# 1

# What Is XML?

*XML (Extensible Markup Language)* is a buzzword you will see everywhere on the Internet, but it's also a rapidly maturing technology with powerful real-world applications, particularly for the management, display, and organization of data. Together with its many related technologies, which are covered in later chapters, XML is an essential technology for anyone working with data, whether publicly on the web or privately within your own organization. This chapter introduces you to some XML basics and begins to show you why learning about it is so important.

This chapter covers the following:

- The two major categories of computer file types — binary files and text files — and the advantages and disadvantages of each
- The history behind XML, including other markup languages such as SGML and HTML
- How XML documents are structured as hierarchies of information
- A brief introduction to some of the other technologies surrounding XML, which you will work with throughout the book
- A quick look at some areas where XML is useful

*While there are some short examples of XML in this chapter, you aren't expected to understand what'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.*



## Of Data, Files, and Text

XML is a technology concerned with the description and structuring of *data*, so before you can really delve into the concepts behind XML, you need to understand how computers store and access data. For our purposes, computers understand two kinds of data files: binary files and text files.

## 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 a 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 instance, when a document is created with Microsoft Word, the program creates a binary file with an extension of "doc," in its own proprietary format. The programmers who wrote Word decided to insert certain binary codes into the document to denote bold text, codes to denote page breaks, and other codes for all of the information that needs to go into a "doc" file. When you open a document in Word, it interprets those codes and displays the properly formatted text or prints it to the printer.

The codes inserted into the document are *meta data*, or information about information. Examples could be "this word should be in bold," "that paragraph should be centered," and so on. This meta data is really what differentiates one file type from another; the different types of files use different kinds of meta data. For example, a word processing document has different meta data than a spreadsheet document, because they are describing different things. Not so obviously, documents from different word processing applications, such as Microsoft Word and WordPerfect, also have different meta data, because the applications were written differently (see Figure 1-1).

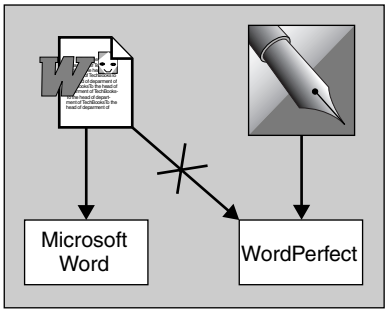


**Figure 1-1**

You can't assume that a document created with one word processor will be readable by another, because the companies who write word processors all have their own proprietary formats for their data files. Word documents open in Microsoft Word, and WordPerfect documents open in WordPerfect.

Luckily, most word processors come with translators or import utilities, which can translate documents from other word processors into formats that can be understood natively. If I have Microsoft Word installed on my computer and someone gives me a WordPerfect document, I might be able to import it into Word so that I can read the document. Of course, many of us have seen the garbage that sometimes occurs as a result of this translation; sometimes applications are not as good as we'd like them to be at converting the information.

Binary file formats are advantageous because it is easy for computers to understand these binary codes—meaning that they can be processed much faster than nonbinary formats—and they are very efficient for storing this meta data. There is also a disadvantage, as you've seen, in that binary files are proprietary. You might not be able to open binary files created by one application in another application, or even in the same application running on another platform.

## Text Files

Like binary files, *text files* are also streams of bits. However, in a text file these bits are grouped together in standardized ways, so that they always form numbers. These numbers are then further mapped to characters. For example, a text file might contain the following bits:

```
1100001
```

This group of bits would be translated as the number `97`, which could then be further translated into the letter `a`.

> *This example makes a number of assumptions. A better description of how numbers are represented in text files is given in the "Encoding" section in Chapter 2.*

Because of these standards, text files can be read by many applications, and can even be read by humans, using a simple text editor. If I create a text document, anyone in the world can read it (as long as they understand English, of course) in any text editor they wish. Some issues still exist, such as the fact that different operating systems treat line-ending characters differently, but it is much easier to share information when it's contained in a text file than when the information is in a binary format.

Figure 1-2 shows some of the applications on my machine that are capable of opening text files. Some of these programs only allow me to *view* the text, while others will let me *edit* it as well.

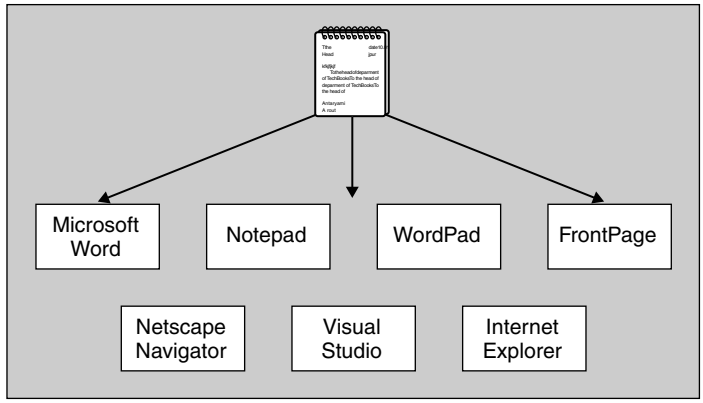


**Figure 1-2**

In its early days, the Internet was almost completely text-based, which enabled people to communicate with relative ease. This contributed to the explosive rate at which the Internet was adopted, and to the ubiquity of applications such as e-mail, the World Wide Web, newsgroups, and so on.

The disadvantage of text files is that adding other information — our meta data, in other words — is more difficult and bulky. For example, most word processors enable you to save documents in text form, but if you do, you can't mark a section of text as bold or insert a binary picture file. You will simply get the words with none of the formatting.

## A Brief History of Markup

You can see that there are advantages to binary file formats (easy to understand by a computer, compact, the ability to add meta data), as well as advantages to text files (universally interchangeable). Wouldn't it be ideal if there were a format that combined the universality of text files with the efficiency and rich information storage capabilities of binary files?

This idea of a universal data format is not new. In fact, for as long as computers have been around, programmers have been trying to find ways to exchange information between different computer programs. An early attempt to combine a universally interchangeable data format with rich information storage capabilities was *Standard Generalized Markup Language (SGML)*. SGML is a text-based language that can be used to mark up data — that is, add meta data — in a way that is *self-describing*. (You'll see in a moment what self-describing means.)

SGML was designed to be a standard way of marking up data for any purpose, and took off mostly in large document management systems. When it comes to huge amounts of complex data, a lot of considerations must be taken into account, so SGML is a very complicated language. However, with that complexity comes power.

A very well-known language based on the SGML work is the *HyperText Markup Language (HTML)*. HTML uses many of SGML's concepts to provide a universal markup language for the display of information, and the linking of different pieces of information. The idea was that any HTML document (or web page) would be presentable in any application that was capable of understanding HTML (termed a *web browser*). A number of examples are given in Figure 1-3.

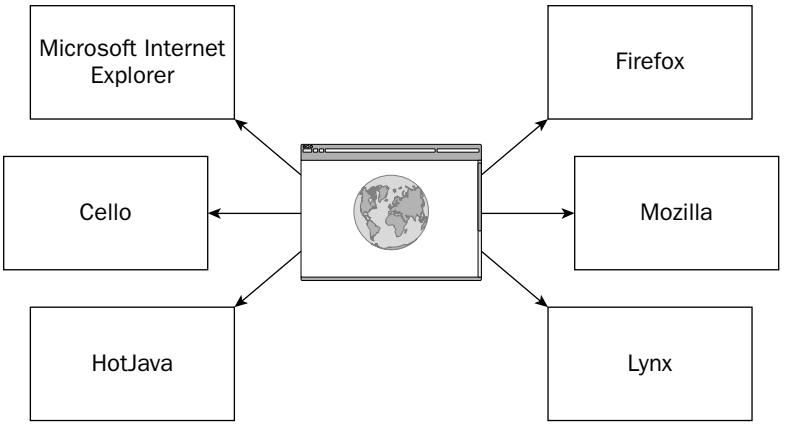


Figure 1-3

Not only would that browser be able to display the document, but if the page contained links (termed *hyperlinks*) to other documents, the browser would also be able to seamlessly retrieve them as well.

Furthermore, because HTML is text-based, anyone can create an HTML page using a simple text editor, or any number of web page editors, some of which are shown in Figure 1-4.

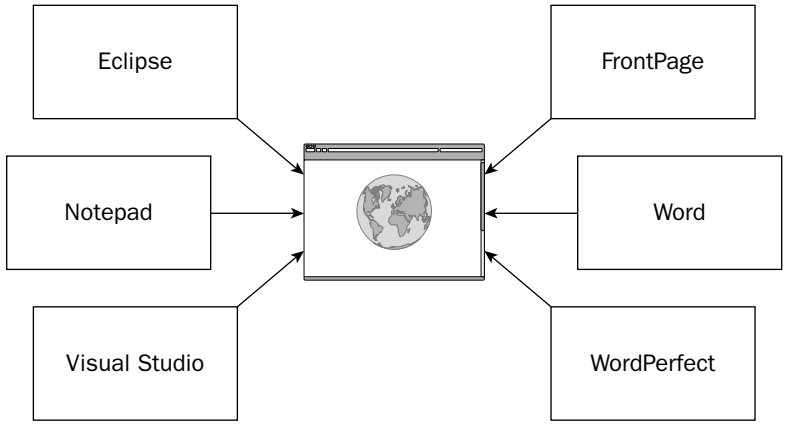


Figure 1-4

Even many word processors, such as WordPerfect and Word, allow you to save documents as HTML. Think about the ramifications of Figures 1-3 and 1-4: Any HTML editor, including a simple text editor, can create an HTML file, and that HTML file can then be viewed in any web browser on the Internet!

# So What Is XML?

Unfortunately, SGML is such a complicated language that it's not well suited for data interchange over the web. In addition, although HTML has been incredibly successful, it's limited in scope: It is only intended for displaying documents in a browser. The tags it makes available do not provide any information about the content they encompass, only instructions about how to display that content. This means that you could create an HTML document that displays information about a person, but that's about all you could do with the document. You couldn't write a program to figure out from that document which piece of information relates to the person's first name, for example, because HTML doesn't have any facilities to describe this kind of specialized information. In fact, HTML wouldn't even know that the document was about a person at all. Extensible Markup Language (XML) was created to address these issues.

> *Note that despite the acronym, it's spelled "Extensible," not "eXtensible." Mixing these up is a common mistake.*

XML is a subset of SGML, with the same goals (markup of any type of data), but with as much of the complexity eliminated as possible. XML was designed to be fully compatible with SGML, meaning any document that follows XML's syntax rules is by definition also following SGML's syntax rules, and can therefore be read by existing SGML tools. It doesn't go both ways, however, so an SGML document is not necessarily an XML document.

It is important to realize that XML is not really a "language" at all, but a standard for creating languages that meet the XML criteria (we go into these rules for creating XML documents in Chapter 2). In other words, XML describes a syntax that you use to create your own languages. For example, suppose you have data about a name, and you want to be able to share that information with others as well as use that information in a computer program. Instead of just creating a text file like this:

```
John Doe
```

or an HTML file like this

```
<html>
<head><title>Name</title></head>
<body>
<p>John Doe</p>
</body>
</html>
```

you might create an XML file like the following:

```
<name>
  <first>John</first>
  <last>Doe</last>
</name>
```

Even from this simple example, you can see why markup languages such as SGML and XML are called "self-describing." Looking at the data, you can easily tell that this is information about a `<name>`, and you can see that there is data called `<first>` and more data called `<last>`. You can give the tags any names you like, but if you're going to use XML, you might as well use it right and give things *meaningful* names.

You can also see that the XML version of this information is much larger than the plain-text version. Using XML to mark up data adds to its size, sometimes enormously, but achieving small file sizes isn't one of the goals of XML; it's only about making it easier to write software that accesses the information, by giving structure to the data.

> **This larger file size should not deter you from using XML. The advantages of easier-to-write code far outweigh the disadvantages of larger bandwidth issues.**

If bandwidth is a critical issue for your applications, you can always compress your XML documents before sending them across the network — compressing text files yields very good results.

If you're running Internet Explorer 5 or later, you can view the preceding XML in your browser, as shown in the following Try It Out. (You can also use other web browsers, such as Firefox, to display the XML examples in this chapter. All of the screenshots shown, however, are of Internet Explorer 6.)

## Try It Out — Opening an XML File in Internet Explorer

1. Open Notepad and type in the following XML:

```
<name>
  <first>John</first>
  <last>Doe</last>
</name>
```

2. Save the document to your hard drive as `name.xml`. If you're using Windows XP, be sure to change the Save as Type drop-down option to All Files. (Otherwise, Notepad will save the document with a `.txt` extension, causing your file to be named `name.xml.txt`.) You might also want to change the Encoding drop-down to Unicode, as shown in Figure 1-5. (Find more information on encodings in Chapter 2.)

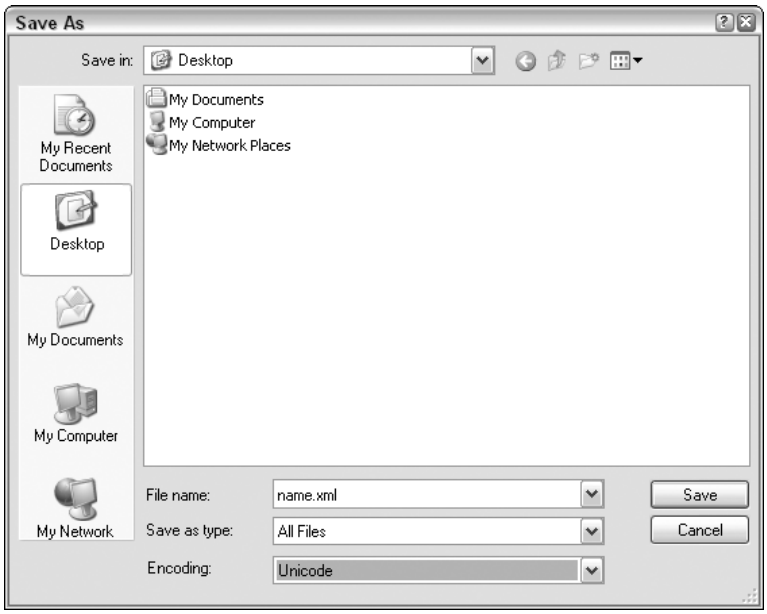


Figure 1-5

3. You can then open the file in Internet Explorer (for example, by double-clicking on the file in Windows Explorer), where it will look something like Figure 1-6.


Figure 1-6

### How It Works

Although your XML file has no information concerning display, the browser formats it nicely for you, with your information in bold and your markup displayed in different colors. In addition, `<name>` is collapsible, like your file folders in Windows Explorer. Try clicking on the minus sign (–) next to `<name>` in the browser window. It should then look like Figure 1-7.

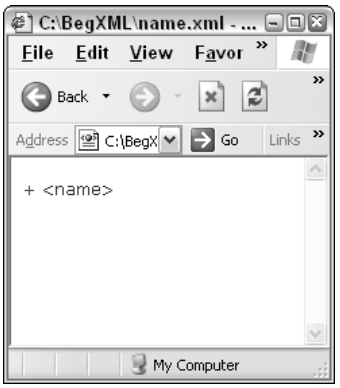


Figure 1-7

For large XML documents, where you only need to concentrate on a smaller subset of the data, this feature can be quite handy. This is one reason why Internet Explorer can be so helpful when authoring XML: It has a default *stylesheet* built in, which applies this default formatting to any XML document.

> *XML styling is accomplished through another document dedicated to the task, called a* stylesheet*. In a stylesheet, the designer specifies rules that determine the presentation of the data. The same stylesheet can then be used with multiple documents to create a similar appearance among them. A variety of languages can be used to create stylesheets. Chapter 8 explains a transformation stylesheet language called Extensible Stylesheet Language Transformations (XSLT), and Chapter 17 looks at a stylesheet language called Cascading Style Sheets (CSS).*

As you'll see in later chapters, you can also create your own stylesheets for displaying XML documents. This way, the same data that your applications use can also be viewed in a browser. In effect, by combining XML data with stylesheets, you can separate your data from your presentation. That makes it easier to use the data for multiple purposes (as opposed to HTML, which doesn't provide any separation of data from presentation — in HTML, *everything* is presentation).

## What Does XML Buy Us?

I can hear what some of you are thinking. Why go to the trouble of creating an XML document? Wouldn't it be easier to just make up some rules for a file about names, such as "The first name starts at the beginning of the file, and the last name comes after the first space?" That way, your application could still read the data, but the file size would be much smaller.

As a partial answer, suppose that we want to add a middle name to our example:

```
John Fitzgerald Doe
```

Okay, no problem. We'll just modify our rules to say that everything after the first space and up to the second space is the middle name, and everything after the second space is the last name. However, if there is no second space, we have to assume that there is no middle name, and the first rule still applies. We're still fine, unless a person happens to have a name like the following:

```
John Fitzgerald Johansen Doe
```

Whoops! There are two middle names in there. The rules get more complex. While a human might be able to tell immediately that the two middle words compose the middle name, it is more difficult to program this logic into a computer program. We won't even discuss "John Fitzgerald Johansen Doe the 3rd"!

Unfortunately, when it comes to problems like this, many software developers simply define more restrictive rules, instead of dealing with the complexities of the data. In this example, a software developer might decide that a person can only have *one* middle name, and the application won't accept anything more than that.

> *This is pretty realistic, I might add. My full name is David John Bartlett Hunter, but because of the way in which many computer systems are set up, a lot of the bills I receive are simply addressed to David John Hunter or David J. Hunter. Maybe I can find some legal ground to stop paying my bills, but in the meantime, my vanity takes a blow every time I open my mail.*

This example is probably not all that hard to solve, but it highlights one of the major focuses behind XML. Programmers have been structuring their data in an infinite variety of ways, and every new way of structuring data brings a new methodology for pulling out the information we need. With those new methodologies comes a lot of experimentation and testing to get it just right. If the data changes, the methodologies also have to change, and testing and tweaking has to begin again. XML offers a standardized way to get the information we need, no matter how we structure it.

In addition, remember how trivial this example is. The more complex the data you have to work with, the more complex the logic you'll need to do that work. You'll appreciate XML the most in larger applications.

## XML Parsers

If we just follow the rules specified by XML, we can be sure that getting at our information will be easy. This is because there are programs called *parsers* that can read XML syntax and extract the information for us. We can use these parsers within our own programs, meaning our applications will never have to look at the XML directly; a large part of the workload will be done for us.

> *Parsers are also available for parsing SGML documents, but they are much more complex than XML parsers. Because XML is a subset of SGML, it's easier to write an XML parser than an SGML parser.*

In the past, before these parsers were around, a lot of work would have gone into the many rules we were looking at (such as the rule that the middle name starts after the first space, and so on), but with our data in XML format, we can just give an XML parser a file like this:

```
<name>
  <first>John</first>
  <middle>Fitzgerald Johansen</middle>
  <last>Doe</last>
</name>
```

The parser can tell us that there is a piece of data called `<middle>`, and that the information stored there is `Fitzgerald Johansen`. The parser writer didn't have to know any rules about where the first name ends and where the middle name begins, because the parser simply uses the `<middle>` and `</middle>` tags to determine where the data begins and ends. The parser didn't have to know anything about my application at all, nor about the types of XML documents the application works with. The same parser could be used in my application, or in a completely different application. The language my XML is written in doesn't matter to the parser either; XML written in English, Chinese, Hebrew, or any other language could all be read by the same parser, even if the person who wrote it didn't understand any of these languages.

> **Just as any HTML document can be displayed by any web browser, any XML document can be read by any XML parser, regardless of what application was used to create it, or even what platform it was created on. This goes a long way toward making your data universally accessible.**

There's another added benefit here: If I had previously written a program to deal with the first XML format, which had only a first and last name, that application could also accept the new XML format, without me having to change the code. Because the parser takes care of the work of getting data out of the document for us, you can add to your XML format without breaking existing code, and new applications can take advantage of the new information if they wish. If we were using our previous text-only format, any time we changed the data at all, every application using that data would have to be modified, retested, and redeployed.

As long as an existing application were simply looking for information called "first" and information called "last," it would continue to work, even if we added to the document. Of course, if we *subtracted* information from our `<name>` example, or changed the names we used for the data, we would still have to modify our applications to deal with the changes.

Because it's so flexible, XML is targeted to be the basis for defining data exchange languages, especially for communication over the Internet. The language facilitates working with data within applications, such as an application that needs to access the previously listed `<name>` information, but it also facilitates sharing information with others. We can pass our `<name>` information around the Internet and, even without our particular program, the data can still be read. People can pull the file up in a regular text editor and look at the raw XML if they like, or open it in a viewer such as Internet Explorer.

## Why "Extensible?"

Because we have full control over the creation of our XML document, we can shape the data in any way we wish, so that it makes sense for our particular application. If we don't need the flexibility of our `<name>` example, and don't need to know which part of the "name" is the "first name," and which is the "last name," we could decide to describe a person's name in XML like this:

```
<designation>John Fitzgerald Johansen Doe</designation>
```

If we want to create data in a way that only one particular computer program will ever use, we can do so; and if we decide that we want to share our data with other programs, or even other companies across the Internet, XML gives us the flexibility to do that as well. We are free to structure the same data in different ways that suit the requirements of an application or category of applications.

*This is where the* extensible *in Extensible Markup Language comes from: Anyone is free to mark up data in any way using the language, even if others are doing it in completely different ways.*

HTML, on the other hand, is not extensible, because you can't add to the language; you have to use the tags that are part of the HTML specification. For example, web browsers can understand the following:

```
<p>This is a paragraph.</p>
```

The `<p>` tag is a predefined HTML tag. However, web browsers can't understand the following:

```
<paragraph>This is a paragraph.</paragraph>
```

The `<paragraph>` tag is not a predefined HTML tag.

The benefits of XML become even more apparent when people use the same format to do common things, because this allows us to interchange information much more easily. There have already been numerous projects to produce industry-standard vocabularies to describe various types of data. For example, *Scalable Vector Graphics (SVG)* is an XML vocabulary for describing two-dimensional graphics (we'll look at SVG in Chapter 19); *MathML* is an XML vocabulary for describing mathematics as a basis for machine-to-machine communication; *Chemical Markup Language (CML)* is an XML vocabulary for the management of chemical information. The list goes on and on. Of course, you could write your own XML vocabularies to describe this type of information if you so wished, but if you use a common format, there is a better chance that you will be able to produce software that is immediately compatible with other software. Better yet, you can reuse code already written to work with these formats.

Because XML is so easy to read and write in your programs, it is also easy to convert between different vocabularies when required. For example, if you want to represent mathematical equations in your particular application in a certain way, but MathML doesn't quite suit your needs, you can create your own vocabulary. If you want to export your data for use by other applications, you might convert the data in your vocabulary to MathML for the other applications to read. In fact, Chapter 8 covers a technology called *XSLT*, which was created for transforming XML documents from one format to another, and which could potentially make these kinds of transformations very simple.

## HTML and XML: Apples and Red Delicious Apples

What HTML does for display, XML is designed to do for data exchange. Sometimes XML isn't up to a certain task, just as HTML is sometimes not up to the task of displaying certain information. How many of us have Adobe Acrobat readers installed on our machines for those documents on the web that HTML just can't display properly? When it comes to display, HTML does a good job most of the time, and those who work with XML believe that, most of the time, XML will do a good job of communicating information. Just as HTML authors sometimes sacrifice precise layout and presentation for the sake of making their information accessible to all web browsers, XML developers sacrifice the small file sizes of proprietary formats for the flexibility of universal data access.

Of course, a fundamental difference exists between HTML and XML: HTML is designed for a *specific* application, to convey information to humans (usually visually, through a web browser), whereas XML has no specific application; it is designed for whatever use you need it for.

This is an important concept. Because HTML has its specific application, it also has a finite set of specific markup constructs (`<p>`, `<ul>`, `<h2>`, and so on), which are used to create a correct HTML document. In theory, we can be confident that any web browser will understand an HTML document because all it has to do is understand this finite set of tags. In practice, of course, I'm sure you've come across web pages that displayed properly in one web browser and not in another, but this is usually a result of nonstandard HTML tags, which were created by browser vendors instead of being part of the HTML specification itself.

On the other hand, if you create an XML document, you can be sure that any XML parser will be able to retrieve information from that document, even though you can't guarantee that any application will be able to understand *what that information means*. That is, just because a parser can tell you that there is a piece of data called `<middle>` and that the information contained therein is `Fitzgerald Johansen`, it doesn't mean that there is any software in the world that knows what a `<middle>` is, what it is used for, or what it means.

In other words, you can create XML documents to describe any information you want, but before XML can be considered useful, applications must be written that understand it. Furthermore, in addition to the capabilities provided by the base XML specification, there are a number of related technologies, some of which are covered in later chapters. These technologies provide more capabilities for us, making XML even more powerful than we've seen so far.

> *Some of these technologies exist only in draft form, so exactly how powerful these tools will be, or in what ways they'll be powerful, is yet to be seen. Other technologies, however, have been in use for a number of years, and are already proving useful in real-world applications.*

# Hierarchies of Information

The syntactical constructs that make up XML are discussed in the next chapter, but first it might be useful to examine how data is structured in an XML document.

When it comes to large, or even moderate, amounts of information, it's usually better to group it into related subtopics, rather than to have all of the information presented in one large blob. For example, this chapter is divided into subtopics, and further subdivided into paragraphs. Similarly, a tax form is divided into subsections, across multiple pages. This makes the information easier to comprehend, as well as making it more accessible.

Software developers have been using this paradigm for years, using a structure called an *object model*. In an object model, all of the information being modeled is divided into various objects, and the objects themselves are then grouped into a hierarchy.

## Hierarchies in HTML

For example, when working with Dynamic HTML (DHTML), an object model is available for working with HTML documents, called the *Document Object Model (DOM)*. This enables us to write code in an HTML document, such as the following JavaScript:

```
alert(document.title);
```

Here we are using the `alert()` function to pop up a message box indicating the title of an HTML document. That's achieved by accessing an object called `document`, which contains all of the information needed about the HTML document. The `document` object includes a property called `title`, which returns the title of the current HTML document.

> *The information that the object provides appears in the form of* properties, *and the functionality available appears in the form of* methods.

## Hierarchies in XML

XML also groups information in hierarchies. The items in our documents relate to each other in parent/child and sibling/sibling relationships.

> **These "items" are called *elements*. Chapter 2 provides a more precise definition of what exactly an element is. For now, just think of them as the individual pieces of information in the data.**

Consider our `<name>` example, shown hierarchically in Figure 1-8.

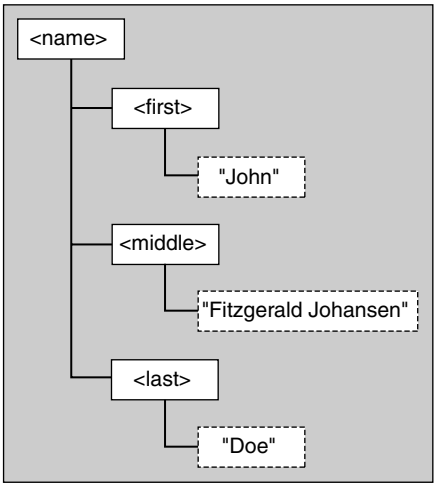


**Figure 1-8**

`<name>` is a parent of `<first>`. `<first>`, `<middle>`, and `<last>` are all siblings to each other (they are all children of `<name>`). Note also that the text is a child of the element. For example, the text `John` is a child of `<first>`.

This structure is also called a *tree,* and any parts of the tree that contain children are called *branches*, while parts that have no children are called *leaves.*

*These are fairly loose terms, rather than formal definitions, which simply facilitate discussing the tree-like structure of XML documents. You might have seen the term "twig" in use, although it is much less common than "branch" or "leaf."*

Because the `<name>` element has only other elements for children, and not text, it is said to have *element content*. Conversely, because `<first>`, `<middle>`, and `<last>` have only text as children, they are said to have *simple content*.

Elements can contain both text and other elements, in which case they are said to have *mixed content,* as shown in the following example:

```
<doc>
  <parent>this is some <em>text</em> in my element</parent>
</doc>
```

Here, `<parent>` has three children:

- A text child containing the text `this is some`
- An `<em>` child
- Another text child containing the text `in my element`

The structure is shown in Figure 1-9.

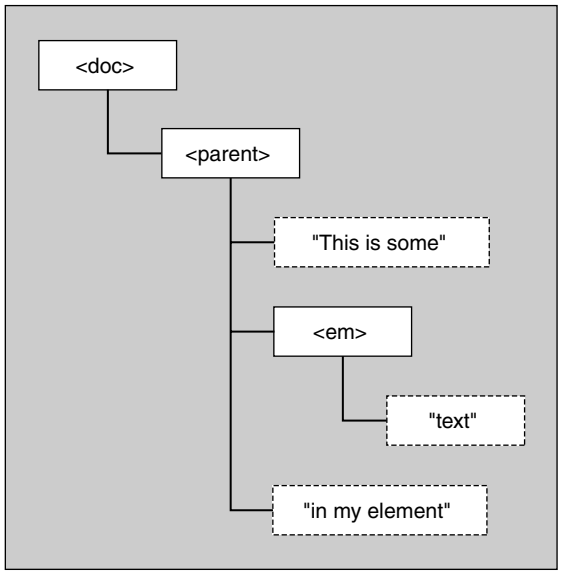


**Figure 1-9**

Relationships can also be defined by making the family tree analogy work a little bit harder: `<doc>` is an *ancestor* of `<em>`; `<em>` is a *descendant* of `<doc>`.

Once you understand the hierarchical relationships between your items (and the text they contain), you'll have a better understanding of the nature of XML. You'll also be better prepared to work with some of the other technologies surrounding XML, which make extensive use of this paradigm.

> *Chapter 11 gives you an opportunity to work with the document object model (DOM) mentioned earlier, which enables you to programmatically access the information in an XML document using this tree structure.*

## What's a Document Type?

XML's beauty comes from its ability to create a document to describe any information we want. It's completely flexible in terms of how we structure our data, but eventually we're going to want to settle on a particular design for our information, and specify "to adhere to our XML format, structure the data like this."

For example, when we created our `<name>` XML above, we created *structured data*. Not only did we include all of the information about a name, but our hierarchy also contains implicit information about how some pieces of data relate to other pieces (our `<name>` contains a `<first>`, for example).

More important, we also created a specific set of elements, which is called a *vocabulary*. That is, we defined a number of XML elements that all work together to form a name: `<name>`, `<first>`, `<middle>`, and `<last>`.

But wait; it's even more than that! The most important thing we created was a *document type*. We created a specific type of document, which must be structured in a specific way, to describe a specific type of information. Although we haven't explicitly defined them yet, there are certain rules to which the elements in our vocabulary must adhere in order for our `<name>` document to conform to our document type. For example:

- The top-most element must be the `<name>` element.
- The `<first>`, `<middle>`, and `<last>` elements must be children of that element.
- The `<first>`, `<middle>`, and `<last>` elements must be in that order.
- There must be information in the `<first>` element and in the `<last>` element, but there doesn't have to be any information in the `<middle>` element.

Unfortunately, there is nothing in our XML document itself which indicates what these rules are; we would have to write any applications that use this data to know the rules, and make sure that they're obeyed. In later chapters, you'll see different syntaxes that you can use to formally define an XML document type. Some XML parsers know how to read these syntaxes, and can use them to determine whether your XML document really adheres to the rules in the document type or not. This is good, because the more work the parser does, the less work your application has to do!

However, all of the syntaxes used to define document types so far are lacking; they can provide some type checking, but not enough for many applications. Furthermore, they can't express the human meaning of terms in a vocabulary. For this reason, when creating XML document types, human-readable documentation should also be provided. For our `<name>` example, if we want others to be able to use the same format to describe names in their XML, we should provide them with documentation to describe how it works.

In real life, this human-readable documentation is often used in conjunction with one or more of the syntaxes available. Ironically, the self-describing nature of XML can sometimes make this human-readable documentation even more important. Often, because the data is already labeled within the document structure, it is assumed that people working with the data will be able to infer its meaning, which can be dangerous if the inferences are incorrect, or even just different from the original author's intent.

## No, Really — What's a Document Type?

Well, okay, maybe I was a little bit hasty in labeling our `<name>` example a "document type." The truth is that others who work with XML may call it something different.

One of the problems people encounter when they communicate is that they sometimes use different terms to describe the same thing, or, even worse, use the same term to describe different things. For example, I might call the thing that I drive a car, whereas someone else might call it an auto, and someone else again might call it a G-class vehicle. Furthermore, when I say car I *usually* mean a vehicle that has four wheels, is made for transporting passengers, and is smaller than a truck. (Notice how fuzzy this definition is, and that it depends further on the definition of a truck.) When someone else uses the word car, or if I use the word car in certain circumstances, it may instead just mean a land-based motorized vehicle, as opposed to a boat or a plane.

The same thing is true in XML. When you're using XML to create document types, you don't really have to think (or care) about the fact that you're creating document types; you just design your XML in a way that makes sense for your application, and then use it. If you ever did think about exactly what you were creating, you might have called it something other than a document type.

> **We picked the terms "document type" and "vocabulary" for this book because they do a good job of describing what we need to describe, but they are not universal terms used throughout the XML community. Regardless of the terms you use, the concepts are very important.**

# Origin of the XML Standards

One of the reasons why HTML and XML are so successful is that they're *standards*. That means anyone can follow the specification and the solutions they develop will be able to interoperate. So who creates these standards?

## What Is the World Wide Web Consortium?

The *World Wide Web Consortium (W3C)* was started in 1994, according to its website (`www.w3.org`), "to lead the World Wide Web to its full potential by developing common protocols that promote its evolution and ensure its interoperability." Recognizing this need for standards, the W3C produces *Recommendations,* or *specifications,* that describe the basic building blocks of the web. They call them "recommendations" instead of "standards" because it is up to others to follow the recommendations to provide the interoperability.

Their most famous contribution to the web is the HTML Recommendation; when web browser producers claims that their product follows version 3.2 or 4.01 of the HTML Recommendation, they're talking about the recommendation developed under the authority of the W3C.

Recommendations from the W3C are so widely implemented because the creation of these standards is a somewhat open process: Any company or individual can join the W3C's membership, and membership allows these companies or individuals to take part in the standards process. This means that web browsers such as Mozilla Firefox and Microsoft Internet Explorer are more likely to implement the same version of the HTML Recommendation, because developers of both applications were involved in the evolution of that recommendation.

Because of the interoperability goals of XML, the W3C is a good place to develop standards around the technology. Most of the technologies covered in this book are based on standards from the W3C: the XML 1.0 Recommendation, the XSLT Recommendation, the XPath Recommendation, and so on.

## Components of XML

Structuring information is a pretty broad topic, and it would be futile to try to define a specification to cover it fully. For this reason, a number of interrelated specifications and recommendations all work together to form the XML family of technologies, with each specification covering different aspects of communicating information. Here are some of the more important ones:

- *XML 1.0* is the base recommendation upon which the XML family is built. It describes the syntax that XML documents have to follow, the rules that XML parsers have to follow, and anything else you need to know to read or write an XML document. It also defines document type definitions (DTDs), although they sometimes are treated as a separate technology.
- Because we can make up our own structures and element names for our documents, *DTDs* and *schemas* provide ways to define our document types. We can check to ensure that other documents adhere to these templates, and other developers can produce compatible documents. DTDs and schemas are discussed in Chapters 4 and 5, respectively.
- *Namespaces* provide a means to distinguish one XML vocabulary from another, which enables us to create richer documents by combining multiple vocabularies into one document type. Namespaces are discussed in detail in Chapter 3.
- *XPath* describes a querying language for addressing parts of an XML document. This enables applications to ask for a specific piece of an XML document, instead of having to always deal with one large chunk of information. For example, XPath could be used to get "all the last names" from a document. We discuss XPath in Chapter 7.
- As mentioned earlier, sometimes we may want to display our XML documents. For simpler cases, we can use *Cascading Style Sheets (CSS)* to define the presentation of our documents. For more complex cases, we can use *Extensible Stylesheet Language (XSL)*; this consists of *XSLT*, which can transform our documents from one type to another, and *formatting objects*, which deal with display. XSLT is covered in Chapter 8, and CSS is covered in Chapter 17.
- Although the syntax for HTML and the syntax for XML look very similar, they are actually not the same — XML's syntax is much more rigid than that of HTML. This means that an XML parser cannot necessarily read an HTML document. This is one of the reasons why *XHTML* was created — an XML version of HTML. XHTML is very similar to HTML, so HTML developers

will have no problem working with XHTML, but the syntax used is more rigid and is readable by XML parsers (since XHTML *is* XML). XHTML is discussed in Chapter 18.

- The *XQuery* Recommendation is designed to provide a means of querying data directly from XML documents on the web. It is discussed in Chapter 9.
- To provide a means for more traditional applications to interface with XML documents, there is a document object model (DOM), discussed in Chapter 11. An alternative way for programmers to interface with XML documents from their code is to use the Simple API for XML (SAX), which is the subject of Chapter 12.
- In addition to the specifications and recommendations for the various XML technologies, some specifications also exist for specific XML document types:
  - The *RDF Site Summary (RSS)* specification is used by websites that want to syndicate news stories (or similar content that can be treated similarly to news stories), for use by other websites or applications. RSS is discussed in Chapter 13.
  - The *Scalable Vector Graphics (SVG)* specification is used to describe two-dimensional graphics, and is discussed in Chapter 19.

# Where XML Can Be Used, and What You Can Use It For

XML can be used anywhere. It is platform- and language-independent, which means it doesn't matter that one computer may be using, for example, a Visual Basic application on a Microsoft operating system, and another computer might be a UNIX machine running Java code. Anytime one computer program needs to communicate with another program, XML is a potential fit for the exchange format. The following are just a few examples, and such applications are discussed in more detail throughout the book.

## *Reducing Server Load*

Web-based applications can use XML to reduce the load on the web servers by keeping all information on the client for as long as possible, and then sending the information to those servers in one big XML document.

For example, a consulting company may write a timesheet application whereby employees can enter how much time they've spent on different tasks; the time entered would be used to bill their clients appropriately. Although employees would often have more than one task to fill, the application could cache all of that data in the browser until the user was finished, meaning that the browser wouldn't have to send or receive any data from the web server. Then, when the user is completely finished, an XML document could be sent to the server, with all of the user's data.

## *Website Content*

It was mentioned earlier that there are technologies — such as CSS and XSLT — that can be used to transform XML from one format to another, or to "style" XML for viewing in a browser. This allows for some very powerful applications of your data.

For example, the W3C uses XML to publish its recommendations. These XML documents can then be transformed into HTML for display (by XSLT), or transformed into a number of other presentation formats. Because all of the presentation formats come from the same XML data file, this solution is faster and less error-prone than having someone re-enter the data in different formats.

Some websites also use XML entirely for their content, where traditionally HTML would have been used. This XML can then be transformed into HTML via XSLT, or displayed directly in browsers via CSS. In fact, the web servers can even determine dynamically what kind of browser is retrieving the information, and then decide what to do — for example, transform the XML into HTML for older browsers, and just send the XML straight to the client for newer browsers, reducing the load on the server.

As an author, I could also use this concept for my writing. After writing a chapter for a book I'm working on, saving it as XML could give me a lot of flexibility:

- I could use a technology such as CSS to make the chapter available on my website.
- I could use a technology such as XSLT to create a "stripped down" version of the chapter if I wanted to publish the content in a magazine article. For example, I might ignore certain aspects of the chapter in the magazine article that I would want to show up in the book. To give myself the most flexibility, I would probably alter the markup in the content in such a way that I could indicate to myself where it should appear: book, magazine article, web, or all of the above.
- I could even transform the XML to a different XML format, which could be understood by a word processor, so that I could further edit it. Most modern word processors — such as Microsoft Word and OpenOffice.org Writer — understand XML formats.

In fact, this can be generalized to *any* content. If your data is in XML, you can use it for any purpose. Presentation on the web is just one possibility.

## Distributed Computing

XML can also be used as a means for sending data for distributed computing, where objects on one computer call objects on another computer to do work. There have been numerous standards for distributed computing, such as DCOM, CORBA, and RMI/IIOP, but as Chapters 14 and 15 show, using XML and HTTP with technologies like web services and/or SOAP enables this to occur even through a firewall, which would normally block such calls, providing greater opportunities for distributed computing.

## e-Commerce

*e-commerce* is another one of those buzzwords that you hear everywhere now. Companies are discovering that by communicating via the Internet, instead of by more traditional methods (such as faxing, human-to-human communication, and so on), they can streamline their processes, decreasing costs and increasing response times. Whenever one company needs to send data to another, XML is the perfect format for the exchange.

When the companies involved in the exchange have some kind of ongoing relationship, this is known as *business-to-business* (B2B) e-commerce. *Business-to-consumer* (B2C) transactions also take place — a system you may have used if you bought this book on the Internet. Both types of e-commerce have their potential uses for XML.

XML is also a good fit for many other applications. After reading this book, you should be able to decide when XML will work in your applications and when it won't.

# Summary

This chapter provided an overview of what XML is and why it's so useful. You've seen the advantages of text and binary files, and the way that XML combines the advantages of both, while eliminating most of the disadvantages. You have also seen the flexibility you can enjoy in creating data in any format you wish.

Because XML is a subset of a proven technology, SGML, there are many years of experience behind the standard. In addition, because other technologies are built around XML, you can create applications that are as complex or simple as your situation warrants.

Much of the power that we get from XML comes from the standard way in which documents must be written. Chapter 2 takes a closer look at the rules for creating well-formed XML.

# Exercise Questions

Suggested solutions to these questions can be found in Appendix A.

## Question 1

Modify the `<name>` XML document you've been working with to include the person's title (e.g., Mr., Ms., Dr., and so on).

## Question 2

The `<name>` example we've been using so far has been in English, but XML is language-agnostic, so you can create XML documents in any language you wish. Therefore, create a new French document type to represent a name. You can use the following table for the names of the XML elements.

| English | French |
|---|---|
| name | identité |
| first | prénom |
| last | nom |
| middle | deuxième-prénom |