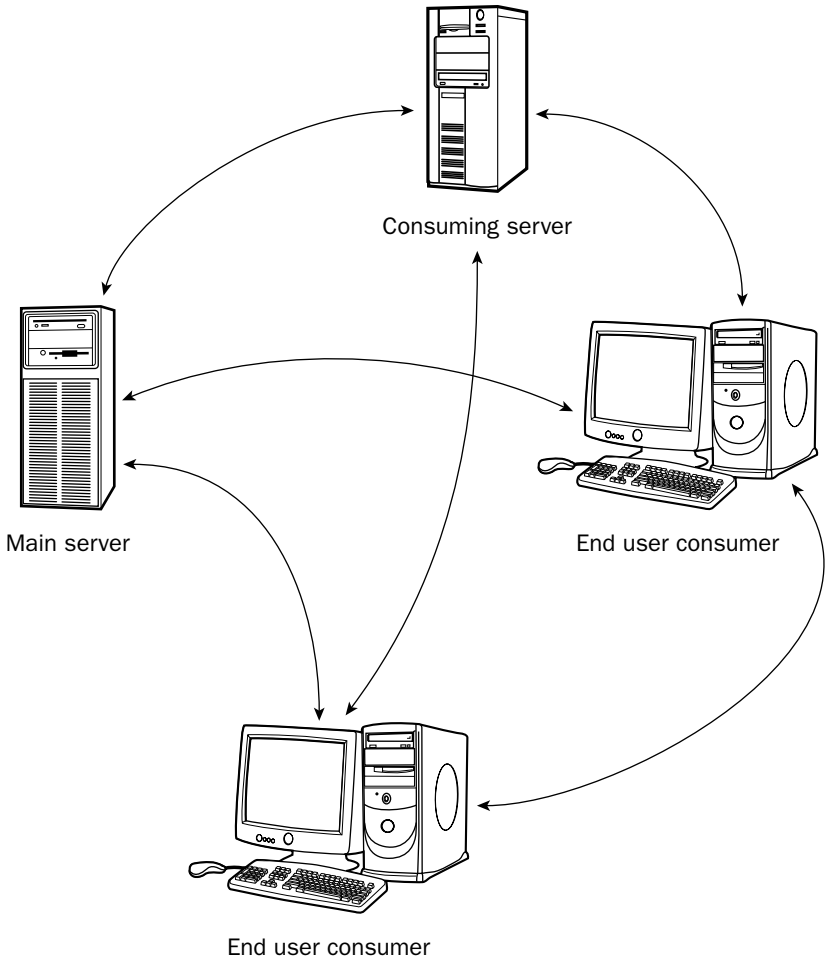

Figure 1-1

The first problem of computer-to-human communication of data and logic was really solved in a large way with the advent of HTML (also known as *HyperText Markup Language*). This markup language packaged data and logic in a way that allowed users to view it via applications specifically designed to present it (the birth of the browser as we know it). Now with HTML and browser applications in place, end users could work through data and logic remotely without too much of a problem.

With that said, it really isn't all about humans is it? There was also a need for other servers, processes, applications, and whatnot to access and act upon data and logic stored elsewhere on a network or across the planet. This created a pursuit to find the best way of moving this data and logic from point A to point B.

It was a tough task. The varying sources of data were often not compatible with the platform where the data was to be served up. A common way to structure and represent the data was needed. Of course, many solutions were proposed — some of which were pretty exciting.

The idea was to *mark up* a document in a manner that enabled the document to be understood across working boundaries. Many systems existed to mark up documents so that other applications could easily understand them. Applying markup to a document means adding descriptive text around items contained in the document so that another application or another instance of an application can decipher the contents of the document.

For instance, Microsoft Word provides markup around the contents of document. What markup is really needed? Well, as you type words into Microsoft Word, you are also providing data to be housed in the document. The reason you don't simply use Microsoft Notepad is that Word gives you the extra capability to change the way in which the data is represented. What this really means is that you can apply *metadata* around the data points contained in the document. For instance, you can specify whether a word, paragraph, or page is bolded, italicized, or underlined. You can specify the size of the text and the color. You can actually alter the data quite a bit. Word takes your instructions and applies a markup language around the data.

Like Word, XML uses markup to provide metadata around data points contained within the document to further define the data element. XML provides such an easy means of creating and presenting markup that it has become the most popular way to apply metadata to data.

In its short lifetime, XML has become the standard for data representation. XML came into its own when the W3C (*The World Wide Web Consortium*) realized that it needed a markup language to represent data that could be used and consumed regardless of the platform. When XML was created in 1998, it was quickly hailed as the solution for data transfer and data representation across varying systems.

In the past, one way to represent data was to place the data within a comma-, tab-, or pipe-delimited text file. Listing 1-1 shows an example of this:

Listing 1-1: An example of a pipe-delimited data representation

```
Bill|Evjen|Technical Architect|Lipper|10/04/2001|St. Louis, Missouri|3
```

These kinds of data representations are in use today. The individual pieces of data are separated by pipes, commas, tabs, or any other characters. Looking at this collection of items, it is hard to tell what the data represents. You might be able to get a better idea based on the file name, but the meaning of the date and the number 3 is not that evident.

On the other hand, XML relates data in a self-describing manner so that any user, technical or otherwise, can decipher the data. Listing 1-2 shows how the same piece of data is represented using XML.

Listing 1-2: Representing the data in an XML document

```
<?xml version="1.0" encoding="UTF-8" ?>
<Employee>
   <FirstName>Bill</FirstName>
   <LastName>Evjen</LastName>
```

(continued)

Listing 1-2 *(continued)*

```
      <JobTitle>Technical Architect</JobTitle>
      <Company>Lipper</Company>
      <StartDate>10/04/2001</StartDate>
      <WorkLocation>St. Louis, Missouri</WorkLocation>
      <NumberOfDependents>3</NumberOfDependents>
   </Employee>
```

You can now tell, by just looking at the data in the file, what the data items mean and how they relate to one another. The data is laid out in such a simple format that is quite possible for even a non-technical person to understand the data. You can also have a computer process work with the data in an automatic fashion.

When you look at this XML file, you may notice how similar XML is to HTML. Both markup languages are related, but HTML is used to mark up text for presentation purposes whereas XML is used to mark up text for data representation purposes.

Both XML and HTML have their roots in the Standard Generalized Markup Language (SGML), which was created in 1986. SGML is a complex markup language that was also used for data representation. With the explosion of the Internet, however, the W3C realized that it needed a universal way to represent data that would be easier to use than SGML. That realization brought forth XML.

XML has a distinct advantage over other forms of data representation. The following list represents some of the reasons XML has become as popular as it is today:

- XML is easy to understand and read.
- A large number of platforms support XML and are able to manage it through an even larger set of tools available for XML data reading, writing, and manipulation.
- XML can be used across open standards that are available today
- XML allows developers to create their own data definitions and models of representation.
- Because a large number of XML tools are available, XML is simpler to use than binary formats when you want to represent complex data structures.

XML Syntax and Rules

Building an XML document properly means that you have to follow specific rules that have been established for the structure of the document. These rules of XML are defined by the XML specification found at `w3.org/TR/REC-xml`. If you have an XML document that follows the rules diligently, it is a *well-formed* XML document.

You want to make sure that the rules are followed closely because if the rules defined in the XML specification are observed, you can use various XML processors (or *parsers*) to work with your documents in an automatic fashion.

XML Parsers

You might not realize it, but you probably already have an XML parser on your computer. A number of computer vendors have provided XML parsers and have even gone as far as to include these parsers in applications that you use each and everyday. The following is a list of some of the main parsers.

- **Microsoft's Internet Explorer XML Parser** — The most popular XML parser on the market is actually embedded in the number-one browser on the market. Microsoft's Internet Explorer comes with a built-in XML parser — Microsoft's XML Parser. Internet Explorer 5.5 comes with Microsoft's XML Parser 2.5 whereas Internet Explorer 6.0 comes with the XML Parser 3.0. This parser includes a complete implementation of XSL Transformations (XSLT) and XML Path Language (XPath) and incorporates some changes to work with Simple API for XML (SAX2). You can get the XML Parser 3.0 via an Internet Explorer download, or you can download the parser directly from `microsoft.com/downloads/details.aspx?familyid=4A3AD088-A893-4F0B-A932-5E024E74519F&displaylang=en`
- **Mozilla's XML Parser (also the Firefox XML Parser)** — Like Internet Explorer, Mozilla includes support for XML parsing. Mozilla has the built-in capability to display XML with CSS.*
- **Apache Xerces** — This open source XML parser can be found online at `http://xerces.apache.org/` and comes under the Apache Software License. This parser is available for Java, C++, and a Perl implementation that makes use of the C++ version. Apache Xerces was originally donated to Apache by IBM in 1999. Until 2004, Apache Xerces was a subproject of the Apache XML Project found at `http://xml.apache.org/`.
- **IBM's XML Parser for Java** — Also known as Xml4j, this parser has become the Apache Xerces2 Java Parser found at `http://xerces.apache.org/xerces2-j/`.
- **Oracle XML Parser** — Oracle provides XML parsers for Java, C, C++, and PL/SQL through its Oracle XML Developer's Kit 10g found at `oracle.com/technology/tech/xml/xdkhome.html`.
- **Expat XML Parser** — Written by James Clark, the tech-lead of the W3C's XML activity that created the XML 1.0 Specification, you can find the Expat parser as a SourceForge project found at `http://expat.sourceforge.net/`. Expat is currently in version 2.0.

XML Elements and Tags

When reading and conversing about XML, you come across the terms *element* and *tag* quite often. What's the difference between the two? Many individuals and organizations incorrectly use these terms interchangeably. Each term has a distinct meaning.

An XML element is the means to provide metadata around text to give it further meaning. For instance, you might be presented with the following bit of XML:

```
<City>Saint Charles</City>
```

In this case, the element is the entire item displayed. XML uses *tags* to surround text in order to provide the appropriate metadata. Figure 1-2 shows the pieces of this bit of code.

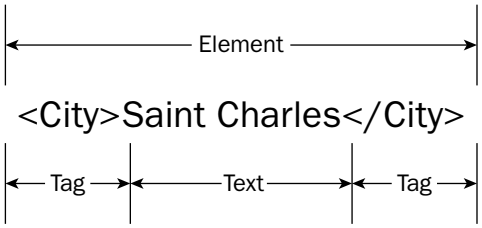

Figure 1-2

From this, you can see that everything from the starting `<City>` to the ending `</City>` is the XML element. An XML element is made up a *start tag*, which precedes the text to be defined, as well as an *end tag*, which comes at the end of the text to be defined. In this case, the start tag is `<City>` and the end tag is `</City>`.

Element Syntax

If there is text to be marked up with XML, then an XML element must contain start and end tags. XML is very strict about its rules, and you must follow them just as strictly if you want to ensure that your XML document is well-formed.

XML Elements Must Have a Start and End Tag

Unlike HTML, where you can bend the rules of the syntax utilized, XML requires a start and end tag if an element contains any text. The following shows two XML elements together.

```
<City>Saint Charles</City>
<State>Missouri</State>
```

Naming Conventions for Elements

You can choose any name that suits your fancy for the elements of your XML document. With that said however, certain rules do restrict the names that you can use for elements.

Element names must start with a letter from an alphabet of one of the languages of the world or an underscore. You cannot start an element name with a number or a special character (such as !, @, #, $, %, and so on).

Examples of improper naming of XML elements include the following:

- `<123Industries></123Industries>`
- `<#Alpha></#Alpha>`
- `<!Yellow></!Yellow>`

Examples of well-formed XML elements include these:

```
<StLouisCardinals></StLouisCardinals>

<Item123></Item123>

<_Wowzer></_Wowzer>

<__></__>
```

Element names cannot contain spaces. This means that the following XML element name is improper and not well-formed:

```
<Bill Evjen></Bill Evjen>
```

Element names cannot start with the word `XML` in any case. For example, the following element names are improper and not well-formed:

```
<xml></xml>

<XML></XML>

<XmlLover></XmlLover>

<XML_Element1></XML_Element1>
```

After you have defined the first character of your XML element, you can subsequently use numbers, periods, underscores, or hyphens. The following are examples of well-formed XML:

```
<St.Louis_Cardinals></St.Louis_Cardinals>

<Item1></Item1>

<Address-Present></Address-Present>
```

Immediately after the opening < and </ of the XML tags, you must start the element name. You cannot have a space first. This means that the following XML element is improper:

```
< Item1></ Item>
```

Although a space is not allowed preceding the element name, you can have a space trailing the element name before the closing of the tag. This use is illustrated in the following example:

```
<Item1 ></Item1 >
```

XML Elements Must Be Properly Nested

When XML documents contain more than one XML element (which they invariably do), you must properly nest the XML elements. You are required to open and close these elements in a logical order. When looking at the preceding XML fragment, you can see that the `<City>` tag is closed with a `</City>` tag before the `<State>` opening tag is utilized. The following fragment is *not* well-formed.

```
<City>Saint Charles
<State></City>Missouri</State>
```

However, you are not required to always close an element before starting another one. In fact, the opposite is true. XML allows for a hierarchical view of the data that it represents. This means that you can define child data of parent data directly in your XML documents; this enables you to show a relationship between the data points.

```
<Location>
   <City>Saint Charles</City>
   <State>Missouri</State>
   <Country>USA</Country>
</Location>
```

The indenting of the XML file is done for readability and is not required for a well-formed document.

This XML fragment starts with the opening XML element `<Location>`. Before the `<Location>` element is closed however, three other XML elements are defined — thereby further defining the item. The `<Location>` element here contains three *subelements* — `<City>`, `<State>`, and `<Country>`. Being subelements, these items must also be closed properly before the `<Location>` element is closed with a `</Location>` tag.

You can also continue the nesting of these elements so that they are aligned hierarchically as deep as you wish. For instance, you can use the following structure for your nested XML fragment.

```
<Person>
   <Name>Bill Evjen</Name>
   <Location>
      <City>Saint Charles</City>
      <State>
         <Name>Missouri</Name>
         <StateCode>MO</StateCode>
      </State>
      <Country>USA</Country>
   </Location>
</Person>
```

In this case, the `<Person>` element contains two child elements or subelements — `<Name>` and `<Location>`. The `<Name>` element is a simple element, whereas the `<Location>` element is further nested two more times with additional subelements.

Empty Elements

If the text or item you want to define in your XML document is null or not present for some reason, you can represent this item through the use of an empty XML element. An empty XML element takes the following format:

```
<Age/>
```

In this case, the XML element is still present, but is represented as an empty value through a single XML tag. When representing an empty element, you do not need an opening and closing tag, but instead just a single tag which ends with `/>`.

In addition to the empty element representation shown here, you can also have a space between the word used to define the tag and the closing of the tag.

```
<Age />
```

In addition to using a single tag to represent an empty element, you can also use the standard start and end tags with no text to represent an empty element. This is illustrated here:

```
<Person>
   <Name>Bill Evjen</Name>
   <Age></Age>
   <Location>
      <City>Saint Charles</City>
      <State>
         <Name>Missouri</Name>
         <StateCode>MO</StateCode>
      </State>
      <Country>USA</Country>
   </Location>
</Person>
```

Tag Syntax

Tags are defined using greater-than/less-than signs (`<Tag>`). A start tag has a textual name preceded with a < and ending with a >. An end tag must have the same textual name as its start tag, but it is preceded by a `</` as opposed to a <. The end tag is finalized with a > just as the start tag is.

The words you use for tag names are entirely up to you, but some basic rules govern how you build tags. The first rule is that the case used for the start tag and the end tag must be the same. Therefore, the `<Location>` tag is not the same as `<location>`. For instance, this is considered improper or malformed XML:

```
<Country>USA</country>
```

Because XML is case-sensitive, the tags shown here are actually completely different tags and, therefore, don't match. For your XML to be well-formed, the XML tags must be of the same case.

```
<Country>USA</Country>
```

Because XML does understand case, you could, theoretically, have the following XML snippet in your XML document:

```
<Name>Bill Evjen</Name>
<name>Bill</name>
```

Although completely legal, you shouldn't actually implement this idea because it causes confusion and can lead to some improper handling of your XML documents. Remember that you want to build XML documents that are easily understandable by the programmers who will build programs that process these documents.

XML Text

The text held within an XML element can be whatever you wish. The entire point of the XML document is to hold information using XML elements as markup. Remember a few rules, however, when you are representing content within your XML elements.

Text Length

You have no rules on the length of the text contained within your XML documents. This means that the content can be of any length you deem necessary.

```
<Message>
   This can go on and on and on and on and on and on and on and on and on
   and on and on ...
</Message>
```

Content

You might think that the content of an XML element is just text for humans to read, but an XML element really can contain just about anything. For instance, you can use binary code to represent an image or other document and then stick this item in your XML document. This is illustrated here with a partial element:

```
<base64Binary>/9j/4AAQSkZJRgABAAgEASABIAAD/7RNoGUGhvdG9ZaG9wIDMuMAA4QklNAAQBIAAAAAQ
 ABOEJJTQQNAAAAAAEAAAAAeDhCSu0D8 ...
</base64Binary>
```

Spoken Languages

Of course, XML is for the world, and this means that you can write content in any language you want. Here are some examples of proper XML:

```
<Message> 私は別の言語を話している。</Message>

<Message>_ _____ __ _____ _____.</Message>

<Message>Estoy hablando otra lengua.</Message>

<Message> 我講其它語言。</Message>
```

When working with an XML file that contains a fragment such as this, notice that the XML parser has no problem working with the content. See Figure 1-3.

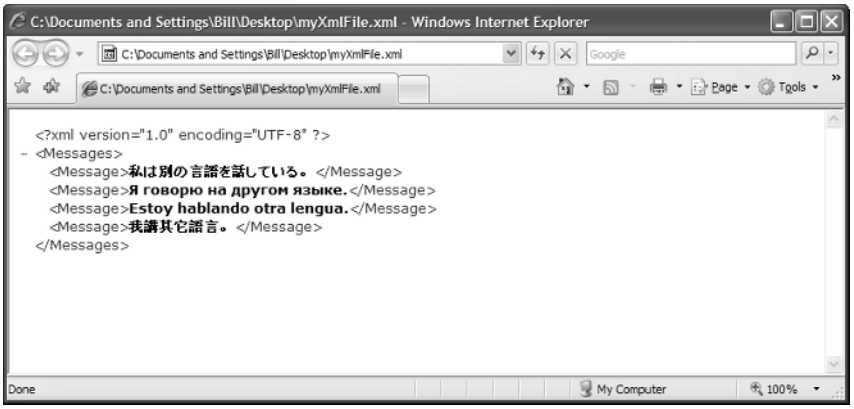

Figure 1-3

Whitespace

Whitespace is a special character in its own right. Whitespace is the space, line feeds, tabs, and carriage returns within your XML document. An example XML document containing various whitespace elements is presented here:

```
<Movies>
   <Favorites>
      <Title>Happy  Gilmore</Title>
      <Title>Grease</Title>
      <Title>Lawrence
             of
             Arabia</Title>
      <Title>Star Wars -         The Empire Strikes Back</Title>
   </Favorites>
</Movies>
```

HTML parsers do a good job of ignoring the whitespace contained within a document. In fact, in HTML, if you want to force the HTML parsers to interpret the whitespace contained within an HTML document, you have to put `<pre>` tags around the text.

XML works in the opposite manner. All whitespace is preserved in XML documents. This means that if there are two spaces between two words, these spaces are maintained by any XML parser and, consequently, they pass to the consuming application. The consuming application can choose whether to process the whitespace. Certain applications or processes strip the whitespace from the document, and others do not.

For example, Microsoft's Internet Explorer receives whitespace from the XML document and then strips it out in the consumption process. The previous XML document produces the results shown in Figure 1-4 when it is viewed in Internet Explorer.

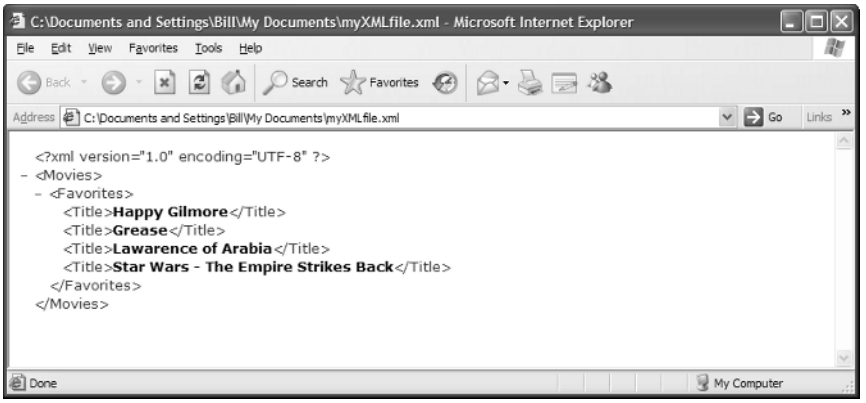

Figure 1-4

Entity References

Although you can put about just anything in the text part of an XML element, some characters cannot be contained as a value within an XML element. Take a look at the following code to see if you can tell where a problem might occur.

Incorrect usage of text within an XML element

```
<Value>Do if 5 < 3</Value>
```

You should be able to tell right away that a processing error will occur because of the character directly after the 5. The *greater than* sign is used to close an XML tag; but here it is used as a textual value within an XML element, and so it will confuse the XML parser. In fact, if you run this in Internet Explorer, you are presented with the error directly. See Figure 1-5.

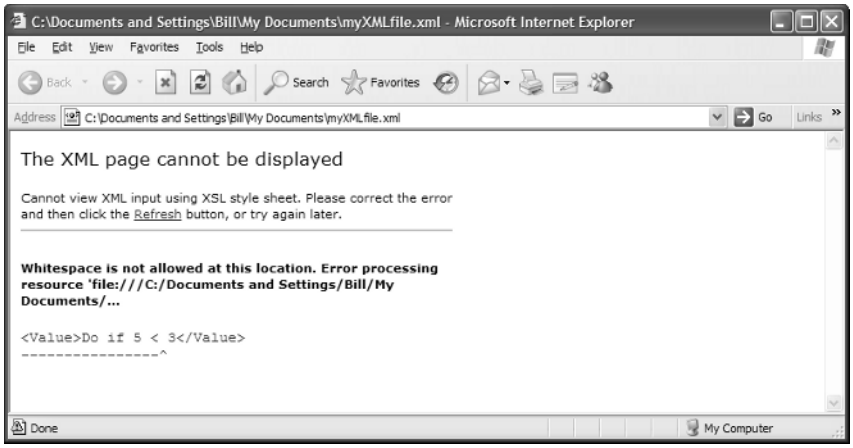

Figure 1-5

You can see that a parsing error is automatically thrown because the XML parser thinks that the less than sign is actually the start of the closing tag of the element. The space behind the character causes the parser to throw an error because it sees a whitespace problem.

The trick is to encode this character so that the XML parser can treat it in the appropriate manner. Five characters that cause an error and, therefore, must be encoded are shown in the following table with their encoded values.

Character	Entity
<	<
>	>
“	"
‘	'
&	&

With this knowledge, you can now write the XML element as follows:

```
<Value>Do if 5 &lt; 3</Value>
```

If you run this element in Internet Explorer, you get the correct output as presented in Figure 1-6.

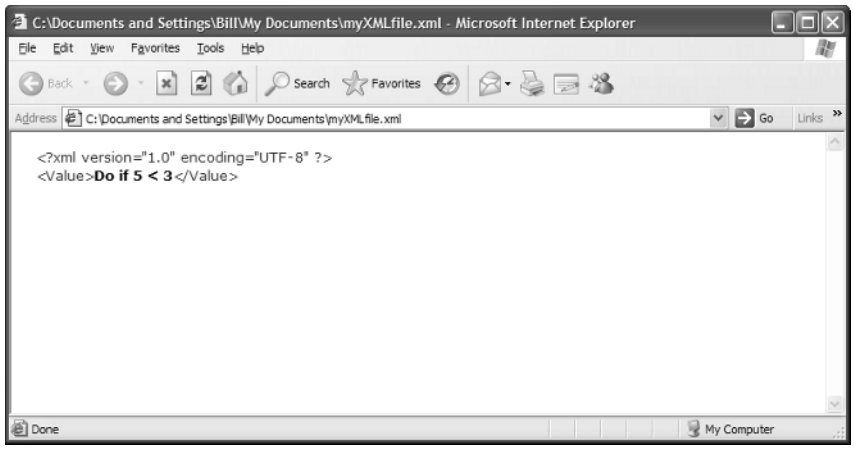

Figure 1-6

The XML with the encoded character was passed to the XML parser used by Internet Explorer, and the XSL stylesheet then converts the encoded character to the format in which it should be represented visually.

CDATA Sections

One way to work with some of the character entities that XML parsers can't easily interpret is to encode the character.

```
<Value>Do if 5 &lt; 3</Value>
```

When you have a lot of items that need this type of encoding (especially if you are representing computer code within your XML document), you should check out the CDATA section capability found in XML.

Creating a CDATA section within the text value of your XML element allows you, with little work on your part, to use as many difficult characters as you wish. Representing the previous code within a CDATA section is accomplished in the following manner:

```
<Value><![CDATA[Do if 5 < 3]]></Value>
```

You can use this method to represent large content sets that might require a lot of escape sequences. This method is shown in Listing 1-3.

Listing 1-3: Representing text using the CDATA section

```
<?xml version="1.0" encoding="UTF-8" ?>
<Value>
   <![CDATA[
   <script runat="server">
    protected void DropDownList1_SelectedIndexChanged(object sender, EventArgs e)
    {
        string[] CarArray = new string[4] {"Ford", "Honda", "BMW", "Dodge"};
        string[] AirplaneArray = new string[3] {"Boeing 777", "Boeing 747",
           "Boeing 737"};
        string[] TrainArray = new string[3] {"Bullet Train", "Amtrack", "Tram"};

        if (DropDownList1.SelectedValue == "Car") {
            DropDownList2.DataSource = CarArray; }
        else if (DropDownList1.SelectedValue == "Airplane") {
            DropDownList2.DataSource = AirplaneArray; }
        else {
            DropDownList2.DataSource = TrainArray;
        }

        DropDownList2.DataBind();
        DropDownList2.Visible = true;
    }

    protected void Button1_Click(object sender, EventArgs e)
    {
        Response.Write("You selected <b>" +
           DropDownList1.SelectedValue.ToString() + ": " +
           DropDownList2.SelectedValue.ToString() + "</b>");
    }
   </script>
   ]]>
</Value>
```

The start of the CDATA section is defined with `<![CDATA[`. After this entry, you can place as much text as you wish. The XML parser looks for a closing `]]>` before ending the CDATA section. To make this work, be careful that you don't have this sequence of characters in your text. The previous XML document displayed in IE is shown in Figure 1-7.

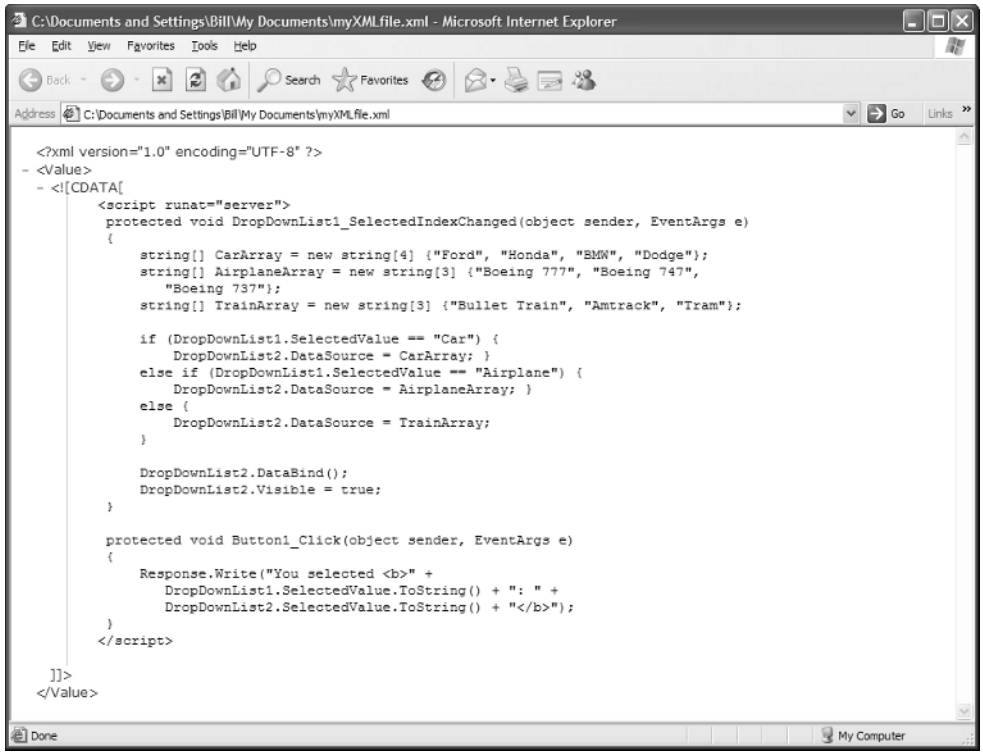

Figure 1-7

The XML Document

Now that you have studied the pieces that make up an XML document, you can turn your attention to the entire XML document.

Creating an .xml File

Like all files, XML files have a file extension. In many cases, XML files have an `.xml` file extension, but this is not the only one used. Certain XML files have their own file extensions. For instance, if you have the .NET Framework installed on your computer, it includes many configuration files with a `.config` file extension. If you look at one of these `.config` files within Microsoft's Notepad, you see that they are indeed XML files. (See Figure 1-8.)

XML file are created in a number of ways. Many outstanding tools are out there to help you with the construction and creation of XML files. Tools such as Altova's XMLSpy or Microsoft's Visual Studio 2005 give you what you need to get the job done. Because this is the first XML file you are working with, this example concentrates on building the XML file using Microsoft's Notepad.

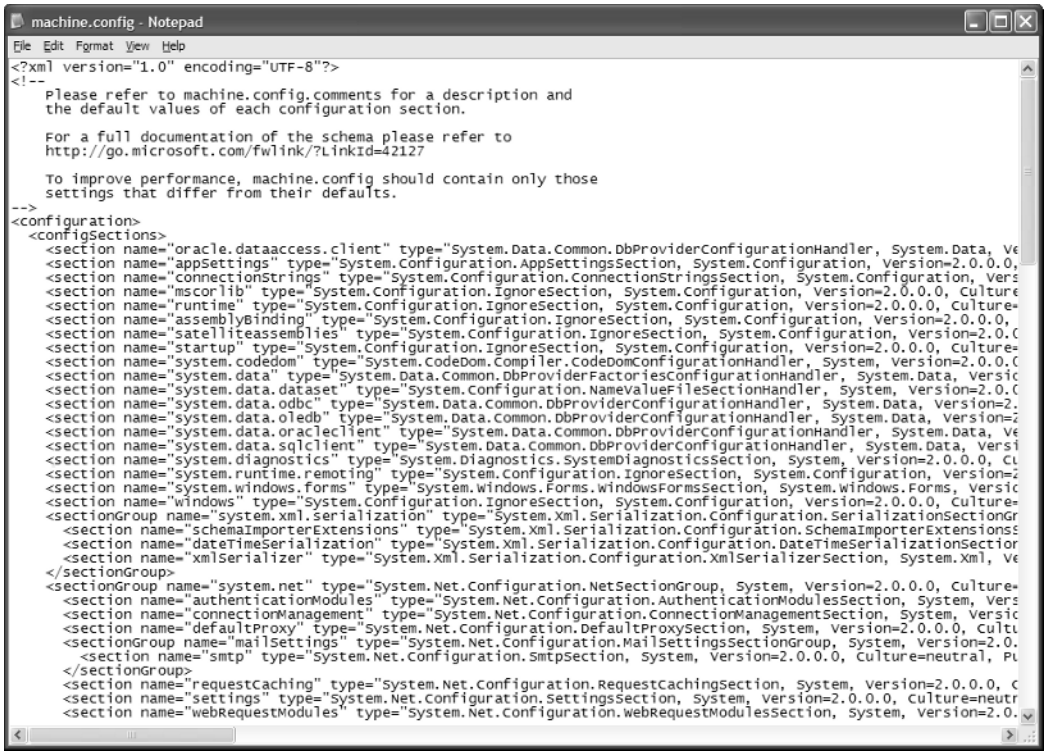

Figure 1-8

XML tools are covered in Chapter 2.

Now, open Notepad and type the XML shown in Listing 1-4.

Listing 1-4: Creating an XML file

```
<?xml version="1.0" encoding="UTF-8" ?>
<Process>
   <Name>Bill Evjen</Name>
   <Address>123 Main Street</Address>
   <City>Saint Charles</City>
   <State>Missouri</State>
   <Country>USA</Country>
   <Order>
      <Item>52-inch Plasma</Item>
      <Quantity>1</Quantity>
   </Order>
</Process>
```

After your have typed this small XML document into Notepad, save it as `myXMLfile.xml`. Make sure you save it as presented in Figure 1-9.

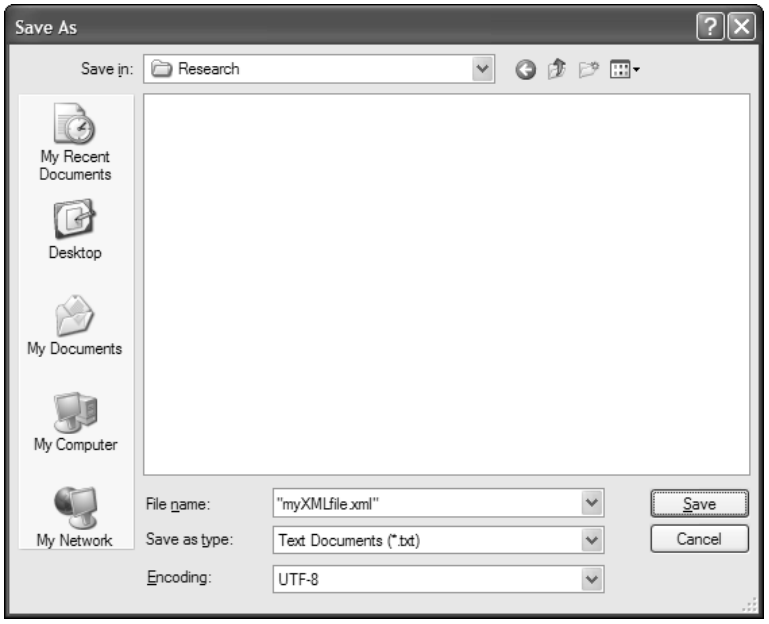

Figure 1-9

Put the name of the file along with the file extension `.xml` in quotes within the Filename text box of the Save As dialog. This ensures that the file won't be saved with a `.txt` file extension. You also want to change the encoding of the file from ANSI to UTF-8 because this is the format used for many XML files. This enables any XML parsers to interpret a larger character base than otherwise.

After you do this, click the Save button and you can then double-click on the new `.xml` file. This opens up in Internet Explorer if you are using a Windows operating system. You might be wondering why Internet Explorer is the default container for XML files. You can actually manually change this yourself by going into the Properties dialog of one of your XML files. Figure 1-10 shows the Properties dialog of the newly created `myXMLfile.xml`.

From the Properties dialog, you can see in the first section that the `Opens with` property is set to Internet Explorer. You can easily change the application used to open the file by clicking the Change button.

Internet Explorer produces the results presented in Figure 1-11.

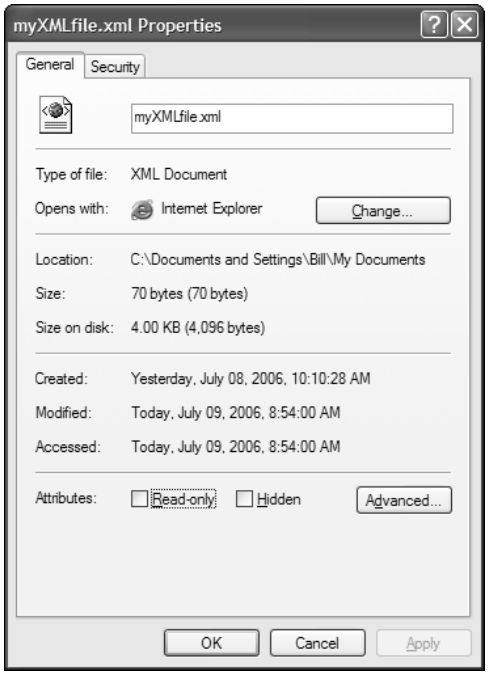

Figure 1-10

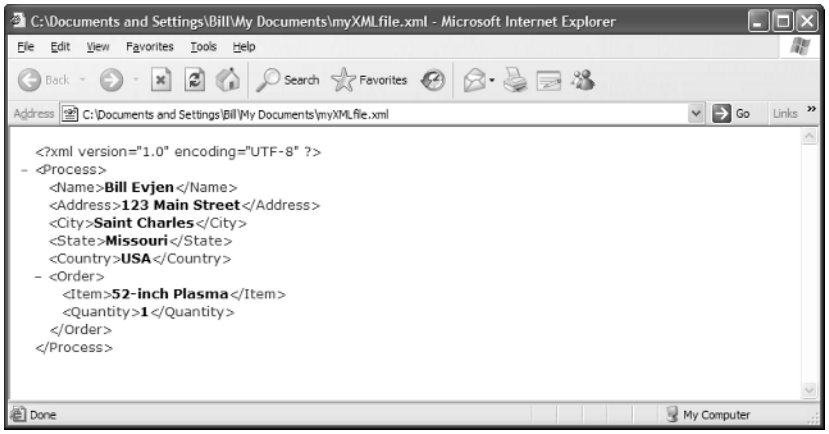

Figure 1-11

You can see that XML files are color-coded for easy viewing. IE is using the Microsoft XML parser and then applies an XSL stylesheet to beautify the results. The interesting part of the XML document as it is presented in IE is that you can expand and collapse the child nodes for easy readability. By default, the XML document is presented with the entire document expanded, but you can start collapsing nodes by clicking the minus button next to nodes that have children associated with them. Figure 1-12 shows various nodes collapsed in IE.

Using Mozilla's Firefox for the same XML file produces the results presented in Figure 1-13.

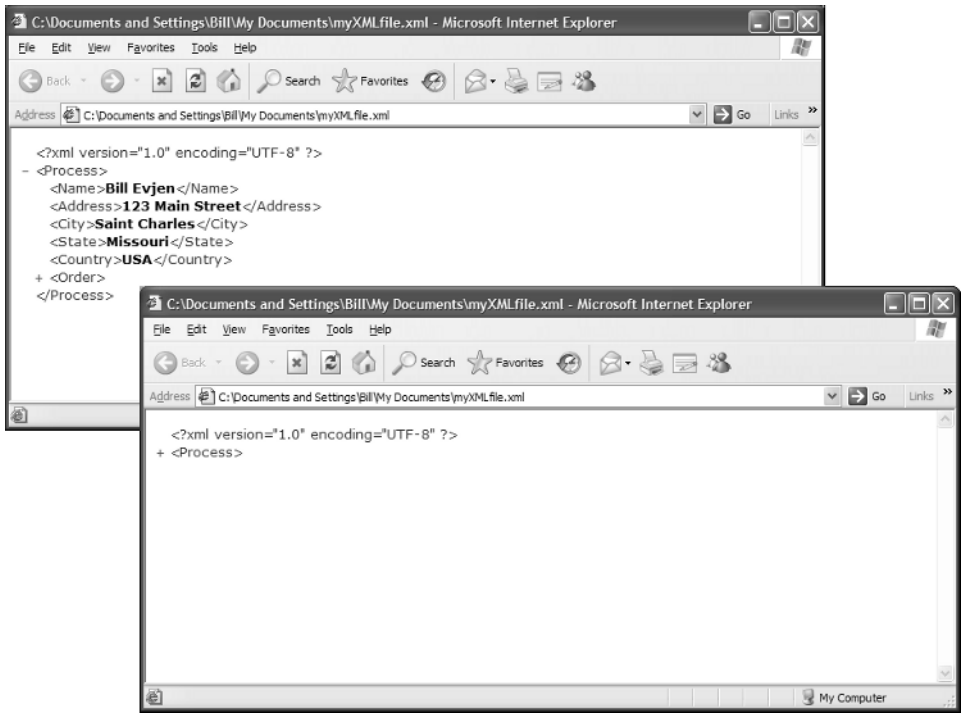

Figure 1-12

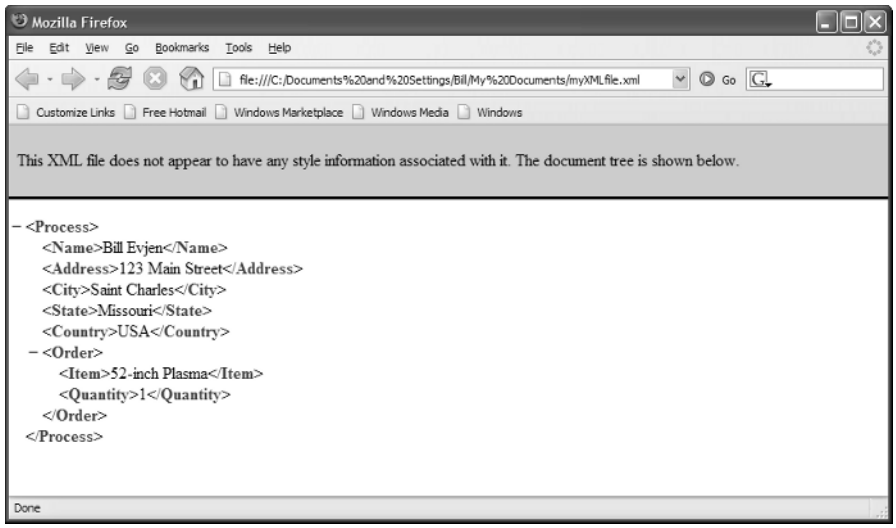

Figure 1-13

As you can see from Figure 1-13, Firefox also allows you to expand and collapse the nodes of the XML document.

Now that you understand a little more about the XML document, the next sections review the construction of the XML document.

The XML Declaration

Typically, you place a declaration at the top of your XML file stating that this is an XML file. It is called the *XML declaration*. It is recommended, but not required, that you use an XML declaration. The XML should be written in the following format:

```
<?xml version="1.0" ?>
```

In this case, the XML declaration starts with a `<?xml` and ends with a `?>`. Within the declaration, you can include a couple of different key/value pairs to further define the XML document to help parsers understand how to process the forthcoming XML document.

If you include the XML declaration, the only required attribute is the `version` attribute. The other possible attributes are `encoding` and `standalone`. One difference between this set of attributes in the XML declaration and normal attributes that you would find in any other XML element is that `version`, `encoding`, and `standalone` are required in this particular order when other attributes have no such requirements.

The version Attribute

The `version` attribute allows you to specify the version of the XML used in the XML document.

```
<?xml version="1.0" ?>
```

This preceding XML declaration signifies that version 1.0 of XML is used in the XML document. All values defined by the version and other attributes in the XML declaration must be placed within quotes. It is important to note that at present, the only version you can use is 1.0.

The encoding Attribute

`encoding` explains how a computer interprets 1's and 0's. These 1's and 0's are put together to represent various characters, and the computer can interpret these digits in a number of different ways.

The United States and its computer encoding formats evolved around an encoding technology called ANCII. ANCII, *American Standard Code for Information Interchange*, is very limiting in that it only allows 256 possible values. It works with the English language, but it is less effective when working with multiple languages with their many characters.

Because XML was developed by an international organization, it is uses Unicode for encoding an XML document. So far, you have seen the use of UTF-8 used for encoding some of the XML documents in this chapter.

```
<?xml version="1.0" encoding="UTF-8" ?>
```

The more common encoding formats used for XML documents are UTF-8 and UTF-16. The difference between the two is that UTF-8 can result in smaller file size because it can use either a single byte (mostly for English characters) or a double byte for other characters. UTF-16 uses a double-byte for all characters.

XML parsers must understand at least UTF-8 and UTF-16. If no encoding directive is provided, then UTF-8 is assumed. Some common character sets are presented in the following table.

Character Set Code Name	Coverage
US-ASCII	English
UTF-8	Compressed Unicode
UTF-16	Compressed UCS
windows-1252	Microsoft Windows Western European
windows-1250	Microsoft Windows Central European
windows-1251	Microsoft Windows Cyrillic
windows-1253	Microsoft Windows Greek
ISO-8859-1	Latin 1, Western European
ISO-8859-2	Latin 2, Eastern European
ISO-8859-3	Latin 3, Southern European
ISO-8859-4	Latin 4, Northern European
ISO-2022-JP	Japanese

The standalone Attribute

Another optional attribute that can be contained within the XML declaration is the `standalone` attribute. The `standalone` attribute signifies whether the XML document requires any other files in order to be understood or whether the file can be completely understood as a *standalone* file. By default, the value of this attribute is set to `no`.

```
<?xml version="1.0" standalone="no" ?>
```

If the document does not depend upon other documents in order to be complete, set the `standalone` attribute to reflect this.

```
<?xml version="1.0" standalone="yes" ?>
```

XML Comments

As in HTML, you can easily place XML comments inside your XML documents. Comments placed inside the XML document are ignored by any XML parser. Listing 1-5 shows how you can add comments to the XML document displayed in Listing 1-4.

Listing 1-5: Adding comments to the previous XML document

```
<?xml version="1.0" encoding="UTF-8" ?>
<Process>
   <!-- Be sure to check name against customer database later -->
   <Name>Bill Evjen</Name>
   <Address>123 Main Street</Address>
   <City>Saint Charles</City>
   <State>Missouri</State>
   <Country>USA</Country>
   <Order>
      <Item>52-inch Plasma</Item>
      <Quantity>1</Quantity>
   </Order>
</Process>
```

An XML comment starts with a `<!--` and ends with a `-->`. Anything found in between these two items is considered the comment and is ignored by the XML parser. You are not required to put XML comments on a single line. You can break them up in multiple lines if you wish. This is illustrated in Listing 1-6.

Listing 1-6: Adding comments on multiple lines

```
<?xml version="1.0" encoding="UTF-8" ?>
<Process>
   <!--
        Be sure to
        check name against
        customer database later
    -->
   <Name>Bill Evjen</Name>
   <Address>123 Main Street</Address>
   <City>Saint Charles</City>
   <State>Missouri</State>
   <Country>USA</Country>
   <Order>
      <Item>52-inch Plasma</Item>
      <Quantity>1</Quantity>
   </Order>
</Process>
```

If you open the XML document in IE, you see that the Microsoft XML parser did indeed interpret the XML comment as a comment because it is shown in gray, unlike the other XML elements. This is illustrated in Figure 1-14.

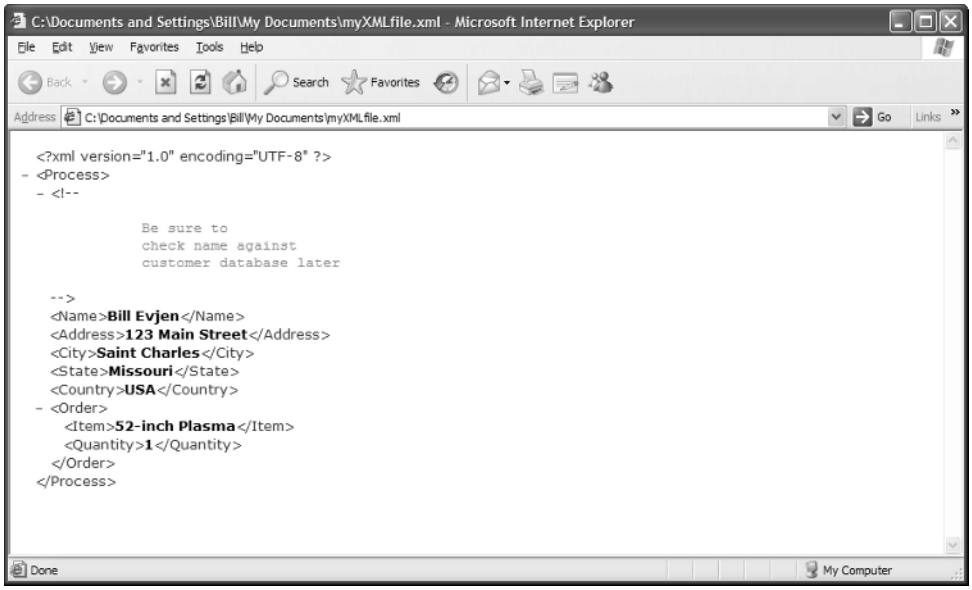

Figure 1-14

XML Processing Instructions

An XML processing instruction lets you direct computer process reactions. You won't see XML processing instructions used in the XML documents you interact with mainly because they are not accepted by all parsers and are often difficult to deal with. An example of an XML processing instruction is illustrated in Listing 1-7.

Listing 1-7: Using XML processing instructions

```
<?xml version="1.0" encoding="UTF-8" ?>
<Process>
   <?CustomerInput INPUT:Evjen?>
   <Name>Bill Evjen</Name>
   <Address>123 Main Street</Address>
   <City>Saint Charles</City>
   <State>Missouri</State>
   <Country>USA</Country>
   <Order>
      <Item>52-inch Plasma</Item>
      <Quantity>1</Quantity>
   </Order>
</Process>
```

In this case, the CustomerInput application that interprets this XML document can use the INPUT instruction and accomplish something with the `Evjen` value as provided by the XML processing instruction.

You would use the following syntax for processing instructions:

```
<?[target] [data]?>
```

The `target` part of the statement is a required item and it must be named in an XML-compliant way. The `data` item itself can contain any character sequence except for the `?>` set of characters, which signify the closing of a processing instruction.

Attributes

So far, you have seen the use of XML elements and tags and how they work within an XML document. One item hasn't yet been discussed — XML attributes. XML elements provide values via an attribute that is basically a key/value pair contained within the start tag of an XML element. The use of an XML attribute is presented in Listing 1-8.

Listing 1-8: Creating an XML file

```
<?xml version="1.0" encoding="UTF-8" ?>
<Process>
   <Name>Bill Evjen</Name>
   <Address type="Home">123 Main Street</Address>
   <City>Saint Charles</City>
   <State>Missouri</State>
   <Country>USA</Country>
   <Order count="1">
      <Item>52-inch Plasma</Item>
      <Quantity>1</Quantity>
   </Order>
</Process>
```

In this case, this XML document has two attributes — `type` and `count`. All attributes must include the name of the attribute followed by an equal sign and the value of the attribute contained within quotes (single or double). It is illegal not to include the quotes.

Illegal XML Element

```
<Address type=Home>123 Main Street</Address>
```

Naming Attributes

The names you give your attributes must follow the same rules that you follow when naming XML elements. This means that the following attributes are illegal.

Illegal Attribute Names

```
<myElement 123type="Value"></myElement>

<myElement #type="Value"></myElement>

<myElement .type="Value"></myElement>
```

Empty Attributes

If an attribute doesn't have a value, you can represent this empty or null value as two quotes next to each other as illustrated here:

```
<Address type="">123 Main Street</Address>
```

Attribute Names Must Be Unique

All attribute names must be unique within an XML element. You cannot have two attributes with the same name in the same element (as presented here):

```
<Address type="Home" type="Mail">123 Main Street</Address>
```

If you use this XML element, you would get an error when parsing the XML document — as illustrated in Figure 1-15.

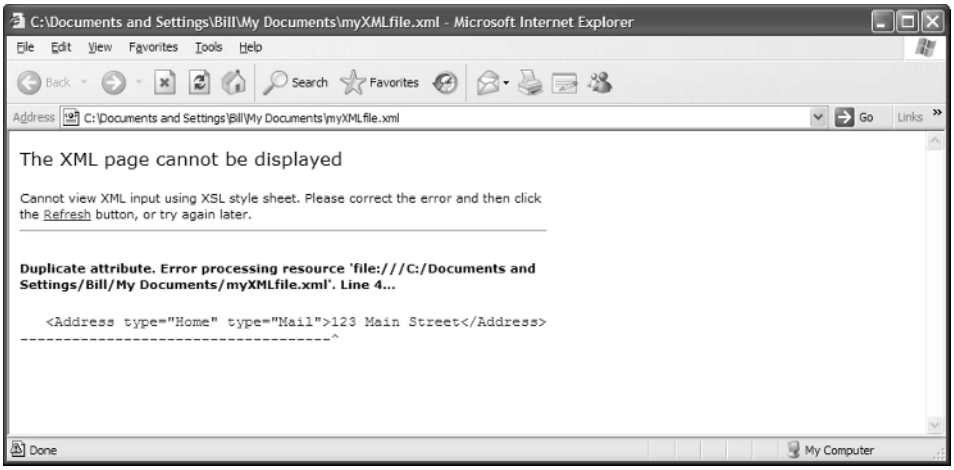

Figure 1-15

This only applies to attributes that are contained within the same XML element. You can have similar attribute names if they are contained within different XML elements. This scenario is shown in Listing 1-9.

Listing 1-9: Creating an XML file

```
<?xml version="1.0" encoding="UTF-8" ?>
<Process>
   <Name>Bill Evjen</Name>
   <Address type="Home">123 Main Street</Address>
   <City>Saint Charles</City>
   <State>Missouri</State>
   <Country>USA</Country>
   <Order type="Express">
      <Item>52-inch Plasma</Item>
      <Quantity>1</Quantity>
   </Order>
</Process>
```

In this case, you can see that there are two attributes that use the name `type`, but because they are contained within different XML elements, you won't encounter any errors in their use.

The xml:lang Attribute

Two built-in XML element attributes can be used in XML documents — `xml:lang` and `xml:space`. The first of these, `xml:lang`, allows you to specify the language of the item represented as the value in the XML element.

You can use as a value either the ISO 639 standard (found at `http://ftp.ics.uci.edu/pub/ietf/http/related/iso639.txt`), the ISO 3166 standard (found at `http://ftp.ics.uci.edu/pub/ietf/http/related/iso3166.txt`), or the IANA language codes (found at `iana.org/assignments/lang-tags/`).

Listing 1-10 shows how you might represent the earlier XML fragment when using the `xml:lang` attribute with the ISO 639 standard.

Listing 1-10: Using ISO 639 with the xml:lang attribute

```
<?xml version="1.0" encoding="UTF-8" ?>
<TranslatedMessages>
   <Message xml:lang="jp">  私は別の言語を話している。</Message>
   <Message xml:lang="ru">_ ______ __ ______ _____.</Message>
   <Message xml:lang="es">Estoy hablando otra lengua.</Message>
   <Message xml:lang="zh">  我講其它語言。</Message>
</TranslatedMessages>
```

You can see that the ISO 639 standard is simply a two-letter code that signifies the language to use. The problem with this standard is that it really only allows for 676 languages to be represented and a significant number more than that are used in the world. The ISO 3166 standard has a similar problem, but represents a country as well as a language (as presented in Listing 1-11).

Listing 1-11: Using ISO 3166 with the xml:lang attribute

```
<?xml version="1.0" encoding="UTF-8" ?>
<TranslatedMessages>
   <Message xml:lang="jp-JP">  私は別の言語を話している。</Message>
   <Message xml:lang="ru-RU">_ ______ __ ______ _____.</Message>
   <Message xml:lang="es-ES">Estoy hablando otra lengua.</Message>
   <Message xml:lang="zh-CN">  我講其它語言。</Message>
</TranslatedMessages>
```

The xml:space Attribute

When you really want to preserve your whitespace, one option is to also include the attribute `xml:space` within your XML elements where whitespace needs to be maintained. For instance, Listing 1-12 shows using the `xml:space` attribute for maintaining whitespace.

Listing 1-12: Using the xml:space attribute

```
<?xml version="1.0" encoding="UTF-8" ?>
<Movies>
   <Favorites>
      <Title xml:space="default">Happy  Gilmore</Title>
      <Title>Grease</Title>
      <Title xml:space="preserve">Lawrence
```

```
            of
            Arabia</Title>
      <Title xml:space="preserve">Star Wars -      The Empire Strikes Back</Title>
   </Favorites>
</Movies>
```

Two possible values exist for the `xml:space` attribute — `default` and `preserve`. A value of `default` means that the whitespace should not be preserved, and a value of `preserve` means that the whitespace should remain intact.

Notice that the XML parsers do not strip whitespace or act upon the whitespace in any manner because of these attribute settings. Instead, the `xml:space` attribute is simply an instruction to the consuming application that is can choose to act upon if desired.

XML Namespaces

Because developers create their own tag names in XML, you must use *namespaces* to define particular elements. If you are using two XML files within your application and the two files use some of the same tags, namespaces become very important. Even though the tags have the same name, they might have different meanings. For instance, compare the two XML files in Listings 1-13 and 1-14.

Listing 1-13: Book.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<Book>
   <Title>Professional ASP.NET 2.0</Title>
   <Price>49.99</Price>
   <Year>2005</Year>
</Book>
```

Now take a look at the second XML file. You should be able to see where the problem lies.

Listing 1-14: Author.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<Author>
   <Title>Mr.</Title>
   <FirstName>Bill</FirstName>
   <LastName>Evjen</LastName>
</Author>
```

A conflict exists with the `<Title>` tag. If you are using both these XML files, you might be able to tell the difference between the tags by just glancing at them; but computers are unable to decipher the difference between two tags with the same name.

The solution to this problem is to give the tag an identifier that enables the computer to tell the difference between the two tags. Do this is by using the XML namespace attribute, `xmlns`. Listing 1-15 shows how you would differentiate between these two XML files by using XML namespaces.

Listing 1-15: Revised Book.xml using an XML namespace

```
<?xml version="1.0" encoding="UTF-8" ?>
<Book xmlns="http://www.xmlws101.com/xmlns/book">
   <Title>Professional ASP.NET 2.0</Title>
   <Price>49.99</Price>
   <Year>2005</Year>
</Book>
```

Notice that you now have added the XML namespace attribute to your root element `<Book>`. Now look at the second file (Listing 1-16).

Listing 1-16: Revised Author.xml using an XML namespace

```
<?xml version="1.0" encoding="UTF-8" ?>
<Author xmlns="http://www.xmlws101.com/xmlns/author">
   <Title>Mr.</Title>
   <FirstName>Bill</FirstName>
   <LastName>Evjen</LastName>
</Author>
```

In this example, the `<Author>` element contains an XML namespace that uniquely identifies this XML tag and all the other tags that are contained within it. Note that you could have put an XML namespace directly in the `<Title>` tag if you wished. By putting the `xmlns` attribute in the root element, not only do you uniquely identify the root element, but you also identify all the child elements contained within the root element.

The value of the `xmlns` attribute is the Universal Resource Identifier (URI). It is not required that the URI be a Web site URL as shown in the example, but this is usually a good idea. The URI can be anything that you wish it to be. For example, it could just as easily be written as `xmlns="myData"` or `xmlns="12345"`. But with this kind of URI, you are not guaranteed any uniqueness because another URI in another file may use the same value. Therefore, it is common practice to use a URL, and this practice serves two purposes. First, it is guaranteed to be unique. A URL is unique, and using it as your URI ensures that your URI won't conflict with any other. The other advantage to using a URL as the URI is that it also identifies where the data originates.

You don't have to point to an actual file. In fact, it is usually better not to do that, but instead to use something like the following:

```
xmlns="http://www.xmlws101.com/[Namespace Name]"
```

If the XML file has an associated XSD file, another option is to point to this file. The XSD file defines the schema of the XML file.

The style of XML namespaces used thus far is referred to as a default namespace. The other type of XML namespaces that is available for your use is a qualified namespace. To understand why you need another type of XML namespace, take a look at the example in Listing 1-17.

Listing 1-17: Author.xml using multiple XML namespaces

```
<?xml version="1.0" encoding="UTF-8" ?>
<Author xmlns="http://www.firstserver.com/xmlns/author">
   <Title xmlns="http://www.secondserver.com/title">Mr.</Title>
   <FirstName xmlns="http://www.thirdserver.com/fn">Bill</FirstName>
   <LastName xmlns="http://www.thirdserver.com/ln">Evjen</LastName>
</Author>
```

As you can see in this example, you use a number of different XML namespaces to identify your tags. First, your `<Author>` tag is associated with the XML namespace from the first server. The `<Title>` tag is associated with the second server, and the `<FirstName>` and `<LastName>` tags are associated with the third server. XML allows you to associate your tags with more than one namespace throughout your document.

The problem is that you might have hundreds or thousands of nodes within your XML document. If one of the namespaces that is repeated throughout the document changes, you have a lot of changes to make throughout the document.

Using qualified namespaces enables you to construct the XML document so that if you need to make a change to a namespace that is used throughout the document, you only have to do it in one spot. The change is reflected throughout the document. Look at Listing 1-18 to see an XML document that uses qualified namespaces.

Listing 1-18: Author.xml using qualified XML namespaces

```
<?xml version="1.0" encoding="UTF-8" ?>
<AuthorNames:Author
 xmlns:AuthorNames="http://www.firstserver.com/xmlns/author"
 xmlns:AuthorDetails="http://www.secondserver.com/xmlns/details">
   <AuthorNames:Title>Mr.</AuthorNames:Title>
   <AuthorNames:FirstName>Bill</AuthorNames:FirstName>
   <AuthorNames:LastName>Evjen</AuthorNames:LastName>
   <AuthorDetails:Book>Professional ASP.NET 2.0</AuthorDetails:Book>
</AuthorNames:Author>
```

In this document, you use an explicit declaration of the namespace. Explicit declarations of namespace prefixes use attribute names beginning with `xmlns:` followed by the prefix. So in the first node, you have explicitly declared two namespaces that you use later in the document.

```
xmlns:AuthorNames="http://www.firstserver.com/xmlns/author"
xmlns:AuthorDetails="http://www.secondserver.com/xmlns/details"
```

Notice that the declaration name follows a colon after `xmlns`. In this example, you declare two qualified namespaces: `AuthorNames` and `AuthorDetails`. Later, when you want to associate an element with one of these explicit declarations, you can use the shorthand (or prefix) to substitute for the full namespace. In your document, you do this by using `<AuthorNames:LastName>` and `<AuthorDetails:Book>`.

Summary

This chapter is a simple review of XML. It is meant to serve as a primer for you to get up to speed in dealing with the technologies that use XML. These technologies are covered throughout the rest of this book.

This chapter looked at the basics of XML, its syntax, and the rules that go into building XML documents. In addition, this chapter reviewed how to build an XML document and to view it in the IE or Firefox.

Finally, this chapter explained how to use namespaces with XML documents. The next chapter covers building and working with XML documents using a number of different developer tools.