CHAPTER 1

Essential XML

Welcome to the world of Extensible Markup Language, XML. This book is your guided tour to that world, so have no worries—you've come to the right place. That world is large and expanding in unpredictable ways every minute, but we're going to become familiar with the lay of the land in detail here. And there's a lot of territory to cover because XML is getting into the most amazing places, and in the most amazing ways, these days.

XML is a language defined by the World Wide Web Consortium (W3C, `www.w3c.org`), the body that sets the standards for the Web, and this first chapter is all about getting a solid overview of that language and how you can use it. For example, you probably already know that you can use XML to create your own elements, thus creating a customized markup language for your own use. In this way, XML supercedes other markup languages such as Hypertext Markup Language (HTML); in HTML, all the elements you use are predefined—and there are not enough of them. In fact, XML is a metamarkup language because it lets you create your own markup languages.

Markup Languages

Markup languages are all about describing the form of the document—that is, the way the content of the document should be interpreted. The markup language that most people are familiar with today is, of course, HTML, which you use to create standard Web pages. Here's an example HTML page:

Listing ch01_01.html

```
<HTML>
    <HEAD>
        <TITLE>Hello From HTML</TITLE>
    </HEAD>
    <BODY>
        <CENTER>
            <H1>
               Hello From HTML
            </H1>
        </CENTER>
        Welcome to the wild and woolly world of HTML.
    </BODY>
</HTML>
```

You can see the results of this HTML in Figure 1-1 in Netscape Navigator. Note that the HTML markup in this page—that is, *tags* such as `<HEAD>`, `<CENTER>`, `<H1>`, and so on—is there to give directions to the browser. That's what markup does; it specifies directions on the way the content is to be interpreted.

Figure 1-1 An HTML page in a browser.

When you think of markup in terms of specifying how the content of a document is to be handled, it's easy to see that there are many kinds of markup languages all around already. For example, if you use a word processor to save a document in Rich Text Format (RTF), you'll find all kinds of markup codes embedded in the document. Here's an example; in this case, I've just created an RTF file with the letters `abc` underlined and in bold using Microsoft Word—try searching for the actual text (hint: it's near the very end):

```
{\rtf1\ansi\ansicpg1252\uc1 \deff0\deflang1033
\deflangfe1033{\fonttbl{\f0\froman\fcharset0\fprq2{\*\panose
020206030504050203040}Times New Roman;}}{\colortbl;\red0
\green0\blue0;\red0\green0\blue255;\red0\green255\blue255;
\red0\green255\blue0;\red255\green0\blue255;\red255\green0
\blue0;\red255\green255\blue0;\red255\green255\blue255;\red0
\green0\blue128;\red0\green128\blue128;\red0\green128\blue0;
\red128\green0\blue128;\red128\green0\blue0;\red128\green128
\blue0;\red128\green128\blue128;\red192\green192\blue192;}
{\stylesheet{\widctlpar\adjustright \fs20\cgrid \snext0 Normal;}
{\*\cs10 \additive Default Paragraph Font;}}{\info{\title  }
{\author Steven Holzner}{\operator Steven Holzner}{\creatim
\yr2000\mo\dy\hr\min}{\revtim\yr2000\mo4\dy17\hr13\min55}
{\version1}{\edmins1}{\nofpages1}{\nofwords0}{\nofchars1}
{\*\company SteveCo}{\nofcharsws1}{\vern89}}\widowctrl\ftnbj
\aenddoc\formshade\viewkind4\viewscale100\pgbrdrhead\pgbrdrfoot
\fet0\sectd \psz1\linex0\endnhere\sectdefaultcl {\*\pnseclvl1
\pnucrm\pnstart1\pnindent720\pnhang{\pntxta .}}{\*\pnseclvl2
\pnucltr\pnstart1\pnindent720\pnhang{\pntxta .}}{\*\pnseclvl3
\pndec\pnstart1\pnindent720\pnhang{\pntxta .}}{\*\pnseclvl4
\pnlcltr\pnstart1\pnindent720\pnhang{\pntxta )}}{\*\pnseclvl5
\pndec\pnstart1\pnindent720\pnhang{\pntxtb (}{\pntxta )}}
{\*\pnseclvl6\pnlcltr\pnstart1\pnindent720\pnhang{\pntxtb (}
{\pntxta )}}{\*\pnseclvl7\pnlcrm\pnstart1\pnindent720\pnhang
{\pntxtb (}{\pntxta )}}{\*\pnseclvl8\pnlcltr\pnstart1
\pnindent720\pnhang{\pntxtb (}{\pntxta )}}{\*\pnseclvl9\pnlcrm
\pnstart1\pnindent720\pnhang{\pntxtb (}{\pntxta )}}\pard\plain
\sl480\slmult1\widctlpar\adjustright \fs20\cgrid {\b\fs24\ul abc }{\b\ul \par }}
```

The markup language that most people are familiar with these days is HTML, but it's easy to see how that language doesn't provide enough power for anything beyond creating standard Web pages.

HTML 1.0 consisted of only a dozen or so tags, but the most recent version, HTML 4.01, consists of almost 100—and if you include the other tags added by the major browsers, that number is closer to 120. But as handling data on the Web and other nets intensifies, it's clear that 120 tags isn't enough—and, in fact, you can never have enough.

For example, what if your hobby was building model ships and you wanted to exchange specifications with others on the topic? HTML doesn't include tags such as <BEAMWIDTH>, <MIZZENHEIGHT>, <DRAFT>, <SHIPCLASS>, and the others you might want. What if you were a major bank that wanted to exchange financial data with other institutions—would you prefer tags such as <B>, <UL>, and <FONT>, or tags such as <FISCALYEAR>, <ACCOUNTNUMBER>, <TRANSFERACCOUNT>, and others? (In fact, such markup languages as Extensible Business Reporting Language exist now—and they're built on XML.)

What if you were a Web browser manufacturer and wanted to create your own markup language to let people configure your browser, adding scrollbars, toolbars, and other elements? You might create your own markup language to do that; in fact, Netscape has done just that with the XML-based User Interface Language, which we'll see in this chapter.

The upshot is that there are as many reasons to create markup languages as there are ways of handling data—and, of course, that's unlimited. That's where XML comes in: It's a metamarkup specification that lets you create your own markup languages.

What Does XML Look Like?

So what does XML look like and how does it work? Here's an example that mimics the HTML page just introduced:

Listing ch01_02.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<DOCUMENT>
     <GREETING>
          Hello From XML
     </GREETING>
     <MESSAGE>
          Welcome to the wild and woolly world of XML.
     </MESSAGE>
</DOCUMENT>
```

We'll see the parts of an XML document in detail in the next chapter, but in overview, here's how this one works: I start with the XML *processing instruction* <?xml version="1.0" encoding="UTF-8"?> (all XML processing instructions start with <? and end with ?>), which indicates that I'm using XML version 1.0, the only version currently defined, and using the UTF-8 *character encoding*, which means that I'm using an 8-bit condensed version of Unicode (more on this later in the chapter):

```
<?xml version="1.0" encoding="UTF-8"?>
<DOCUMENT>
    <GREETING>
        Hello From XML
    </GREETING>
    <MESSAGE>
        Welcome to the wild and woolly world of XML.
    </MESSAGE>
</DOCUMENT>
```

Next, I create a new *tag* named `<DOCUMENT>`. As we'll see in the next chapter, you can use any name, not just `DOCUMENT`, for a tag, as long as the name starts with a letter or underscore (`_`) and the following characters consist of letters, digits, underscores, dots (`.`), or hyphens (`-`), but no spaces. In XML, tags always start with `<` and end with `>`.

XML documents are made up of XML *elements*, and (much like HTML) you create XML elements with an opening tag (such as `<DOCUMENT>`), followed by any element content (if any) (such as text or other elements) and ending with the matching closing tag that starts with `</` (such as `</DOCUMENT>`). (There are additional rules we'll see in the next chapter if the element doesn't contain any content.) It's necessary to enclose the entire document, except for processing instructions, in one element, called the *root element*, and that's the `<DOCUMENT>` element here:

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

Now I'll add a new element that I made up, `<GREETING>`, which encloses text content (in this case, that's `Hello From XML`), to this XML document, like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<DOCUMENT>
    <GREETING>
        Hello From XML
    </GREETING>
    .
    .
    .
</DOCUMENT>
```

Next, I can add a new element as well, <MESSAGE>, which also encloses text content:

```
<?xml version="1.0" encoding="UTF-8"?>
<DOCUMENT>
    <GREETING>
        Hello From XML
    </GREETING>
    <MESSAGE>
        Welcome to the wild and woolly world of XML.
    </MESSAGE>
</DOCUMENT>
```

Now the <DOCUMENT> root element contains two elements—<GREETING> and <MESSAGE>. And each of the <GREETING> and <MESSAGE> elements holds text. In this way, I've created a new XML document.

Note the similarity of this document to the HTML page we saw earlier. Note also, however, that in the HTML document, all the tags were predefined and a Web browser knows how to handle them. Here we've just created these tags, <DOCUMENT>, <GREETING>, and <MESSAGE>, from thin air—how can we use an XML document like this one? What would a browser make of these new tags?

What Does XML Look Like in a Browser?

It turns out that a browser such as Microsoft Internet Explorer version 5 or later lets you display raw XML documents directly. For example, if I saved the XML document just created in a document named ch01_02.xml and opened that document in Internet Explorer, I'd see something like Figure 1-2.

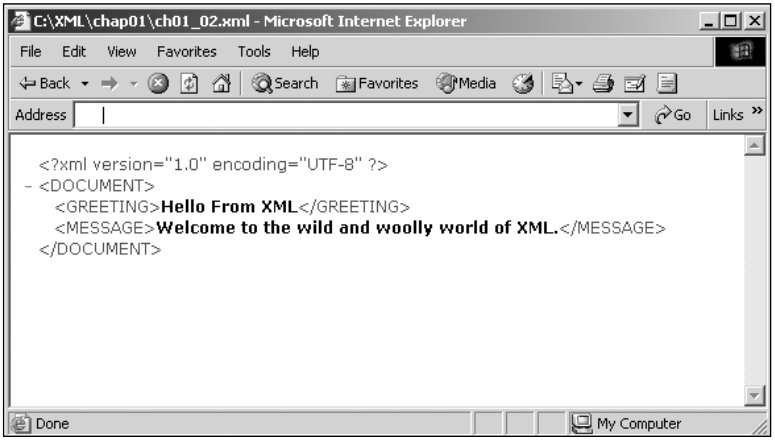

Figure 1-2 An XML document in Internet Explorer.

You can see the complete XML document in Figure 1-2, but it's nothing like the image you see in Figure 1-1; there's no particular formatting at all. So now that we've created our own markup elements, how do you tell a browser how to display them?

In fact, many people who are new to XML find the claim that you can use XML to create new markup languages very frustrating—after all, what then? It turns out that it's up to you to assign meaning to the new elements you create, and you can do that in two main ways. First, you can use a *stylesheet* to indicate to a browser how you want the content of the elements you've created formatted. The second way is to use a programming language, such as Java or JavaScript, to handle the XML document in programming code. We'll see both ways throughout this book, and I'll take a look at them in overview in this chapter as well. I'll start by adding a stylesheet to the XML document we've already created.

There are two main ways of specifying styles when you format XML documents: Cascading Style Sheets (CSS) and the Extensible Stylesheet Language (XSL). We'll see both in this book; here, I'll apply a CSS stylesheet using the XML processing instruction `<?xml-stylesheet type="text/css" href="ch01_04.css"?>`, which tells the browser that the type of the stylesheet I'll be using is CSS and that its name is ch01_04.css (and because I'm not giving any path to the stylesheet, the browser will assume that it's in the same directory as the XML document itself):

Listing ch01_03.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/css" href="ch01_04.css"?>
<DOCUMENT>
     <GREETING>
          Hello From XML
     </GREETING>
     <MESSAGE>
          Welcome to the wild and woolly world of XML.
     </MESSAGE>
</DOCUMENT>
```

Here's what ch01_04.css itself looks like. In this case, I'm customizing the <GREETING> element to display its content in red, centered in the browser and in 36-point font, and the <MESSAGE> element to display its text in black 18-point font. The display: block part indicates that I want the content of these elements to be displayed in a block, which translates here to being displayed on its own line:

Listing ch01_04.css

```
GREETING {display: block; font-size: 36pt; color: #FF0000; text-align:
➥center}
MESSAGE (display: block; font-size: 18pt; color: #000000}
```

You can see the results in two browsers that support XML in Figures 1-3 and 1-4. Figure 1-3 shows this XML document in Netscape 6, and Figure 1-4 shows the same document in Internet Explorer. As you can see, we've formatted the XML document as we like it using stylesheets—in fact, this result is already an advance over HTML because we've been able to format exactly how we want to display the text, not just rely on predefined elements such as <H1>.

Figure 1-3 An XML document in Netscape Navigator.

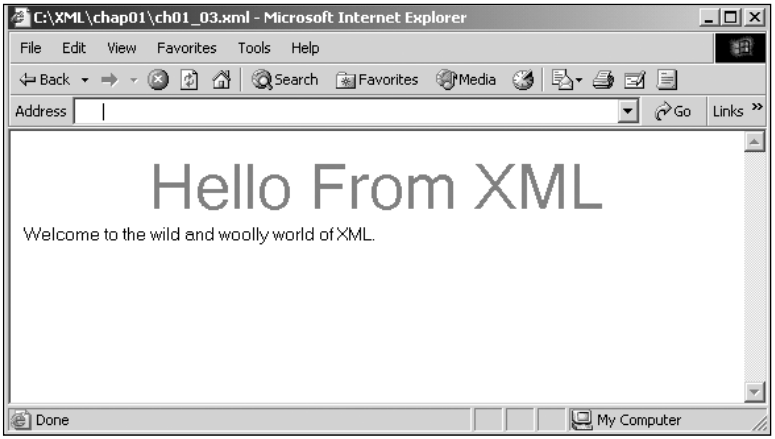

Figure 1-4 An XML document in Internet Explorer.

That gives us a taste of XML. Now we've seen how to create a first XML document and how to use a stylesheet to display it in a browser. We've seen what it looks like—so what's so great about XML? Take a look at the overview, coming up next.

What's So Great About XML?

XML is so popular for many reasons. I'll examine some of them here as part of our overview of where XML is today. My own personal favorite is that XML allows easy data handling and exchange, and I'm going to start with that.

Easy Data Exchange

I've been involved with computing for a long time, and one of the things I've watched with misgiving is the growth of proprietary data formats. In earlier days, programs could exchange data easily because data was stored as text. Today, however, you need conversion programs or modules to let applications transfer data between themselves. In fact, proprietary data formats have become so complex that frequently one version of a complex application can't even read data from an earlier version of the same application.

In XML, data and markup is stored as text that you yourself can configure. If you like, you can use XML editors, as we'll see, to create XML documents. If something goes wrong, however, you can examine or modify the document directly because it's all just text. The data is also not encoded in some way that has been patented or copyrighted, which some formats are, so it's more accessible.

You might think that binary formats would be more efficient because they can store data more compactly, but that's not the way things have worked out. Microsoft Corporation, for example, is notorious for turning out huge applications that store even simple data in huge files (the not-so-affectionate name for this is "bloatware"). If you store only the letters `abc` in a Microsoft Word 2000 document, you might be surprised to find that the document is something like 20,000 bytes. A similar XML file might be 30 or 40 bytes. Even large amounts of data are not necessarily stored efficiently; Microsoft Excel, for example, routinely creates large files that are five times as long as the corresponding text, and Microsoft Access XP creates files that start at 96KB.

In addition, when you standardize markup languages, many different people can use them; I'll take a look at that next.

Customizing Markup Languages

As we've already seen, you can create customized markup languages using XML, and that represents its extraordinary power. When you and a number of other people agree on a markup language, you can create customized browsers or applications that handle that language. Hundreds of such languages are already being standardized now, including these:

- Banking Industry Technology Secretariat (BITS)
- Financial Exchange (IFX)
- Bank Internet Payment System (BIPS)
- Telecommunications Interchange Markup (TIM)
- Schools Interoperability Framework (SIF)
- Common Business Library (xCBL)
- Electronic Business XML Initiative (ebXML)
- Product Data Markup Language (PDML)
- Financial Information eXchange protocol (FIX)
- The Text Encoding Initiative (TEI)

Some customized markup languages, such as Chemical Markup Language (CML), let you represent complex molecules graphically, as we'll see later in this chapter. And you can imagine how useful a language would be that creates graphical building plans for architects when you open a document in a browser.

Not only can you create custom markup languages, but you can extend them using XML as well. So, if someone creates a markup language based on XML, you can add the extensions you want easily. In fact, that's what's happening now with Extensible Hypertext Markup Language (XHTML), which I'll take a look at briefly in this chapter and in detail later in the book. Using XHTML, you can add your own elements to what a browser displays as normal HTML.

Self-Describing Data

The data in XML documents is self-describing. Take a look at this document:

Listing ch01_05.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<DOCUMENT>
    <GREETING>
        Hello From XML
    </GREETING>
    <MESSAGE>
        Welcome to the wild and woolly world of XML.
    </MESSAGE>
</DOCUMENT>
```

Based solely on the names we've given to each XML element here, you can figure out what's going on. This document has a greeting and a message to impart. Even if you came back to this document years later, you could figure out what's going on. This means that XML documents are, to a large extent, self-documenting. (We'll also see in the next chapter that you can add explicit comments to XML files.)

Structured and Integrated Data

Another powerful aspect of XML is that it lets you specify not only data, but also the structure of that data and how various elements are integrated into other elements. This is important when you're dealing with complex and important data. For example, you could represent a long bank statement in HTML, but in XML, you can actually build in the semantic rules that specify the structure of the document so that the document can be checked to make sure it's set up correctly.

Take a look at this XML document:

Listing ch01_06.xml

```
<?xml version="1.0"?>
<SCHOOL>
   <CLASS type="seminar">
        <CLASS_TITLE>XML In The Real World</CLASS_TITLE>
        <CLASS_NUMBER>6.031</CLASS_NUMBER>
        <SUBJECT>XML</SUBJECT>
        <START_DATE>6/1/2002</START_DATE>
        <STUDENTS>
             <STUDENT status="attending">
                  <FIRST_NAME>Edward</FIRST_NAME>
                  <LAST_NAME>Samson</LAST_NAME>
             </STUDENT>
             <STUDENT status="withdrawn">
                  <FIRST_NAME>Ernestine</FIRST_NAME>
                  <LAST_NAME>Johnson</LAST_NAME>
             </STUDENT>
        </STUDENTS>
   </CLASS>
</SCHOOL>
```

Here I've set up an XML seminar and added two students to it. As we'll see in Chapter 2, "Creating Well-Formed XML Documents" and Chapter 3, "Valid Documents: Creating Document Type Definitions," using XML, you can specify, for example, that each `<STUDENT>` element needs to enclose a `<FIRST_NAME>` and a `<LAST_NAME>` element, that the `<START_DATE>` element can't go in the `<STUDENTS>` element, and more.

In fact, this emphasis on the correctness of documents is strong in XML. In HTML, a Web author could (and frequently did) write sloppy HTML, knowing that the Web browser would take care of any syntax problems (some Web authors even exploited this intentionally to create special effects in some browsers). In fact, some people estimate that 50% or more of the code in modern browsers is there to take care of sloppy HTML in Web pages. For that kind of reason, the story is different in XML. In XML, browsers are supposed to check your document; if there's a problem, they are not supposed to proceed any further. They should let you know about the problem, but that's as far as they're supposed to go.

So how does an XML browser check your document? There are two main checks that XML browsers can make: checking that your document is *well-formed* and checking that it's *valid*. We'll see what these terms mean in more detail in the next chapter, and I'll look at them in overview here.

Well-Formed XML Documents

What does it mean for an XML document to be well formed? To be well formed, an XML document must follow the syntax rules set up for XML by the World Wide Web Consortium in the XML 1.0 specification (which you can find at `www.w3.org/TR/REC-xml`, and which we'll discuss in more detail in the next chapter). Informally, well-formedness often means that the document must contain one or more elements, and one element, the root element, must contain all the other elements. Also, each element must nest inside any enclosing elements properly. For example, this document is not well formed because the `</GREETING>` closing tag comes after the opening `<MESSAGE>` tag for the next element:

```
<?xml version="1.0" encoding="UTF-8"?>
<DOCUMENT>
    <GREETING>
        Hello From XML
    <MESSAGE>
    </GREETING>
        Welcome to the wild and woolly world of XML.
    </MESSAGE>
</DOCUMENT>
```

Valid XML Documents

Most XML browsers will check your document to see if it is well formed. Some of them can also check whether it's valid. An XML document is valid if there is a document type definition (DTD) or XML schema associated with it, and if the document complies with that DTD or schema.

A document's DTD or schema specifies the correct syntax of the document, as we'll see in Chapter 3, "Valid Documents: Creating Document Type Definitions," for DTDs and Chapter 5, "Creating XML Schemas," for schemas. For example, DTDs can be stored in a separate file, or they can be stored in the document itself using a `<!DOCTYPE>` element. Here's an example in which I add a `<!DOCTYPE>` element to the greeting XML document we developed earlier:

Listing ch01_07.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/css" href="first.css"?>
<!DOCTYPE DOCUMENT [
    <!ELEMENT DOCUMENT (GREETING, MESSAGE)>
```

continues

Listing ch01_07.xml Continued

```
    <!ELEMENT GREETING (#PCDATA)>
    <!ELEMENT MESSAGE (#PCDATA)>
]>
<DOCUMENT>
    <GREETING>
        Hello From XML
    </GREETING>
    <MESSAGE>
        Welcome to the wild and woolly world of XML.
    </MESSAGE>
</DOCUMENT>
```

We'll see more about DTDs in Chapter 3, but this DTD indicates that you can have <GREETING> and <MESSAGE> elements inside a <DOCUMENT> element, that the <DOCUMENT> element is the root element, and that the <GREETING> and <MESSAGE> elements can hold text.

Most XML browsers will check XML documents for well-formedness, but only a few will check for validity. I'll talk more about where to find XML validators later in this chapter.

We've gotten an overview of XML documents now, including how to display them using stylesheets and what a well-formed and valid document is. However, this is only part of the story: Many XML documents are not designed to be displayed in browsers at all—or, even if they are, they are not designed to be used with modern stylesheets (such as browsers that convert XML into industry-specific graphics such as molecular structure, physics equations, or even musical scales). The more powerful use of XML involves *parsing* an XML document to break it down into its component parts and then handling the resulting data yourself. I'll take a look at a few ways of parsing XML data that are available to us next.

Parsing XML Yourself

Say that you have this XML document, ch01_02.xml, which we developed earlier in this chapter:

```
<?xml version="1.0" encoding="UTF-8"?>
<DOCUMENT>
    <GREETING>
        Hello From XML
    </GREETING>
    <MESSAGE>
        Welcome to the wild and woolly world of XML.
    </MESSAGE>
</DOCUMENT>
```

Now say that you wanted to extract the greeting `Hello From XML` from this XML document. One way of doing that is by using XML data islands in Internet Explorer and then using a scripting language, such as JavaScript, to extract and display the text content of the `<GREETING>` element. Here's how that looks in a Web page:

Listing ch01_08.html

```
<HTML>
     <HEAD>
          <TITLE>
               Finding Element Values in an XML Document
          </TITLE>

          <XML ID="firstXML" SRC="ch01_02.xml"></XML>

          <SCRIPT LANGUAGE="JavaScript">
               function getData()
               {
                    xmldoc= document.all("firstXML").XMLDocument;

                    nodeDoc = xmldoc.documentElement;
                    nodeGreeting = nodeDoc.firstChild;

                    outputMessage = "Greeting: " +
                          nodeGreeting.firstChild.nodeValue;
                    message.innerHTML=outputMessage;
               }
          </SCRIPT>
     </HEAD>

     <BODY>
          <CENTER>
               <H1>
                    Finding Element Values in an XML Document
               </H1>

               <DIV ID="message"></DIV>
               <P>
               <INPUT TYPE="BUTTON" VALUE="Get The Greeting"
                    ONCLICK="getData()">
          </CENTER>
     </BODY>
</HTML>
```

This Web page displays a button with the caption `Get The Greeting`, as you see in Figure 1-5. When you click the button, the JavaScript code in this page reads in ch01_02.xml, extracts the text from the `<GREETING>` element, and displays that text, as you can also see in Figure 1-5. In this way, you can see how you can create applications that handle XML documents in a customized way—even customized XML browsers.

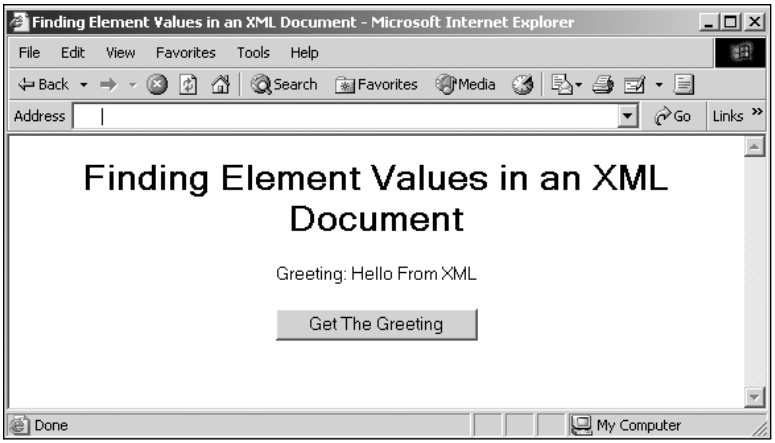

Figure 1-5 Extracting data from an XML document in Internet Explorer.

We'll see more about using JavaScript to work with XML later in this book. I'll also cover how JavaScript itself works first, so that if you haven't used it before, there won't be a problem.

Although JavaScript is useful for lightweight XML uses, most XML programming is done in Java today, and we'll take advantage of that in this book as well. Here's an example Java program, ch01_09.java. This program also reads ch01_02.xml and extracts the text content of the `<GREETING>` element (we'll see how to run this kind of program in Chapter 10, "Understanding Java"):

Listing ch01_09.java

```
import javax.xml.parsers.*;
import org.w3c.dom.*;
import java.io.*;

public class ch01_09
{
    static public void main(String[] argv)
    {
```

```
        try {

        DocumentBuilderFactory dbf =
            DocumentBuilderFactory.newInstance();

        DocumentBuilder db = null;
        try {
            db = dbf.newDocumentBuilder();
        }
        catch (ParserConfigurationException pce) {}

        Document doc = null;
            doc = db.parse("ch01_02.xml");

            for (Node node = doc.getDocumentElement().getFirstChild();
                node != null; node = node.getNextSibling()) {

                if (node instanceof Element) {
                    if (node.getNodeName().equals("GREETING")) {

                        StringBuffer buffer = new StringBuffer();

                        for (Node subnode = node.getFirstChild();
                            subnode != null; subnode =
subnode.getNextSibling()){
                            if (subnode instanceof Text) {
                                buffer.append(subnode.getNodeValue());
                            }
                        }
                        System.out.println(buffer.toString());
                    }
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

When you run this program, the output looks like this (I'll use `%` to stand for the command-line prompt in this book. If you're using UNIX, this prompt might look familiar, or your prompt might look something like `\home\steve:`. If you're using Windows, you get a command-line prompt by opening an MS-DOS window, and your prompt might look something like `C:\XML>`):

```
%java ch01_09
        Hello From XML
```

(Note that this program returns all the text in the <GREETING> element, including the leading spaces.) We'll see how to use Java to parse XML documents later in this book, in Chapter 11, "Java and the XML DOM," and Chapter 12, "Java and SAX." I'll also cover the Java you need to know before getting into the programming; if you haven't programmed in Java before, that won't be a problem.

We've gotten a good overview of how XML works. Now it's time to take a look at how it's already working in the real world, starting with an overview of the XML resources available to you.

XML Resources

A great many XML resources are available to you online. Because it's very important to know about them to get a solid background in XML, I'm going to list them here.

The XML specification is defined by the World Wide Web Consortium (W3C), and that's where you should start looking for XML resources. Here's a good starter list (we'll see all these topics in this book):

- www.w3c.org/xml—The World Wide Web Consortium's main XML site, the starting point for all things XML.
- www.w3.org/XML/1999/XML-in-10-points—"XML in 10 Points" (actually only seven), an XML overview.
- www.w3.org/TR/REC-xml—The official W3C recommendation for XML 1.0, the current (and only) version. It won't be easy to read, however—that's what this book is all about, translating that kind of document to English.
- www.w3.org/TR/xml-stylesheet/—All about using stylesheets and XML.
- www.w3.org/TR/REC-xml-names/—All about XML namespaces.
- www.w3.org/Style/XSL/—All about Extensible Stylesheet Language, XSL.
- www.w3.org/TR/xslt—All about XSL transformations, XSLT.
- www.w3.org/XML/Activity.html—An overview of current XML activity at W3C.
- www.w3.org/TR/xmlschema-0/, www.w3.org/TR/xmlschema-1/, and www.w3.org/TR/xmlschema-2/—XML Schemas, the alternative to DTDs.

- www.w3.org/TR/xlink/—The XLinks specification.
- www.w3.org/TR/xptr—The XPointers specification.
- www.w3.org/TR/xhtml1/—The XHTML 1.0 specification.
- www.w3.org/TR/xhtml11/—The XHTML 1.1 specification.
- www.w3.org/DOM/—The W3C Document Object Model, DOM.

Many non-W3C XML resources are out there, too (a casual search for the word *XML* on the Web turns up more than 13 million matches). Here's a list to get started with:

- www.xml.com XML.com—This site is filled with XML resources, discussions, and notifications of public events.
- www.oasis-open.org—OASIS, the Organization for the Advancement of Structured Information Standards, is dedicated to the adoption of product-independent formats such as XML.
- XML.org www.xml.org—XML.ORG is designed to provide information about the use of XML in industrial and commercial settings. It's hosted by OASIS and is a reference for XML vocabularies, DTDs, schemas, and namespaces.
- http://msdn.microsoft.com/xml/default.asp—Microsoft's XML page.

There are also quite a few XML tutorials out there online (searching for "XML Tutorial" brings up more than 2,300 matches). Here are a few to start with:

- www-105.ibm.com/developerworks/education.nsf/xml-onlinecourse-bytitle/8C8A8628B3DD7EDB852567BD000A8A64?OpenDocument—IBM's free tutorials.
- www.ucc.ie/xml/—A comprehensive Frequently Asked Questions (FAQ) list about XML, kept up by some of the contributors to the W3C's XML Working Group. Considered by many the definitive FAQ on XML.
- http://msdn.microsoft.com/xml/tutorial/default.asp—Microsoft's XML tutorial.
- www.xml.com/pub/98/10/guide0.html—XML.com's XML overview.

In addition, you might find some newsgroups on Usenet useful (note that your news server might not carry all these groups):

- `comp.text.xml`—A good general-purpose, free-floating XML forum.
- `microsoft.public.dotnet.xml`—XML discussions and questions concerning using XML with Microsoft's .NET initiative.
- `microsoft.public.xml`—The general Microsoft XML forum.

That's a good start on XML resources available on the Internet (note that you can use a search engine such as `http://groups.google.com` to search these groups for XML material). What about XML software? I'll take a look at what's out there, starting with XML editors.

XML Editors

To create the XML documents we'll use in this book, all you need is a text editor of some kind, such as vi, emacs, pico, Macintosh's BBEdit or SimpleText, or Windows Notepad or Windows WordPad. By default, XML documents are supposed to be written in Unicode, although in practice you can write them in ASCII—and nearly all of them are written that way so far. Just make sure that when you write an XML document, you save it in your editor's plain text format.

Using Windows Text Editors

Windows text editors such as WordPad have an annoying habit of appending the extension .txt to a filename if they don't understand the extension you've given the file. That's not a problem with .xml files because WordPad understands the extension .xml. However, if you try to save, for example, an XML-based User Interface Language document with the correct extension, .xul, WordPad will give it the extension .xul.txt. To avoid that, place the name of the file in quotation marks when you save it, as in "scrollbars.xul".

However, it can be a lot easier to use an actual XML editor, which is designed explicitly for the job of handling XML. Here's a list of some programs that let you edit your XML:

- **Adobe FrameMaker,** `www.adobe.com`—Adobe includes great but expensive XML support in FrameMaker.
- **XML Pro,** `www.vervet.com/`—Costly but powerful XML editor.
- **XML Writer,** `http://xmlwriter.net/`—Color syntax highlighting, nice interface.
- **XML Notepad,** `msdn.microsoft.com/xml/notepad/intro.asp`—Microsoft's free XML editor, but a little obscure to use.
- **eNotepad,** `www.edisys.com/Products/eNotepad/enotepad.asp`—A WordPad replacement that does well with XML and has a good user interface.
- **XML Spy,** `www.xmlspy.com/`—A good user interface, and easy to use.

You can see XML Spy at work in Figure 1-6, XML Writer in Figure 1-7, XML Notepad in Figure 1-8, and eNotepad in Figure 1-9.

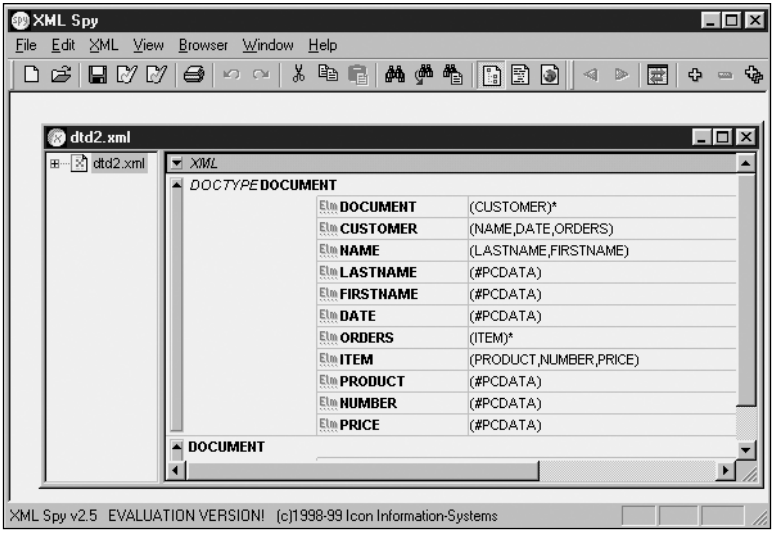

Figure 1-6 XML Spy editing XML.

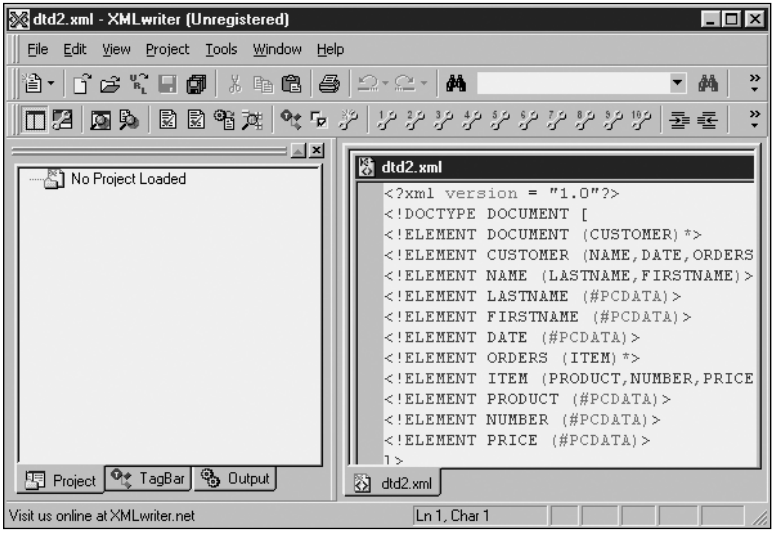

Figure 1-7
XML Writer editing XML.

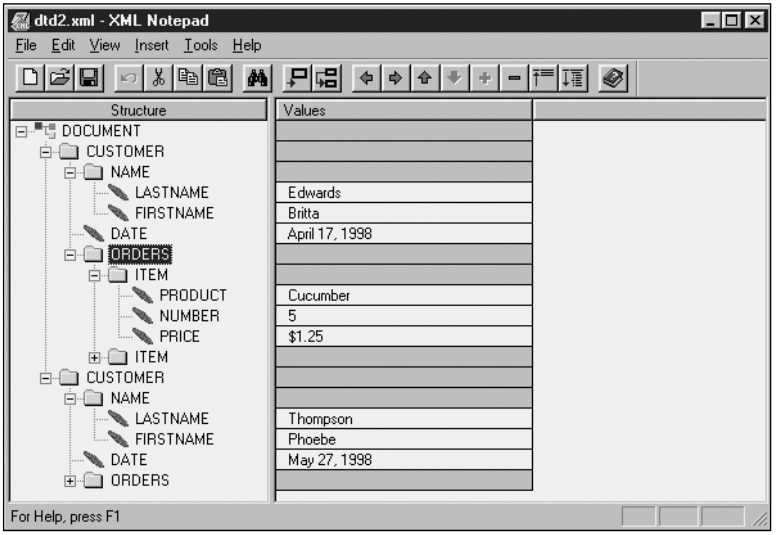

Figure 1-8
XML Notepad editing XML.

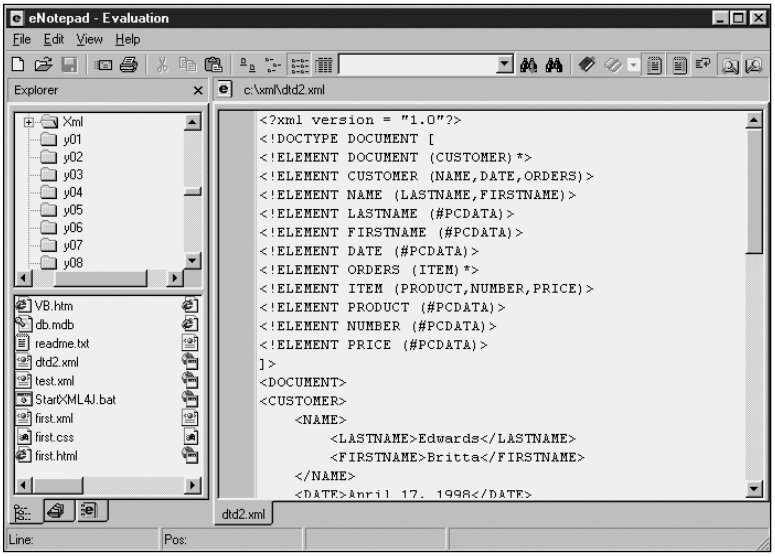

Figure 1-9 eNotepad editing XML.

Now that we've gotten an overview of creating XML documents, what about XML browsers? The list is more limited, but there are a few out there. See the next topic.

XML Browsers

Creating a true XML browser is not easy because, besides XML, you have to support a style language such as CSS or XSL. You should also support a scripting language such as JavaScript. These are heavy requirements for most third-party vendors, so the true XML browsers out there are few. But there are a few.

Internet Explorer 6

Whether you love or hate Microsoft, the fact is that Internet Explorer is currently the most powerful XML browser available; you can get it at `www.microsoft.com/windows/ie/downloads/ie6/default.asp` (although, like many other URLs on the Microsoft site, that URL might change—if so, go to `www.microsoft.com/windows/ie/`).

Internet Explorer 6.0 can display XML documents directly, as you saw in Figure 1-2, and can also handle them in scripting languages (JScript, Microsoft's version of JavaScript, and Microsoft's VBScript are supported). There is good stylesheet support, and other useful features that we'll see in

this book (including the <XML> element, which lets you create XML data islands that you can load XML documents into), and ways of binding XML to ActiveX Data Object (ADO) database recordsets. Internet Explorer 6 is the browser we use most in this book.

There's no question that Microsoft's XML commitment is strong—XML has been integrated even into the Office 2000 suite of applications—but Microsoft sometimes swerves significantly from the W3C standards (that is, when they're not the ones writing those standards themselves).

Netscape Navigator 6

You can get Netscape Navigator 6 or later at http://wp.netscape.com/download, which has some XML support. You can see Netscape Navigator 6 at work in Figure 1-3 shown earlier in this chapter.

As with Internet Explorer, support for stylesheets is good in Netscape Navigator. Netscape Navigator 6 also supports the XML-based User Interface Language (XUL), which lets you configure the browser. Netscape Navigator 7 is available in prerelease form as of this writing, but it's not yet stable enough for standard use.

Jumbo

There are relatively few other true XML browsers out there. One of the more famous ones is Jumbo, an XML browser designed to work with XML and the Chemical Markup Language (CML). You can pick up Jumbo for free at www.xml-cml.org/. This browser cannot only display XML (although not with stylesheets), but it also can use CML to draw molecules, as you see in Figure 1-10.

Although there are relatively few real XML browsers out there, there *are* a large number of XML *parsers*. You can use these parsers to read XML documents and break them up into their component parts.

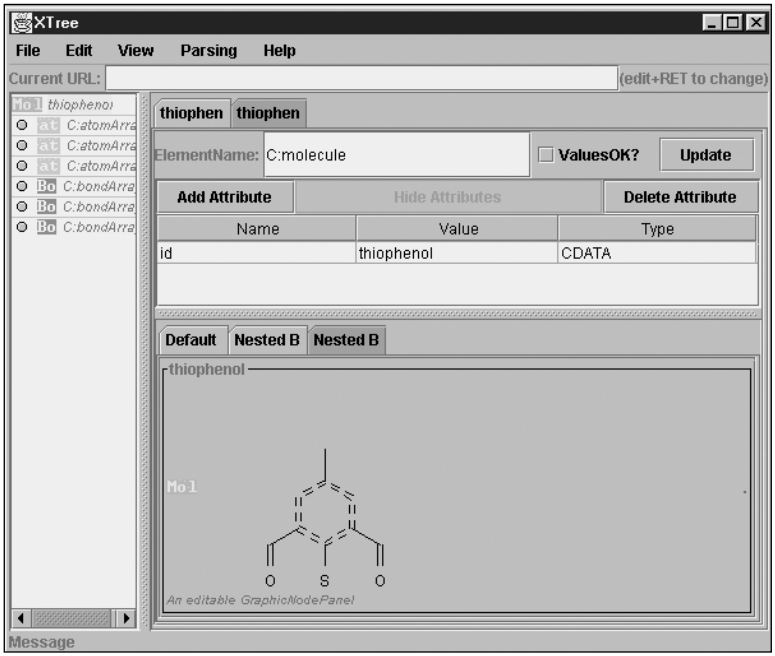

Figure 1-10 Jumbo at work.

XML Parsers

XML parsers are software packages that you use as part of an application such as Oracle 8i, which includes good XML support, or as part of your own programs. You use an XML parser to dissect XML documents and gain access to the data in them. Starting in Chapter 10, you'll see how to put Sun Microsystems's Java language to work with XML. The current version of Java, version 1.4, contains a great deal of built-in support for working with XML. Here's a list of some of the other XML parsers out there:

- **SAX: The Simple API for XML**—SAX is a well-known parser written by David Megginson, et al. (`www.megginson.com/SAX/index.html`), that uses event-based parsing I'll use SAX in this book.
- **expat**—This is a famous XML parser written in the C programming language by James Clark (`www.jclark.com/xml/expat.html`). This is the parser that's used in Netscape Navigator 6 and in the Perl language's XML::Parser module.

- **expat as a Perl Module (XML::Parser)**—This parser is maintained by Clark Cooper (`ftp://ftp.perl.org/pub/CPAN/modules/by-module/XML/`).
- **TclExpat**—This is a parser written for use in the Tcl programming language (`http://tclxml.sourceforge.net/`)
- **LT XML**—This is an XML developers' toolkit from the Language Technology Group at the University of Edinburgh (`www.ltg.ed.ac.uk/software/xml/`).
- **XML for Java (XML4J)**—From IBM AlphaWorks (`www.alphaworks.ibm.com/tech/xml4j`), this is a famous and very widely used XML parser that adheres well to the W3C standards.
- **XML Microsoft's Validating XML Processor**—This parser and various tools, samples, tutorials, and online documentation can be found at `msdn.microsoft.com/xml/default.asp`.
- **XP**—This is a nonvalidating XML processor written in Java by James Clark (`www.jclark.com/xml/xp/index.html`).
- **Python and XML Processing Preliminary XML Parser**—This parser offers XML support to the Python programming language (`www.python.org/topics/xml/`).
- **XML Testbed**—This one was written by Steve Withall (`www.w3.org/XML/1998/08withall/`).
- **SXP Silfide XML Parser (SXP)**—This is another famous XML parser and, in fact, a complete XML application programming interface (API) for Java (`www.loria.fr/projets/XSilfide/EN/sxp/`).

Parsers will break up your document into its component pieces and make them accessible to other parts of a program, as we'll see later in this book; some parsers check for well-formedness, and fewer check for document validity. However, if you just want to check that your XML is both well formed and valid, all you need is an XML *validator*.

XML Validators

How do you know if your XML document is well formed and valid? One way is to check it with an XML validator, and there are plenty out there to choose from. Validators are packages that will check your XML and give you feedback. Here's a list of some of the XML validators on the Web:

- **W3C XML validator** (`http://validator.w3.org/`)—The official W3C HTML validator. Although it's officially for HTML, it also includes some XML support. Your XML document must be online to be checked with this validator.
- **Tidy** (`www.w3.org/People/Raggett/tidy/`)—Tidy is a beloved utility for cleaning up and repairing Web pages, and it includes limited support for XML. Your XML document must be online to be checked with this validator.
- `http://www.xml.com/pub/a/tools/ruwf/check.html`—This is XML.com's XML validator based on the Lark processor. Your XML document must be online to be checked with this validator.
- `www.ltg.ed.ac.uk/~richard/xml-check.html`—The Language Technology Group at the University of Edinburgh's validator, based on the RXP parser. Your XML document must be online to be checked with this validator.
- `http://www.stg.brown.edu/service/xmlvalid/`—The excellent XML validator from the Scholarly Technology Group at Brown University. This is the only online XML validator that I know of that allows you to check XML documents that are not online. You can use the Web page's file upload control to specify the name of the file on your hard disk that you want to have uploaded and checked.

To see a validator at work, take a look at Figure 1-11. There, I'm asking the XML validator from the Scholarly Technology Group to validate the XML document c:\xml\ch01_02.xml, where I've purposely exchanged the order of the `<MESSAGE>` and `</GREETING>` tags:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DOCUMENT [
    <!ELEMENT DOCUMENT (GREETING, MESSAGE)>
    <!ELEMENT GREETING (#PCDATA)>
    <!ELEMENT MESSAGE (#PCDATA)>
]>
<DOCUMENT>
    <GREETING>
        Hello From XML
```

continues

```
    <MESSAGE>
    </GREETING>
        Welcome to the wild and woolly world of XML.
    </MESSAGE>
</DOCUMENT>
```

You can see the results in Figure 1-12; the validator is indicating that there is a problem with these two tags.

XML validators give you a powerful way of checking your XML documents—and that's useful because XML is much stricter about making sure a document is correct than HTML browsers (recall that XML browsers are not supposed to make attempts to fix XML documents if they find a problem—they're just supposed to stop loading the document).

We've gotten a good overview of XML already in this chapter, and in a few pages I'll start taking a look at a number of XML languages that are already developed. However, there are a few more topics that are useful to cover first, especially if you have programmed in HTML and want to know the differences between XML and HTML.

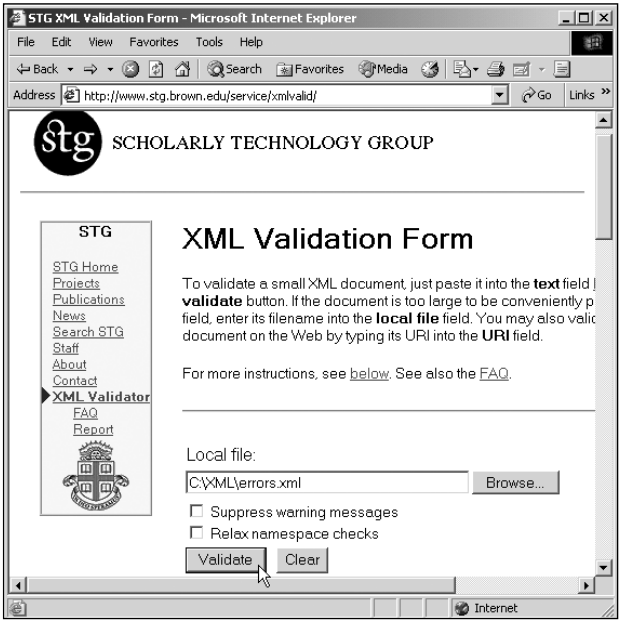

Figure 1-11 Using an XML validator.

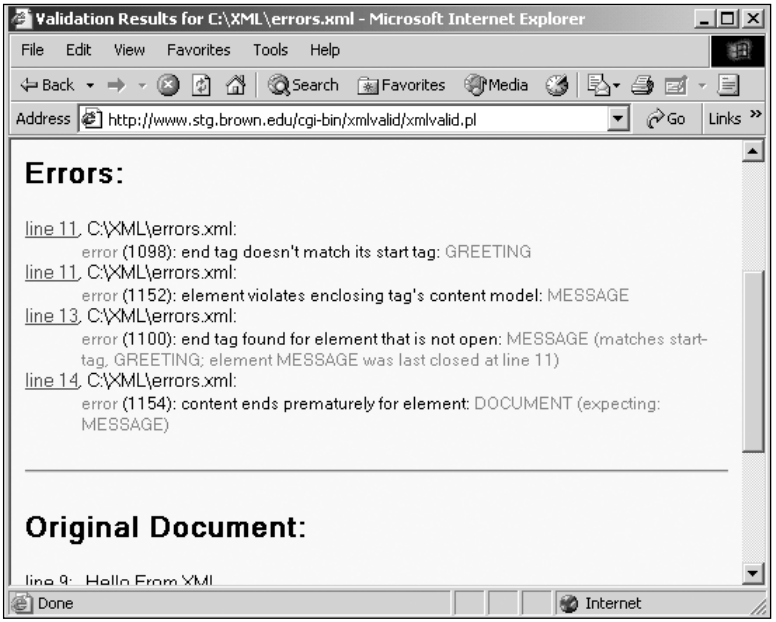

Figure 1-12 The results from an XML validator.

CSS and XSL

Stylesheets are becoming increasing important in HTML because, in HTML 4, many built-in style features such as the `<CENTER>` element have become deprecated (declared obsolete) in favor of stylesheets. However, most HTML programming ignores stylesheets entirely.

The story is different in XML, however. In XML, you create your own elements, which means that if you want a browser to display them, you have to tell it how. This is both good and bad: good because it allows you to use the powerful Cascading Style Sheet (CSS) and Extensible Styleheet Language (XSL) specifications to customize the appearance of your XML elements far beyond what's possible with standard HTML, but bad because it can demand a lot of additional work. (One way of getting around the necessity of designing your own stylesheets is to use an established XML language that has its own stylesheets.)

All this is to say that XML defines the structure and semantics of the document, not its format; if you want to display XML directly, you can either use the default presentation in Internet Explorer or use a stylesheet to set up the presentation yourself.

Two main ways exist for specifying a stylesheet for an XML document: with CSS and with XSL, both of which I'll dig into in this book. CSS is popularly used with HTML documents and is widely supported. Using CSS, you can specify the formatting of individual elements, create style classes, and set up fonts, colors, and even placement of elements in the page.

XSL, on the other hand, is ultimately a better choice to work with XML documents because it's more powerful (in fact, XSL stylesheets are themselves well-formed XML documents). XSL documents are made up of rules that are applied to XML documents. When a pattern that you've specified in the XSL document is recognized in the XML document, the rules transform the matched XML into something entirely new. You can even transform XML into HTML this way.

Whereas CSS can only set the format and placement of elements, XSL can reorder elements in a document, change them entirely, display some but hide others, select styles based not just on elements but also on element attributes (XML elements can have attributes just as HTML elements can, and I'll introduce them in the next chapter), select styles based on element location, and much more. There are two ways to approach XSL: with XSL transformations and XSL formatting objects. We'll take a look at both in this book.

Here are some good online resources for stylesheets:

- `www.w3.org/Style/CSS/`—The W3C outline and overview of CSS programming
- `www.w3.org/TR/REC-CSS1/`—The W3C CSS1 specification
- `www.w3.org/TR/REC-CSS2/`—The W3C CSS2 specification
- `www.w3.org/Style/XSL/`—The W3C XSL page

XLinks and XPointers

It's hard to imagine the World Wide Web without hyperlinks, and, of course, HTML documents excel at letting you link from one to another. How about XML? In XML, it turns out, you use XLinks and XPointers.

XLinks let any element become a link, not just a single element as with the HTML `<A>` element. That's a good thing because XML doesn't have a built-in `<A>` element. In XML, you define your own elements, and it only makes sense that you can define which of those represent links to other documents.

In fact, XLinks are more powerful than simple hyperlinks. XLinks can be bidirectional, allowing the user to return after following a link. They can even be multidirectional—in fact, they can be sophisticated enough to point to the nearest mirror site from which a resource can be fetched.

XPointers, on the other hand, point not to a whole document, but to a part of a document. In fact, XPointers are smart enough to point to a specific element in a document, or the second instance of such an element, or the 11,904th instance. They can even point to the first child element of another element, and so on. The idea is that XPointers are powerful enough to locate specific parts of another document without forcing you to add more markup to the target document.

On the other hand, the whole idea of XLinks and XPointers is relatively new and not fully implemented in any browser. We will see what's possible today later in this book.

Here are some XLink and XPointer references online—take a look for more information on these topics:

- `www.w3.org/TR/xlink/`—The W3C XLink page
- `www.w3.org/TR/xptr`—The W3C XPointer page

URLs Versus URIs

Having discussed XLinks and XPointers, I should also mention that the XML specification expands the idea of standard URLs (uniform resource locators) into URIs (uniform resource identifiers).

URLs are well understood and well supported on the Internet today. On the other hand, as you'd expect given the addition of XLinks and XPointers to XML, the idea of URIs is more general than simple URLs.

URIs represent a way of finding resources on the Internet, and they center more on the resource than the actual location. The idea is that, at least in theory, URIs can locate the nearest mirror site for a resource or even track a document that has been moved from one location to another.

In practice, the concept of URIs is still being developed, and most software will still handle only URLs.

URI: Formal Definition

You might want to look up the formal definition of URIs, which you can find in its entirety at `www.ics.uci.edu/pub/ietf/uri/rfc2396.txt`.

ASCII, Unicode, and the Universal Character System

The actual characters in documents are stored as numeric codes, and today the most common code set is the American Standard Code for Information Interchange (ASCII). ASCII codes extend from 0 to 127; for example, the ASCII code for *A* is 65, the ASCII code for *B* is 66, and so on.

On the other hand, the World Wide Web is just that today—worldwide. And plenty of scripts are not handled by ASCII, including Bengali, Armenian, Hebrew, Thai, Tibetan, Japanese Katakana, Arabic, and Cyrillic.

For that reason, the default character set specified for XML by W3C is Unicode, not ASCII. Unicode codes are made up of 2 bytes, not 1, so they extend from 0 to 65,535 instead of just 0 to 255 (however, to make things easier, the Unicode codes 0 to 255 do indeed correspond to the ASCII 0 to 255 codes). Unicode can therefore include many of the symbols commonly used in worldwide character and ideograph sets today.

Only about 40,000 Unicode codes are reserved at this point (of which about 20,000 codes are used for Han ideographs, although there are more than 80,000 such ideographs defined and 11,000 for Korean Hangul syllables).

In practice, Unicode support, like many parts of the XML technology, is not fully supported on most platforms today. Windows 95/98 does not have full support for Unicode, although Windows NT and Windows 2000 come much closer (and XML Spy lets you use Unicode to write XML documents in Windows NT). What this means most often is that XML documents are written in simply ASCII or in UTF-8, which is a compressed version of Unicode that uses 8 bits to represent characters (in practice, this is well suited to ASCII documents because multiple bytes are needed for many non-ASCII symbols and because ASCII documents converted to Unicode are two times as long). Here's how to specify the UTF-8 character encoding in an XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<DOCUMENT>
     <GREETING>
          Hello From XML
     </GREETING>
     <MESSAGE>
          Welcome to the wild and woolly world of XML.
     </MESSAGE>
</DOCUMENT>
```

In fact, the default for XML processors today is to assume that your document is in UTF-8, so if you omit the encoding specification, UTF-8 is assumed. So if you're writing XML documents in ASCII, you'll have no trouble.

Actually, not even Unicode has enough space for all symbols in common use, so a new specification, the Universal Character System (UCS, also called ISO 10646) uses 4 bytes per symbol. This gives it a range of two billion symbols—far more than needed. You can specify that you want to use pure Unicode encoding in your XML documents by using the UCS-2 encoding (also called ISO-10646-UCS-2), which is compressed 2-byte UCS. You can also use UTF-16, which is a special encoding that represents UCS symbols using 2 bytes so that the result corresponds to UCS-2. Straight UCS encoding is referred to as UCS-4 (also called ISO-10646-UCS-4).

You can write documents in a local character set and use a translation utility to translate them to Unicode, or you can insert the actual Unicode codes directly into your documents. For example, the Unicode for π is 0x3C0 in hexadecimal, so you can insert π into your document with the character entity (more on entities in the next chapter) `&#x03C0;`.

More character sets are available than those mentioned here; for a longer list, take a look at the list posted by the Internet Assigned Numbers Authority (IANA), at `www.iana.org/assignments/character-sets`.

Converting ASCII to Unicode

If you want to convert ASCII files to straight Unicode, you can use the native2ascii program that comes with Sun Microsystems's Java Software Development Kit. Using this tool, you can convert to Unicode like this: `native2ascii file.txt file.uni`. You can also convert to a number of other encodings besides Unicode, such as compressed Unicode, UTF-8.

XML Applications

We've seen a lot of theory in this chapter, so I'm going to spend the rest of this chapter taking a look at how XML is used today in the real world. The world of XML is huge these days; in fact, XML is now used internally even in Netscape and Microsoft products, as well as installations of programming languages such as Perl. You can find a good list of organizations that produce their own XML-based languages at `www.xml.org/xml/marketplace_company.jsp`.

It's useful and encouraging to see how XML is being used today in these XML-based languages. Here's a new piece of terminology: As you know, XML is a metamarkup language, so it's actually used to create languages. The languages so created are applications of XML; as a result, they're called *XML applications*.

Note that the term *XML application* means an application of XML to a specific domain, such as MathML, the mathematics markup language; it does not refer to a program that uses XML (a fact that causes a lot of confusion among people who know nothing about XML).

Thousands of XML applications are around today, and we'll see some of them here. You can see the advantage to various groups when defining their own markup languages. For example, physicists or chemists can use the symbols and graphics of their discipline in customized browsers. In fact, I'll start with Chemical Markup Language (CML) now.

XML at Work: Chemical Markup Language

Peter Murray-Rust developed CML as a very early XML application, so it has been around quite a while. Many people think of CML as a sort of HTML+Molecules, and that's not a bad characterization. Using CML, you can display the structure of complex molecules.

With CML, chemists can create and publish molecule specifications for easy interchange. Note that the real value of this is not so much in looking at individual chemicals as it is in being able to search CML repositories for molecules matching specific characteristics.

I've already mentioned a famous CML browser available: Jumbo, which you can download for free from `www.xml-cml.org/jumbo.html`. Jumbo is not only for handling CML; you can also use it to display the structure of an XML document in general. However, there's no question that the novelty of Jumbo is that it can use CML to create graphical representations of molecules.

We've already seen an example in Jumbo in Figure 1-10, where Jumbo is displaying the molecule thiophenol. Here is the file thiophenol.xml that it's reading to display that molecule:

```
<?jumbo:namespace ns="http://www.xml-cml.org" prefix="C"
    java="jumbo.cmlxml.*Node" ?>
<C:molecule id="thiophenol">
    <C:atomArray builtin="elsym">
        C C C C C C S C C O O
```

```
    </C:atomArray>
    <C:atomArray builtin="x2" type="float">
        0 0.866 0.866 0 -0.866 -0.866
        0.0 0.0  1.732 -1.732 1.732 -1.732
    </C:atomArray>
    <C:atomArray builtin="y2" type="float">
        1 0.5   -0.5  -1.0 -0.5   0.5
        -2.0  2.0  1.0   1.0    2.0   2.0
    </C:atomArray>
    <C:bondArray builtin="atid1">
        1 2 3 4 5 6 1 4 2 9  6  10
    </C:bondArray>
    <C:bondArray builtin="atid2">
        2 3 4 5 6 1 8 7 9 11 10 12
    </C:bondArray>
    <C:bondArray builtin="order" type="integer">
        4 4 4 4 4 4 1 1 1 2  1  2
    </C:bondArray>
</C:molecule>
```

XML at Work: Mathematical Markup Language

Mathematical Markup Language was designed to fill a significant gap in Web documents: equations. In fact, Tim Berners-Lee first developed the World Wide Web at CERN so that high-energy physicists could exchange papers and documents. However, there has been no way to display true equations in Web browsers for nearly a decade.

Mathematical Markup Language (MathML) fixes that. MathML is itself a W3C specification, and you can find it at `www.w3.org/Math/`. Using MathML, you can display equations and all kinds of mathematical terms. (It's not powerful enough for many specialized areas of the sciences or mathematics yet, but it's growing all the time.)

Because of the limited audience for this kind of presentation, no major browser yet supports MathML. However, the Amaya browser, which is W3C's own testbed browser for testing new HTML and XHTML elements (but it's not, unfortunately, an XML browser) has some limited support. You can download Amaya for free from `www.w3.org/Amaya/`.

Here's a MathML document that displays the equation $3Z^2 - 6Z + 12 = 0$ (this document uses an XML namespace, which we'll see more about in the next chapter):

```
<?xml version="1.0"?>
<html xmlns:m="http://www.w3.org/TR/REC-MathML/">
<math>
     <m:mrow>
          <m:mrow>
               <m:mn>3</m:mn>
               <m:mo>&InvisibleTimes;</m:mo>
               <m:msup>
                    <m:mi>Z</m:mi>
                    <m:mn>2</m:mn>
               </m:msup>
               <m:mo>-</m:mo>
               <m:mrow>
                    <m:mn>6</m:mn>
                    <m:mo>&InvisibleTimes;</m:mo>
                    <m:mi>Z</m:mi>
               </m:mrow>
               <m:mo>+</m:mo>
               <m:mn>12</m:mn>
          </m:mrow>
          <m:mo>=</m:mo>
          <m:mn>0</m:mn>
     </m:mrow>
</math>
```

You can see the results of this document in the Amaya browser in Figure 1-13.

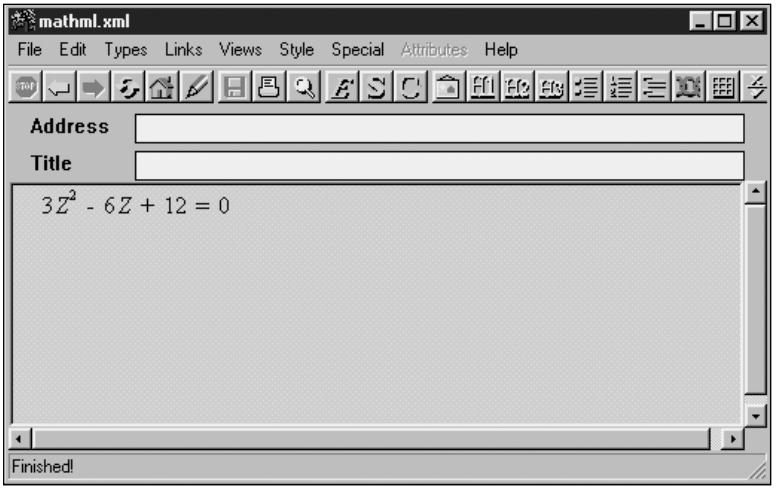

Figure 1-13 Displaying MathML in the Amaya browser.

XML at Work: Synchronized Multimedia Integration Language

Synchronized Multimedia Integration Language (SMIL, pronounced "smile") has been around for quite some time. It's a W3C standard that you can find more about at `www.w3.org/AudioVideo/#SMIL`.

SMIL attempts to fix a problem with modern "multimedia" browsers. Usually, such browsers can handle only one aspect of multimedia at a time—video, audio, or images—and never more than that. SMIL lets you create television-like fast cuts and true multimedia presentations by letting you specify when various multimedia files are played.

The idea is that SMIL lets you specify what multimedia files are played when; SMIL itself does not describe or encapsulate any multimedia itself.

Microsoft, Macromedia, and Compaq have a semicompeting specification, HTML+TIME, which I'll take a look at next. Microsoft hasn't implemented much SMIL in Internet Explorer yet because of this reason. You can find a SMIL applet written in Java at `www.empirenet.com/~joseram`, as well as some stunning examples of symphonies coordinated with images.

Here's an example SMIL document that creates a multimedia sequence playing mozart1.wav and amadeus1.mov, displaying mozart1.htm, then playing mozart2.wav and amadeus2.mov, and displaying mozart2.htm:

```
<?xml version="1.0"?>
<!DOCTYPE smil PUBLIC "-//W3C//DTD SMIL 1.0//EN"
  "http://www.w3.org/TR/REC-smil/SMIL10.dtd">
<smil>
    <body>
        <seq id="mozart">
            <audio src="mozart1.wav"/>
            <video src="amadeus1.mov"/>
            <text src="mozart1.htm"/>
            <audio src="mozart2.wav"/>
            <video src="amadeus2.mov"/>
            <text src="mozart2.htm"/>
        </seq>
    </body>
</smil>
```

XML at Work: HTML+TIME

Microsoft, Macromedia, and Compaq have a multimedia alternative to SMIL called Timed Interactive Multimedia Extension (referred to as HTML+TIME), which is an XML application. Whereas SMIL documents let you manipulate other files, HTML+TIME lets you handle both HTML and multimedia presentations in the same page.

HTML+TIME is not nearly as powerful as SMIL, but Microsoft has shown relatively little interest in SMIL. You can find out about HTML+TIME at `msdn.microsoft.com/workshop/Author/behaviors/time.asp`. HTML+TIME is implemented in the Internet Explorer as a *behavior*, which is a construct in Internet Explorer that lets you separate code from data. You can find more information about Internet Explorer behaviors at `msdn.microsoft.com/workshop/c-frame.htm#/workshop/author/default.asp`.

Here's an example HTML+TIME document that displays the words `Hello`, `there`, `from`, `HTML+TIME`, spacing the words' appearance apart by 2 seconds and then repeating:

Listing ch01_10.html

```
<HTML>
    <HEAD>
        <TITLE>
            Using HTML+TIME
        </TITLE>
        <STYLE>
            .time {behavior: url(#default#time);}
        </STYLE>
    </HEAD>

    <BODY>
        <DIV CLASS="time" t:REPEAT="5" t:DUR="10" t:TIMELINE="par">
            <DIV CLASS="time" t:BEGIN="0" t:DUR="10">Hello</DIV>
            <DIV CLASS="time" t:BEGIN="2" t:DUR="10">there</DIV>
            <DIV CLASS="time" t:BEGIN="4" t:DUR="10">from</DIV>
            <DIV CLASS="time" t:BEGIN="6" t:DUR="10">HTML+TIME.</DIV>
        </DIV>
    </BODY>
</HTML>
```

You can see the results of this HTML+TIME document in Figure 1-14.

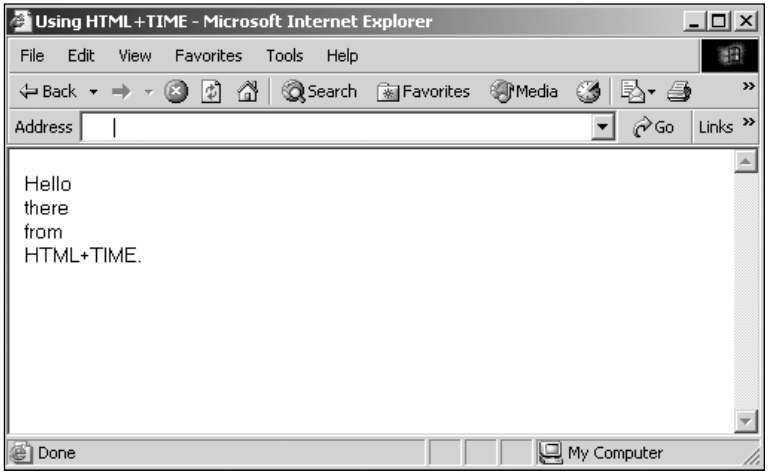

Figure 1-14 An HTML+TIME document at work.

HTML+TIME actually builds on SMIL to a great extent. The example from the previous topic on SMIL would look this way in HTML+TIME:

```
<t:seq id="mozart">
    <t:audio src="mozart1.wav"/>
    <t:video src="amadeus1.mov"/>
    <t:textstream src="mozart1.htm"/>
    <t:audio src="mozart2.wav"/>
    <t:video src="amadeus2.mov"/>
    <t:textstream src="mozart2.htm"/>
</seq>
```

XML at Work: XHTML

One of the biggest XML applications around today is XHTML, the translation of HTML 4.0 into XML by W3C. It's attracting a lot of attention. I'll dig into XHTML in some depth in this book.

W3C introduced XHTML to bridge the gap between HTML and XML, and to introduce more people to XML. XHTML is simply an application that mimics HTML 4.0 in such a way that you can display the results—true XML documents—in current Web browsers. XHTML is an exciting development in the XML world, and we'll be spending some time with it later in this book.

Here are some XHTML resources online:

- `www.w3.org/MarkUp/Activity.html`—The W3C Hypertext Markup activity page, which has an overview of XHTML
- `www.w3.org/TR/xhtml1/`—The XHTML 1.0 specification (in more common use than XHTML 1.1 today)
- `www.w3.org/TR/xhtml11/`—The XHTML 1.1 working draft of the XHTML 1.1 module-based specification

XHTML 1.0 comes in three different versions: transitional, frameset, and strict. The transitional version is the most popular because it supports HTML more or less as it's used today. The frameset version supports XHTML documents that display frames (this version is different than the transitional version because documents in the transitional version are based on the `<body>` element, whereas documents that use frames are based on the `<frameset>` element). The strict version omits all the HTML elements deprecated in HTML 4.0 (of which there were quite a few).

XHTML 1.1 is a form of the XHTML 1.0 strict version made a little more strict by omitting support for some elements and adding support for a few more (such as `<ruby>` for annotated text). You can find a list of the differences between XHTML 1.0 and XHTML 1.1 at `www.w3.org/TR/xhtml11/changes.html#a_changes`.

Here's an example XHTML document using the XHTML 1.0 transitional DTD. You can display this document in any standard HTML browser (note that tag names are all in lowercase in XHTML):

Listing ch01_11.html

```
<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
    <head>
        <title>
            Web page number one!
        </title>
    </head>

    <body>
        <h1>
            Welcome to XHTML!
        </h1>
        <center>
```

```
        This is simple text that appears in this page.
            <p>
                Here's a new paragraph!
            </p>
        </center>
    </body>
</html>
```

You can see the results of this XHTML in Figure 1-15. Writing XHTML is a lot like HTML, except that you have to adhere to XML syntax (such as making sure that every element has a closing tag).

Figure 1-15 Displaying XHTML.

XML at Work: Microsoft's .NET

Microsoft's .NET initiative is based substantially on XML, which it uses to send data back and forth between .NET components. In .NET, you don't usually see the XML—it's handled behind the scenes automatically—but it's there.

Here's an example in VB .NET that will expose the behind-the-scenes XML: The data in .NET datasets is transported using XML, and this example explicitly writes the `authors` database table of the `pubs` example database to an XML file when the user clicks a button. When the user clicks another button, the code reads that file back into a second .NET dataset. You can see this example at work in Figure 1-16 (which also displays the data in the `authors` table).

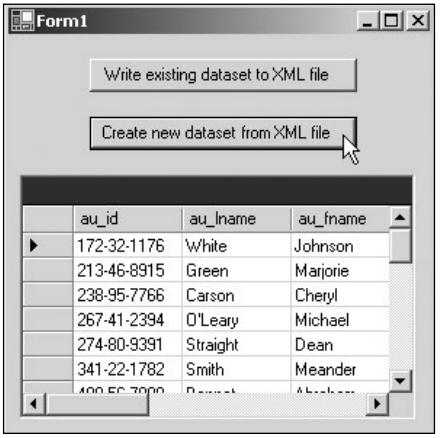

Figure 1-16 Writing data in XML in VB .NET.

Here's the VB .NET code—when the user clicks the "Write existing dataset to XML file" button you see in Figure 1-16, the `authors` table in the dataset is written to the file dataset.xml; and when the user clicks the "Create new dataset from XML file" button, a new dataset is created and reads its data in from dataset.xml:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    DataSet11.Clear()
    OleDbDataAdapter1.Fill(DataSet11)
    DataSet11.WriteXml("dataset.xml")
End Sub

Private Sub Button2_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button2.Click
    Dim ds As New DataSet()
    ds.ReadXml("dataset.xml")
    DataGrid1.SetDataBinding(ds, "authors")
End Sub
```

You can see the dataset's data in the dataset.xml file, which looks like this—it's pure XML (and matches the data you see in Figure 1-16):

```
<?xml version="1.0" standalone="yes"?>
<DataSet1 xmlns="http://www.tempuri.org/DataSet1.xsd">
  <authors>
    <au_id>172-32-1176</au_id>
    <au_lname>White</au_lname>
    <au_fname>Johnson</au_fname>
    <phone>408 496-7223</phone>
    <address>10932 Bigge Rd.</address>
    <city>Menlo Park</city>
    <state>CA</state>
```

```
    <zip>94025</zip>
    <contract>true</contract>
  </authors>
  <authors>
    <au_id>213-46-8915</au_id>
    <au_lname>Green</au_lname>
    <au_fname>Marjorie</au_fname>
    <phone>415 986-7020</phone>
         .
         .
         .
```

And that provides us with a glimpse at the actual XML used behind the scenes to transport data in .NET—something that's usually handled automatically.

XML at Work: Open Software Description

Open Software Description (OSD) was developed by Marimba and Microsoft, and you can find more about this XML application at `www.w3.org/TR/NOTE-OSD.html`. OSD allows you to specify how and when software is updated via the Internet.

Not everyone thinks OSD is a great idea—after all, many users want control over when their software is updated. New versions might have incompatibilities with old versions, for example.

Here's an example .osd file that handles updates for a word processor named SuperDuperTextPro from SuperDuperSoft:

```
<?xml version="1.0"?>
<CHANNEL HREF="http://www.superdupersoft.com/updates.html">
    <TITLE>
        SuperDuperTextPro Updates
    </TITLE>
    <USAGE VALUE="SoftwareUpdate"/>
    <SOFTPKG
        HREF="http://updates.superdupersoft.com/updates.html"
        NAME="{34567A7E-8BE7-99C0-8746-0034829873A3}"
        VERSION="2,4,6">
        <TITLE>
            SuperDuperTextPro
        </TITLE>
        <ABSTRACT>
            SuperDuperTextPro version 206 with sideburns!!!
        </ABSTRACT>
        <IMPLEMENTATION>
             <CODEBASE HREF=
                  "http://www.superdupersoft.com/new.exe"/>
        </IMPLEMENTATION>
    </SOFTPKG>
</CHANNEL>
```

XML at Work: Scalable Vector Graphics

Scalable Vector Graphics (SVG) is another W3C-based XML application that is a good idea but that has found only limited implementation so far (notably, in such programs as CorelDraw and various Adobe products such as Adobe Illustrator). Using SVG, you can draw two-dimensional graphics using markup. You can find the SVG specification at `www.w3.org/TR/SVG/` and an overview at `www.w3.org/Graphics/SVG/Overview.htm8`.

Note that because SVG describes graphics, not text, it's harder for current browsers to implement, and are no browsers today have full SVG implementations. Other graphics standards have been proposed, such as the Precision Graphics Markup Language (PGML) proposed to the W3C (`www.w3.org/TR/1998/NOTE-PGML`) by IBM, Adobe, Netscape, and Sun.

Here's an example PGML document that draws a blue box:

```
<?xml version="1.0"?>
<!DOCTYPE pgml SYSTEM "/DTDs/pgml.dtd">
<pgml>
  <group fillcolor="blue">
    <path>
      <moveto x="0" y="0"/>
      <lineto x="0" y="1000"/>
      <lineto x="1000" y="1000"/>
      <lineto x="1000" y="0"/>
      <closepath/>
    </path>
  </group>
</pgml>
```

XML at Work: Vector Markup Language

Vector Markup Language (VML) is an alternative to SVG that is implemented in Microsoft Internet Explorer. You can find out more about VML at `www.w3.org/TR/NOTE-VML`. Using VML, you can draw many vector-based graphics figures; here's an example that draws a yellow oval, a blue box, and a red squiggle:

Listing ch01_12.html

```
<HTML xmlns:v="urn:schemas-microsoft-com:vml">

    <HEAD>
        <TITLE>
            Using Vector Markup Language
        </TITLE>
```

```
        <STYLE>
        v\:* {behavior: url(#default#VML);}
        </STYLE>
    </HEAD>

    <BODY>
        <CENTER>
            <H1>
                Using Vector Markup Language
            </H1>
        </CENTER>
        <P>
        <v:oval STYLE='width:100pt; height:75pt'
            fillcolor="yellow"> </v:oval>
        <P>
        <v:rect STYLE='width:100pt; height:75pt' fillcolor="blue"
            strokecolor="red" STROKEWEIGHT="2pt"/>
        <P>
        <v:polyline
            POINTS="20pt,55pt,100pt,-10pt,180pt,65pt,260pt,25pt"
            strokecolor="red" STROKEWEIGHT="2pt"/>
    </BODY>
</HTML>
```

You can see the results of this VML in Figure 1-17.

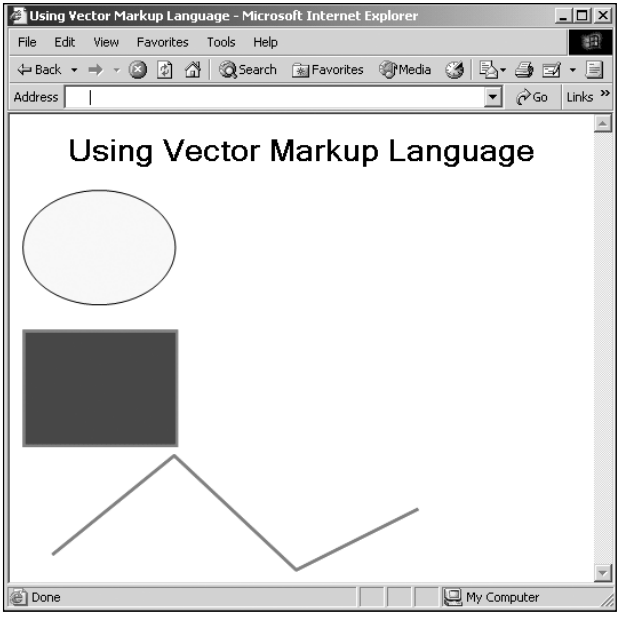

Figure 1-17 Vector Markup Language at work.

Extensible Business Reporting Language

Extensible Business Reporting Language (XBRL, formerly named XFRML) is an open specification that uses XML to describe financial statements. You can find more on XBRL at `www.xbrl.org/`. Using XBRL, you can codify business financial statements in a way that makes it easy to search them en masse and review them quickly, extracting the information you want.

Here's a sample XBRL document that gives you an idea of what this application looks like at work:

```
<?xml version="1.0" encoding="utf-8" ?>
    <group xmlns="http://www.xbrl.org/us/aicpa-us-gaap"
        xmlns:gpsi="http://www.xbrl.org/TaxonomyCustom.xsd"
        id="543-AB" entity="NASDAQ:GPSI" period="1999-05-31"
        schemaLocation="http://www.xbrl.org/TaxonomyCustom.xsd"
        scaleFactor="6" precision="9" type="USGAAP:Financial"
        unit="ISO4217:USD" decimalPattern="" formatName="">
        <item id="IS-025"
            type="operatingExpenses.researchExpense"
            period="P1Y/1999-05-31">20427</item>
        <item id="IS-026"
            type="operatingExpenses.researchExpense"
            period="P1Y/1998-05-31">12586</item>
    </group>
    <group type="gpsi:detail.quarterly" period="1998-05-31">
        <item period="1997-06-01/1998-07-31">0.12</item>
        <item period="1997-09-01/1997-11-30">0.16</item>
        <item period="1997-12-01/1998-02-28">0.17</item>
        <item period="1998-03-01/1998-05-31">-0.12</item>
        <item period="1998-06-01/1998-05-31">0.33</item>
    </group>
    <group type="gpsi:detail.quarterly" period="1999-05-31">
        <item period="1998-06-01/1998-08-31">0.15</item>
        <item period="1998-09-01/1998-11-30">0.20</item>
        <item period="1998-12-01/1999-02-28">0.23</item>
        <item period="1999-03-01/1999-05-31">0.28</item>
        <item period="1998-06-01/1999-05-31">0.86</item>
    </group>
    <group type="gpsi:detail.quarterly" period="1998-05-31">
        <item period="1997-06-01/1998-07-31">0.11</item>
        <item period="1997-09-01/1997-11-30">0.15</item>
        <item period="1997-12-01/1998-02-28">0.17</item>
        <item period="1998-03-01/1998-05-31">-0.12</item>
        <item period="1998-06-01/1998-05-31">0.32</item>
    </group>
```

Resource Description Framework

Resource Description Framework (RDF) is an XML application that specializes in metadata—that is, data about other data. You use RDF to specify information about other resources, such as Web pages, movies, automobiles, or practically anything. You can find more information about RDF at `www.w3.org/RDF/`; I'll be discussing it later in the book as well.

Using RDF, you create vocabularies that describe resources. For example, the Dublin Core is an RDF vocabulary that handles metadata for Web pages; you can find more information about it at `http://dublincore.org/`. Using the Dublin Core, you can specify a great deal of information about Web pages. The Dublin Core is designed ultimately to replace the unsystematic use of `<META>` tags in today's pages. When systemized, that information will be much more tractable to Web search engines.

Here's an example RDF page using the Dublin Core that gives information about a Web page:

```
<RDF:RDF xmlns:RDF="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:DC="http://purl.org/DC/">
     <RDF:Description about="http://www.starpowder.com/xml">
          <DC:Format>HTML</DC:Format>
          <DC:Language>en</DC:Language>
          <DC:Date>2002-02-02</DC:date>
          <DC:Type>tutorial</DC:Type>
          <DC:Title>Welcome to XML!</DC:Title>
     </RDF:Description>
</RDF:RDF>
```

Note that many more XML applications exist than can be covered in one chapter—and plenty of them work behind the scenes. As mentioned earlier, Microsoft's .NET initiative uses XML extensively internally. Microsoft Office 2000 and Office XP can handle HTML as well as other types of documents, but HTML doesn't allow it to store everything it needs in a document; thus, it also includes some XML behind the scenes (in fact, Office 2000 and XP's vector graphics are done using VML). Even relatively early versions of Netscape Navigator allowed you to look for sites much like the current one you're viewing; to do that, it connected to a program that uses XML internally. As you can see, XML is everywhere you look on the Internet.

And that's it for our overview chapter. We've gotten a solid foundation in XML here, and it's a good place to begin. The next step is to get systematic and to start getting all the actual ground rules for creating XML documents under our belts. I'll turn to that in Chapter 2.