

Book I

Describing Data with XML

The 5th Wave

By Rich Tennant



"We're here to clean the code."

Contents at a Glance

Chapter 1: Introducing XML	9
Chapter 2: Dissecting an XML Document	23
Chapter 3: Creating Well-Formed XML Documents	39
Chapter 4: Namespaces — Avoiding Naming Conflicts	49

Chapter 1: Introducing XML

In This Chapter

- ✓ Exploring the roots of XML
- ✓ Discovering why XML is a natural way to describe information
- ✓ Finding out how XML is used
- ✓ Exploring the strengths of XML

Throughout the past decade, a myriad of technologies have been touted as the “next big thing.” Yet, despite their initial acclaim and fanfare, most of these technologies are exposed as pretenders and eventually fade into that Great Fad Wasteland in the Sky, alongside the Pet Rock, Rubik’s Cube, and a host of teeny pop idols.

Extensible Markup Language (XML), however, was not one of those wannabes. Yes, XML was bandied about by industry pundits from the start as the “next big thing.” But as the years go by, XML is proving itself worthy of all the hype, as it is being used in practical and significant ways by untold thousands of developers.

As you begin your journey through the world of XML, this chapter introduces you to Extensible Markup Language and helps you understand where it came from, what it can do for you, and why XML is a natural way to work with all types of information.

Tipping a Hat to HTML

XML owes a lot to Hypertext Markup Language (HTML) for its emergence as the “technology of choice” for many developers today. First, if it wasn’t for the widespread popularity of HTML, markup languages would have never been considered as a viable way to work with data. Second, if it wasn’t for the major shortcomings of HTML, the need for proposing an alternative wouldn’t have been compelling to so many people.

HTML: The information blender

The term “HTML” is synonymous with the World Wide Web. Heck, even my mother has heard of HTML. From the very get-go, HTML has been used as the standard data format to display content over the Web. Documents written in the markup language are stored on Web servers and then delivered to

a Web browser when its Web address (also called a *URL*) is requested. The Web browser, which knows how to work with HTML documents, processes the document and renders it as a Web page.

HTML is a tag-based language that has a defined set of formatting tags or *elements* to define formatting instructions. An element is simply a special code inside a document that tells the browser how to display the information it contains. For example, suppose you'd like to format a sentence in a Web page so that it looks like this:

The **dog** ate *my* homework.

The HTML code you'd write might look something like this:

```
<p>The <b>dog</b> ate <i>my</i> homework.</p>
```

The `p` element is used to define a paragraph, so that everything within the open and close `p` tags is considered a paragraph. The `b` element is for bolded text, so text within the `b` element is bolded. Finally, the `i` element is for italicized text, so words inside the `i` element are italicized.

HTML is popular not only because you can display cool graphics and play annoying MIDI sounds when a Web page is loaded, but, as you can see from the simple example shown above, it is also very easy to learn and work with. Because HTML is just plain text, you can use a visual Web page software to design a Web page, but you can also use plain old Notepad and achieve the exact same result.

When people saw how easy it was to work with HTML, everyone and their brother began to build Web sites. As the Web became more and more popular, businesses in turn began to discover new ways in which they could reach customers through this emerging technology.

But the more people who began to develop innovative new ways to utilize the Web, the more the Web began to resemble an ever-expanding house with an inadequate infrastructure. Let me explain. Imagine a new furnace is installed into a one bedroom house and is designed to effectively heat that size of home. Yet, while keeping the same furnace, suppose you expand the house, constructing a second story and adding on more and more rooms, ultimately transforming the 1,000 square-foot dwelling into a 4,500 square-foot mansion. When the bitter cold of the winter comes, it would become very obvious to the homeowners that a furnace that worked well for a small home doesn't do nearly so well when its job becomes four times more difficult.

In the same way, HTML became a victim of the success of the Web. It was originally designed solely as a way to format and layout information. However, because people began adding more and more uses for the Web,

HTML has been forced to do far more than was ever intended for it, much like that little furnace in the big mansion.

One of the most prolific examples of HTML's inadequacy on the Web is displaying and manipulating information that is stored in a database. Consider a typical example of the Slurpbucks coffee roaster company who wants to allow customers to order coffee beans over the Web. The information about each coffee product is stored in a database that looks like the following:

<i>Coffee</i>	<i>Origin</i>	<i>Taste</i>	<i>Price</i>	<i>Availability</i>	<i>Best With</i>
Guatemalan Express	Guatemala	Mild and Bland	11.99	Year-round	Breakfast
Costa Rican Deacon	Costa Rica	Exotic and Untamed	12.99	Year-round	Dessert
Ethiopian Sunset Supremo	Ethiopia	Cheap but Expensive	14.99	Limited	Chocolate
Kenyan Elephantismo	Kenya	Solid yet Understated	13.99	Year-round	Elephant Ears

In order to be able to present this list of coffee beans on the Web, this information has to be extracted from a database and then converted into an HTML table that the browser can display. The HTML code would look something like this:

```
<table border="1">
  <tr colspan="5">
    <th>Coffee</th>
    <th>Origin</th>
    <th>Taste</th>
    <th>Price</th>
    <th>Availability</th>
    <th>Best With</th>
  </tr>
  <tr colspan="5">
    <td>Guatemalan Express</td>
    <td>Guatemala</td>
    <td>Mild and Bland</td>
    <td>11.99</td>
    <td>Year-round</td>
    <td>Breakfast</td>
  </tr>
  <tr colspan="5">
    <td>Costa Rican Deacon</td>
    <td>Costa Rica</td>
    <td>Exotic and Untamed</td>
    <td>12.99</td>
    <td>Year-round</td>
    <td>Dessert</td>
  </tr>
```

```

</tr>
<tr colspan="5">
  <td>Ethiopian Sunset Supremo</td>
  <td>Ethiopia</td>
  <td>Cheap But Expensive</td>
  <td>14.99</td>
  <td>Limited</td>
  <td>Chocolate</td>
</tr>
<tr colspan="5">
  <td>Kenyan Elephantismo</td>
  <td>Guatemala</td>
  <td>Solid yet Understated</td>
  <td>13.99</td>
  <td>Year-round</td>
  <td>Elephant Ears</td>
</tr>
</table>

```

When you first look at this table in a Web browser, you might think “cool beans” because HTML provided a way to display a database table inside of a browser. While that is useful, only when you start to look closer do you begin to see the major weakness of HTML starting to appear in the cracks.

In this simple example, I transform meaningful collections of data into a table that displays inside of a browser. But the information is now not useful for much else. While inside the database, the prices of the coffees all were identified as “price” fields, and the database record served as the means to store a group of related information. However, when this information was outputted to HTML, the pieces of information lost their identity. They are now just raw pieces of text that have row and column formatting instructions around them. Any concept of a “coffee” record is now gone.

If Slurpbucks wanted to display only product information on its Web site and nothing else, then this loss of identity is no big deal. But like most other people who use the Web, Slurpbucks is discovering that they have needs that go far beyond that. They’d like to be able to work with the same database data inside the browser to perform additional functionality and would like to easily exchange data with the server.

Slurpbucks can create a Moo-Cow-A-Chino drink by mixing espresso, chocolate syrup, frothed milk, and whip cream together. But after these ingredients are blended, there’s no practical way to extract the chocolate syrup from the drink. In the same way, it is hard to extract useful information from HTML documents because the data is mixed into the display instructions.

Therefore, you can think of HTML acting as an “information blender”: start with a cup or two of data and then add a pinch of formatting instructions into a blender. Turn the blender’s power on high, and, after a few seconds,

out comes a pureed mixture of the two. The HTML page may look great, but don't try to use the raw materials again for a different purpose.

XML: Giving data the focus

XML was developed as a solution to the “information blender” effect of HTML. XML allows you to work with structured information on the Web without requiring the data to lose its identity. The inventors of XML wanted to be able to work with structured data in a format that was

- ◆ **Similar to HTML** so that it can be easy to read and work with, but
- ◆ **Different enough from HTML** that XML could effectively describe data in a manner that was separate from display instructions

Therefore, the inventors wanted to preserve the basic markup language capabilities of HTML. But instead of using the markup tags for formatting purposes, they wanted to use the tags for describing the data inside of it.

For example, suppose you start with a group of related information, expressed as a row in a database table:

<i>Coffee</i>	<i>Origin</i>	<i>Taste</i>	<i>Price</i>	<i>Availability</i>	<i>Best With</i>
Guatemalan Express	Guatemala	Mild and Bland	11.99	Year-round	Breakfast

As I show in the preceding section, if you output this cluster of information into HTML markup code, you lose its collective identity:

```
<tr colspan="5">
  <td>Guatemalan Express</td>
  <td>Guatemala</td>
  <td>Mild and Bland</td>
  <td>11.99</td>
  <td>Year-round</td>
  <td>Breakfast</td>
</tr>
```

In contrast, XML combines these two concepts (markup language and data identity) by using HTML-like markup code to describe and structure the data:

```
<coffee>
  <name>Guatemalan Express</name>
  <origin>Guatemala</origin>
  <taste>Mild and Bland</taste>
  <price>11.99</price>
  <availability>Year-round</availability>
  <bestwith>Breakfast</bestwith>
</coffee>
```

What is the W3C?

XML was developed by a group known as the World Wide Web Consortium, or the W3C. The W3C is a group of Web stakeholders, including universities, companies, and other technology geeks that come together under the W3C umbrella to define Web-related standards (known technically as “W3C recommendations”).

In addition to XML itself, the W3C defines recommendations for HTML and many XML-related technologies you’ll be exploring throughout this book, such as XSLT (Extensible Stylesheet Language Transformations), XPath, and XLink. You can visit their Web site at www.w3.org.

XML: A Natural Way to Describe Stuff

Another reason why XML is so popular is that it is such a natural way to describe the data that you work with on a day-in, day-out basis. Most every kind of information you deal with has an implicit structure to it and can be expressed as XML.

Consider the following letter to the editor:

November 31, 2003

Dear Recipe Digest:

Thanks for your recent article on snicker-doodle noodles. However, I don’t think you gave enough credit to the cook who invented the original snicker-doodle noodle – Peter Pasta.

*Warm Regards,
Mrs. Peter Pasta*

Instead of thinking of the letter as a set of grouped paragraphs, it actually has far more structure to it. For example, the above letter could be expressed as XML as the following code:

```
<letter>
  <date>November 31, 2003</date>
  <salutation>Dear Recipe Digest:</salutation>
  <body>Thanks for your recent article on snicker-doodle
    noodles. However, I don't think you gave enough credit to
    the cook who invented the original snicker-doodle noodle
    - Peter Pasta.</body>
  <closing>Warm Regards,</closing>
  <signature>Mrs. Peter Pasta</signature>
</letter>
```

Or consider the dialogue from a movie, *The Princess Bride*:

Inigo Montoya: "Who are you?"

Dread Pirate Roberts: "No one of consequence."

Inigo Montoya: "I must know."

Dread Pirate Roberts: "Get used to disappointment."

Expressed as XML, this text could be structured as follows:

```
<dialogue film="The Princess Bride">
  <InigoMontoya>Who are you?</InigoMontoya>
  <DreadPirateRoberts>No one of
    consequence.</DreadPirateRoberts>
  <InigoMontoya>I must know.</InigoMontoya>
  <DreadPirateRoberts>Get used to
    disappointment.</DreadPirateRoberts>
</dialogue>
```

Finally, those Slurpbucks coffee beans could also be represented as an XML structure:

```
<coffees>
  <region name="Latin America">
    <coffee name="Guatemalan Express" origin="Guatemala">
      <taste>Mild and Bland</taste>
      <price>11.99</price>
      <availability>Year-round</availability>
      <bestwith>Breakfast</bestwith>
    </coffee>
    <coffee name="Costa Rican Deacon" origin="Costa Rica">
      <taste>Exotic and Untamed</taste>
      <price>12.99</price>
      <availability>Year-round</availability>
      <bestwith>Dessert</bestwith>
    </coffee>
  </region>
  <region name="Africa">
    <coffee name="Ethiopian Sunset Supremo" origin="Ethiopia">
      <taste>Cheap but Expensive</taste>
      <price>14.99</price>
      <availability>Limited</availability>
      <bestwith>Chocolate</bestwith>
    </coffee>
    <coffee name="Kenyan Elephantismo" origin="Kenya">
      <taste>Solid yet Understated</taste>
      <price>3.99</price>
```

```
<availability>Year-round</availability>
<bestwith>Elephant Ears</bestwith>
</coffee>
</region>
</coffees>
```

From these examples, you can see XML describes information in a very logical and straightforward manner — *put descriptive tags before and after the text values and you’ve just about got an XML document!*

While HTML is standardized with a fixed set of formatting elements, the only thing standardized about XML is its syntax rules, not the actual tag names. This aspect of XML shows you why it is so flexible in most any situation.



A set of defined XML tags used for a particular purpose is called an *XML vocabulary*.

The Life of an XML Document

XML may be a great technology, but unless you are among the geekiest of folks, you’re not going to want to spend your days looking “under the hood” at a raw XML document. Clearly then, in order to make XML usable in the real world, you need to do something to an XML document to make it presentable or to be otherwise useful.

It may be thus helpful to think about the life of an XML document as flowing through three stages: create, parse, and consume. Each of these stages are described below and shown in Figure 1-1.

- ◆ **Stage 1: Creating an XML document.** The first stage is to create an XML document either by hand using an editor, such as Notepad, or to export data from another software application into XML format. Many databases now allow you to export data as XML documents, along with such everyday software applications as Microsoft Excel or Word.
- ◆ **Stage 2: Parsing the XML document.** After you’ve created the XML document, the second stage is to parse (or process) the document to check to see whether the document structure and content is acceptable for using the document in Stage 3.

Many software applications that consume XML (see Stage 3) have an XML parser built into them. For example, if you display an XML document inside Microsoft Internet Explorer, it first parses the XML document and then renders it inside of the browser window.

Additionally, there are also several freely available XML parsers that enable you to work with XML documents for your own unique uses. These are discussed in greater detail in Book I, Chapter 3.

- ♦ **Stage 3: Consuming the XML document.** The final stage is actually doing something with the XML document that has been “approved” by the parser. Whereas an HTML document has one primary purpose — display in a Web browser — an XML document actually has several possible ways of being utilized by a variety of applications, as Figure 1-1 illustrates.

Some applications can render the XML and display the document in a presentable fashion. However, an XML document by itself does not contain formatting instructions, so the basic view of an XML document inside an application is usually going to either be the raw file (such as in Microsoft WordPad, shown in Figure 1-2) or perhaps as a tree structure (such as in Internet Explorer, shown in Figure 1-3). You discover in Book III that you can apply formatting instructions to XML documents using Cascading Style Sheets (CSS) or Extensible Stylesheet Language (XSL) to make XML documents look presentable in the fashion of an HTML document.

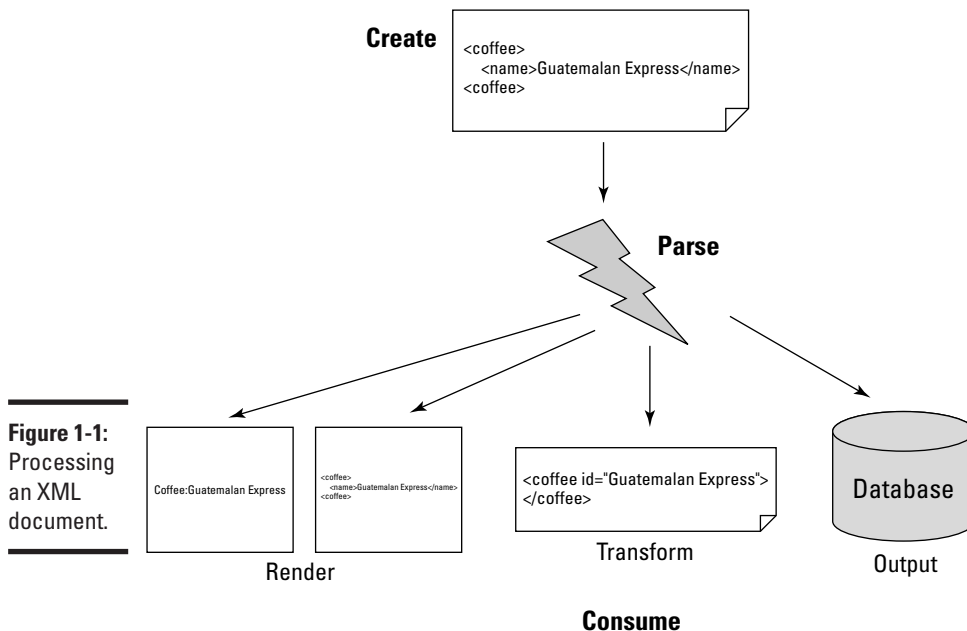
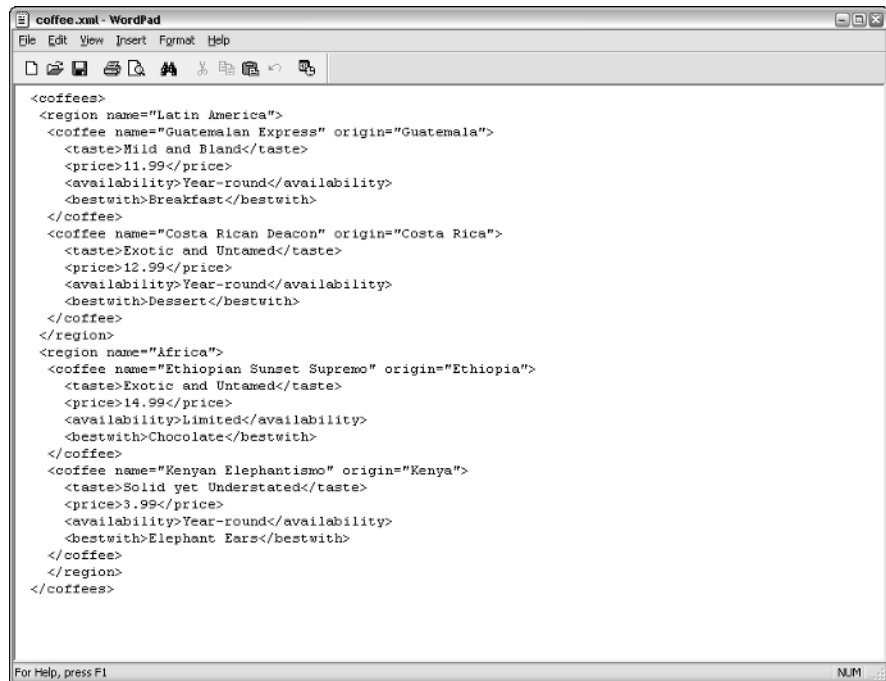


Figure 1-1:
Processing
an XML
document.

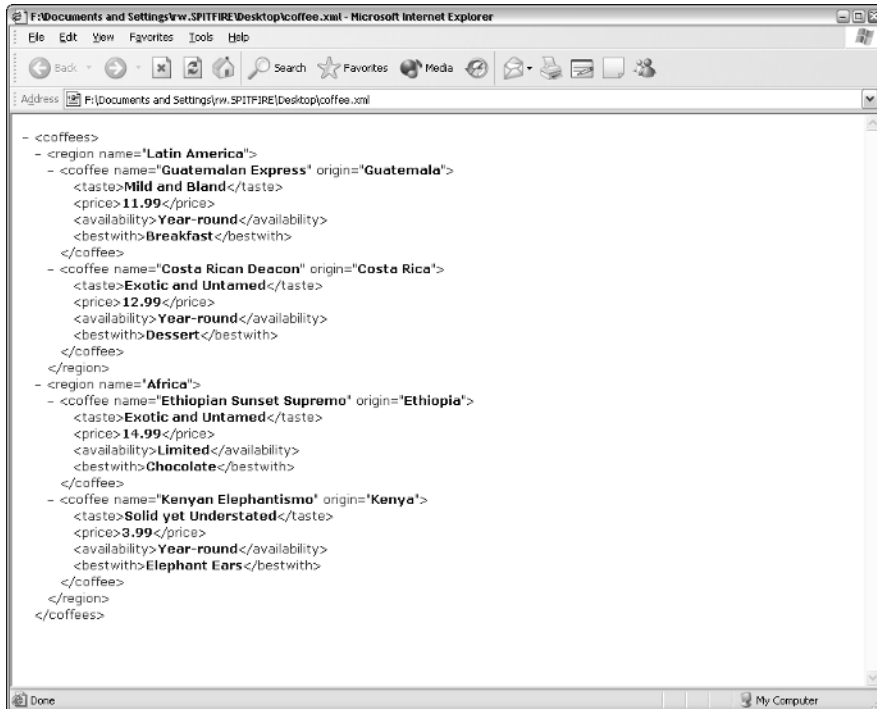
Figure 1-2:
Microsoft
WordPad
rendering
an XML
document
as plain text.



XML documents are used for more purposes than just rendering, however. Perhaps you need to change the document into a completely new structure for another purpose. XSLT is the XML-based technology that enables you to perform exactly these kind of transformations and is covered in detail in Book IV. Or perhaps you have a need to move the XML data into a traditional relational database. Many databases today support importing of XML documents.

XML is an extensible, flexible technology for defining data. The ways in which you can utilize XML are as extensive, too.

Figure 1-3:
Internet
Explorer
rendering
an XML
document
as a tree.



Yes, but Is XML Useful?

By now, you've discovered some of the basics of what XML is and how it works, but some of the implications of using XML in the real world may not be immediately apparent. While there are many reasons why XML is a useful technology, the following are but a sampling of some of these key uses.

- ◆ **XML allows you to create your own markup language.** Unlike HTML, XML has no predefined set of elements. You can create your own elements based on the specific needs of your application, company, or industry. Consider the following XML code snippet:

```
<coffee>
  <name>Guatemalan Express</name>
  <origin>Guatemala</origin>
  <taste>Mild and Bland</taste>
  <price>11.99</price>
  <availability>Year-round</availability>
  <bestwith>Breakfast</bestwith>
</coffee>
```

By reading the element names and then looking at their content, you can easily determine that the structure describes the details of a

particular coffee. Check out Book I, Chapter 2 for more information on how to create your own set of XML elements.

- ◆ **XML data can be presented in a variety of ways.** XML concentrates on describing data, not on the appearance of the data. However, as I mention earlier in the chapter, you can display XML documents in a variety of ways based on CSS and XSL technologies. XML's ability to separate data from the presentation instructions allows you to change the way an XML document is displayed without changing the actual data.
- ◆ **XML enables more robust and efficient information exchange over the Web.** XHTML is considered the "next generation" version of HTML and adds many of the advantages of XML to the older HTML language. (See Book V, Chapter 4 for more on XHTML.) When browsers eventually support it, the data exchange between a browser and a Web server will be greatly optimized. For example, rather than reloading an entire HTML page, XHTML will enable you to download only the portion of the page that has changed since the last reload.
- ◆ **XML is well-suited as means of transferring data across platforms.** Businesses depend on reliable data interchange within and across companies. However, the age-old challenge in transferring data has been that everyone stores data in a different structure using different software tools. Fortune 500 companies have turned to such technologies as EDI (Electronic Data Interchange) as a standardized way to transfer data across companies. But EDI is very expensive and not practical except for the largest of companies. In contrast, XML provides a low-cost, low-barriers to entry solution to provide a universal data format that makes communication across heterogeneous platforms possible.

Yup, that "X" stands for *extensible*

Working with HTML and other legacy technologies is something like a goldfish swimming in a bowl. A goldfish can survive in a small bowl, but the goldfish's life is characterized by the severe limits of its confines. Studies show that even after you set a goldfish free in a lake, the goldfish will continue to swim in small circles, not realizing the vastness of its new domain.

In the same way, as you start to work with XML, you may find yourself "swimming in small circles" as you grapple with the extensible nature of XML. Those accustomed to working with HTML can be so used to describing data in a

way that makes sense to HTML's fishbowl that they find it hard to believe that XML actually enables them to describe data like it should be done — without forcing it into an artificial structure imposed by something outside.

Therefore, when you are defining an XML structure to describe data, don't fall into the trap of defining the information based on how it is going to be presented or utilized. Instead, try and look at the natural organization and types of data that you are trying to structure and produce an XML vocabulary accordingly.

XML and EDI

EDI (Electronic Data Interchange) is a technology that has been used over the last 40 years to transfer documents and data between businesses. Many see XML as a successor to this technology due to three reasons:

- ✓ **Lower cost:** XML does not require a high priced infrastructure like EDI does. Therefore, XML is a practical technology that even businesses large and small can utilize.
- ✓ **More flexible:** XML enables you to store data in a structured format using your own XML vocabulary. In contrast, EDI is far less

flexible and does not, for example, allow individual business units in a company to define their own business rules. Instead, it has a fixed transaction set, which is required to be followed by all business units.

- ✓ **Easy to interpret:** EDI requires specialized translation software to map the EDI formats to the formats used by individual trading partners. In contrast, XML documents can be easily interpreted by the target systems by using an everyday XML parser, many of which are available for free download on the Internet.

Chapter 2: Dissecting an XML Document

In This Chapter

- ✓ Discovering the basic building blocks of an XML document
- ✓ Knowing how to name elements and attributes
- ✓ Exploring XML declarations, processing instructions, and comments
- ✓ Looking at the document as a tree
- ✓ Comparing XML and HTML

Okay class, it's dissection time. But don't worry if you're squeamish — unlike your high school biology course, you won't have to put on a lab smock, smell formaldehyde, or gaze upon a cut-up frog. Instead, you can slice and dice your way through the relatively clean confines of an XML document.

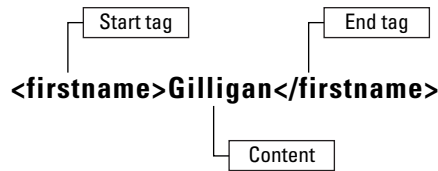
In this chapter, you explore the innards of an XML document and discover what it takes to build one. You begin by looking at *elements* (the basic building blocks of XML) and then go on to explore a document's organization and structure.

Elements: The Basic Building Blocks

If you study the human body and want to understand how it works, you'd probably want to start with the cell, the basic building block of a human. Similarly, when you want to understand how XML works, you have to start with its most fundamental unit, the element.

An *element* is made up of a start tag, an end tag, and the content in between the two (see Figure 2-1). The *start tag* is enclosed in angle brackets and is given a name that appropriately describes the data inside the element. The *end tag* is also enclosed in angle brackets but is prefixed with a forward slash (/) after the opening bracket, followed by the same identifier as its companion start tag. The *content* is the stuff that falls in between these tags.

Figure 2-1:
An element
is the basic
building
block of
XML.



An element can contain text, other elements, or both inside its start and end tags. For example, each of the following are valid XML elements:

```
<quote>We thought you was a toad.</quote>
```

```
<quote>
  <spokenby>Delmar</spokenby>
  <source>O Brother, Where Art Thou</source>
  <text>We thought you was a toad.</text>
</quote>
```

```
<quote><spokenby>Delmar</spokenby> said: We thought you was a
  toad.</quote>
```

The first element contains text, the second element contains elements nested inside of it, and the third element contains a mixture of both text and elements.



If you've ever viewed the source of an HTML file, the syntax shown above should look familiar to you, since HTML follows a similar tag/content structure. However, XML permits a shortcut syntax that you can use that is not available in HTML. When you have an element that doesn't contain any content, you can combine the start and end tags to form an *empty element*. To combine, take the forward slash (/) from the end tag and add it onto the end of the start tag, like this:

```
<combo></combo>
```

is the same as:

```
<combo/>
```

To illustrate, suppose the `quote` element below has a favorite element that is used to denote the favorite film quote in the document:

```
<quote>
  <spokenby>Delmar</spokenby>
  <source>O Brother, Where Art Thou</source>
```

```
<text>We thought you was a toad.</text>
<favorite></favorite>
</quote>
```

Because the `favorite` element is used as flag, it doesn't need any content. Therefore, the `favorite` element syntax can be reduced to an empty tag, as shown in the following code:

```
<quote>
  <spokenby>Delmar</spokenby>
  <source>O Brother, Where Art Thou</source>
  <text>We thought you was a toad.</text>
  <favorite/>
</quote>
```

Attributes: Describing Elements

While elements contain information and a descriptive tag name that labels the content, you often have additional qualities that you'd like to associate with that piece of data. For example, consider the following `quotes` element that contains several `quote` elements:

```
<quotes>
  <quote>We thought you was a toad.</quote>
  <quote>To infinity, and beyond!</quote>
  <quote>It seems to me that, aside from being a little
    mentally ill, she's pretty normal.</quote>
  <quote>More isn't always better, Linus. Sometimes it's just
    more.</quote>
  <quote>Who are you really, and what were you before? What
    did you do, and what did you think, huh?</quote>
  <quote>Well, it's Groundhog Day... again...</quote>
  <quote>To the pain...</quote>
  <quote>Only one is a wanderer; two together are always
    going somewhere.</quote>
</quotes>
```

For each of the `quote` elements, suppose that you'd like to add an ID to identify the element and a category to classify each of the quotes. You can do so by adding attributes to the inside of the element's start tag. An *attribute* consists of a name and value bound together in a *name-value pair*. A name-value pair is formatted as follows:

```
name="value"
```

or

```
name='value'
```



The pair must always be connected using the equal sign, and the attribute value must always be enclosed in either single or double quotes.

You can add as many attributes to an XML element as you wish, so long as the names are unique and you separate the name-value pairs with one or more spaces. In this example, I'll add two attributes to each `quote` element:

```
<quotes>
  <quote id="101" film="O Brother Where Art Thou"
    category="Comedy">
    We thought you was a toad.</quote>
  <quote id="102" film="Toy Story" category="Children">
    To infinity, and beyond!</quote>
  <quote id="103" film="Benny & Joon"
    category="Comedy/Drama">
    It seems to me that, aside from being a little mentally
    ill, she's pretty normal.</quote>
  <quote id="104" film="Sabrina" category="Poignant">
    More isn't always better, Linus. Sometimes it's just
    more.</quote>
  <quote id="105" film="Casablanca" category="Romance">
    Who are you really, and what were you before? What did
    you do, and what did you think, huh?</quote>
  <quote id="106" film="Groundhog Day" category="Comedy">
    Well, it's Groundhog Day... again...</quote>
  <quote id="107" film="Princess Bride" category="Comedy">
    To the pain...</quote>
  <quote id="101" film="Vertigo" category="Romance">
    Only one is a wanderer; two together are always going
    somewhere.</quote>
</quotes>
```

What's In a Name?

In “XMLSpeak,” the name of an element or attribute is referred to as a name token or *NMTOKEN*. As I discuss in Book I, Chapter 1, XML has no fixed set of elements or attributes like HTML does. You are free then to name your NMTokens in any way that you please. However, be aware of the rules concerning permissible characters and case sensitivity.

Legal characters

There are some rules that you need to keep in mind as you work with NMTokens:

- ◆ **Starting character.** NMTokens must begin with an alphabetic letter or an underscore. You can't use a number or a special character.

- ◆ **Valid characters.** NMTokens can include any alphanumeric character, underscores, hyphens, and periods.
- ◆ **Invalid characters.** NMTokens can't include any white space or any other special character not mentioned in the valid characters list.

Table 2-1 lists some examples displaying valid and invalid element names.

Valid Name	Invalid Name
<darnquote>	<#&\$!quote>
<film_quote>	<film quote>
<quote1>	<1quote>
<film.quote>	<.film.quote>
<_quote>	<quote>>

Case sensitivity

Unlike HTML, which ignores case altogether, XML element and attribute names are case sensitive. As a result, each of the following are considered different element names because their case is different:

```
<filmquote/>
<FILMQUOTE/>
<filmQuote/>
```



To avoid confusion, you should consistently use either all lower, all upper-case, or mixed case for the XML documents you create. I recommend standardizing on lowercase — it's the convention most developers follow.

Looking at XML through HTML eyes

You may be approaching XML after having already worked with HTML to create Web pages. If so, as you begin working with XML, you'll naturally look at the markup language through HTML eyes.

If you have an HTML background, you definitely have a head start as you begin to learn the syntax of XML. Use this section as a primer to

reinforce some of the similarities and differences in the syntax of the two markup languages.

Similarities

1. Element is basic building block.

The element is the basic building block of XML, much like in HTML. For example, a paragraph in HTML is defined as:

(continued)

(continued)

```
<p>Only one is a wanderer; two
  together are always going
  somewhere.</p>
```

Similarly, a quote in XML can be defined to look something like this:

```
<quote>Only one is a wanderer;
  two together are always
  going somewhere.</quote>
```

2. Attributes are used to describe elements.

XML follows HTML's practice of using attributes (name-value pairs) inside elements to provide additional descriptive information about an element. For example, the `bgcolor` attribute provides additional information related to HTML's body element:

```
<body bgcolor="#FFFFFF">
</body>
```

In the same way, XML uses name-value pairs, as shown below:

```
<quote id="101">
  Only one is a wanderer; two
  together are always going
  somewhere.
</quote>
```

In this XML snippet, the `id` attribute provides additional information related to the `quote` element.

Differences

HTML and XML have definite similarities, but there are also important variations in syntax rules as well. The three most important are as follows.

1. XML is well-formed.

HTML has always been laid back in some of its syntax requirements, not always forcing you to have closing tags on some of the elements, such as the paragraph (`<p>`) element. For example, both of the following lines are valid HTML:

```
<p>Hello. My name is Inigo
  Montoya. You killed my
  father. Prepare to die.
```

and

```
<p>Hello. My name is Inigo
  Montoya. You killed my
  father. Prepare to die.</p>
```

In contrast, XML is much more rigid: All XML documents must be "well-formed," meaning that every begin tag needs to have a matching end tag:

```
<source>Princess
  Bride</source>
```

2. XML allows "shortcuts".

While XML requires any element to have an open and closed tag pair, it does allow you to combine the two tags if the element is *empty*, meaning that no text is provided between the open and closed tags. For example, the following two lines are equivalent:

```
<device
  id="3838-2020"></device>
```

and

```
<device id="3838-2020"/>
```

In contrast, HTML does not allow for empty tags using this notation.

3. XML is case sensitive.

HTML is case insensitive. As long as you spell out the tag syntax correctly, the document is processed appropriately. For example, each of the following are valid and identical HTML statements:

```
<body
  bgcolor="#FFFFFF"></body>
<BODY
  BGCOLOR="#FFFFFF"></BODY>
<Body
  Bgcolor="#FFFFFF"></Body>
```

On the other hand, XML is case sensitive, so the following statements are not considered equal:

```
<quote>Get used to disappoint-  
ment.</quote>  
<QUOTE>Get used to disappoint-  
ment.</QUOTE>
```

If you can read HTML, you'll quickly discover that XML looks very much like HTML in terms of its markup language structure.

Putting the Pieces Together

When you've got a basic understanding of elements and attributes, you can begin to look at the XML document as a whole. An XML document is a collection of elements that describe a particular set of data. As you find out in Book I, Chapter 3, a valid XML document has a single element that serves as the container for all other elements in the file. This "mother-of-all-elements" is known as the *root element* (or sometimes called the *document element*).

To illustrate, consider the following XML structure of film quotes. The `filmquotes` element serves as the root element for this XML document because it contains all the other elements:

```
<filmquotes>  
  <quote id="101">  
    <spokenby>Sam</spokenby>  
    <source>Benny and Joon</source>  
    <text>It seems to me that, aside from being a little  
      mentally ill, she's pretty normal.</text>  
  </quote>  
  <quote id="102">  
    <spokenby>Sabrina</spokenby>  
    <source>Sabrina</source>  
    <text>More isn't always better, Linus. Sometimes it's  
      just more.</text>  
  </quote>  
  <quote id="103">  
    <spokenby>Delmar</spokenby>  
    <source>O Brother, Where Art Thou</source>  
    <text>We thought you was a toad.</text>  
  </quote>  
  <quote id="104">  
    <spokenby>Inigo Montoya</spokenby>  
    <source>Princess Bride</source>  
    <text>Hello. My name is Inigo Montoya. You killed my  
      father. Prepare to die.</text>  
  </quote>  
</filmquotes>
```

The `filmquotes` element contains the `quote` four elements, but you'll notice that the `quote` elements also contain elements themselves. As a result, `filmquotes` and `quote` are considered *container elements* because they enclose other elements inside their start and end tags. In contrast, the `spokenby`, `source`, and `text` elements don't contain any elements inside them.

When an element is inside a container element, it is considered a *child element* or *sub element*. Therefore, the `quote` elements are child elements of `filmquotes`, and the `spokenby`, `source`, and `text` elements are children of `quote`.



Keep in mind as you begin to create XML documents that there can't be more than one root element per document or you'll get errors trying to process the document. Therefore, you couldn't simply have the four `quote` elements be free to roam about outside of a container element in a document.

In addition to the root element, you also have additional parts that may be part of an XML document: XML declaration, processing instructions, comments, and entities.

In the end, designing an XML structure is more an art than a science. The longer you work with the data, the more you begin to get a feel for which data bits work best as attributes and which work as elements inside the container element for your particular needs.

Six one way, half a dozen the other

Because elements can contain attributes as well as other elements, it can be confusing to determine whether a given piece of data should be in the form of an element or attribute. After all, consider the following three `quote` elements. Each of them are valid ways of containing the inter-related information inside of an element:

Mixed

```
<quote id="101"
  category="Comedy">We
  thought you was a
  toad.</quote>
```

All elements

```
<quote>
  <id>101</id>
```

```
<category>Comedy</comedy>
<text>We thought you was a
  toad.</text>
</quote>
```

All attributes

```
<quote id="101"
  category="Comedy" text="We
  thought you was a toad."/>
```

While each of these may be valid from an XML syntax perspective, which is the best way to express the data? In one sense, you could apply the old adage, "six one way, half a dozen the other" and say that it really doesn't matter. However, there are considerations to think about:

- ✓ **Verboseness of content.** If the content that you are storing is sizable, then you'll usually want to use an element. For example, because the text of the quotes in the elements can span multiple lines, putting the text inside an element is easier to work with than as an attribute value, which must be enclosed in quotes. For example, the first element below is far harder to work with than the second:

```
<quote id="105" text="Who are
you
really, and what were you
before? What did
you do, and what did you
think, huh?"
category="Comedy"/>
```

```
<quote id="105"
category="Comedy">
Who are you really, and what
were you before? What did
you do, and what did you
```

```
think, huh?
</quote>
```

- ✓ **Presentable data.** In some XML implementations, it is more straightforward to format and display element content than it is attribute values. Therefore, if you are certain that the data bit you are using must be presented to the user, then you may want to consider using an element instead of an attribute name-value pair.

- ✓ **Manageability.** Attributes can be easier to manage within an element than is a set of nested elements. For example, the following element:

```
<quote id="101"
category="Comedy"/>
```

can be easier to work with than the following nested element:

```
<quote>
<id>101</id>
<category>Comedy</category>
</quote>
```

Declaring the XML document

An XML document is a normal text file that has a structure beginning with a root element that in turn contains more elements. However, in order to identify a XML document, you need to add an *XML declaration* at the top of your file that simply says to an XML processor, "Hey, I'm an XML file. Please process me!"

An XML declaration is a specific kind of *processing instruction* (PI), which is a special kind of XML markup tag that is used to instruct the XML processor on how to parse the document. While a PI is included inside the document, it is not considered part of the document's XML data and wouldn't ever be shown to a user or outputted in another fashion. You may find it helpful to think of a PI as being akin to Post-It note that you affix to a report you give to your coworkers for review. The yellow-sticky note tells your coworkers special instructions on what to double-check. But those notes are removed before you give the final report to your boss. In the same way, PIs are stripped out of the document by the processor during its parsing process.

A PI has a special syntax that uses a `<?` to begin the tag and a `?>` to end the tag. Inside the brackets, the processing instruction must always be lowercase:

```
<? processinginstruction ?>
```



A processing instruction is a single lowercase tag enclosed with the `<?` and `?>` characters. It doesn't have a corresponding end tag like a normal XML element does.

A basic XML declaration uses the following syntax:

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

The `<?xml ?>` identifies the tag as an XML declaration, and the `version` attribute denotes the version of XML that the document conforms to. Currently, there is only a single version of XML, so you'll always use `1.0` as the value. This instruction then tells the processor that the document should be parsed as if it is an XML 1.0 document.

The XML declaration is placed at the top of the XML document, before the root element, as shown in Listing 2-1.

Listing 2-1 filmquotes.xml

```
<?xml version="1.0"?>
<filmquotes>
  <quote id="101">
    <spokenby>Sam</spokenby>
    <source>Benny and Joon</source>
    <text>It seems to me that, aside from being a little
      mentally ill, she's pretty normal.</text>
  </quote>
  <quote id="102">
    <spokenby>Sabrina</spokenby>
    <source>Sabrina</source>
    <text>More isn't always better, Linus. Sometimes it's
      just more.</text>
  </quote>
  <quote id="103">
    <spokenby>Delmar</spokenby>
    <source>O Brother, Where Art Thou</source>
    <text>We thought you was a toad.</text>
  </quote>
  <quote id="104">
    <spokenby>Inigo Montoya</spokenby>
    <source>Princess Bride</source>
    <text>Hello. My name is Inigo Montoya. You killed my
      father. Prepare to die.</text>
  </quote>
</filmquotes>
```



Even though you can have only one root element, the XML declaration — nor any PI for that matter — is considered part of the actual document structure and therefore is not considered an additional root element.

There are also two other optional attributes that you can add to the XML declaration, as shown in the following example:

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

The `standalone` attribute specifies whether or not the document is independent of other files for its markup declarations. If you use a `no` value, then you tell the XML processor that you are using *external markup declarations* within the document, which are usually defined in a document type definition (DTD). You can discover more about *external markup declarations* in Book II, Chapter 2. If you don't include the `standalone` attribute, then the processor defaults the value to `yes`.

The `encoding` attribute specifies the encoding scheme used in the XML document. The *encoding scheme* is the standard character set of a language. The XML processor uses this encoding information to know how to work with the data contained in the XML document. *UTF-8* is the standard character set used to create pages written in English.



UTF stands for UCS (Universal Character Set) Transformation Format. This character set uses eight bits of information to represent each character. Therefore, *UTF-8* stands for an 8-bit character set. *UTF-8* supports characters compatible with computing systems based on ASCII text. However, if you need to work with information in other languages, such as Japanese Katana and Cyrillic, you need to set the encoding attribute to *UTF-16*. *UTF-16* character set uses 16-bits to store a character.

Getting descriptive with comments

A *comment* is descriptive text included in your XML document for “behind-the-scenes” use that the XML processor ignores during its parsing process. You may, for instance, want to use a comment to label a particular structure, describe the purpose of a given element, or jot a note about something you need to add to the document later. A comment is something used for development purposes only and is never output by the XML processor.

Just like in HTML, a comment is any text that is surrounded by a `<!--` prefix and a `-->` suffix. For example, the following verbose XML document will be parsed by an XML processor and produce the same output as the comment-free structure shown earlier in the chapter:

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

```
<!-- Developed by: Gargan Zola -->
```

```
<!-- Last modified: 6/22 -->

<filmquotes>
  <!-- Note to self: Double-check this one again after seeing
    the movie -->
  <quote id="101">
    <spokenby>Sam</spokenby>
    <source>Benny and Joon</source>
    <text>It seems to me that, aside from being a little
      mentally ill, she's pretty normal.</text>
  </quote>
  <quote id="102">
    <spokenby>Sabrina</spokenby>
    <source>Sabrina</source>
    <text>More isn't always better, Linus. Sometimes it's
      just more.</text>
  </quote>
  <!-- This is my favorite quote! -->
  <quote id="103">
    <spokenby>Delmar</spokenby>
    <source>O Brother, Where Art Thou</source>
    <text>We thought you was a toad.</text>
  </quote>
  <quote id="104">
    <spokenby>Inigo Montoya</spokenby>
    <!-- Hey, no one will ever read this comment when it
      is output! -->
    <source>Princess Bride</source>
    <text>Hello. My name is Inigo Montoya. You killed my
      father. Prepare to die.</text>
  </quote>
</filmquotes>
```



Use comments freely in your XML documents. Doing so can make your XML document easier to read and work with than without them. This is especially true if more than one person works on an XML document.

You can do pretty much anything you want with comments, but there are a few rules that you need to follow:

- ◆ **Location.** You can add comments anywhere inside the XML document except: (a) above the XML declaration and (b) inside a start or end tag.

Therefore, the following are *invalid* comments:

```
<!-- Developed by: Gargan Zola -->
<?xml version="1.0"?>
```

and:

```
<quote <!-- This is my favorite quote! --> id="103">
```

- ♦ **Permissible text.** You can type most any text inside a comment and span as many lines as you wish. However, you can't use two consecutive hyphens or you'll get an error. Consider the following *invalid* example:

```
<!-- Developed by -- Gargan Zola -->
```

Adding aliases to your documents

An *entity* is a shortcut or an alias to another piece of information. XML allows you to declare an entity either inside your document or in an external DTD. For example, if you have a long piece of text that you need to use throughout a document, you can define an entity once and then reference the entity in your document, saving you space and time.

XML also provides some built-in entities to enable you to use characters that have special meaning for XML documents. Suppose the content of an XML element needs to use an opening angle bracket, as shown in the following example:

```
<expression>(5*5)<(6*6)</expression>
```

An XML processor would generate an error trying to process that element because the `<` character must always refer to the start of an XML element. Therefore, to get around that problem, you can use the built-in entity `<`; to represent the angle bracket in the expression:

```
<expression>(5*5)&lt;(6*6)</expression>
```

When the XML processor parses the element, it recognizes the entity and replaces the `<` with a `<` when it is rendered or output.

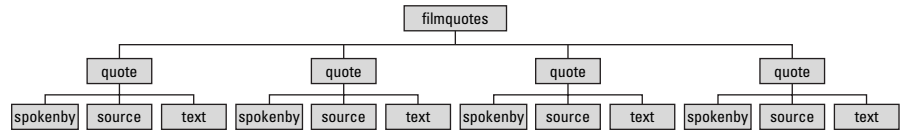


Entities are closely associated with DTDs. See Book II, Chapter 2 for complete details on using entities with your XML documents.

Thinking of Documents as Trees

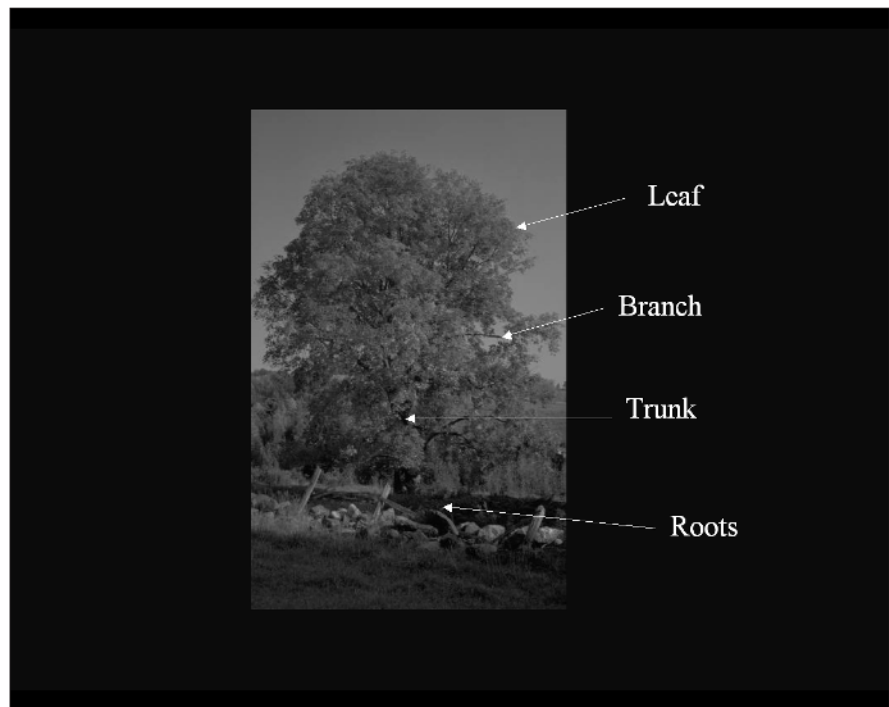
So far, as you've explored the innards of an XML document, the focus has been on the syntax of the XML elements and document as a whole. I've alluded to the hierarchical structure of an XML document, but the focus has thus far been on a "flat" document. However, XML documents have a very clear hierarchy to them when you start to examine the various layers of a document. For example, Figure 2-2 illustrates the hierarchy of elements from the `filmsquotes.xml` file shown in Listing 2-1.

Figure 2-2:
Hierarchy of
elements.



As you can see, an XML document has a natural hierarchy or tree-like structure. In the same way, a tree has a built-in structure or hierarchy to its various parts (see Figure 2-3). At the base layer is a root system directly connected to the trunk, which is the most noticeable part of the tree's structure. At the next level, you have a multitude of branches, some large and some small, that directly connect with the tree trunk. These branches then either connect to smaller branches or else to leaves. In turn, the tinier branches connect to leaves or to even smaller branches. If you started at the trunk of the tree, you could trace a path to each branch and leaf within the interconnected hierarchy of tree limbs.

Figure 2-3:
Various
parts of a
tree.



An XML document is quite similar in structure. A document's root element is the counterpart to a tree trunk, because it is the container of all other elements in the document. Elements directly inside of the root element are the equivalent of the first level of branches of a tree. Some of these elements will contain additional elements, like smaller branches on a tree.

When you start working with XSLT later in this book, you discover more about how to work with an XML document as a tree. See Book IV, Chapter 1 for more details.

Chapter 3: Creating Well-Formed XML Documents

In This Chapter

- ✓ Exploring “well-formedness” and why it is critical to XML
- ✓ Creating your first well-formed XML document

Close only counts in horseshoes and hand grenades.

Boy, how many times I’ve heard that expression from someone, stressing the fact that simply being close oftentimes isn’t good enough. In fact, consider the following scenarios:

- ◆ A student receiving a 1599 on the SAT nears perfection, but still loses out on a scholarship to the other student who received a perfect 1600.
- ◆ A baseball team mounts an incredible 5-run comeback in the bottom of the 9th inning but goes on to lose in the 11th by the homerun of the opposing team.
- ◆ While driving your brand new car down a road, you see a tree falling into the road. You swerve to avoid it; despite your valiant attempt, your car is totaled just the same, as the tree clips the tail end of the vehicle.

Being close in situations like these may provide a “moral victory,” but in the harsh world that you and I live in, “moral victories” have no substance to them and only serve to make it easier for us to sleep at night.

In the same way, an XML document must be syntactically perfect to be a true XML document, adhering to the syntax rules discussed in Book I, Chapter 2. “Being close,” such as missing a closing tag or mixing the case of elements, isn’t good enough. Any error, regardless of how small it is, renders an XML document as unacceptable for processing by an XML parser.

In this chapter, I discuss what it means to have a well-formed XML document as well as discuss the how-to steps of structuring a XML document that goes beyond “being close.”

Understanding “Well-Formedness”

Given the amazing popularity of diets these days, you can see that people both young and old are obsessed with being “well-formed.” XML is similarly fanatical with well-formedness. It may not care about carbohydrates or fat grams, but XML is focused on the syntax of the XML document.

A *well-formed document* is one that is syntactically correct. For example, Figures 3-1 and 3-2 show examples of a well-formed document and a poorly formed document. An XML parser processes a well-formed document but spits back an incorrect document.

Figure 3-1:
A well-
formed XML
document.

```
<beverage type="Cola">
  <size>Small</size>
  <size>Medium</size>
  <size>Large</size>
  <value>6<7</value>
  <special/>
</beverage>
```

Figure 3-2:
A poorly-
formed XML
document
with several
syntax
problems.

```
<beverage type=Cola>
  <size>Small</size>
  <size>Medium</size>
  <size>Large</size>
  <value>6<7</value>
  <special>
</Beverage>
```

Quotes missing

Illegal character being used in content

Empty tag is not terminated

Case difference

Book I, Chapter 2 discusses the syntax rules of XML, but use Table 3-1 as a quick checklist.

Table 3-1 Well-Formed Document Checklist	
Question	Checkbox
Document has a single root element	☐
Element and attribute names have consistent use of case	☐
All non-empty elements have a closing tag	☐

Question	Checkbox
All empty elements terminate properly	☐
No elements are overlapped and follow the “last open, first closed” rule	☐
Attribute values are enclosed by quotation marks	☐
Entities are used in place of special characters (such as <code>&lt;</code> instead of <code><</code>)	☐



While the terms sound synonymous, a *well-formed document* is not the same as a *valid document*. A valid document refers to one whose structure is tested against a Document Type Definition (DTD) or an XML Schema by validating parser. Book II covers the issue of validating XML documents.

Writing a Well-Formed XML Document

If you’ve read Book I, Chapter 2 and the first part of this chapter, you know how an XML document should look to be well-formed. However, it’s time now to dive in and create an XML document from scratch — from concept to completion. In doing so, I follow a five-step process:

- ◆ Structuring your document.
- ◆ Declaring an XML document.
- ◆ Adding the root element.
- ◆ Adding the remaining data.
- ◆ Viewing the XML document

Each of these steps are discussed below.

Step 1: Structuring your document

Perhaps the most critical step in creating a well-formed document is to determine the appropriate XML structure that best describes the data. Consider, for example, the structured information shown in Table 3-2. It displays a listing of entrées on a menu for the world-famous Gilligan’s By The Sea restaurant.

Table 3-2		List of Entrees for Gilligan’s By The Sea		
ID	Name	Features	Fat Grams	Description
1	Sunburnt Chicken	Salad, vegetables, baked potato, and dessert	23	Chicken prepared so hot, you’ll need a gallon of soda to wash it down. Bring sunscreen!

(continued)

Table 3-2 (continued)

<i>ID</i>	<i>Name</i>	<i>Features</i>	<i>Fat Grams</i>	<i>Description</i>
2	Filet Mig's None	Soup, vegetables, baked potato, and dessert	0	Our master chef Mig prepares a uniquely no-fat filet mignon. You won't believe how great it tastes!
3	Chicken Parmashaun	Soup, pasta, baked potato, and dessert	20	Our award-winning chicken parmesan prepared especially for you by our master chef Shaun.
4	Eggs Benelux	Bacon, sausage, and toast	35	No matter the time of day, enjoy our scrumptious breakfast cooked by our famous Belgian and Dutch master chefs.
5	Jerk Chicken	Soup, vegetables, and dessert	5	A delicious hot Jamaican dish prepared by our most obnoxious master chef.
6	Gusto Spaghetti	Soup, salad, and dessert	55	Our famous master chef Boyd Ardee prepares a succulent dish of spaghetti with zesty gusto!

This entrée list provides an example of how information that you and I work with every day has a natural structure to it. (See Book I, Chapter 1 for more on this topic.)

Suppose Gilligan's By The Sea wants to be able to work with their data more efficiently, storing the information in one place but using it to do a variety of tasks, such as outputting menus and table tents, updating an online menu on their Web site, and exchanging data with its sister restaurant Ginger & Mary Ann's. XML is seen as the solution, so the first responsibility is to convert the menu entrée list into an XML document structure.

1. Pick the organizing element that can serve as the container.

Because all entrées are part of the menu, the menu is a logical choice to serve as the container.

```
<menu>  
<menu>
```

2. Identify the second level of elements.

The entrée is the primary piece of data inside a menu and serves as the main element for the XML document. All the data inside the rows of the table naturally can be contained by an *entree* element. The structure now looks like:

```
<menu>
  <entree></entree>
</menu>
```

3. Determine the child elements and attributes.

Each of the columns in the table can potentially be a child element of the `entree` element or else an attribute. (Book I, Chapter 2 discusses the nuances between deciding whether attribute or child element is the best choice.) For this structure, I'm going to use the ID as an attribute of the `entree` element but classify the other columns as child elements. The updated structure is now:

```
<menu>
  <entree id="">
    <name></name>
    <features></features>
    <fatgrams></fatgrams>
    <description></description>
  </entree>
</menu>
```

The structure I have defined now adequately accounts for the entrée information that I have on hand, yet is extensible as needs change over time.

Step 2: Declaring an XML document

The preliminaries are now over; the time for creating the document can begin. Using Notepad or another editor, create a new text file and enter an XML declaration at the start of the file:



TIP

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

See Book I, Chapter 2 for more on XML declarations.

Step 3: Adding the document root

After the document has been designated an XML file using the XML declaration, you can add the root element, which is the element that contains all the other elements within the document.



REMEMBER

You can't have more than one root element defined or you'll get an error.

Looking back at Step 1, you can see that the `menu` element was selected to be the root element and so is entered here:

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

</menu>
```



The root element is often used as a location to define additional information pertaining to the XML document, such as namespaces (see Book I, Chapter 4) and XML schemas (see Book II, Chapter 6).

Step 4: Adding the structured data

The menu container element is now ready to house the entrée data. Using the `entree` element structure that was developed in Step 1, can add each of the entrees to the document:

```
<?xml version="1.0" encoding="utf-8" ?>
<menu>
  <entree id="1">
    <name>Sunburnt Chicken</name>
    <fatgrams>23</fatgrams>
    <features>Salad, Vegetables, Baked Potato, and
    Dessert</features>
    <description>Chicken prepared so hot, you'll need a
    gallon of soda to wash it down. Bring
    sunscreen!</description>
  </entree>
  <entree id="2">
    <name>Filet Mig's None</name>
    <fatgrams>0</fatgrams>
    <features>Soup, Vegetables, Baked Potato, and
    Dessert</features>
    <description>Our master chef Mig prepares a uniquely no-
    fat filet mignon. You won't believe how great it
    tastes!</description>
  </entree>
  <entree id="3">
    <name>Chicken Parmashaun</name>
    <fatgrams>20</fatgrams>
    <features>Soup, Pasta, Baked Potato, and
    Dessert</features>
    <description>Our award-winning Chicken Parmesan prepared
    especially for you by our master chef
    Shaun.</description>
  </entree>
  <entree id="4">
    <name>Eggs Benelux</name>
    <fatgrams>35</fatgrams>
    <features>Bacon, Sausage, and Toast</features>
    <description>No matter the time of day, enjoy our
    scrumptious breakfast cooked by our famous Belgian and
    Dutch master chefs.</description>
  </entree>
```

```
<entree id="5">
  <name>Jerk Chicken</name>
  <fatgrams>5</fatgrams>
  <features>Soup, Vegetables, and Dessert</features>
  <description>A delicious hot Jamaican dish prepared by
    our most obnoxious master chef.</description>
</entree>
<entree id="6">
  <name>Gusto Spaghetti</name>
  <fatgrams>55</fatgrams>
  <features>Soup, Salad, and Dessert</features>
  <description>Our famous master chef Boyd Ardee prepares a
    succulent dish of spaghetti with zesty
    gusto!</description>
</entree>
</menu>
```



For each successive level of the document hierarchy, I indent two spaces to make the document structure easier to read. However, this indentation is purely optional.

After you've entered the data, save the file as menu.xml.

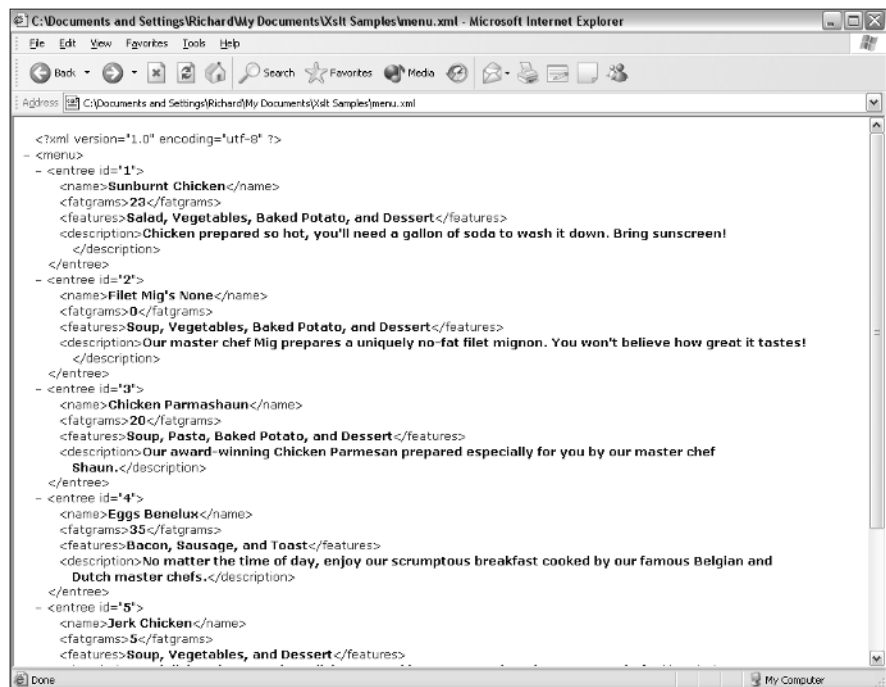
Step 5: Viewing the document

You can now view the results of the document in an XML document reader. No, you shouldn't need to download any new software to perform this task. Actually, the newer versions of Microsoft Internet Explorer and Netscape Navigator both have XML parsers built into them and display the XML document as a tree.

If you typed in the document as shown in the previous steps, the XML document is well-formed and will display properly by the document reader. To view, locate the menu.xml file you just saved in Windows Explorer and double-click it to open in your Web browser. Figure 3-3 shows the result when viewed in Internet Explorer 6.0.

However, because an XML document must be well-formed in order to be displayed, check out what happens when an error is in the document. Suppose I remove the menu element from the file so that I no longer have a single root element in the file. When I try to open up this incorrect document in Internet Explorer, the browser encounters the error during processing and then displays that error information inside its window instead of the document, as shown in Figure 3-4.

Figure 3-3:
Viewing an
XML
document in
Internet
Explorer.



The next steps

The menu.xml document is completed and viewable by an XML document reader. However, this basic XML document is often just the starting point to doing more. Therefore, consider jumping to one of the following parts of the book:

- ◆ To add an increased level of validation to the structure, see Book 2.
- ◆ To display this XML document in a presentable, formatted manner, see Book 3.
- ◆ To transform this document into HTML or a different XML structure, see Book 4.

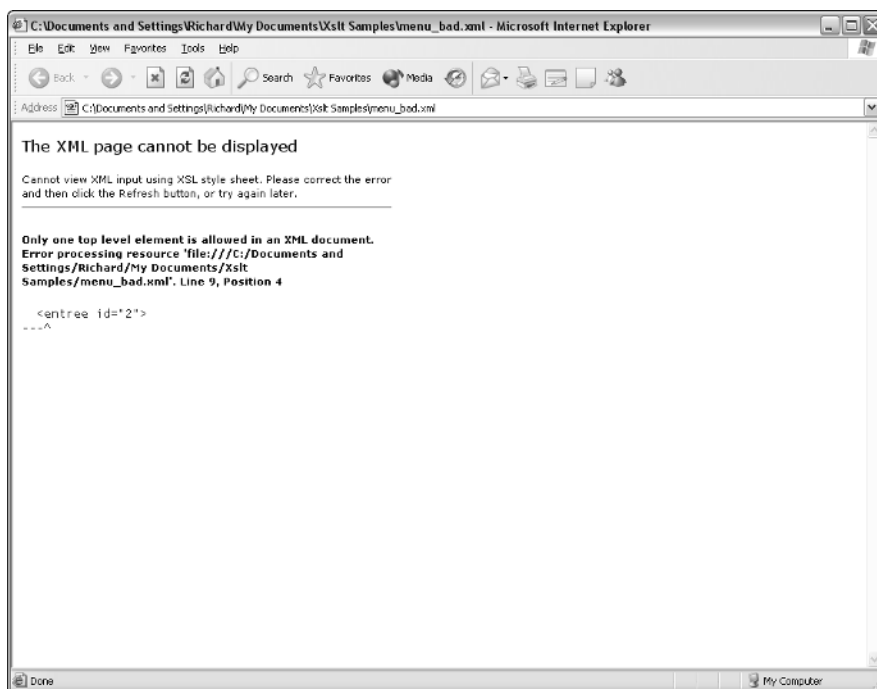


Figure 3-4:
An error is
displayed by
the browser
during
parsing of
document.

Chapter 4: Namespaces — Avoiding Naming Conflicts

In This Chapter

- ✓ Exploring the need for namespaces
- ✓ Declaring namespaces
- ✓ Finding out how attributes work with namespaces
- ✓ Using multiple namespaces
- ✓ Overriding namespace declarations
- ✓ Undeclaring namespaces

In a classic *Seinfeld* episode, George Costanza becomes frantic when his girlfriend wants to meet his sidekick friends, Jerry, Kramer, and Elaine. Up until that point, his life was neatly compacted into two separate worlds, which he termed “Relationship George” and “Independent George.” But if these two worlds are allowed to collide, then something earth-shattering will happen: Independent George will cease to exist. Or, in the memorable words of George, “A George divided against itself cannot stand!”

George’s dilemma humorously illustrates a fundamental problem that XML has. One of XML’s major advantages is that it is completely extensible, enabling you to create your own tags to describe data in a manner that works best for you. The problem becomes when you use the same name for an element as someone else used for his. Therefore, when you try to combine XML elements together from different worlds, the clashing names can cause a Costanzaesque accident. After all, an XML document divided against cannot stand!

While not helping out George’s condition, namespaces are the solution around naming conflicts in XML. In this chapter, you discover what namespaces are and why they’re necessary. I show you how to declare namespaces in your XML documents, how to combine multiple namespaces in a single document, and how to override and undeclare namespaces.



Even if you don’t plan on sharing data outside your organization, you still need to know how to work with namespaces because many XML technologies (such as XML Schemas, XSLT, XLink, and XForms) make heavy use of them. Without namespaces being used, for example, it would be impossible to tell if you wanted to use `<element>` to describe an XML element in an XML schema or refer to an element in a chemistry periodic table.

Keeping Your Distance with Namespaces

XML uses *namespaces* to distinguish a set of element names from another group of names. The need for namespaces becomes obvious when you consider the flexibility of the XML language. Unlike HTML, XML has no pre-defined set of element names. As a result, the naming of elements and attributes is left entirely up to the one writing the XML documents. While the flexibility is critical to meeting the purposes of XML, it does leave open one potential problem: the possibility of two or more developers using an identical element name.

Consider, for example, DirecTV develops an XML vocabulary using `dish` as an element to describe those space-age ornaments that hug houses in suburbia. For example:

```
<dish>
  <model>A32201-WZ-TURBO</model>
  <size>2.0 x 2.0 x .75</size>
  <weight>27 lbs.</weight>
</dish>
```

Meanwhile, suppose Pfaltzgraff has their own `dish` element to describe those round things that you eat off of:

```
<dish>
  <model>Ashton</model>
  <shape>Hexagon</shape>
  <weight>14 ounces</weight>
</dish>
```

If DirecTV and Pfaltzgraff never exchange data outside of their respective companies, then they could use their XML vocabulary as is. But suppose they both wish to share data with outside suppliers and retailers, who in turn tried to combine them into a common data pool. If this were to happen, the chances of a DirecTV `dish` colliding with a Pfaltzgraff `dish` is very high:

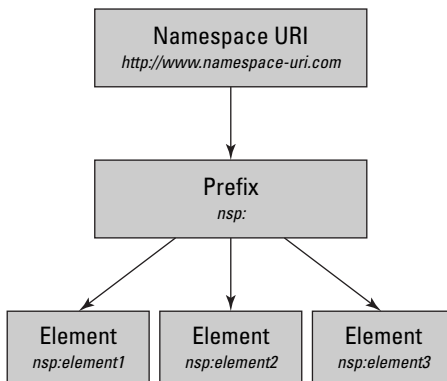
```
<!-- dish elements below have differing structures -->
<supplierorder>
  <dish>
    <model>A32201-WZ-TURBO</model>
    <size>2.0 x 2.0 x .75</size>
    <weight>27 lbs.</weight>
  </dish>
  <dish>
    <model>Ashton</model>
    <shape>Hexagon</shape>
    <weight>14 ounces</weight>
  </dish>
</supplierorder>
```

The end result of this collision is that a customer gets a dinner plate hung on their roof or that a satellite dish becomes the new avant-garde way to serve pizza.

Namespaces were introduced as a way to avoid this name collision by linking a collection of names with a URI (Uniform Resource Identifier). A *URI* is the method in which resources on the Web are named, the most widespread form being URLs (Uniform Resource Locator) or Web addresses (for example, `http://www.dummies.com`). Because URIs are unique, you can be sure that the namespace associated with the URI is one of a kind.

When you define a namespace, you usually declare a URI once in a document and then refer to it elsewhere in the document by using a *namespace prefix* (also known as *namespace identifier* or *abbreviation*), as shown in Figure 4-1. This namespace identifier is used as a prefix onto element and/or attribute names.

Figure 4-1:
A URI is
linked to a
namespace
identifier.



Understanding URIs

While URI sounds like the name of a former Soviet leader in the early 1980s, the acronym actually is a common term used for all types of names and addresses that access the resources on the Web. URI (pronounced *you-are-I*) is a superset of two types of identifiers:

✓ **Uniform Resource Locator (URL).** URL identifies the location of a resource. An example of a URL is `http://www.dummies.com`.

✓ **Uniform Resource Name (URN).** You can also use a URN to identify the location of a resource and retrieve the resource. A URN is more robust than a URL, and it can refer to more than one URL. A URN is written in the form `urn:unique-string`. For example, you can specify a URN as `urn:dummies.xml.allinone.2003` and use it as a namespace URI for all documents relating to this book.

(continued)

(continued)

A namespace uses a URI as a *string* to uniquely identify a group of elements and attributes. However, even though they look the same, don't confuse a namespace URI with a Web address that you can actually visit with your browser. A namespace URI doesn't actually point to a resource on the Internet. It is just a unique identifier that's used to resolve conflicts between elements that have the same name.

For example, XSLT (discussed in Book IV) uses the `http://www.w3.org/1999/XSL/Transform` URI to identify its namespace, as shown in the following code:

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

During processing, the XSLT processor doesn't try to look up a data definition resource at `http://www.w3.org/1999/XSL/Transform`. Instead, an XSLT-enabled parser knows that this unique string identifies a namespace reserved for XSLT.

You can assign any name or string as a URI so long as it conforms to the URL or URN syntax requirements. However, referring to the following guidelines helps you to ensure uniqueness of the URI:

- ✓ **Use a URI that you control.** In order to ensure the uniqueness of the URI, the recommended strategy is to use a domain name that is under the developer's control. For example, if `www.notathing.com` is your Web address, you can feel confident that using a namespace URI, such as `http://www.notathing.com/spec/2003`, ensures that the specified URI is unique.
- ✓ **Avoid using a relative URI.** A relative URI is one that refers to a location that is relative to the current location or directory. Although there is no restriction on the use of relative URIs, it is recommended you use a *fully qualified name* (complete address of the resource) to specify a URI.

Declaring Namespaces

Namespaces are declared by adding an `xmlns` attribute (or another name that uses an `xmlns: prefix`) to the outermost element in which you want to use the namespace. The syntax for explicitly declaring a namespace is:

```
<myelement xmlns:prefix="URI">
</myelement>
```

In the preceding syntax,

- ◆ `xmlns` is a reserved attribute or prefix used to declare namespaces.
- ◆ `prefix` is the optional alias for the namespace.
- ◆ `URI` is a unique string used to identify a namespace.

What's in a name?

When you are adding your own namespace prefix, keep in mind the following naming do's and don'ts:

- ✓ DO use any legal XML name. A legal XML name must begin with a letter or an underscore (_). Subsequent letters may include letters, digits, underscores, or periods (.).
- ✓ DO NOT use any name that uses `xml` or `xmlns` in it. These are reserved for XML-related namespaces.
- ✓ DO NOT use any name that resembles any of the existing XML-related namespace prefixes, such as `xsl` or `xlink`. See the "XML world loves namespaces" sidebar later in the chapter for a list of these prefix conventions.

You can also define a namespace without using the optional prefix. To do so, see the "Declaring a Default Namespaces" section.

For example, suppose I want to add namespaces to the two `dish` structures described in the previous section to ensure that they were treated as separate element types. For the DirecTV `dish` element, I use the following namespace declaration:

```
xmlns:dtv="http://www.directv.com/2003/dish"
```

The `dtv` is the namespace prefix that is bound to the `http://www.directv.com/2003/dish` URI. I add this declaration to the `dish` element and then prefix each element name with the prefix to associate these elements with the declared namespace:

```
<dtv:dish xmlns:dtv="http://www.directv.com/2003/dish">
  <dtv:model>A32201-WZ-TURBO</dtv:model>
  <dtv:size>2.0 x 2.0 x .75</dtv:size>
  <dtv:weight>27 lbs.</dtv:weight>
</dtv:dish>
```

For the Pfaltzgraf `dish` element, I use a different namespace declaration and add it to the structure:

```
<pf:dish xmlns:pf="http://www.pfaltzgraf.com/dish">
  <pf:model>Ashton</pf:model>
  <pf:shape>Hexagon</pf:shape>
  <pf:weight>14 ounces</pf:weight>
</pf:dish>
```

When you declare a namespace, the element in which you define it dictates the namespace's scope. A namespace can be declared at the root level of a document or it can be declared inside any element of the XML structure.

The XML world loves namespaces

When you venture beyond basic XML and begin working with its related technologies, such as XSLT, XLink, and XForms, you'll see the

widespread usage of common namespace prefixes. Here are the common namespace prefixes for XML technologies.

Technology	Common Prefix	Namespace URI
XSLT	Xsl	http://www.w3.org/1999/XSL/Transform
XLink	Xlink	http://www.w3.org/1999/xlink
XForms	Xforms	http://www.w3.org/2002/xforms/cr
XHTML	Xhtml	http://www.w3.org/2002/06/xhtml12
XML Schema	xs or xsd	http://www.w3.org/2001/XMLSchema
XML Events	Ev	http://www.w3.org/2001/xml-events

However, a namespace is visible to only the element in which it is declared and all of the child elements of that container element. Therefore, in the examples above, the `dtv` namespace is available by DirecTV's `dish` element and its child elements, while the `pf` namespace is available by Pfatzgraff's `dish` element and its children.



The binding of the namespace prefix to the URI is what is important, not the literal namespace prefix. The prefix can actually be any label you choose. For example, the following code snippet shows another valid namespace option:

```
<!-- awesomedishnamespace used as namespace prefix -->
<awesomedishnamespace:dish
  xmlns:awesomedishnamespace="http://www.directv.com/2003/dish" >
  <awesomedishnamespace:model>A32201-WZ-
    TURBO</awesomedishnamespace:model>
  <awesomedishnamespace:size>2.0 x 2.0 x
    .75</awesomedishnamespace:size>
  <awesomedishnamespace:weight>27
    lbs.</awesomedishnamespace:weight>
</awesomedishnamespace:dish>
```

Declaring a Default Namespace

In addition to binding a prefix to a namespace, you can declare a default namespace as well. A *default namespace* applies a namespace to the element

in which it is declared and all of its child elements and does so without adding a prefix before the element name.

To declare a default namespace, add a `xmlns` attribute to your element rather than a `xmlns:prefix` attribute:

```
<myelement xmlns="URI">  
</myelement>
```

For example, the following code snippet defines `http://www.directv.com/2003/dish` as the namespace for `dish` and all of its child elements.

```
<dish xmlns="http://www.directv.com/2003/dish">  
  <model>A32201-WZ-TURBO</model>  
  <size>2.0 x 2.0 x .75</size>  
  <weight>27 lbs.</weight>  
</dish>
```



Default namespaces don't apply directly to attributes. See the "How Namespaces Apply to Attributes" section for details.



The default namespace doesn't override namespace prefixes inside its scope. Elements with existing prefix namespaces preserve their namespaces. For example, consider the following `dish` element:

```
<dish xmlns="http://www.directv.com/2003/dish"  
      xmlns:pic="http://www.directv.com/pics">  
  <model>A32201-WZ-TURBO</model>  
  <size>2.0 x 2.0 x .75</size>  
  <weight>27 lbs.</weight>  
  <pic:graphic>turbo.jpg</pic:graphic>  
</dish>
```

As you can see, both a default namespace and an explicitly named namespace were defined in the `dish` element. Because `model`, `size`, and `weight` have no prefix, they are bound to the default namespace (`http://www.directv.com/2003/dish`), while the `pic:graphic` element is bound to the `http://www.directv.com/pics` namespace.



Default declarations are useful for long documents that have a lot of markup. For such documents, you may find it inconvenient to add the namespace prefix with each element name.

How Namespaces Apply to Attributes

When you explicitly define a namespace and append its prefix on element names, the attributes that are contained within the element are "piggybacked"

on the element, and so are automatically assigned the same namespace. For example, consider the following `dish` element:

```
<dtv:dish id="1029-2a"
  xmlns:dtv="http://www.directv.com/2003/dish">
  <dtv:model year="2003">A32201-WZ-TURBO</dtv:model>
  <dtv:size>2.0 x 2.0 x .75</dtv:size>
  <dtv:weight type="pounds">27 lbs.</dtv:weight>
</dtv:dish>
```

In this example, the `id`, `year`, and `type` attributes are all considered to be part of the `http://www.directv.com/2003/dish` namespace because their containing elements all have the `dtv` prefix. This assumption saves you from the need for adding prefixes to both the element and attribute names. While the code snippet below is functionally equivalent to that shown above, it is messier to work with, especially with complex XML structures:

```
<dtv:dish dtv:id="1029-2a"
  xmlns:dtv="http://www.directv.com/2003/dish">
  <dtv:model dtv:year="2003">A32201-WZ-TURBO</dtv:model>
  <dtv:size>2.0 x 2.0 x .75</dtv:size>
  <dtv:weight dtv:type="pounds">27 lbs.</dtv:weight>
</dtv:dish>
```

However, the “piggybacking” effect does not apply to default namespaces. Therefore, in the case below in which I declare a default namespace, the `id`, `year`, and `type` attributes are *not* considered to be part of the `http://www.directv.com/2003/dish` namespace:

```
<dish id="1029-2a"
  xmlns="http://www.directv.com/2003/dish">
  <model year="2003">A32201-WZ-TURBO</model>
  <size>2.0 x 2.0 x .75</size>
  <weight type="pounds">27 lbs.</weight>
</dish>
```

Therefore, using a default namespaces yet needing to apply a namespace to those attributes, you’d have to declare an explicit namespace and bind to the attributes:

```
<dish dtv:id="1029-2a"
  xmlns="http://www.directv.com/2003/dish"
  xmlns:dtv="http://www.directv.com/2003/dish">
  <model dtv:year="2003">A32201-WZ-TURBO</model>
  <size>2.0 x 2.0 x .75</size>
  <weight dtv:type="pounds">27 lbs.</weight>
</dish>
```

Using Multiple Namespaces

Explicitly defined namespace declarations may seem like a pain to work with because you need to add the prefix with each and every element that you want to bind to the namespace. If you are always working within the same namespace, then default namespaces are much easier to use. However, prefix usage becomes critical when you start working with multiple namespaces within the same document.

Multiple namespaces can then be combined inside a single XML document, enabling you, for example, to distinguish between the two distinct `dish` elements:

```
<supplierorder
  xmlns:dtv="http://www.directv.com/2003/dish"
  xmlns:pf="http://www.pfaltzgraf.com/dish">
  <dtv:dish>
    <dtv:model>A32201-WZ-TURBO</dtv:model>
    <dtv:size>2.0 x 2.0 x .75</dtv:size>
    <dtv:weight>27 lbs.</dtv:weight>
  </dtv:dish>
  <pf:dish>
    <pf:model>Ashton</pf:model>
    <pf:shape>Hexagon</pf:shape>
    <pf:weight>14 ounces</pf:weight>
  </pf:dish>
</supplierorder>
```

Because `dtv` and `pf` are declared at the root element level, they can be accessed anywhere inside of the document.

Overriding Namespace Declarations

When you are working with multiple namespaces at various levels of an XML document, you may have an element that needs to be bound to a different namespace that is the default within its scope. In XML, you can override the namespace declaration by declaring another default namespace. In this section, you find out how to override both default and explicit namespace declarations.

Overriding default namespace declarations

You can override a default namespace declaration in an XML document by simply declaring another default namespace, which automatically replaces the initial declaration within the scope of the second declaration. Consider

the following example of XML code that overrides the default namespace declaration that refers to the `Books.dtd` with the namespace that refers to the `BookOrder.dtd`. In this case, the default namespace has been overridden to identify the `Quantity` element as belonging to the `BookOrder.dtd` DTD.

```
<supplierorder xmlns="http://www.supplierme.com">
  <dish xmlns="http://www.directv.com/2003/dish">
    <model>A32201-WZ-TURBO</model>
    <size>2.0 x 2.0 x .75</size>
    <weight>27 lbs.</weight>
  </dish>
  <dish xmlns="http://www.pfaltzgraf.com/dish">
    <model>Ashton</model>
    <shape>Hexagon</shape>
    <weight>14 ounces</weight>
  </dish>
</supplierorder>
```

Overriding explicit namespace declarations

While a far less common situation, you can also override an explicitly defined namespace by re-declaring the namespace prefix to a new URI. For example, the `ds` prefix is initially defined to the `http://www.directv.com/2003/dish` namespace and then when it gets to the second dish element, it is reassigned to the `http://www.pfaltzgraf.com/dish` namespace:

```
<supplierorder xmlns:ds="http://www.directv.com/2003/dish">
  <ds:dish>
    <ds:model>A32201-WZ-TURBO</ds:model>
    <ds:size>2.0 x 2.0 x .75</ds:size>
    <ds:weight>27 lbs.</ds:weight>
  </ds:dish>
  <ds:dish xmlns:pf="http://www.pfaltzgraf.com/dish">
    <ds:model>Ashton</ds:model>
    <ds:shape>Hexagon</ds:shape>
    <ds:weight>14 ounces</ds:weight>
  </ds:dish>
</supplierorder>
```

Undeclaring Namespaces

After you declare a namespace, the namespace declaration remains in scope until the last descendant (unless the namespace declaration is overridden) in a document. The last descendant is the inner-most child element in an XML document. However, sometimes, you may not need to bind certain descendants with the declared namespace in its scope. For example, some elements in an XML document might not be referenced to in a namespace. Therefore, you may need to *undeclare* the namespace, which means you need to remove certain elements from the scope of the namespace.

An explicit namespace cannot be undeclared. However, you can undeclare a default namespace. To undeclare a default namespace, you simply need to declare the default namespace with an empty name (zero-length URI). For example, in the code snippet below, the `http://www.directv.com/2003/dish` namespace is declared as the default namespace and would normally be in effect until the closing `dish` element. However, the `size` element declares an empty namespace, which makes both the `size` and `weight` elements to not be considered part of any namespace:

```
<dish xmlns="http://www.directv.com/2003/dish">
  <model>A32201-WZ-TURBO</model>
  <size xmlns="">2.0 x 2.0 x .75</size>
  <weight>27 lbs.</weight>
</dish>
```

