2

Understanding XML

THIS chapter describes Extensible Markup Language (XML) and its related specifications. It also gives you practice in writing XML data so that you can become comfortably familiar with XML syntax.

Note: The XML files mentioned in this chapter can be found in `<INSTALL>/j2eetutorial14/examples/xml/samples/`.

Introduction to XML

This section covers the basics of XML. The goal is to give you just enough information to get started so that you understand what XML is all about. (You'll learn more about XML in later sections of the tutorial.) We then outline the major features that make XML great for information storage and interchange, and give you a general idea of how XML can be used.

What Is XML?

XML is a text-based markup language that is fast becoming the standard for data interchange on the web. As with HTML, you identify data using *tags* (identifiers enclosed in angle brackets: <. . .>). Collectively, the tags are known as markup.

But unlike HTML, XML tags *identify* the data rather than specify how to display it. Whereas an HTML tag says something like, "Display this data in bold font"

(...), an XML tag acts like a field name in your program. It puts a label on a piece of data that identifies it (for example, <message>...</message>).

Note: Because identifying the data gives you some sense of what it *means* (how to interpret it, what you should do with it), XML is sometimes described as a mechanism for specifying the *semantics* (meaning) of the data.

In the same way that you define the field names for a data structure, you are free to use any XML tags that make sense for a given application. Naturally, for multiple applications to use the same XML data, they must agree on the tag names they intend to use.

Here is an example of some XML data you might use for a messaging application:

```
<message>
  <to>you@yourAddress.com</to>
  <from>me@myAddress.com</from>
  <subject>XML Is Really Cool</subject>
  <text>
    How many ways is XML cool? Let me count the ways...
  </text>
</message>
```

Note: Throughout this tutorial, we use boldface text to highlight things we want to bring to your attention. XML does not require anything to be in bold!

The tags in this example identify the message as a whole, the destination and sender addresses, the subject, and the text of the message. As in HTML, the <to> tag has a matching end tag: </to>. The data between the tag and its matching end tag defines an element of the XML data. Note, too, that the content of the <to> tag is contained entirely within the scope of the <message>..</message> tag. It is this ability for one tag to contain others that lets XML represent hierarchical data structures.

Again, as with HTML, whitespace is essentially irrelevant, so you can format the data for readability and yet still process it easily with a program. Unlike HTML, however, in XML you can easily search a data set for messages containing, say, “cool” in the subject, because the XML tags identify the content of the data rather than specify its representation.

Tags and Attributes

Tags can also contain attributes—additional information included as part of the tag itself, within the tag's angle brackets. The following example shows an email message structure that uses attributes for the `to`, `from`, and `subject` fields:

```
<message to="you@yourAddress.com" from="me@myAddress.com"
     subject="XML Is Really Cool">
  <text>
    How many ways is XML cool? Let me count the ways...
  </text>
</message>
```

As in HTML, the attribute name is followed by an equal sign and the attribute value, and multiple attributes are separated by spaces. Unlike HTML, however, in XML commas between attributes are not ignored; if present, they generate an error.

Because you can design a data structure such as `<message>` equally well using either attributes or tags, it can take a considerable amount of thought to figure out which design is best for your purposes. Designing an XML Data Structure (page 76), includes ideas to help you decide when to use attributes and when to use tags.

Empty Tags

One big difference between XML and HTML is that an XML document is always constrained to be *well formed*. There are several rules that determine when a document is well formed, but one of the most important is that every tag has a closing tag. So, in XML, the `</to>` tag is not optional. The `<to>` element is never terminated by any tag other than `</to>`.

Note: Another important aspect of a well-formed document is that all tags are completely nested. So you can have `<message>..<to>..</to>..</message>`, but never `<message>..<to>..</message>..</to>`. A complete list of requirements is contained in the list of XML frequently asked questions (FAQ) at `http://www.ucc.ie/xml/#FAQ-VALIDWF`. (This FAQ is on the W3C "Recommended Reading" list at `http://www.w3.org/XML/`.)

Sometimes, though, it makes sense to have a tag that stands by itself. For example, you might want to add a tag that flags the message as important: `<flag/>`.

This kind of tag does not enclose any content, so it's known as an *empty* tag. You create an empty tag by ending it with /> instead of >. For example, the following message contains an empty flag tag:

```
<message to="you@yourAddress.com" from="me@myAddress.com"
     subject="XML Is Really Cool">
  <flag/>
  <text>
    How many ways is XML cool? Let me count the ways...
  </text>
</message>
```

Note: Using the empty tag saves you from having to code <flag></flag> in order to have a well-formed document. You can control which tags are allowed to be empty by creating a schema or a document type definition, or DTD (page 1390). If there is no DTD or schema associated with the document, then it can contain any kinds of tags you want, as long as the document is well formed.

Comments in XML Files

XML comments look just like HTML comments:

```
<message to="you@yourAddress.com" from="me@myAddress.com"
     subject="XML Is Really Cool">
  <!-- This is a comment -->
  <text>
    How many ways is XML cool? Let me count the ways...
  </text>
</message>
```

The XML Prolog

To complete this basic introduction to XML, note that an XML file always starts with a *prolog*. The minimal prolog contains a declaration that identifies the document as an XML document:

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

The declaration may also contain additional information:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
```

The XML declaration is essentially the same as the HTML header, `<html>`, except that it uses `<?..?>` and it may contain the following attributes:

- `version`: Identifies the version of the XML markup language used in the data. This attribute is not optional.
- `encoding`: Identifies the character set used to encode the data. `ISO-8859-1` is `Latin-1`, the Western European and English language character set. (The default is 8-bit Unicode: `UTF-8`.)
- `standalone`: Tells whether or not this document references an external entity or an external data type specification. If there are no external references, then "yes" is appropriate.

The prolog can also contain definitions of *entities* (items that are inserted when you reference them from within the document) and specifications that tell which tags are valid in the document. Both declared in a document type definition (DTD, page 1390) that can be defined directly within the prolog, as well as with pointers to external specification files. But those are the subject of later tutorials. For more information on these and many other aspects of XML, see the Recommended Reading list on the W3C XML page at `http://www.w3.org/XML/`.

Note: The declaration is actually optional, but it's a good idea to include it whenever you create an XML file. The declaration should have the version number, at a minimum, and ideally the encoding as well. That standard simplifies things if the XML standard is extended in the future and if the data ever needs to be localized for different geographical regions.

Everything that comes after the XML prolog constitutes the document's *content*.

Processing Instructions

An XML file can also contain *processing instructions* that give commands or information to an application that is processing the XML data. Processing instructions have the following format:

```
<?target instructions?>
```

`target` is the name of the application that is expected to do the processing, and *`instructions`* is a string of characters that embodies the information or commands for the application to process.

Because the instructions are application-specific, an XML file can have multiple processing instructions that tell different applications to do similar things, although in different ways. The XML file for a slide show, for example, might have processing instructions that let the speaker specify a technical- or executive-level version of the presentation. If multiple presentation programs were used, the program might need multiple versions of the processing instructions (although it would be nicer if such applications recognized standard instructions).

Note: The target name "xml" (in any combination of upper- or lowercase letters) is reserved for XML standards. In one sense, the declaration is a processing instruction that fits that standard. (However, when you're working with the parser later, you'll see that the method for handling processing instructions never sees the declaration.)

Why Is XML Important?

There are a number of reasons for XML's surging acceptance. This section lists a few of the most prominent.

Plain Text

Because XML is not a binary format, you can create and edit files using anything from a standard text editor to a visual development environment. That makes it easy to debug your programs, and it makes XML useful for storing small amounts of data. At the other end of the spectrum, an XML front end to a database makes it possible to efficiently store large amounts of XML data as well. So XML provides scalability for anything from small configuration files to a company wide data repository.

Data Identification

XML tells you what kind of data you have, not how to display it. Because the markup tags identify the information and break the data into parts, an email program can process it, a search program can look for messages sent to particular people, and an address book can extract the address information from the rest of the message. In short, because the different parts of the information have been identified, they can be used in different ways by different applications.

Stylability

When display is important, the stylesheet standard, XSL (page 1391), lets you dictate how to portray the data. For example, consider this XML:

```
<to>you@yourAddress.com</to>
```

The stylesheet for this data can say

1. Start a new line.
2. Display "To:" in bold, followed by a space
3. Display the destination data.

This set of instructions produces:

```
To: you@yourAddress
```

Of course, you could have done the same thing in HTML, but you wouldn't be able to process the data with search programs and address-extraction programs and the like. More importantly, because XML is inherently style-free, you can use a completely different stylesheet to produce output in Postscript, TEX, PDF, or some new format that hasn't even been invented. That flexibility amounts to what one author described as "future proofing" your information. The XML documents you author today can be used in future document-delivery systems that haven't even been imagined.

Inline Reusability

One of the nicer aspects of XML documents is that they can be composed from separate entities. You can do that with HTML, but only by linking to other documents. Unlike HTML, XML entities can be included "inline" in a document. The included sections look like a normal part of the document: you can search the whole document at one time or download it in one piece. That lets you modularize your documents without resorting to links. You can single-source a section so that an edit to it is reflected everywhere the section is used, and yet a document composed from such pieces looks for all the world like a one-piece document.

Linkability

Thanks to HTML, the ability to define links between documents is now regarded as a necessity. Appendix B discusses the link-specification initiative. This initia-

tive lets you define two-way links, multiple-target links, expanding links (where clicking a link causes the targeted information to appear inline), and links between two existing documents that are defined in a third.

Easily Processed

As mentioned earlier, regular and consistent notation makes it easier to build a program to process XML data. For example, in HTML a <dt> tag can be delimited by </dt>, another <dt>, <dd>, or </dl>. That makes for some difficult programming. But in XML, the <dt> tag must always have a </dt> terminator, or it must be an empty tag such as <dt/>. That restriction is a critical part of the constraints that make an XML document well formed. (Otherwise, the XML parser won't be able to read the data.) And because XML is a vendor-neutral standard, you can choose among several XML parsers, any one of which takes the work out of processing XML data.

Hierarchical

Finally, XML documents benefit from their hierarchical structure. Hierarchical document structures are, in general, faster to access because you can drill down to the part you need, as if you were stepping through a table of contents. They are also easier to rearrange, because each piece is delimited. In a document, for example, you could move a heading to a new location and drag everything under it along with the heading, instead of having to page down to make a selection, cut, and then paste the selection into a new location.

How Can You Use XML?

There are several basic ways to use XML:

- Traditional data processing, where XML encodes the data for a program to process
- Document-driven programming, where XML documents are containers that build interfaces and applications from existing components
- Archiving—the foundation for document-driven programming—where the customized version of a component is saved (archived) so that it can be used later

- Binding, where the DTD or schema that defines an XML data structure is used to automatically generate a significant portion of the application that will eventually process that data

Traditional Data Processing

XML is fast becoming the data representation of choice for the web. It's terrific when used in conjunction with network-centric Java platform programs that send and retrieve information. So a client-server application, for example, could transmit XML-encoded data back and forth between the client and the server.

In the future, XML is potentially the answer for data interchange in all sorts of transactions, as long as both sides agree on the markup to use. (For example, should an email program expect to see tags named `<FIRST>` and `<LAST>`, or `<FIRSTNAME>` and `<LASTNAME>`?) The need for common standards will generate a lot of industry-specific standardization efforts in the years ahead. In the meantime, mechanisms that let you "translate" the tags in an XML document will be important. Such mechanisms include projects such as the Resource Description Framework initiative (RDF, page 1395), which defines meta tags, and the Extensible Stylesheet Language specification (XSL, page 1391), which lets you translate XML tags into other XML tags.

Document-Driven Programming

The newest approach to using XML is to construct a document that describes what an application page should look like. The document, rather than simply being displayed, consists of references to user interface components and business-logic components that are "hooked together" to create an application on-the-fly.

Of course, it makes sense to use the Java platform for such components. To construct such applications, you can use JavaBeans components for interfaces and Enterprise JavaBeans components for the business logic. Although none of the efforts undertaken so far is ready for commercial use, much preliminary work has been done.

Note: The Java programming language is also excellent for writing XML-processing tools that are as portable as XML. Several visual XML editors have been written for the Java platform. For a listing of editors, see `http://www.xml.com/pub/pt/3`.

For processing tools and other XML resources, see Robin Cover's SGML/XML web page at `http://xml.coverpages.org/software.html`.

Binding

After you have defined the structure of XML data using either a DTD or one of the schema standards, a large part of the processing you need to do has already been defined. For example, if the schema says that the text data in a `<date>` element must follow one of the recognized date formats, then one aspect of the validation criteria for the data has been defined; it only remains to write the code. Although a DTD specification cannot go the same level of detail, a DTD (like a schema) provides a grammar that tells which data structures can occur and in what sequences. That specification tells you how to write the high-level code that processes the data elements.

But when the data structure (and possibly format) is fully specified, the code you need to process it can just as easily be generated automatically. That process is known as *binding*—creating classes that recognize and process different data elements by processing the specification that defines those elements. As time goes on, you should find that you are using the data specification to generate significant chunks of code, and you can focus on the programming that is unique to your application.

Archiving

The Holy Grail of programming is the construction of reusable, modular components. Ideally, you'd like to take them off the shelf, customize them, and plug them together to construct an application, with a bare minimum of additional coding and additional compilation.

The basic mechanism for saving information is called *archiving*. You archive a component by writing it to an output stream in a form that you can reuse later. You can then read it and instantiate it using its saved parameters. (For example, if you saved a table component, its parameters might be the number of rows and columns to display.) Archived components can also be shuffled around the web and used in a variety of ways.

When components are archived in binary form, however, there are some limitations on the kinds of changes you can make to the underlying classes if you want to retain compatibility with previously saved versions. If you could modify the archived version to reflect the change, that would solve the problem. But that's

hard to do with a binary object. Such considerations have prompted a number of investigations into using XML for archiving. But if an object's state were archived in text form using XML, then anything and everything in it could be changed as easily as you can say, "Search and replace."

XML's text-based format could also make it easier to transfer objects between applications written in different languages. For all these reasons, there is a lot of interest in XML-based archiving.

Summary

XML is pretty simple and very flexible. It has many uses yet to be discovered, and we are only beginning to scratch the surface of its potential. It is the foundation for a great many standards yet to come, providing a common language that different computer systems can use to exchange data with one another. As each industry group comes up with standards for what it wants to say, computers will begin to link to each other in ways previously unimaginable.

Generating XML Data

This section takes you step by step through the process of constructing an XML document. Along the way, you'll gain experience with the XML components you'll typically use to create your data structures.

Writing a Simple XML File

You'll start by writing the kind of XML data you can use for a slide presentation. To become comfortable with the basic format of an XML file, you'll use your text editor to create the data. You'll use this file and extend it in later exercises.

Creating the File

Using a standard text editor, create a file called `slideSample.xml`.

Note: Here is a version of it that already exists: `slideSample01.xml`. (The browsable version is `slideSample01-xml.html`.) You can use this version to compare your work or just review it as you read this guide.

Writing the Declaration

Next, write the *declaration*, which identifies the file as an XML document. The declaration starts with the characters <?, which is also the standard XML identifier for a *processing instruction*. (You'll see processing instructions later in this tutorial.)

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

This line identifies the document as an XML document that conforms to version 1.0 of the XML specification and says that it uses the 8-bit Unicode character-encoding scheme. (For information on encoding schemes, see Appendix A.)

Because the document has not been specified as `standalone`, the parser assumes that it may contain references to other documents. To see how to specify a document as `standalone`, see The XML Prolog (page 36).

Adding a Comment

Comments are ignored by XML parsers. A program will never see them unless you activate special settings in the parser. To put a comment into the file, add the following highlighted text.

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

<!-- A SAMPLE set of slides -->
```

Defining the Root Element

After the declaration, every XML file defines exactly one element, known as the *root element*. Any other elements in the file are contained within that element. Enter the following highlighted text to define the root element for this file, `slideshow`:

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

<!-- A SAMPLE set of slides -->

<slideshow>

</slideshow>
```

Note: XML element names are case-sensitive. The end tag must exactly match the start tag.

Adding Attributes to an Element

A slide presentation has a number of associated data items, none of which requires any structure. So it is natural to define these data items as attributes of the `slideshow` element. Add the following highlighted text to set up some attributes:

```
...
  <slideshow
     title="Sample Slide Show"
     date="Date of publication"
     author="Yours Truly"
     >
  </slideshow>
```

When you create a name for a tag or an attribute, you can use hyphens (`-`), underscores (`_`), colons (`:`), and periods (`.`) in addition to characters and numbers. Unlike HTML, values for XML attributes are always in quotation marks, and multiple attributes are never separated by commas.

Note: Colons should be used with care or avoided, because they are used when defining the namespace for an XML document.

Adding Nested Elements

XML allows for hierarchically structured data, which means that an element can contain other elements. Add the following highlighted text to define a slide element and a title element contained within it:

```
<slideshow
   ...
   >

    <!-- TITLE SLIDE -->
   <slide type="all">
```

```
    <title>Wake up to WonderWidgets!</title>
  </slide>

</slideshow>
```

Here you have also added a `type` attribute to the slide. The idea of this attribute is that you can earmark slides for a mostly technical or mostly executive audience using `type="tech"` or `type="exec"`, or identify them as suitable for both audiences using `type="all"`.

More importantly, this example illustrates the difference between things that are more usefully defined as elements (the `title` element) and things that are more suitable as attributes (the `type` attribute). The visibility heuristic is primarily at work here. The title is something the audience will see, so it is an element. The type, on the other hand, is something that never gets presented, so it is an attribute. Another way to think about that distinction is that an element is a container, like a bottle. The type is a characteristic of the *container* (tall or short, wide or narrow). The title is a characteristic of the *contents* (water, milk, or tea). These are not hard-and-fast rules, of course, but they can help when you design your own XML structures.

Adding HTML-Style Text

Because XML lets you define any tags you want, it makes sense to define a set of tags that look like HTML. In fact, the XHTML standard does exactly that. You'll see more about that toward the end of the SAX tutorial. For now, type the following highlighted text to define a slide with a couple of list item entries that use an HTML-style `<em>` tag for emphasis (usually rendered as italicized text):

```
...
<!-- TITLE SLIDE -->
<slide type="all">
  <title>Wake up to WonderWidgets!</title>
</slide>

<!-- OVERVIEW -->
<slide type="all">
  <title>Overview</title>
  <item>Why <em>WonderWidgets</em> are great</item>
  <item>Who <em>buys</em> WonderWidgets</item>
</slide>

</slideshow>
```

Note that defining a *title* element conflicts with the XHTML element that uses the same name. Later in this tutorial, we discuss the mechanism that produces the conflict (the DTD), along with possible solutions.

Adding an Empty Element

One major difference between HTML and XML is that all XML must be well formed, which means that every tag must have an ending tag or be an empty tag. By now, you're getting pretty comfortable with ending tags. Add the following highlighted text to define an empty list item element with no contents:

```
    ...
    <!-- OVERVIEW -->
    <slide type="all">
       <title>Overview</title>
       <item>Why <em>WonderWidgets</em> are great</item>
       <item/>
       <item>Who <em>buys</em> WonderWidgets</item>
    </slide>

</slideshow>
```

Note that any element can be an empty element. All it takes is ending the tag with /> instead of >. You could do the same thing by entering <item></item>, which is equivalent.

Note: Another factor that makes an XML file well formed is proper nesting. So some_text is well formed, because the ... sequence is completely nested within the .. tag. This sequence, on the other hand, is not well formed: some_text.

The Finished Product

Here is the completed version of the XML file:

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

<!--  A SAMPLE set of slides  -->

<slideshow
  title="Sample Slide Show"
  date="Date of publication"
  author="Yours Truly"
  >

  <!-- TITLE SLIDE -->
  <slide type="all">
    <title>Wake up to WonderWidgets!</title>
  </slide>

  <!-- OVERVIEW -->
  <slide type="all">
    <title>Overview</title>
    <item>Why <em>WonderWidgets</em> are great</item>
    <item/>
    <item>Who <em>buys</em> WonderWidgets</item>
  </slide
</slideshow>
```

Save a copy of this file as `slideSample01.xml` so that you can use it as the initial data structure when experimenting with XML programming operations.

Writing Processing Instructions

It sometimes makes sense to code application-specific processing instructions in the XML data. In this exercise, you'll add a processing instruction to your `slideSample.xml` file.

Note: The file you'll create in this section is `slideSample02.xml`. (The browsable version is `slideSample02-xml.html`.)

As you saw in Processing Instructions (page 37), the format for a processing instruction is <?*target data*?>, where *target* is the application that is expected to do the processing, and *data* is the instruction or information for it to process.

Add the following highlighted text to add a processing instruction for a mythical slide presentation program that will query the user to find out which slides to display (technical, executive-level, or all):

```
<slideshow
  ...
  >

  <!-- PROCESSING INSTRUCTION -->
  <?my.presentation.Program QUERY="exec, tech, all"?>

  <!-- TITLE SLIDE -->
```

Notes:

- The data portion of the processing instruction can contain spaces or it can even be null. But there cannot be any space between the initial <? and the target identifier.
- The data begins after the first space.
- It makes sense to fully qualify the target with the complete web-unique package prefix, to preclude any conflict with other programs that might process the same data.
- For readability, it seems like a good idea to include a colon (:) after the name of the application:

  ```
  <?my.presentation.Program: QUERY="..."?>
  ```

  The colon makes the target name into a kind of "label" that identifies the intended recipient of the instruction. However, even though the W3C spec allows a colon in a target name, some versions of Internet Explorer 5 (IE5) consider it an error. For this tutorial, then, we avoid using a colon in the target name.

Save a copy of this file as `slideSample02.xml` so that you can use it when experimenting with processing instructions.

Introducing an Error

The parser can generate three kinds of errors: a fatal error, an error, and a warning. In this exercise, you'll make a simple modification to the XML file to introduce a fatal error. Later, you'll see how it's handled in the Echo application.

Note: The XML structure you'll create in this exercise is in `slideSampleBad1.xml`. (The browsable version is `slideSampleBad1-xml.html`.)

One easy way to introduce a fatal error is to remove the final / from the empty `item` element to create a tag that does not have a corresponding end tag. That constitutes a fatal error, because all XML documents must, by definition, be well formed. Do the following:

1. Copy `slideSample02.xml` to `slideSampleBad1.xml`.
2. Edit `slideSampleBad1.xml` and remove the character shown here:

```
...
<!-- OVERVIEW -->
   <slide type="all">
   <title>Overview</title>
   <item>Why <em>WonderWidgets</em> are great</item>
   <item~~/~~>
   <item>Who <em>buys</em> WonderWidgets</item>
</slide>
...
```

This change produces the following:

```
...
<item>Why <em>WonderWidgets</em> are great</item>
<item>
<item>Who <em>buys</em> WonderWidgets</item>
...
```

Now you have a file that you can use to generate an error in any parser, any time. (XML parsers are required to generate a fatal error for this file, because the lack of an end tag for the <item> element means that the XML structure is no longer well formed.)

Substituting and Inserting Text

In this section, you'll learn about

- Handling special characters (<, &, and so on)
- Handling text with XML-style syntax

Handling Special Characters

In XML, an entity is an XML structure (or plain text) that has a name. Referencing the entity by name causes it to be inserted into the document in place of the entity reference. To create an entity reference, the entity name is surrounded by an ampersand and a semicolon, like this:

```
&entityName;
```

Later, when you learn how to write a DTD, you'll see that you can define your own entities so that `&yourEntityName;` expands to all the text you defined for that entity. For now, though, we'll focus on the predefined entities and character references that don't require any special definitions.

Predefined Entities

An entity reference such as `&` contains a name (in this case, amp) between the start and end delimiters. The text it refers to (&) is substituted for the name, as with a macro in a programming language. Table 2–1 shows the predefined entities for special characters.

Table 2–1 Predefined Entities

Character	Name	Reference
&	ampersand	`&`
<	less than	`<`
>	greater than	`>`
"	quote	`&quot;`
'	apostrophe	`&apos;`

Character References

A character reference such as `&#147;` contains a hash mark (#) followed by a number. The number is the Unicode value for a single character, such as 65 for the letter A, 147 for the left curly quote, or 148 for the right curly quote. In this case, the "name" of the entity is the hash mark followed by the digits that identify the character.

Note: XML expects values to be specified in decimal. However, the Unicode charts at `http://www.unicode.org/charts/` specify values in hexadecimal! So you'll need to do a conversion to get the right value to insert into your XML data set.

Using an Entity Reference in an XML Document

Suppose you want to insert a line like this in your XML document:

```
Market Size < predicted
```

The problem with putting that line into an XML file directly is that when the parser sees the left angle bracket (<), it starts looking for a tag name, throws off the parse. To get around that problem, you put `<` in the file instead of <.

Note: The results of the next modifications are contained in `slideSample03.xml`.

Add the following highlighted text to your `slideSample.xml` file, and save a copy of it for future use as `slideSample03.xml`:

```
<!-- OVERVIEW -->
<slide type="all">
   <title>Overview</title>
   ...
</slide>

<slide type="exec">
   <title>Financial Forecast</title>
   <item>Market Size &lt; predicted</item>
   <item>Anticipated Penetration</item>
   <item>Expected Revenues</item>
   <item>Profit Margin</item>
</slide>

</slideshow>
```

When you use an XML parser to echo this data, you will see the desired output:

```
Market Size < predicted
```

You see an angle bracket (<) where you coded `<`, because the XML parser converts the reference into the entity it represents and passes that entity to the application.

Handling Text with XML-Style Syntax

When you are handling large blocks of XML or HTML that include many special characters, it is inconvenient to replace each of them with the appropriate entity reference. For those situations, you can use a `CDATA` section.

Note: The results of the next modifications are contained in `slideSample04.xml`.

A `CDATA` section works like `<pre>...</pre>` in HTML, only more so: all whitespace in a `CDATA` section is significant, and characters in it are not interpreted as XML. A `CDATA` section starts with `<![CDATA[` and ends with `]]>`.

Add the following highlighted text to your `slideSample.xml` file to define a `CDATA` section for a fictitious technical slide, and save a copy of the file as `slideSample04.xml`:

```
    ...
    <slide type="tech">
      <title>How it Works</title>
      <item>First we fozzle the frobmorten</item>
      <item>Then we framboze the staten</item>
      <item>Finally, we frenzle the fuznaten</item>
      <item><![CDATA[Diagram:
        frobmorten <---------------- fuznaten
          |            <3>             ^
          | <1>                         | <1> = fozzle
          V                             | <2> = framboze
        staten-------------------------+ <3> = frenzle
                 <2>
      ]]></item>
    </slide>
  </slideshow>
```

When you echo this file with an XML parser, you see the following output:

```
Diagram:
frobmorten <---------------- fuznaten
  |            <3>             ^
  | <1>                        | <1> = fozzle
  V                            | <2> = framboze
staten--------------------------+ <3> = frenzle
         <2>
```

The point here is that the text in the `CDATA` section arrives as it was written. Because the parser doesn't treat the angle brackets as XML, they don't generate the fatal errors they would otherwise cause. (If the angle brackets weren't in a `CDATA` section, the document would not be well formed.)

Creating a Document Type Definition

After the XML declaration, the document prolog can include a DTD, which lets you specify the kinds of tags that can be included in your XML document. In addition to telling a validating parser which tags are valid and in what arrangements, a DTD tells both validating and nonvalidating parsers where text is expected, which lets the parser determine whether the whitespace it sees is significant or *ignorable*.

Basic DTD Definitions

To begin learning about DTD definitions, let's start by telling the parser where text is expected and where any text (other than whitespace) would be an error. (Whitespace in such locations is ignorable.)

Note: The DTD defined in this section is contained in `slideshow1a.dtd`. (The browsable version is `slideshow1a-dtd.html`.)

Start by creating a file named `slideshow.dtd`. Enter an XML declaration and a comment to identify the file:

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

<!--
  DTD for a simple "slide show"
-->
```

Next, add the following highlighted text to specify that a `slideshow` element contains `slide` elements and nothing else:

```
<!-- DTD for a simple "slide show" -->

<!ELEMENT slideshow (slide+)>
```

As you can see, the DTD tag starts with `<!` followed by the tag name (`ELEMENT`). After the tag name comes the name of the element that is being defined (`slideshow`) and, in parentheses, one or more items that indicate the valid contents for that element. In this case, the notation says that a `slideshow` consists of one or more `slide` elements.

Without the plus sign, the definition would be saying that a `slideshow` consists of a single `slide` element. The qualifiers you can add to an element definition are listed in Table 2–2.

Table 2–2 DTD Element Qualifiers

Qualifier	Name	Meaning
?	Question mark	Optional (zero or one)
*	Asterisk	Zero or more
+	Plus sign	One or more

You can include multiple elements inside the parentheses in a comma-separated list and use a qualifier on each element to indicate how many instances of that element can occur. The comma-separated list tells which elements are valid and the order they can occur in.

You can also nest parentheses to group multiple items. For an example, after defining an `image` element (discussed shortly), you can specify `((image, title)+)` to declare that every `image` element in a slide must be paired with a `title` element. Here, the plus sign applies to the `image/title` pair to indicate that one or more pairs of the specified items can occur.

Defining Text and Nested Elements

Now that you have told the parser something about where *not* to expect text, let's see how to tell it where text *can* occur. Add the following highlighted text to define the `slide`, `title`, `item`, and `list` elements:

```
<!ELEMENT slideshow (slide+)>
<!ELEMENT slide (title, item*)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT item (#PCDATA | item)* >
```

The first line you added says that a slide consists of a `title` followed by zero or more `item` elements. Nothing new there. The next line says that a title consists entirely of *parsed character data* (`PCDATA`). That's known as "text" in most parts of the country, but in XML-speak it's called "parsed character data." (That distinguishes it from `CDATA` sections, which contain character data that is not parsed.) The `#` that precedes `PCDATA` indicates that what follows is a special word rather than an element name.

The last line introduces the vertical bar (`|`), which indicates an *or* condition. In this case, either `PCDATA` or an `item` can occur. The asterisk at the end says that either element can occur zero or more times in succession. The result of this specification is known as a *mixed-content model*, because any number of `item` elements can be interspersed with the text. Such models must always be defined with `#PCDATA` specified first, followed by some number of alternate items divided by vertical bars (`|`), and an asterisk (`*`) at the end.

Save a copy of this DTD as slideSample1a.dtd for use when you experiment with basic DTD processing.

Limitations of DTDs

It would be nice if we could specify that an `item` contains either text, or text followed by one or more list items. But that kind of specification turns out to be hard to achieve in a DTD. For example, you might be tempted to define an `item` this way:

```
<!ELEMENT item (#PCDATA | (#PCDATA, item+)) >
```

That would certainly be accurate, but as soon as the parser sees `#PCDATA` and the vertical bar, it requires the remaining definition to conform to the mixed-content model. This specification doesn't, so you get can error that says `Illegal mixed`

`content model for 'item'. Found &#x28; ...`, where the hex character 28 is the angle bracket that ends the definition.

Trying to double-define the item element doesn't work either. Suppose you try a specification like this:

```
<!ELEMENT item (#PCDATA) >
<!ELEMENT item (#PCDATA, item+) >
```

This sequence produces a "duplicate definition" warning when the validating parser runs. The second definition is, in fact, ignored. So it seems that defining a mixed-content model (which allows `item` elements to be interspersed in text) is the best we can do.

In addition to the limitations of the mixed-content model we've mentioned, there is no way to further qualify the kind of text that can occur where `PCDATA` has been specified. Should it contain only numbers? Should it be in a date format, or possibly a monetary format? There is no way to specify such things in a DTD.

Finally, note that the DTD offers no sense of hierarchy. The definition of the `title` element applies equally to a `slide` title and to an `item` title. When we expand the DTD to allow HTML-style markup in addition to plain text, it would make sense to, for example, restrict the size of an `item` title compared with that of a `slide` title. But the only way to do that would be to give one of them a different name, such as `item-title`. The bottom line is that the lack of hierarchy in the DTD forces you to introduce a "hyphenation hierarchy" (or its equivalent) in your namespace. All these limitations are fundamental motivations behind the development of schema-specification standards.

Special Element Values in the DTD

Rather than specify a parenthesized list of elements, the element definition can use one of two special values: `ANY` or `EMPTY`. The `ANY` specification says that the element can contain any other defined element, or `PCDATA`. Such a specification is usually used for the root element of a general-purpose XML document such as you might create with a word processor. Textual elements can occur in any order in such a document, so specifying `ANY` makes sense.

The `EMPTY` specification says that the element contains no contents. So the DTD for email messages that let you flag the message with `<flag/>` might have a line like this in the DTD:

```
<!ELEMENT flag EMPTY>
```

Referencing the DTD

In this case, the DTD definition is in a separate file from the XML document. With this arrangement, you reference the DTD from the XML document, and that makes the DTD file part of the *external subset* of the full document type definition for the XML file. As you'll see later on, you can also include parts of the DTD within the document. Such definitions constitute the *local subset* of the DTD.

Note: The XML written in this section is contained in `slideSample05.xml`. (The browsable version is `slideSample05-xml.html`.)

To reference the DTD file you just created, add the following highlighted line to your `slideSample.xml` file, and save a copy of the file as `slideSample05.xml`:

```
<!--  A SAMPLE set of slides  -->

<!DOCTYPE slideshow SYSTEM "slideshow.dtd">

<slideshow
```

Again, the DTD tag starts with `<!`. In this case, the tag name, `DOCTYPE`, says that the document is a `slideshow`, which means that the document consists of the `slideshow` element and everything within it:

```
<slideshow>
...
</slideshow>
```

This tag defines the `slideshow` element as the root element for the document. An XML document must have exactly one root element. This is where that element is specified. In other words, this tag identifies the document *content* as a `slideshow`.

The `DOCTYPE` tag occurs after the XML declaration and before the root element. The `SYSTEM` identifier specifies the location of the DTD file. Because it does not start with a prefix such as `http:/` or `file:/`, the path is relative to the location of the XML document. Remember the `setDocumentLocator` method? The parser is using that information to find the DTD file, just as your application would use it to find a file relative to the XML document. A `PUBLIC` identifier can also be used to specify the DTD file using a unique name, but the parser would have to be able to resolve it.

The `DOCTYPE` specification can also contain DTD definitions within the XML document, rather than refer to an external DTD file. Such definitions are contained in square brackets:

```
<!DOCTYPE slideshow SYSTEM "slideshow1.dtd" [
  ...local subset definitions here...
]>
```

You'll take advantage of that facility in a moment to define some entities that can be used in the document.

Documents and Data

Earlier, you learned that one reason you hear about XML *documents*, on the one hand, and XML *data*, on the other, is that XML handles both comfortably, depending on whether text is or is not allowed between elements in the structure.

In the sample file you have been working with, the `slideshow` element is an example of a *data element*: it contains only subelements with no intervening text. The `item` element, on the other hand, might be termed a *document element*, because it is defined to include both text and subelements.

As you work through this tutorial, you will see how to expand the definition of the title element to include HTML-style markup, which will turn it into a document element as well.

Defining Attributes and Entities in the DTD

The DTD you've defined so far is fine for use with a nonvalidating parser. It tells where text is expected and where it isn't, and that is all the nonvalidating parser pays attention to. But for use with the validating parser, the DTD must specify the valid attributes for the different elements. You'll do that in this section, and then you'll define one internal entity and one external entity that you can reference in your XML file.

Defining Attributes in the DTD

Let's start by defining the attributes for the elements in the slide presentation.

Note: The XML written in this section is contained in `slideshow1b.dtd`. (The browsable version is `slideshow1b-dtd.html`.)

Add the following highlighted text to define the attributes for the `slideshow` element:

```
<!ELEMENT slideshow (slide+)>
<!ATTLIST slideshow
     title    CDATA    #REQUIRED
     date     CDATA    #IMPLIED
     author   CDATA    "unknown"
>
<!ELEMENT slide (title, item*)>
```

The DTD tag `ATTLIST` begins the series of attribute definitions. The name that follows `ATTLIST` specifies the element for which the attributes are being defined. In this case, the element is the `slideshow` element. (Note again the lack of hierarchy in DTD specifications.)

Each attribute is defined by a series of three space-separated values. Commas and other separators are not allowed, so formatting the definitions as shown here is helpful for readability. The first element in each line is the name of the attribute: `title`, `date`, or `author`, in this case. The second element indicates the type of the data: `CDATA` is character data—unparsed data, again, in which a left angle bracket (<) will never be construed as part of an XML tag. Table 2–3 presents the valid choices for the attribute type.

Table 2–3 Attribute Types

Attribute Type	Specifies...
`(value1 | value2 | ...)`	A list of values separated by vertical bars
`CDATA`	Unparsed character data (a text string)
`ID`	A name that no other ID attribute shares
`IDREF`	A reference to an ID defined elsewhere in the document
`IDREFS`	A space-separated list containing one or more ID references
`ENTITY`	The name of an entity defined in the DTD

Table 2–3 Attribute Types

Attribute Type	Specifies...
ENTITIES	A space-separated list of entities
NMTOKEN	A valid XML name composed of letters, numbers, hyphens, underscores, and colons
NMTOKENS	A space-separated list of names
NOTATION	The name of a DTD-specified notation, which describes a non-XML data format, such as those used for image files. (This is a rapidly obsolescing specification which will be discussed in greater length towards the end of this section.)

When the attribute type consists of a parenthesized list of choices separated by vertical bars, the attribute must use one of the specified values. For an example, add the following highlighted text to the DTD:

```
<!ELEMENT slide (title, item*)>
<!ATTLIST slide
     type   (tech | exec | all) #IMPLIED
>
<!ELEMENT title (#PCDATA)>
<!ELEMENT item (#PCDATA | item)* >
```

This specification says that the `slide` element's `type` attribute must be given as `type="tech"`, `type="exec"`, or `type="all"`. No other values are acceptable. (DTD-aware XML editors can use such specifications to present a pop-up list of choices.)

The last entry in the attribute specification determines the attribute's default value, if any, and tells whether or not the attribute is required. Table 2–4 shows the possible choices.

Table 2–4 Attribute-Specification Parameters

Specification	Specifies...
#REQUIRED	The attribute value must be specified in the document.

Table 2–4 Attribute-Specification Parameters

Specification	Specifies...
`#IMPLIED`	The value need not be specified in the document. If it isn't, the application will have a default value it uses.
`"defaultValue"`	The default value to use if a value is not specified in the document.
`#FIXED "fixedValue"`	The value to use. If the document specifies any value at all, it must be the same.

Finally, save a copy of the DTD as `slideshow1b.dtd` for use when you experiment with attribute definitions.

Defining Entities in the DTD

So far, you've seen predefined entities such as `&` and you've seen that an attribute can reference an entity. It's time now for you to learn how to define entities of your own.

Note: The XML you'll create here is contained in `slideSample06.xml`. (The browsable version is `slideSample06-xml.html`.)

Add the following highlighted text to the `DOCTYPE` tag in your XML file:

```
<!DOCTYPE slideshow SYSTEM "slideshow.dtd" [
  <!ENTITY product  "WonderWidget">
  <!ENTITY products "WonderWidgets">
]>
```

The `ENTITY` tag name says that you are defining an entity. Next comes the name of the entity and its definition. In this case, you are defining an entity named `product` that will take the place of the product name. Later when the product name changes (as it most certainly will), you need only change the name in one place, and all your slides will reflect the new value.

The last part is the substitution string that replaces the entity name whenever it is referenced in the XML document. The substitution string is defined in quotes, which are not included when the text is inserted into the document.

Just for good measure, we defined two versions—one singular and one plural—so that when the marketing mavens come up with "Wally" for a product name, you will be prepared to enter the plural as "Wallies" and have it substituted correctly.

Note: Truth be told, this is the kind of thing that really belongs in an external DTD so that all your documents can reference the new name when it changes. But, hey, this is only an example.

Now that you have the entities defined, the next step is to reference them in the slide show. Make the following highlighted changes:

```
<slideshow
  title="WonderWidget&product; Slide Show"
  ...

  <!-- TITLE SLIDE -->
  <slide type="all">
    <title>Wake up to WonderWidgets&products;!</title>
  </slide>

   <!-- OVERVIEW -->
  <slide type="all">
    <title>Overview</title>
    <item>Why <em>WonderWidgets&products;</em> are
great</item>
    <item/>
    <item>Who <em>buys</em> WonderWidgets&products;</item>
  </slide>
```

Notice two points. Entities you define are referenced with the same syntax (`&entityName;`) that you use for predefined entities, and the entity can be referenced in an attribute value as well as in an element's contents.

When you echo this version of the file with an XML parser, here is the kind of thing you'll see:

```
Wake up to WonderWidgets!
```

Note that the product name has been substituted for the entity reference.

To finish, save a copy of the file as `slideSample06.xml`.

Additional Useful Entities

Here are several other examples for entity definitions that you might find useful when you write an XML document:

```
<!ENTITY ldquo  "&#147;"> <!-- Left Double Quote -->
<!ENTITY rdquo  "&#148;"> <!-- Right Double Quote -->
<!ENTITY trade  "&#153;"> <!-- Trademark Symbol (TM) -->
<!ENTITY rtrade "&#174;"> <!-- Registered Trademark (R) -->
<!ENTITY copyr  "&#169;"> <!-- Copyright Symbol -->
```

Referencing External Entities

You can also use the `SYSTEM` or `PUBLIC` identifier to name an entity that is defined in an external file. You'll do that now.

Note: The XML defined here is contained in `slideSample07.xml` and in `copyright.xml`. (The browsable versions are `slideSample07-xml.html` and `copyright-xml.html`.)

To reference an external entity, add the following highlighted text to the `DOCTYPE` statement in your XML file:

```
<!DOCTYPE slideshow SYSTEM "slideshow.dtd" [
  <!ENTITY product  "WonderWidget">
  <!ENTITY products "WonderWidgets">
  <!ENTITY copyright SYSTEM "copyright.xml">
]>
```

This definition references a copyright message contained in a file named `copyright.xml`. Create that file and put some interesting text in it, perhaps something like this:

```
  <!--  A SAMPLE copyright  -->

This is the standard copyright message that our lawyers
make us put everywhere so we don't have to shell out a
million bucks every time someone spills hot coffee in their
lap...
```

Finally, add the following highlighted text to your `slideSample.xml` file to reference the external entity, and save a copy of the file as `slideSample07.html`:

```
<!-- TITLE SLIDE -->
  ...
</slide>

<!-- COPYRIGHT SLIDE -->
<slide type="all">
  <item>&copyright;</item>
</slide>
```

You could also use an external entity declaration to access a servlet that produces the current date using a definition something like this:

```
<!ENTITY currentDate SYSTEM
  "http://www.example.com/servlet/Today?fmt=dd-MMM-yyyy">
```

You would then reference that entity the same as any other entity:

```
  Today's date is &currentDate;.
```

When you echo the latest version of the slide presentation with an XML parser, here is what you'll see:

```
...
<slide type="all">
  <item>
This is the standard copyright message that our lawyers
make us put everywhere so we don't have to shell out a
million bucks every time someone spills hot coffee in their
lap...
  </item>
</slide>
...
```

You'll notice that the newline that follows the comment in the file is echoed as a character, but that the comment itself is ignored. This newline is the reason that the copyright message appears to start on the next line after the `<item>` element instead of on the same line: the first character echoed is actually the newline that follows the comment.

Summarizing Entities

An entity that is referenced in the document content, whether internal or external, is termed a *general entity.* An entity that contains DTD specifications that are referenced from within the DTD is termed a *parameter entity.* (More on that later.)

An entity that contains XML (text and markup), and is therefore parsed, is known as a *parsed entity.* An entity that contains binary data (such as images) is known as an *unparsed entity.* (By its nature, it must be external.) In the next section, we discuss references to unparsed entities.

Referencing Binary Entities

This section discusses the options for referencing binary files such as image files and multimedia data files.

Using a MIME Data Type

There are two ways to reference an unparsed entity such as a binary image file. One is to use the DTD's `NOTATION` specification mechanism. However, that mechanism is a complex, unintuitive holdover that exists mostly for compatibility with SGML documents.

Note: SGML stands for Standard Generalized Markup Language. It was extremely powerful but *so* general that a program had to read the beginning of a document just to find out how to parse the remainder of it. Some very large document-management systems were built using it, but it was so large and complex that only the largest organizations managed to deal with it. XML, on the other hand, chose to remain small and simple—more like HTML than SGML—and, as a result, it has enjoyed rapid, widespread deployment. This story may well hold a moral for schema standards as well. Time will tell.

We will have occasion to discuss the subject in a bit more depth when we look at the `DTDHandler` API, but suffice it for now to say that the XML namespaces standard, in conjunction with the MIME data types defined for electronic messaging attachments, together provide a much more useful, understandable, and extensible mechanism for referencing unparsed external entities.

Note: The XML described here is in `slideshow1b.dtd`. (The browsable version is `slideshow1b-dtd.html`.) It shows how binary references can be made, assuming that the application that will process the XML data knows how to handle such references.

To set up the slide show to use image files, add the following highlighted text to your `slideshow1b.dtd` file:

```
<!ELEMENT slide (image?, title, item*)>
<!ATTLIST slide
     type   (tech | exec | all) #IMPLIED
>
<!ELEMENT title (#PCDATA)>
<!ELEMENT item (#PCDATA | item)* >
<!ELEMENT image EMPTY>
<!ATTLIST image
     alt    CDATA    #IMPLIED
     src    CDATA    #REQUIRED
     type   CDATA    "image/gif"
>
```

These modifications declare `image` as an optional element in a `slide`, define it as empty element, and define the attributes it requires. The `image` tag is patterned after the HTML 4.0 `img` tag, with the addition of an image type specifier, `type`. (The `img` tag is defined in the HTML 4.0 specification.)

The `image` tag's attributes are defined by the `ATTLIST` entry. The `alt` attribute, which defines alternative text to display in case the image can't be found, accepts character data (`CDATA`). It has an implied value, which means that it is optional and that the program processing the data knows enough to substitute something such as "Image not found." On the other hand, the `src` attribute, which names the image to display, is required.

The `type` attribute is intended for the specification of a MIME data type, as defined at `http://www.iana.org/assignments/media-types/`. It has a default value: `image/gif`.

Note: It is understood here that the character data (`CDATA`) used for the type attribute will be one of the MIME data types. The two most common formats are `image/gif` and `image/jpeg`. Given that fact, it might be nice to specify an attribute list here, using something like

```
type ("image/gif", "image/jpeg")
```

That won't work, however, because attribute lists are restricted to name tokens. The forward slash isn't part of the valid set of name-token characters, so this declaration fails. Also, creating an attribute list in the DTD would limit the valid MIME types to those defined today. Leaving it as CDATA leaves things more open-ended so that the declaration will continue to be valid as additional types are defined.

In the document, a reference to an image named "intro-pic" might look something like this:

```
<image src="image/intro-pic.gif", alt="Intro Pic",
type="image/gif" />
```

The Alternative: Using Entity References

Using a MIME data type as an attribute of an element is a flexible and expandable mechanism. To create an external ENTITY reference using the notation mechanism, you need DTD NOTATION elements for JPEG and GIF data. Those can, of course, be obtained from a central repository. But then you need to define a different ENTITY element for each image you intend to reference! In other words, adding a new image to your document always requires both a new entity definition in the DTD and a reference to it in the document. Given the anticipated ubiquity of the HTML 4.0 specification, the newer standard is to use the MIME data types and a declaration such as image, which assumes that the application knows how to process such elements.

Defining Parameter Entities and Conditional Sections

Just as a general entity lets you reuse XML data in multiple places, a parameter entity lets you reuse parts of a DTD in multiple places. In this section you'll see how to define and use parameter entities. You'll also see how to use parameter entities with conditional sections in a DTD.

Creating and Referencing a Parameter Entity

Recall that the existing version of the slide presentation can not be validated because the document uses tags, and they are not part of the DTD. In general, we'd like to use a variety of HTML-style tags in the text of a slide, and not

just one or two, so using an existing DTD for XHTML makes more sense than defining such tags ourselves. A parameter entity is intended for exactly that kind of purpose.

Note: The DTD specifications shown here are contained in `slideshow2.dtd` and `xhtml.dtd`. The XML file that references it is `slideSample08.xml`. (The browsable versions are `slideshow2-dtd.html`, `xhtml-dtd.html`, and `slideSample08-xml.html`.)

Open your DTD file for the slide presentation and add the following highlighted text to define a parameter entity that references an external DTD file:

```
<!ELEMENT slide (image?, title?, item*)>
<!ATTLIST slide
        ...
>

<!ENTITY % xhtml SYSTEM "xhtml.dtd">
%xhtml;

<!ELEMENT title ...
```

Here, you use an `<!ENTITY>` tag to define a parameter entity, just as for a general entity, but you use a somewhat different syntax. You include a percent sign (%) before the entity name when you define the entity, and you use the percent sign instead of an ampersand when you reference it.

Also, note that there are always two steps to using a parameter entity. The first is to define the entity name. The second is to reference the entity name, which actually does the work of including the external definitions in the current DTD. Because the uniform resource identifier (URI) for an external entity could contain slashes (/) or other characters that are not valid in an XML name, the definition step allows a valid XML name to be associated with an actual document. (This same technique is used in the definition of namespaces and anywhere else that XML constructs need to reference external documents.)

Notes:

- The DTD file referenced by this definition is `xhtml.dtd`. (The browsable version is `xhtml-dtd.html`.) You can either copy that file to your system or modify the `SYSTEM` identifier in the `<!ENTITY>` tag to point to the correct URL.

- This file is a small subset of the XHTML specification, loosely modeled after the Modularized XHTML draft, which aims at breaking up the DTD for XHTML into bite-sized chunks, which can then be combined to create different XHTML subsets for different purposes. When work on the modularized XHTML draft has been completed, this version of the DTD should be replaced with something better. For now, this version will suffice for our purposes.

The point of using an XHTML-based DTD is to gain access to an entity it defines that covers HTML-style tags like `<em>` and `<b>`. Looking through `xhtml.dtd` reveals the following entity, which does exactly what we want:

```
<!ENTITY % inline "#PCDATA|em|b|a|img|br">
```

This entity is a simpler version of those defined in the Modularized XHTML draft. It defines the HTML-style tags we are most likely to want to use—emphasis, bold, and break—plus a couple of others for images and anchors that we may or may not use in a slide presentation. To use the `inline` entity, make the following highlighted changes in your DTD file:

```
<!ELEMENT title (~~#PCDATA~~ %inline;)*>
<!ELEMENT item (~~#PCDATA~~ %inline; | item)* >
```

These changes replace the simple `#PCDATA` item with the `inline` entity. It is important to notice that `#PCDATA` is first in the `inline` entity and that `inline` is first wherever we use it. That sequence is required by XML's definition of a mixed-content model. To be in accord with that model, you also must add an asterisk at the end of the `title` definition.

Save the DTD as `slideshow2.dtd` for use when you experiment with parameter entities.

Note: The Modularized XHTML DTD defines both `inline` and `Inline` entities, and does so somewhat differently. Rather than specify `#PCDATA|em|b|a|img|br`, the definitions are more like `(#PCDATA|em|b|a|img|br)*`. Using one of those definitions, therefore, looks more like this:

```
<!ELEMENT title %Inline; >
```

Conditional Sections

Before we proceed with the next programming exercise, it is worth mentioning the use of parameter entities to control *conditional sections*. Although you cannot conditionalize the content of an XML document, you can define conditional sections in a DTD that become part of the DTD only if you specify `include`. If you specify `ignore`, on the other hand, then the conditional section is not included.

Suppose, for example, that you wanted to use slightly different versions of a DTD, depending on whether you were treating the document as an XML document or as a SGML document. You can do that with DTD definitions such as the following:

```
someExternal.dtd:
  <![ INCLUDE [
     ... XML-only definitions
  ]]>
  <![ IGNORE [
     ... SGML-only definitions
  ]]>
  ... common definitions
```

The conditional sections are introduced by `<![`, followed by the `INCLUDE` or `IGNORE` keyword and another `[`. After that comes the contents of the conditional section, followed by the terminator: `]]>`. In this case, the XML definitions are included, and the SGML definitions are excluded. That's fine for XML documents, but you can't use the DTD for SGML documents. You could change the keywords, of course, but that only reverses the problem.

The solution is to use references to parameter entities in place of the `INCLUDE` and `IGNORE` keywords:

```
someExternal.dtd:
  <![ %XML; [
     ... XML-only definitions
  ]]>
  <![ %SGML; [
     ... SGML-only definitions
  ]]>
  ... common definitions
```

Then each document that uses the DTD can set up the appropriate entity definitions:

```
<!DOCTYPE foo SYSTEM "someExternal.dtd" [
  <!ENTITY % XML  "INCLUDE" >
  <!ENTITY % SGML "IGNORE" >
]>
<foo>
  ...
</foo>
```

This procedure puts each document in control of the DTD. It also replaces the `INCLUDE` and `IGNORE` keywords with variable names that more accurately reflect the purpose of the conditional section, producing a more readable, self-documenting version of the DTD.

Resolving a Naming Conflict

The XML structures you have created thus far have actually encountered a small naming conflict. It seems that `xhtml.dtd` defines a `title` element that is entirely different from the `title` element defined in the slide-show DTD. Because there is no hierarchy in the DTD, these two definitions conflict.

Note: The Modularized XHTML DTD also defines a `title` element that is intended to be the document title, so we can't avoid the conflict by changing `xhtml.dtd`. The problem would only come back to haunt us later.

You can use XML namespaces to resolve the conflict. You'll take a look at that approach in the next section. Alternatively, you can use one of the more hierarchical schema proposals described in Schema Standards (page 1392). The simplest way to solve the problem for now is to rename the `title` element in `slideshow.dtd`.

Note: The XML shown here is contained in `slideshow3.dtd` and `slideSample09.xml`, which references `copyright.xml` and `xhtml.dtd`. (The browsable versions are `slideshow3-dtd.html`, `slideSample09-xml.html`, `copyright-xml.html`, and `xhtml-dtd.html`.)

To keep the two title elements separate, you'll create a *hyphenation hierarchy.* Make the following highlighted changes to change the name of the `title` element in `slideshow.dtd` to `slide-title`:

```
<!ELEMENT slide (image?, slide-title?, item*)>
<!ATTLIST slide
        type   (tech | exec | all) #IMPLIED
>

<!-- Defines the %inline; declaration -->
<!ENTITY % xhtml SYSTEM "xhtml.dtd">
%xhtml;

<!ELEMENT slide-title (%inline;)*>
```

Save this DTD as `slideshow3.dtd`.

The next step is to modify the XML file to use the new element name. To do that, make the following highlighted changes:

```
...
<slide type="all">
<slide-title>Wake up to ... </slide-title>
</slide>

...

<!-- OVERVIEW -->
<slide type="all">
<slide-title>Overview</slide-title>
<item>...
```

Save a copy of this file as `slideSample09.xml`.

Using Namespaces

As you saw earlier, one way or another it is necessary to resolve the conflict between the `title` element defined in `slideshow.dtd` and the one defined in `xhtml.dtd` when the same name is used for different purposes. In the preceding exercise, you hyphenated the name in order to put it into a different namespace. In this section, you'll see how to use the XML namespace standard to do the same thing without renaming the element.

The primary goal of the namespace specification is to let the document author tell the parser which DTD or schema to use when parsing a given element. The parser can then consult the appropriate DTD or schema for an element definition. Of course, it is also important to keep the parser from aborting when a "duplicate" definition is found and yet still generate an error if the document references an element such as `title` without *qualifying* it (identifying the DTD or schema to use for the definition).

Note: Namespaces apply to attributes as well as to elements. In this section, we consider only elements. For more information on attributes, consult the namespace specification at `http://www.w3.org/TR/REC-xml-names/`.

Defining a Namespace in a DTD

In a DTD, you define a namespace that an element belongs to by adding an attribute to the element's definition, where the attribute name is `xmlns` ("xml namespace"). For example, you can do that in `slideshow.dtd` by adding an entry such as the following in the `title` element's attribute-list definition:

```
<!ELEMENT title (%inline;)*>
<!ATTLIST title
  xmlns CDATA #FIXED "http://www.example.com/slideshow"
>
```

Declaring the attribute as `FIXED` has several important features:

- It prevents the document from specifying any nonmatching value for the `xmlns` attribute.
- The element defined in this DTD is made unique (because the parser understands the `xmlns` attribute), so it does not conflict with an element that has the same name in another DTD. That allows multiple DTDs to use the same element name without generating a parser error.
- When a document specifies the `xmlns` attribute for a tag, the document selects the element definition that has a matching attribute.

To be thorough, every element name in your DTD would get exactly the same attribute, with the same value. (Here, though, we're concerned only about the `title` element.) Note, too, that you are using a `CDATA` string to supply the URI. In this case, we've specified a URL. But you could also specify a universal resource name (URN), possibly by specifying a prefix such as `urn:` instead of

`http:`. (URNs are currently being researched. They're not seeing a lot of action at the moment, but that could change in the future.)

Referencing a Namespace

When a document uses an element name that exists in only one of the DTDs or schemas it references, the name does not need to be qualified. But when an element name that has multiple definitions is used, some sort of qualification is a necessity.

Note: In fact, an element name is always qualified by its *default namespace*, as defined by the name of the DTD file it resides in. As long as there is only one definition for the name, the qualification is implicit.

You qualify a reference to an element name by specifying the `xmlns` attribute, as shown here:

```
<title xmlns="http://www.example.com/slideshow">
  Overview
</title>
```

The specified namespace applies to that element and to any elements contained within it.

Defining a Namespace Prefix

When you need only one namespace reference, it's not a big deal. But when you need to make the same reference several times, adding `xmlns` attributes becomes unwieldy. It also makes it harder to change the name of the namespace later.

The alternative is to define a *namespace prefix*, which is as simple as specifying `xmlns`, a colon (:), and the prefix name before the attribute value:

```
<SL:slideshow xmlns:SL='http:/www.example.com/slideshow'
   ...>
  ...
</SL:slideshow>
```

This definition sets up `SL` as a prefix that can be used to qualify the current element name and any element within it. Because the prefix can be used on any of

the contained elements, it makes the most sense to define it on the XML document's root element, as shown here.

Note: The namespace URI can contain characters that are not valid in an XML name, so it cannot be used directly as a prefix. The prefix definition associates an XML name with the URI, and that allows the prefix name to be used instead. It also makes it easier to change references to the URI in the future.

When the prefix is used to qualify an element name, the end tag also includes the prefix, as highlighted here:

```
<SL:slideshow xmlns:SL='http:/www.example.com/slideshow'
     ...>
  ...
  <slide>
   <SL:title>Overview</SL:title>
  </slide>
  ...
</SL:slideshow>
```

Finally, note that multiple prefixes can be defined in the same element:

```
<SL:slideshow xmlns:SL='http:/www.example.com/slideshow'
     xmlns:xhtml='urn:...'>
  ...
</SL:slideshow>
```

With this kind of arrangement, all the prefix definitions are together in one place, and you can use them anywhere they are needed in the document. This example also suggests the use of a URN instead of a URL to define the `xhtml` prefix. That definition would conceivably allow the application to reference a local copy of the XHTML DTD or some mirrored version, with a potentially beneficial impact on performance.

Designing an XML Data Structure

This section covers some heuristics you can use when making XML design decisions.

Saving Yourself Some Work

Whenever possible, use an existing schema definition. It's usually a lot easier to ignore the things you don't need than to design your own from scratch. In addition, using a standard DTD makes data interchange possible, and may make it possible to use data-aware tools developed by others.

So if an industry standard exists, consider referencing that DTD by using an external parameter entity. One place to look for industry-standard DTDs is at the web site created by the Organization for the Advancement of Structured Information Standards (OASIS). You can find a list of technical committees at `http://www.oasis-open.org/` or check its repository of XML standards at `http://www.XML.org`.

Note: Many more good thoughts on the design of XML structures are at the OASIS page `http://www.oasis-open.org/cover/elementsAndAttrs.html`.

Attributes and Elements

One of the issues you will encounter frequently when designing an XML structure is whether to model a given data item as a subelement or as an attribute of an existing element. For example, you can model the title of a slide this way:

```
<slide>
  <title>This is the title</title>
</slide>
```

Or you can do it this way:

```
<slide title="This is the title">...</slide>
```

In some cases, the different characteristics of attributes and elements make it easy to choose. Let's consider those cases first and then move on to the cases where the choice is more ambiguous.

Forced Choices

Sometimes, the choice between an attribute and an element is forced on you by the nature of attributes and elements. Let's look at a few of those considerations:

- **The data contains substructures:** In this case, the data item must be modeled as an *element*. It can't be modeled as an attribute, because attributes take only simple strings. So if the title can contain emphasized text (`The <em>Best</em> Choice`) then the title must be an element.
- **The data contains multiple lines:** Here, it also makes sense to use an *element*. Attributes need to be simple, short strings or else they become unreadable, if not unusable.
- **Multiple occurrences are possible:** Whenever an item can occur multiple times, such as paragraphs in an article, it must be modeled as an *element*. The element that contains it can have only one attribute of a particular kind, but it can have many subelements of the same type.
- **The data changes frequently:** When the data will be frequently modified with an editor, it may make sense to model it as an *element*. Many XML-aware editors make it easy to modify element data, whereas attributes can be somewhat harder to get to.
- **The data is a small, simple string that rarely if ever changes:** This is data that can be modeled as an *attribute*. However, just because you *can* does not mean that you should. Check the Stylistic Choices section next, to be sure.
- **The data is confined to a small number of fixed choices:** If you are using a DTD, it really makes sense to use an *attribute*. A DTD can prevent an attribute from taking on any value that is not in the preapproved list, but it cannot similarly restrict an element. (With a schema, on the other hand, both attributes and elements can be restricted, so you could use either element or an attribute.)

Stylistic Choices

As often as not, the choices are not as cut-and-dried as those just shown. When the choice is not forced, you need a sense of "style" to guide your thinking. The question to answer, then, is what makes good XML style, and why.

Defining a sense of style for XML is, unfortunately, as nebulous a business as defining style when it comes to art or music. There are, however, a few ways to

approach it. The goal of this section is to give you some useful thoughts on the subject of XML style.

One heuristic for thinking about XML elements and attributes uses the concept of *visibility*. If the data is intended to be shown—to be displayed to an end user—then it should be modeled as an element. On the other hand, if the information guides XML processing but is never seen by a user, then it may be better to model it as an attribute. For example, in order-entry data for shoes, shoe size would definitely be an element. On the other hand, a manufacturer's code number would be reasonably modeled as an attribute.

Another way of thinking about the visibility heuristic is to ask, who is the consumer and the provider of the information? The shoe size is entered by a human sales clerk, so it's an element. The manufacturer's code number for a given shoe model, on the other hand, may be wired into the application or stored in a database, so that would be an attribute. (If it were entered by the clerk, though, it should perhaps be an element.)

Perhaps the best way of thinking about elements and attributes is to think of an element as a *container*. To reason by analogy, the *contents* of the container (water or milk) correspond to XML data modeled as elements. Such data is essentially variable. On the other hand, the *characteristics* of the container (whether a blue or a white pitcher) can be modeled as attributes. That kind of information tends to be more immutable. Good XML style separates each container's contents from its characteristics in a consistent way.

To show these heuristics at work, in our slide-show example the `type` of the slide (executive or technical) is best modeled as an attribute. It is a characteristic of the slide that lets it be selected or rejected for a particular audience. The `title` of the slide, on the other hand, is part of its contents. The visibility heuristic is also satisfied here. When the slide is displayed, the `title` is shown but the `type` of the slide isn't. Finally, in this example, the consumer of the `title` information is the presentation audience, whereas the consumer of the `type` information is the presentation program.

Normalizing Data

In Saving Yourself Some Work (page 77), you saw that it is a good idea to define an external entity that you can reference in an XML document. Such an entity has all the advantages of a modularized routine: changing that one copy affects every document that references it. The process of eliminating redundancies is

known as *normalizing*, and defining entities is one good way to normalize your data.

In an HTML file, the only way to achieve that kind of modularity is to use HTML links, but then the document is fragmented rather than whole. XML entities, on the other hand, suffer no such fragmentation. The entity reference acts like a macro: the entity's contents are expanded in place, producing a whole document rather than a fragmented one. And when the entity is defined in an external file, multiple documents can reference it.

The considerations for defining an entity reference, then, are pretty much the same as those you would apply to modularized program code:

- Whenever you find yourself writing the same thing more than once, think entity. That lets you write it in one place and reference it in multiple places.
- If the information is likely to change, especially if it is used in more than one place, definitely think in terms of defining an entity. An example is defining `productName` as an entity so that you can easily change the documents when the product name changes.
- If the entity will never be referenced anywhere except in the current file, define it in the local subset of the document's DTD, much as you would define a method or inner class in a program.
- If the entity will be referenced from multiple documents, define it as an external entity, in the same way that you would define any generally usable class as an external class.

External entities produce modular XML that is smaller, easier to update, and easier to maintain. They can also make the resulting document somewhat more difficult to visualize, much as a good object-oriented design can be easy to change, after you understand it, but harder to wrap your head around at first.

You can also go overboard with entities. At an extreme, you could make an entity reference for the word *the*. It wouldn't buy you much, but you could do it.

Note: The larger an entity is, the more likely it is that changing it will have the expected effect. For example, when you define an external entity that covers a whole section of a document, such as installation instructions, then any changes you make will likely work out fine wherever that section is used. But small inline substitutions can be more problematic. For example, if `productName` is defined as an entity and if the name changes to a different part of speech, the results can be unfortunate. Suppose the product name is something like HtmlEdit. That's a verb. So you write a sentence like, "You can HtmlEdit your file...", using the `productName` entity. That sentence works, because a verb fits in that context. But if the name is eventually

changed to “HtmlEditor”, the sentence becomes “You can HtmlEditor your file...”, which clearly doesn’t work. Still, even if such simple substitutions can sometimes get you into trouble, they also have the potential to save a lot of time. (One way to avoid the problem would be to set up entities named `productNoun`, `productVerb`, `productAdj`, and `productAdverb`.)

Normalizing DTDs

Just as you can normalize your XML document, you can also normalize your DTD declarations by factoring out common pieces and referencing them with a parameter entity. Factoring out the DTDs (also known as *modularizing*) gives the same advantages and disadvantages as normalized XML—easier to change, somewhat more difficult to follow.

You can also set up conditionalized DTDs. If the number and size of the conditional sections are small relative to the size of the DTD as a whole, conditionalizing can let you single-source the same DTD for multiple purposes. If the number of conditional sections gets large, though, the result can be a complex document that is difficult to edit.

Summary

Congratulations! You have now created a number of XML files that you can use for testing purposes. Table 2–5 describes the files you have constructed.

Table 2–5 Listing of Sample XML Files

File	Contents
`slideSample01.xml`	A basic file containing a few elements and attributes as well as comments.
`slideSample02.xml`	Includes a processing instruction.
`SlideSampleBad1.xml`	A file that is *not* well formed.
`slideSample03.xml`	Includes a simple entity reference (`<`).
`slideSample04.xml`	Contains a `CDATA` section.

Table 2–5 Listing of Sample XML Files

File	Contents
slideSample05.xml	References either a simple external DTD for elements (slideshow1a.dtd) for use with a nonvalidating parser, or else a DTD that defines attributes (slideshow1b.dtd) for use with a validating parser.
slideSample06.xml	Defines two entities locally (product and products) and references slideshow1b.dtd.
slideSample07.xml	References an external entity defined locally (copyright.xml) and references slideshow1b.dtd.
slideSample08.xml	References xhtml.dtd using a parameter entity in slideshow2.dtd, producing a naming conflict because title is declared in both.
slideSample09.xml	Changes the title element to slide-title so that it can reference xhtml.dtd using a parameter entity in slideshow3.dtd without conflict.