1

What Is XML?

WHAT YOU'LL WILL LEARN IN THIS CHAPTER:

- The story before XML
- How XML arrived
- The basic format of an XML document
- Areas where XML is useful
- A brief introduction to the technologies surrounding, and associated with, XML

XML stands for *Extensible Markup Language* (presumably the original authors thought that sounded more exciting than *EML*) and its development and usage have followed a common path in the software and IT world. It started out more than ten years ago and was originally used by very few; later it caught the public eye and began to pervade the world of data exchange. Subsequently, the tools available to process and manage XML became more sophisticated, to such an extent that many people began to use it without being really aware of its existence. Lately there has been a bit of a backlash in certain quarters over its perceived failings and weak points, which has led to various proposed alternatives and improvements. Nevertheless, XML now has a permanent place in IT systems and it's hard to imagine any non-trivial application that doesn't use XML for either its configuration or data to some degree. For this reason it's essential that modern software developers have a thorough understanding of its principles, what it is capable of, and how to use it to their best advantage. This book can give the reader all those things.

> ***NOTE*** *Although this chapter presents some short examples of XML, you aren't expected to understand all that's going on just yet. The idea is simply to introduce the important concepts behind the language so that throughout the book you can see not only how to use XML, but also why it works the way it does.*

STEPS LEADING UP TO XML: DATA REPRESENTATION AND MARKUPS

There are two main uses for XML: One is a way to represent low-level data, for example configuration files. The second is a way to add metadata to documents; for example, you may want to stress a particular sentence in a report by putting it in *italics* or **bold**.

The first usage for XML is meant as a replacement for the more traditional ways this has been done before, usually by means of lists of name/value pairs as is seen in Windows' *INI* or Java's *Property* files. The second application of XML is similar to how HTML files work. The document text is contained in an overall container, the `<body>` element, with individual phrases surrounded by `<i>` or `<b>` tags. For both of these scenarios there has been a multiplicity of techniques devised over the years. The problem with these disparate approaches has been more apparent than ever, since the increased use of the Internet and extensive existence of distributed applications, particularly those that rely on components designed and managed by different parties. That problem is one of intercommunication. It's certainly possible to design a distributed system that has two components, one outputting data using a Windows INI file and the other which turns it into a Java Properties format. Unfortunately, it means a lot of development on both sides, which shouldn't really be necessary and detracts resources from the main objective, developing new functionality that delivers business value.

XML was conceived as a solution to this kind of problem; it is meant to make passing data between different components much easier and relieve the need to continually worry about different formats of input and output, freeing up developers to concentrate on the more important aspects of coding such as the business logic. XML is also seen as a solution to the question of whether files should be easily readable by software or by humans; XML's aim is to be both. You'll be examining the distinction between data-oriented and document-centric XML later in the book, but for now let's look a bit more deeply into what the choices were before XML when there was need to store or communicate data in an electronic format.

This section takes a mid-level look at data representation, without taking too much time to explain low-level details such as memory addresses and the like. For the purposes here you can store data in files two ways: as binary or as text.

Binary Files

A *binary file*, at its simplest, is just a stream of *bits* (1s and 0s). It's up to the application that created the binary file to understand what all of the bits mean. That's why binary files can only be read and produced by certain computer programs, which have been specifically written to understand them.

For example, when saving a document in Microsoft Word, using a version before 2003, the file created (which has a *doc* extension) is in a binary format. If you open the file in a text editor such as Notepad, you won't be able to see a picture of the original Word document; the best you'll be able to see is the occasional line of text surrounded by gibberish rather than the prose, which could be in a number of formats such as bold or italic. The characters in the document other than the actual text are metadata, literally information about information. Mixing data and metadata is both common and straightforward in a binary file. Metadata can specify things such as which words should be shown in bold, what text is to be displayed in a table, and so on. To interpret this file you the need the help of the application that created it. Without the help of a converter that has in-depth knowledge of the underlying binary format, you won't be able to open a document created in Word with another similar application such as WordPerfect. The main advantage of binary formats is that they are concise and can be expressed in a relatively small space. This means that more files can be stored (on a hard drive, for example) but, more importantly nowadays, less bandwidth is used when transporting these files across networks.

Text Files

The main difference between text and binary files is that text files are human and machine readable. Instead of a proprietary format that needs a specific application to decipher it, the data is such that each group of bits represents a character from a known set. This means that many different applications can read text files. On a standard Windows machine you have a choice of Notepad, WordPad, and others, including being able to use command-line–based utilities such as Edit. Non-Windows machines have a similar wide range insert of programs available, such as Emacs and Vim.

> ***NOTE*** *The way that characters are represented by the underlying data stream is referred to as a file's* encoding. *The specific encoding used is often present as the first few bytes in the file; an application checks these bytes upon opening the file and then knows how to display and manipulate the data. There is also a default encoding if these first few bytes are not present. XML also has other ways of specifying how a file was encoded, and you'll see these later on.*

The ability to be read and understood by both humans and machines is not the only advantage of text files; they are also comparatively easier to parse than binary files. The main disadvantage however, is their size. In order for text files to contain metadata (for example, a stretch of text to be marked as important), the relevant words are usually surrounded by characters denoting this extra information, which are somehow differentiated from the actual text itself. The most common examples of this can be found in HTML, where angle brackets are special symbols used to convey the meaning that anything within them refers to how the text should be treated rather than the actual data. For example, if I want mark a phrase as important I can wrap it like so:

```
<strong>returns must include the item order number</strong>
```

Another disadvantage of text files is their lack of support for metadata. If you open a Word document that contains text in an array of fonts with different styles and save it as a text file, you'll just get a plain rendition; all of the metadata has been lost. What people were looking for was some way to have the best of both worlds — a human-readable file that could also be read by a wide range of applications, and could carry metadata along with its content. This brings us to the subject of markup.

A Brief History of Markup

The advantages of text files made it the preferred choice over binary files, yet the disadvantages were still cumbersome enough that people wanted to also standardize how metadata could be added. Most agreed that markup, the act of surrounding text that conveyed information about the text, was the way forward, but even with this agreed there was still much to be decided. The main two questions were:

- How can metadata be differentiated from the basic text?
- What metadata is allowed?

For example, some documents needed the ability to mark text as bold or italic whereas others were more concerned with who the original document author was, when was it created, and who had subsequently modified it. To cope with this problem a definition called *Standard Generalized Markup Language* was released, commonly shortened to *SGML*. SGML is a step removed from defining an actual markup language, such as the Hyper Text Markup Language, or HTML. Instead it relays how markup languages are to be defined. SGML allows you to create your own markup language and then define it using a standard syntax such that any SGML-aware application can consume documents written in that language and handle them accordingly. As previously noted, the most ubiquitous example of this is HTML. HTML uses angular brackets (< and >) to separate metadata from basic text and also defines a list of what can go into these brackets, such as `em` for emphasizing text, `tr` for table, and `td` for representing tabular data.

THE BIRTH OF XML

SGML, although well thought-out and capable of defining many different types of markup, suffered from one major failing: it was very complicated. All the flexibility came at a cost, and there were still relatively few applications that could read the SGML definition of a markup language and use it to correctly process documents. The concept was correct, but it needed to be simpler. With this goal in mind, a small working group and a larger number of interested parties began working in the mid-1990s on a subset of SGML known as Extensible Markup Language (XML). The first working draft was published in 1996 and two years later the W3C published a revised version as a recommendation on February 10, 1998.

> ***NOTE*** *The* World Wide Web Consortium (W3C) *is the main international standards organization for the World Wide Web. It has a number of working groups targeting different aspects of the Web that discuss standardization and documentation of the different technologies used on the Internet. The standards documents go through various stages such as* Working Draft *and* Candidate Recommendation *before finally becoming a* Recommendation. *This process can take many years. The reason that the final agreement is called a recommendation rather than a standard is that you are still free to ignore what it says and use your own. All web developers know the problems in developing applications that work across all browsers, and many of these problems arise because the browser vendors did not follow a W3C recommendation or they did not implement features before the recommendation was finalized. Most of the XML technologies discussed in this book have a W3C recommendation associated with them, although some don't have a full recommendation because they are still in draft form. Additionally, some XML-related standards originate from outside the W3C, such as SAX which is discussed in Chapter 11, "Event Driven Programming." and therefore they also don't have official W3C recommendations.*

XML therefore derived as a subset of SGML, whereas HTML is an application of SGML. XML doesn't dictate the overall format of a file or what metadata can be added, it just specifies a few rules. That means it retains a lot of the flexibility of SGML without most of the complexity. For example, suppose you have a standard text file containing a list of application users:

```
Joe Fawcett
Danny Ayers
Catherine Middleton
```

This file has no metadata; the only reason you know it's a list of people is your own knowledge and experience of how names are typically represented in the western world. Now look at these names as they might appear in an XML document:

```
<applicationUsers>
  <user firstName="Joe" lastName="Fawcett" />
  <user firstName="Danny" lastName="Ayers" />
  <user firstName="Catherine" lastName="Middleton" />
</applicationUsers>
```

Immediately it's more apparent what the individual pieces of data are, although an application still wouldn't know just from that file how to treat a `user` or what `firstName` means. Using the XML format rather than the plain text version, it's much easier to map these data items within the application itself so they can be handled correctly.

The two common features of virtually all XML file are called *elements* and *attributes*. In the preceding example, the elements are `applicationUsers` and `user`, and the attributes are `firstName` and `lastName`.

A big disadvantage of this metadata, however, is the consequent increase in the size of the file. The metadata adds about 130 extra characters to the file's original 43 character size, an increase of more than 300 percent. The creators of XML decided that the power of metadata warranted this increase and, indeed, one of their maxims during the design was that *terseness is not an aim*, a decision that many would later come to regret.

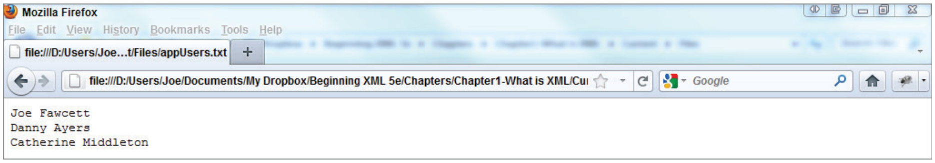
> ***NOTE*** *Later on in the book you'll see a number of ways to minimize the size of an XML file if needed. However, all these methods are, to some extent, a trade-off against readability and ease of use.*

Following is a simple exercise to demonstrate the differences in how applications handle simple text files against how XML is treated. Even though the application, in this case a browser, is told nothing in advance of opening the two files, you'll see how much more metadata is available in the XML version compared to the text one.

TRY IT OUT Opening an XML File in a Browser

This example shows the differences in how XML files can be handled compared to plain text files.

1. Create a new text file in Notepad, or an equivalent simple text editor, and paste in the list of names first shown earlier.
2. Save this file at a convenient location as `appUsers.txt`.
3. Next, open a browser and paste the path to `appUsers.txt` into the address bar. You should see something like Figure 1-1. Notice how it's just a simple list:

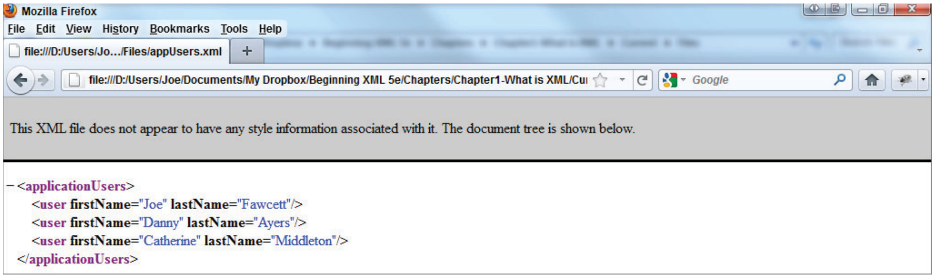

Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///D:/Users/Joe...t/Files/appUsers.txt
Google

file:///D:/Users/Joe/Documents/My Dropbox/Beginning XML 5e/Chapters/Chapter1-What is XML/Cu

```
Joe Fawcett
Danny Ayers
Catherine Middleton
```

FIGURE 1-1

4. Now create another text file based on the XML version and save it as `appUsers.xml`. If you're doing this in Notepad make sure you put quotes around the full name before saving or otherwise you'll get an unwanted `.txt` extension added.
5. Open this file and you should see something like Figure 1-2.


Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///D:/Users/Jo.../Files/appUsers.xml
+
file:///D:/Users/Joe/Documents/My Dropbox/Beginning XML 5e/Chapters/Chapter1-What is XML/Cu
Google

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
- <applicationUsers>
    <user firstName="Joe" lastName="Fawcett"/>
    <user firstName="Danny" lastName="Ayers"/>
    <user firstName="Catherine" lastName="Middleton"/>
  </applicationUsers>
```

FIGURE 1-2

> ***WARNING*** *If you are using Internet Explorer for this or other activities, you'll probably have to go to Tools ➪ Internet Options and choose the Advanced tab. Under the Security section, check the box in front of Allow Active Content to Run in Files on My Computer. This effectively allows script to work on local files.*

As you can see the XML file is treated very differently. The browser has shown the metadata in a different color than the base data, and also allows expansion and contraction of the `applicationUsers` section. Even though the browser has no idea that this file represents three different users, it knows that some of the content is to be handled differently from other parts and it is a relatively straightforward step to take this to the next level and start to process the file in a sensible fashion.

How It Works

Browsers use an XML stylesheet or transformation to display XML files. An XML stylesheet is a text-based file with an XML format that can transform one format into another. They are most commonly used to convert from a particular XML format to another or from XML to HTML, but they can also be used to process plain text. In this case the original XML is transformed into HTML, which permits the styling of elements to give the different colors as well as the ability to expand and contract sections using script. Transformations are covered in depth in Chapter 8, "XSLT."

> ***NOTE*** *If you want to view the default style sheet that Firefox uses to display XML, type* **`chrome://global/content/xml/XMLPrettyPrint.xsl`** *into the Firefox address bar. IE has a similar built-in style sheet but it's not so easily viewable and it's written in an older, and now no longer used, version of XSLT that Microsoft brought out before the current version was standardized.*

> ***NOTE*** *You'll be using a browser a few times in this chapter to view XML files. This has a number of advantages — they're easy, they give reasonable messages if the XML file has errors, and you'd be unlikely to find a machine that doesn't have one. However, for serious development they are not such a good idea, especially if you are trying to convert XML to HTML as you do in the next Try It Out. Because most browsers allow for poorly formed HTML you won't be able to see if what you've produced has errors, and you certainly won't be able to easily debug if something is wrong. For this reason we suggest you use a proper XML editor when developing. Chapter 2, "Well-Formed XML" covers a number of these.*

MORE ADVANTAGES OF XML

One of the aims of XML is to implement a clear separation between data and presentation. This means that the same underlying data can be used in multiple presentation scenarios. It also means that when moving data, across a network for example, bandwidth is not wasted by having to carry redundant information concerned only with the look and feel. This separation is simple with XML as there are no built-in presentational features such as exist in HTML, and is one of its main advantages.

XML Rules

In order to maintain this clear separation, the rules of XML have to be quite strict, but this also works to the user's advantage. For instance, in the `appUsers.xml` file you saw, values of the users' first and last names were within quotes; this is a prerequisite for XML files; therefore, the following would not be considered XML:

```
<applicationUsers>
  <user firstName=Joe lastName=Fawcett />
  <user firstName=Danny lastName=Ayers />
  <user firstName=Catherine lastName=Middleton />
</applicationUsers>
```

The need for quotes in turn makes it easy to tell when certain data is missing, for example here:

```
<applicationUsers>
  <user lastName="Fawcett" />
  <user lastName="Ayers" />
  <user lastName="Middleton" />
</applicationUsers>
```

None of the users has a first name. Now your application may find that acceptable or it may not, but either way it's easier to tell whether the file is legitimate, or *valid* as it's known in XML, when the data is in quotation marks. This means unsuitable files can be rejected at an early stage without causing application errors. Additional ways of validating XML files are covered in Part 2 of this book.

Another advantage is the easy extensibility of XML files. If you want to add more data, perhaps a middle name for example, to the application users' data, you can do that easily by creating a new attribute, middleName:

```
<applicationUsers>
  <user firstName="Joe" middleName="John" lastName="Fawcett" />
  <user firstName="Danny" middleName="John" lastName="Ayers" />
  <user firstName="Catherine" middleName="Elizabeth" lastName="Middleton" />
</applicationUsers>
```

Consider if you had an application that consumed the original version of the data, with just first name and last name stored in the file, and used it to present a list of application users on its main screen. Originally the software was designed to show just the first name and last name of each user but a new requirement demands that the middle name is displayed as well. The newer version of the XML adds the middleName attribute to satisfy this new requirement. Now the older version of the application can still consume this data and simply ignore the middle name information while the new versions can take advantage of it. This is more difficult to accomplish if the data is in the type of simple text file such as appUsers.txt:

```
Joe John Fawcett
Danny John Ayers
Catherine Elizabeth Middleton
```

If the extra data is added to the middle column, the existing application will probably misinterpret it, and even if the middle name becomes the third column it's likely to cause problems parsing the file. This occurs because there are no delimiters specifying where the individual data items begin and end, whereas with the XML version it's easy to separate the different components of a user's name.

Hierarchical Data Representation

Another area where XML-formatted data flourishes over simple text files is when representing a hierarchy; for instance a filesystem. This scenario needs a root with several folders and files; each folder then may have its own subfolders, which can also contain folders and files. This can go on indefinitely. If all you had was a text file, you could try something like this, which has a column representing the path and one to describe whether it's a folder or a file:

```
Path Type
C:\folder
C:\pagefile.sys file
C:\Program Files folder
C:\Program Files\desktop.ini file
C:\Program Files\Microsoft folder
C:\Program Files\Mozilla folder
C:\Windows folder
C:\Windows\System32 folder
```

```
C:\Temp folder
C:\Temp\~123.tmp file
C:\Temp\~345.tmp file
```

As you can see, this is not pretty and the information is hard for us humans to read and quickly assimilate. It would be quite difficult to write code that interprets this neatly. Comparatively, now look at one possible XML version of the same information:

```
<folder name="C:\">
  <folder name="Program Files">
    <folder name="Microsoft">
    </folder>
    <folder name="Mozilla">
    </folder>
  </folder>
  <folder name="Windows>
    <folder name="System32">
    </folder>
  </folder>
  <folder name="Temp">
    <files>
      <file name="~123.tmp"></file>
      <file name="~345.tmp"></file>
    </files>
  </folder>
  <files>
    <file name="pagefile.sys"></file>
  </files>
</folder>
```

This hierarchy is much easier to appreciate. There's less repetition of data and it would be fairly easy to parse.

Interoperability

The main advantage of XML is interoperability. It is much quicker to agree on or publish an XML format and use that to exchange data between different applications (with the associated metadata included in the file) than to have an arbitrary format that requires accompanying information for processing. Due to the high availability of cheap XML parsers and the pieces of software that read XML and enable interrogation of its data, anyone can now publish the format that their application deals with and others can then either consume it or recreate it. One of the best examples of this comes back to the binary files discussed at the beginning of this chapter. Before Microsoft Word 2003, Word used a binary format for its documents. However, creating an application that could read and create these files was a considerable chore and often led to converters that only partially worked. Since Word 2003, all versions of Word can save documents in an XML format with a documented structure. This has meant the ability to read these documents in other applications (OfficeLibre, for example), as well as the ability to create Word documents using even the most basic tools. It also means that corrupted documents, which would previously have been completely lost, can now often be fixed by opening them in a simple text editor and repairing them. With this and the previously discussed advantages, XML is truly the best choice.

***NOTE** OfficeLibre is an open source application that mimics, to a large extent, other office work applications such as Microsoft Office. It was originally called OpenOffice but split off when OpenOffice was taken over by Oracle. You can obtain it at* `www.libreoffice.org`.

XML IN PRACTICE

Since its first appearance in the mid-'90s the actual XML specification has changed little; the main change being more freedom allowed for content. Some characters that were forbidden from earlier versions are now allowed. However, there have been many changes in how and where XML is used and a proliferation of associated technologies, most with their associated standards. There has also been a massive improvement in the tools available to manage XML in its various guises. This is especially true of the past several years, Five years ago any sort of manipulation of XML data in a browser meant reams of custom JavaScript, and even that often couldn't cope with the limited support in many browsers. Now many well-written script libraries exist that make sending, receiving, and processing XML a relatively simple process, as well as taking care of the gradually diminishing differences between the major makes of browser. Another recent change has been a more overall consensus of when *not* to use XML, although plenty of die-hards still offer it as the solution to every problem. Later chapters cover this scenario, as well as others. This section deals with some of the current uses of XML and also gives a foretaste of what is coming in the chapters ahead.

***NOTE** You can find the latest W3C XML Recommendation at* `www.w3.org/TR/xml`.

***NOTE** JSON stands for JavaScript Object Notation and is discussed more in Chapters 14 and 16 which relate to web services and Ajax. If you need more information in the meantime, head to* `www.json.org`.

Data Versus Document

So far the examples you've seen have concentrated on what are known as data-centric uses of XML. This is where raw data is combined with markup to help give it meaning, make it easier to use, and enable greater interoperability. There is a second major use of XML and markup in general, which is known as *document-centric*. This is where more loosely structured content (for example, a chapter from a book or a legal document) is annotated with metadata. HTML is usually considered to be a document-centric use of SGML (and XHTML, is similarly a document-oriented application of XML) because HTML is generally content that is designed to be read by humans rather than data that will be consumed by a piece of software. XML is designed to be read and understood by both humans and software but, as you will see later, the ways of processing the different styles of XML can vary considerably.

Document-centric XML is generally used to facilitate multiple publishing channels and provide ways of reusing content. This is useful for instances in which regular content changes need to be applied to multiple forms of media at once. For example, a few years ago I worked on a system that produced training materials for the financial sector. A database held a large number of articles, quizzes, and revision aids that could be collated into general training materials. These were all in an XML format very similar to XHTML, the XML version of HTML. Once an editor finalized the content in this database, it was transformed using XSLT (as described in Chapter 8) into media suitable for both the Web and a traditional printed output. When using document-centric XML in this sort of system, whenever content changes, it is only necessary to alter the underlying data for changes to be propagated to all forms of media in use. Additionally, when a different form of the content is needed, to support mobile web browsers for example, a new transformation is the only necessary action.

XML Scenarios

In addition to document-centric situations, XML is also frequently used as a means of representing and storing data. The main reasons for this use are XML's flexible nature and the relative ease with which these files can be read and edited by both humans and machines. This section presents some common, relevant scenarios in which XML is used in one way or another, along with some brief reasons why XML is appropriate for that situation.

Configuration Files

Nearly all modern configuration files use XML. Visual Studio project files and the build scripts used by Ant (a tool used to control the software build process in Java) are both examples of XML configuration files. The main reasons for using XML are that it's so much easier to parse than the traditional name/value pair style and it's easy to represent hierarchies.

Web Services

Both the more long-winded SOAP style and the usually terser RESTful web services use XML, although many now have the option to use JSON as well. XML is used both as a convenient way to serialize objects in a cross-platform manner and as a means of returning results in a universally accepted fashion. SOAP-style services (covered in depth in Chapters 15 and 16) are also described using an XML format called *WSDL*, which stands for *Web Services Description Language*. WSDL provides a complete description about a web service and its capabilities, including the format of the initial request, the ensuing response, and details of exactly how to call the service, its hostname, what port it runs on, and the format of the rest of the URL.

Web Content

Although many believe that XHTML (the XML version of HTML) has not really caught on and will be superseded by HTML 5, it's still used extensively on the Web. There's also a lot of content stored as plain XML, which is transformed either server-side or client-side when needed. The reason for storing it as XML can be content re-use as mentioned earlier, but also it can be a way to save on bandwidth and storage. Content that needs to be shown as an HTML table, for example, nearly always takes up less room as XML combined with code to transform it.

Document Management

In addition to XML being used to store the actual content that will be presented via the Web, XML is also used heavily in document-management systems to store and keep track of documents and manage metadata, usually in conjunction with a traditional relational database system. XML is used to store information such as a document's author, the date of creation, and any modifications. Keeping all this extra information together with the actual content means that everything about a document is in one place, making it easier to extract when needed as well as making sure that metadata isn't orphaned, or separated from the data it's describing.

Database Systems

Most modern high-end database systems, such as Oracle and SQL Server, can store XML documents. This is good news because many types of data don't fit nicely into the relational structure (tables and joins) that traditional databases implement. For example, a table of products may need to store some instructions that are in an XML format that will be turned into a web page or a printed manual when needed. This can't be reduced to a simpler form and only needs modifying very rarely, perhaps to insert a new section to support a different language. These modifications are easy and straightforward if the data being manipulated is stored in a database system that has a column designed specifically for XML. This XML should enable updates using the XQuery language, which is briefly covered later in this chapter. Both Oracle and SQL Server, as well as some open source applications such as MySQL, provide such a column type, designed specifically to store XML. These types have methods associated with them that allow for the extraction of particular sections of the XML or for its modification.

Image Representation

Vector images can be represented with XML, the SVG format being the most popular. The advantage of using an XML format over a traditional bitmap when portraying images is that the images can be manipulated far more easily. Scaling and other changes become transformations of the XML rather than complex intensive calculations.

Business Interoperability

Hundreds of industries now have standard XML formats to describe the different entities that are used in day-to-day transactions, which is one of the biggest uses of XML. A brief list includes:

- Medical data
- Financial transactions such as purchasing stocks and shares and exchanging currency

- Commercial and residential properties
- Legal and court records
- Mathematical and scientific formulas

XML Technologies

To enable the preceding scenarios you can use a number of associated technologies, standards, and patterns. The main ones, which are all covered throughout the book, are introduced here to give a broad overview of the world of XML.

XML Parsers

Before any work can be done with an XML document it needs to be parsed; that is, broken down into its constituent parts with some sort of internal model built up. Although XML files are simply text, it is not usually a good idea to extract information using traditional methods of string manipulation such as `Substring`, `Length`, and various uses of regular expressions. Because XML is so rich and flexible, for all but the most trivial processing, code using basic string manipulation will be unreliable.

Instead a number of XML parsers are available — some free, some as commercial products— that facilitate the breakdown and yield more reliable results. You will be using a variety of these parsers throughout this book. One of the reasons to justify using a handmade parser in the early days of XML was that pre-built ones were overkill for the job and had too large a footprint, both in actual size and in the amount of memory they used. Nowadays some very efficient and lightweight parsers are available; these mean developing your own is a waste of resources and not a task to be undertaken lightly.

Some of the more common parsers used today include the following:

- **MSXML (Microsoft Core XML Services):** This is Microsoft's standard set of XML tools including a parser. It is exposed as a number of COM objects so it can be accessed using older forms of Visual Basic (6 and below) as well as from C++ and script. The latest version is 6.0 and, as of this writing it is not being developed further, although service packs are still being released that address bugs and any other security issues. Although you probably wouldn't use this parser when writing your own application from scratch, this is the only option when you need to parse XML from within older versions of Internet Explorer (6 and below). In these browsers the MSXML parser is invoked using ActiveX technology, which can present problems in some secure environments. Fortunately versions 7 and later have a built-in parser and cross-browser libraries. Choose this one in preference if it's available.
- **System.Xml.XmlDocument:** This class is part of Microsoft's .NET library, which contains a number of different classes related to working with XML. It has all the standard *Document Object Model* (DOM, covered in the next section) features plus a few extra ones that, in theory, make life easier when reading, writing, and processing XML. However, since the world is trending away from using the DOM, Microsoft also has a number of other ways of tackling XML, which are discussed in later chapters.

- **Saxon:** Ask any group of XML cognoscenti what the leading XML product is and Saxon will likely be the majority verdict. Saxon's offerings contain tools for parsing, transforming, and querying XML, and it comes from the software house of Dr. Michael Kay, who has written a number of Wrox books on XML and related technologies. Although Saxon offers ways to interact using the document object model, it also has a number of more modern and user-friendly interfaces available. Saxon offers a version for Java and .NET; the basic edition is free to download and use.
- **Java built-in parser:** The Java library has its own parser. It has a reputation for being a bit basic but is suitable for many XML tasks such as parsing and validation of a document. The library is designed such that you can replace the built-in parser with an external implementation such as Xerces from Apache or Saxon.
- **Xerces:** Xerces is implemented in Java and is developed by the famous and open source Apache Software Foundation. It is used as the basis for many Java-based XML applications and is a more popular choice than the parser that comes with Java.

The Document Object Model

Once an XML parser has done its work, it produces an in-memory representation of the XML. This model exposes properties and methods that let you extract information from and also modify the XML. For example, you'll find methods such as `createElement` to manufacture new elements in the document and properties such as `documentElement` that bring back the root element in the document (`applicationUsers` in the example file).

One of the earliest models used was the *Document Object Model* (*DOM*). This model has an associated standard but it doesn't just apply to XML; it also works with HTML documents. At its heart, the DOM is a tree-like representation of an XML document. You can start at the tree's root and move to its different branches, extracting or inserting data as you go. Although the DOM was used for many years, it has a reputation for being a bit unwieldy and difficult to use. It also tends to take up a lot of memory. For example, opening an XML document that is 1MB on a disk can use about 5MB of RAM. This can obviously be a problem if you want to open very large documents. As a result of these problems, a number of other models have sprung up, especially because the DOM is typically only an intermediate step in processing XML; it's not a goal in itself. However, if you need to extract just a few pieces of information from XML or HTML the DOM is widely supported, especially across browsers, and is used a lot by many of the script libraries that are popular nowadays such as jQuery.

DTDs and XML Schemas

Both *document type definitions* (*DTDs*) and *XML Schemas* serve to describe the definition of an XML document, its structure, and what data is allowed where. They can then be used to test whether a document that has been received is consistent with the prescribed format, a process known as validation. DTDs are the older standard and have been around since SGML. They are gradually succumbing to XML Schemas but are still in widespread use particularly with (X)HTML. They also have a few features that XML lacks, such as the ability to create entity declarations (covered in Chapter 4, "Document Type Definitions") and the ability to add default attribute content.

In general, XML Schemas offer more functionality; they also have the advantage of being written in XML so the same tools can be used with both the data and its schema. DTDs on the other hand use a completely different format that is much harder to work with. In addition to assisting with validation, DTDs and XML Schema are also used to help authorship of XML documents. Most modern XML editors allow you to create an XML document based on a specified schema. They prompt you with valid choices from the schema as you're editing and also warn you if you've used an element or attribute in the wrong location. Although many have misgivings about how XML Schemas have developed it's probably true to say that most recently developed XML formats are described using schemas rather than DTDs.

There are also other ways of ensuring the documents you receive are in the correct format, ones that can cope with some scenarios that neither DTDs nor XML Schemas can handle. A selection of these alternatives are covered in Chapter 6, "RELAX NG and Schematron." DTDs and XML Schemas are covered in depth in Chapters 4 and 5, respectively.

NOTE *If you take a look at the source for an XHTML document you'll see the reference to the DTD at the top of the page. It will look something like this:*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

XML Namespaces

XML Namespaces were added to the XML specification sometime after the initial recommendation. They have a reputation for being difficult to understand and also for being poorly implemented. Basically, namespaces serve as a way of grouping XML. For instance, if one or two different formats need to be used together, he element names can be grouped under a namespace; this ensures that there is no confusion about what the elements represent, especially if the authors of the different formats have chosen the same names for some of the elements. The same idea is used in software all the time; in both .NET and Java, for example, you may design a class that represents a type of XML document that you call `XmlDocument`. To prevent that class from conflicting with other classes that might exist with the same name, the class is placed in a namespace. (NET terminology) or a package (Java terminology). So your class may have a full name of `Wrox.Entities.XmlDocument`, which will differentiate it from Microsoft's `System.Xml.XmlDocument`. See Chapter 3 for the full story on namespaces.

XPath

XPath is used in many XML technologies. It enables you to target specific elements or attributes (or the other building blocks you'll meet in the next chapter). It works similar to how paths in a filesystem work, starting at the root and progressing through the various layers until the target is found. For example, with the `appUsers.xml` file, you may want to select all the users. The XPath for this would be:

```
/applicationUsers/user
```

The path starts at the root, represented by a forward slash (/), then selects the applicationUsers element, and then any user elements beneath there. XPaths can be very sophisticated and allow you to traverse the document in a number of different directions as well as target specific parts using predicates, which enable filtering of the results. In addition to being used in XSLT, XPath is also used in XQuery, XML Schemas, and many other XML-related technologies. XPath is dealt with in more detail in Chapter 7, "Extracting Data From XML."

XSLT

One of the main places you find XPath is XSLT. *Extensible Stylesheet Language Transformations (XSLT)* is powerful way to transform files from one format to another. Originally it could only operate on XML files, although the output could be any form of text file. Since version 2.0 however, it also has the capability to use any text file as an input. XSLT is a declarative language and uses templates to define the output that should result from processing different parts of the source files.

XSLT is often used to transform XML to (X)HTML, either server-side or in the browser. The advantages of doing a client-side transformation are that it offloads the presentational side of the process to the application layer that deals with the display. Additionally it frees resources on the server making it more responsive, and it tends to reduce the amount of data transmitted between the server and the browser. This is especially the case when the data consists of many rows of similar data that are to be shown in tabular form. HTML tables are very verbose and can easily double or triple the amount of bandwidth between client and server.

The following Try It Out shows how browsers have been specially designed to be able to accept an XML as an input and transform the data using a specified transformation. You won't be delving too deeply into the XSLT at this stage, (that's left for Chapter 8) but you'll get a good idea of how XML enables you to separate the intrinsic data being shown from the visual side of the presentation.

TRY IT OUT XSLT in the Browser

Use the appUsers.xml file created earlier to produce a demonstration of how a basic transformation can be achieved within a browser:

1. To start, create the following file using any text editor and save it as appUsers.xslt in the same folder as appUsers.xml:

Available for download on Wrox.com

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
      <html>
        <head>
          <title>Application Users</title>
        </head>
        <body>
          <table>
            <thead>
              <tr>
                <th>First Name</th>
                <th>Last Name</th>
              </tr>
            </thead>
            <tbody>
              <xsl:apply-templates select="applicationUsers/user" />
            </tbody>
          </table>
        </body>
      </html>
    </xsl:template>

  <xsl:template match="user">
    <tr>
      <td>
        <xsl:value-of select="@firstName"/>
      </td>
      <td>
        <xsl:value-of select="@lastName"/>
      </td>
    </tr>
  </xsl:template>
</xsl:stylesheet>
```

code snippet appUsers.xslt

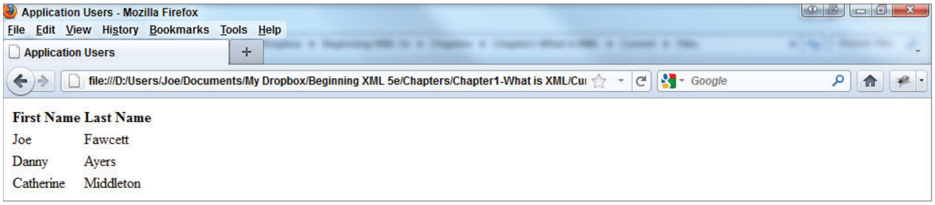
2. Next make a small change to appUsers.xml so that, if it is opened in a browser, the browser will know to use the specified XSLT to transform the XML, rather than the built-in default transformation that was used in earlier examples. Save the modified file as appUsersWithXslt.xml.

Available for download on Wrox.com

```
<?xml-stylesheet type="text/xsl" href="appUsers.xslt" ?>
<applicationUsers>
  <user firstName="Joe" lastName="Fawcett" />
  <user firstName="Danny" lastName="Ayers" />
  <user firstName="Catherine" lastName="Middleton" />
</applicationUsers>
```

code snippet appUsersWithXslt.xml

3. Finally, open appUsersWithXslt.xml in a browser. The results will be similar to Figure 1-3.


Application Users - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Application Users
+
file:///D:/Users/Joe/Documents/My Dropbox/Beginning XML 5e/Chapters/Chapter1-What is XML/Cu
Google

First Name	Last Name
Joe	Fawcett
Danny	Ayers
Catherine	Middleton

FIGURE 1-3

How It Works

When the browser sees the following line at the top of the XML:

```
<?xml-stylesheet type="text/xsl" href="appUsers.xslt" ?>
```

It knows that, instead of the default style sheet that produced the result shown in Figure 1-2, it should use `appUsers.xslt`.

`appUsers.xslt` has two `xsl:templates`. The first causes the basic structure of an HTML file to appear along with the outline of an HTML table. The second template acts on any `user` element that appears in the file and produces one row of data for each that is found. Once the transformation is complete the resultant code is treated as if it were a traditional HTML page. The actual code produced by the transformation is shown here:

```
<html>
  <head>
    <title>Application Users</title>
  </head>
  <body>
    <table>
      <thead>
        <tr>
          <th>First Name</th>
          <th>Last Name</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>Joe</td>
          <td>Fawcett</td>
        </tr>
        <tr>
          <td>Danny</td>
          <td>Ayers</td>
        </tr>
        <tr>
          <td>Catherine</td>
          <td>Middleton</td>
        </tr>
      </tbody>
    </table>
  </body>
</html>
```

XQuery

XQuery shares many features with XSLT and because of this, a common question on the XML development forums is, “Is this a job for XSLT or XQuery?” The answer is, “It depends.” Like XSLT, XQuery can operate against single documents, but it is also often used on large collections, especially those that are stored in a relational database. Say you want to use XQuery to process the appUsers.xml file from the previous examples and again produce an HTML page showing the users in a tabular form. The XQuery needed would look like this:

```
<html>
  <head>
    <title>Application Users</title>
  </head>
  <body>
    <table>
      <thead>
        <tr>
          <th>First Name</th>
          <th>Last Name</th>
        </tr>
      </thead>
      <tbody>
    {for $user in doc("appUsers.xml")/applicationUsers/user
    return <tr><td>{data($user/@firstName)}</td>
    <td>{data($user/@lastName)}</td></tr>}
      </tbody>
    </table>
  </body>
</html>
```

As you can see, a lot of the query mimics the XSLT used earlier. One major difference is that XQuery isn't itself an XML format. This means that it's less verbose to write, making it somewhat simpler to author than XSLT. On the other hand, being as it's not XML, it cannot be authored in standard XML editors nor processed by an XML parser, meaning it needs a specialized editor to write and custom built software to process.

NOTE *There is an XML-based version of XQuery called XQueryX. It has never gained much acceptance and nearly all examples of XQuery online use the simpler non-XML format.*

With regards to authoring XQuery, the main difference in syntax between it and XSLT is that XQuery uses braces ({}) to mark parts of the document that need processing by the engine; the rest of the document is simply output verbatim.

Therefore, in the example the actual code part is this section:

```
{for $user in doc("appUsers.xml")/applicationUsers/user
    return <tr><td>{data($user/@firstName)}</td>
    <td>{data($user/@lastName)}</td></tr>}
```

this uses the doc() function to read an external file, in this case the appUsers.xml file, and then creates one <tr> element for each user element found there. XQuery is covered in depth in Chapter 9.

There are many instances where the choice of XSLT or XQuery is simply a matter of which technology you're happier with. If you want a terser, more readable syntax or you need to process large numbers of documents, particularly those found in databases, then XQuery, with its plain text syntax and functions aimed at document collections, is probably a better choice. If you prefer an XML style syntax that can be easily read by standard XML software, or your goal is to rearrange existing XML into a different format rather than create a whole new structure, then XSLT will most likely be the better option.

XML Pipelines

XML pipelines are used when single atomic steps are insufficient to achieve the output you desire. For example, it may not be possible to design an XML transformation that copes with all the different types of documents your application accepts. You may need to perform a preliminary transform first, depending on the input, and follow with a generalized transformation. Another example might be that the initial input needs validating before being transformed. In the past, these pipelines or workflows have been created in a rather ad hoc manner. More recently, there have been calls for a recognized standard to define how pipelines are described. The W3C recommendation for these standards is called *XProc* and you can find the relevant documentation at `www.w3.org/TR/xproc`. Only a handful of implementations exist at the moment, but if you have the need for this type of workflow it's certainly worth taking a look at XProc rather than re-inventing the wheel.

SUMMARY

- ➤ The situation before XML and the problems with binary and plain text files
- ➤ How XML developed from SGML
- ➤ The basic building blocks of XML: elements and attributes
- ➤ Some of the advantages and disadvantages of XML
- ➤ The difference between data-centric and document-centric XML
- ➤ Some real-world uses of XML
- ➤ The associated technologies such as parsers, schemas, XPath, transformations with XSLT, and XQuery

The next chapter discusses the rules for constructing XML and what different constituent parts can make up a document.

EXERCISES

Answers to the exercises can be found in Appendix A.

1. Change the format of the `appUsers.xml` document to remove the attributes and use elements to store the data.

2. State the main disadvantage to having the file in the format you've just created. Bear in mind that data is often transmitted across networks rather than just being consumed where it is stored.

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY POINTS
Before XML	Most data formats were proprietary, capable of being read by a very small number of applications and not suitable for today's distributed systems.
XML's Goals	To make data more interchangeable, to use formats readable by both humans and machines, and to relieve developers from having to write low-level code every time they needed to read or write data.
Who's In Charge of Standardization?	No one, but many XML specifications are curated by the World Wide Web Consortium, the W3C. These documents are created after a lengthy process of design by committee followed by requests for comments from stakeholders.
Data-centric versus Document-centric	There are two main types of XML formats: those used to store pure data, such as configuration settings, and those used to add metadata to documents, for example XHTML.
What Technologies Rely On XML?	There are hundreds, but the main ones are XML Schemas, to validate that documents are in the correct format; XSLT which is mainly used to convert from one XML format to another; XQuery, which is used to query large document collections such as those held in databases; and SOAP which uses XML to represent the data that is passed to, and returned from, a web service.