Well-Formed XML

WHAT YOU WILL LEARN IN THIS CHAPTER:

- The meaning of well-formed XML
- The constituent parts of an XML document
- How these parts are put together

So far you've looked at the history before XML, why it came about, and some of its advantages and disadvantages. You've also taken a whirlwind tour of some of the technologies associated with XML that are featured in this book.

In this chapter you'll be examining the rules that apply to a document that decide whether or not it *is* XML. This knowledge is needed in two main situations: first, when you're designing an XML format for your own data so that you can be sure that any standard XML parser can handle your document; second, when you are designing a system that will accept XML input from an external source so you'll be sure that the data you receive is legitimate XML. There are, unfortunately, a number of systems that purport to export data as XML but break some of the rules, meaning that unless you can get the problem fixed at source, you have to resort to handling the input using non-XML tools. This makes for a lot of unnecessary development and defeats the object of having a universally recognized method of data representation.

Additionally, you'll take a look at the basic and more advanced building blocks of XML starting with the most common, *elements* and *attributes*, and see how these are used to construct a complete document. You'll also be introduced to the modern terminology that describes these constituent parts; this is one of the major changes from earlier editions of this book as great efforts have been made in the XML world to have a vocabulary that is independent of the technology used to handle XML, yet is precise and extensive enough to enable the XML standards to be clearly written and form the basis for technological development.

WHAT DOES WELL-FORMED MEAN?

To the purist there is no such thing as well-formed XML; a document is either XML and therefore, by definition, well-formed, or it's just text. But in common parlance well-formed XML means a document that follows the W3C's XML Recommendation with all its rules governing the following:

- How the content is separated from the metadata (markup)
- What is used to identify the markup
- What the constituent parts are
- In what order and where these parts can appear

VARYING XML TERMINOLOGY

One small problem that exists when talking about XML is that its constituent parts can be described in many different ways. These varying descriptions have arisen for two reasons:

- The many different technologies associated with XML each have their own jargon; only a few terms are common to all of them. For instance the Document Object Model (covered in Chapter 7) and XSLT (covered in Chapter 8) have very different vocabularies for the same concepts
- The official W3C XML recommendations were finalized long after XML had been in use. The terms used in these documents often differ from the vernacular.

This chapter tries to stick with the terminology used by the W3C in two Recommendations: the first, simply called Extensible Markup Language (`www.w3.org/TR/xml`), describes the lexical representation or, in simpler terms, how XML is created in a text editor. The second, called Infoset Recommendation (`www.w3.org/TR/xml-infoset/`), describes an idealized abstract representation of an XML document.

CREATING XML IN A TEXT EDITOR

Creating XML in a text editor, something as simple as Notepad in Windows or Vim in Linux, is the first place to start when discussing the elements of XML. Throughout the process of creating XML, you gradually build up an example document and, at each stage, identify the constituent parts and what rules need to be followed in their construction.

Forbidden Characters

The first thing to know before writing XML is that a few restrictions exist on what characters are permitted in an XML document. These rules vary slightly depending on whether you're using

version 1.0 or 1.1, the latter being a bit more permissive. Both versions forbid the use of *null* in a document; this is the character represented by 0x0 in hexadecimal. In version 1.0 you are also forbidden to use the characters represented by the hexadecimal codes between 0x01 and 0x19, except for three: the tab (0x9), the newline (0xA), and the carriage return (0xD).

NOTE *These three characters, and a fourth, the standard space character (0x20), are collectively known as whitespace and have special rules governing their treatment in XML. These rules are covered later in the chapter.*

For example, you cannot use the character 0x7, known as the bell character, because it sounds a bell or a beep on some systems. In version 1.1 you can use all these control characters although their use is a little unusual. You see how to specify which version you are using in the next section. A few characters in the Unicode specification also can't be used but you're unlikely to come across these. You can find the full list in the W3C's XML Recommendation.

XML Prolog

The first part of a document is the *prolog*. It is optional so you won't see it every time, but if it does exist it must come first. The prolog begins with an XML *declaration* which, in its simplest form, looks like the following:

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

This declaration contains only one piece of information, the version number, and currently this will always be either 1.0 or 1.1. Sometimes the declaration may also contain information about the encoding used in the document:

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

Here the encoding is specified as UTF-8, a variety of Unicode.

Encoding with Unicode

Encoding is the process of turning characters into their equivalent binary representation. Some encodings use only a single byte, or eight bits; others use more. The disadvantage of using only one byte is that you are limited to how many characters can be encoded without recourse; this can go to such means as having a special sequence of bits to indicate that the next two bytes refer to one character or other similar workarounds. When an XML processor reads a document, it has to know which encoding was used; but, it's a chicken-and-egg situation — if it doesn't know the encoding how can it read what you've put in the declaration? The simple answer to this lies in the fact that the first few bytes of a file can contain a *byte order mark*, or *BOM*. This helps

the parser enough to be able to read the encoding specified in the declaration. Once it knows this it can decode the rest of the document. If, for some reason, the encoding specified is not the actual encoding used you'll most likely get an error, or mistakes will be made interpreting the content. If you want to see the full workings about how encodings are decided the URL is `www.w3.org/TR/2008/REC-xml-20081126/#sec-guessing`.

Unicode is a text encoding specification designed from scratch with internationalization in mind. It tries to define every possible character by giving it a name and a *code point*, which is a number that can be used to represent it. It also assigns various categories to each character such as whether it's a letter, a numeral, or a punctuation mark. You will see how to use these code points when you look at character references later in the chapter.

Two main encoding systems use Unicode: UTF-8 and UTF-16. *UTF* stands for *UCS Transformation Format*, and *UCS* itself means *Universal Character Set*. The number refers to how many bits are used to represent a simple character, either 8 or 16 (one or two bytes, respectively). The reason UTF-8 manages with only one byte whereas UTF-16 needs two is because UTF-8 uses a single byte to represent the more commonly used characters and two or three bytes for the less common ones. UTF-16 uses two bytes for the majority of characters and three bytes for the rest. It's a bit like your keyboard — the lowercase letters and digits require only one key press but by using the Shift key you have access to the uppercase letters and other symbols. The advantage of UTF-16 is that it's easier to decode because of its fixed size of two bytes per character (very few need three); the disadvantage is that file sizes are typically larger than UTF-8 if you are only using the Latin alphabet plus the standard numerals and punctuation marks

All XML processors are mandated to understand UTF-8 and UTF-16 even if those are the only encodings they can read. UTF-8 is the default for documents without encoding information. Despite the advantages of Unicode, many documents use other encodings such as ISO-8859-1, Windows-1252, or EBCDIC (an encoding found on many mainframes). You will also come across files written using ASCII — a basic set of characters that at one time was used for almost all files created. ASCII is a subset of Unicode though so it can be read by any application that understands Unicode.

NOTE *You will often see the side effects of files being encoded in one system and then decoded using another when browsing the Web — seemingly meaningless characters appear interspersed with otherwise readable text. This is a byproduct of the files often being created on one machine, uploaded to a second, the web server, and then read by a third, the one running the browser. If the encoding is not correctly interpreted by all three machines in the chain then you'll get some characters misinterpreted. You'll notice how the gibberish characters are usually those not found in ASCII and hence have different code points in different systems.*

In practical terms the UTF-8 encoding is probably best because it has a wide range of characters and is supported by all XML parsers. UTF-8 encoding is also the default assumed if no specific encoding is declared. If you do run into the problem of creating or reading files encoded with characters UTF-8 doesn't recognize, you should still manage without many problems by just

creating these character yourself. You'll learn how to do this later in the "Entity and Character References" section. Additionally, the Unicode specification grows in time as more characters are added. You can find the current version at `http://unicode.org`.

Completing the Declaration

Now that you have specified the type of encoding you are using, you can finish the declaration. The final part of the declaration is determining whether the document is considered to be *standalone*:


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

Example.xml

Standalone applies only to documents that specify a DTD and then only when that DTD is used to add or change content. `Example.xml` isn't using a DTD (remember that most modern XML formats rely on schemas instead), therefore you can set the standalone declaration to yes or leave it out altogether.

NOTE *DTD stands for document type definition and is a way to specify the format the XML should take as well as describing any default content that should appear and how references within the XML should be interpreted. Chapter 4 is devoted to DTDs.*

If you were to ever use a DTD, an example for an XHTML document would look something like this: `<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">`. Chapter 4 goes into more detail on DTD declarations.

Sometimes there are a few additional, elements to the XML prolog. These optional parts include *comments* and *processing instructions*. Processing instructions are discussed later this chapter. Comments are usually meant for human consumption and are not supposed to be part of the actual data in a document. They are initiated by the sequence `<!--` and terminated by `-->`. Following is `example.xml` with a comment added:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!-- This is a comment that follows the XML declaration -->
```

In general, comments are solely for the benefit of humans; you might want to include the date you created the file, your name, and other author details. However, if you think that the file will only be processed by a software application there's little point inserting them.

Once the XML prolog is finished you need to create the *root element* of the document. The following section details elements and how to create them.

Creating Elements

Elements are the basic building blocks of XML and all documents will have at least one. All elements are defined in one of two ways. At its simplest, an element with content consists of a start tag, which is a left angle bracket (<) followed by the name of the element, such as `myElement`, and then a right angle bracket (>). So a full start tag might be `<myElement>`. To close the element the end tag starts with a left angle bracket, a forward slash, and then the name of the element and a right angle bracket. So the end tag for `<myElement>` would be `</myElement>`. You can add spaces after the name in a start tag, such as `<myElement >`, but not before the name as in `< myElement>`. You can add this to `Example.xml`:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!-- This is a comment that follows the XML declaration -->
  <myElement></myElement>
```

Example.xml

There is an alternative syntax used to define an element, and this can only be used for elements with no content:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!-- This is a comment that follows the XML declaration -->
  <myElement />
```

This sort of element is known as *self-closing*.

Naming Styles

In addition to the two ways to define an element, there are a few different naming styles for elements and, as in many things IT-related, people can get quite evangelical about them. The one thing almost everyone agrees on is to be consistent; choose a style for the document and stick with it. Following are the main contenders for how you should name your elements — the main idea is how you distinguish separate words in an element name:

- **Pascal-casing:** This capitalizes separate words including the first: `<MyElement />`.
- **Camel-casing:** Similar to Pascal except that the first letter is lowercase: `<myElement />`.
- **Underscored names:** Use an underscore to separate words: `<my_element />`.
- **Hyphenated names:** Separate words with a hyphen: `<my-element />`.

While there are many other styles to use, these four seem to work the best.

Naming Specifications

Along with naming styles come a few specific rules used when naming elements that you must follow. These main rules include the following:

- An element name can begin with either an underscore or an uppercase or lowercase letter from the Unicode character set. This means you can use the Roman alphabet used by English and many other Western languages, the Cyrillic one used by Russian and its language

relatives, characters from Greek, or any of the other numerous scripts, such as Thai or Arabic, that are defined in the Unicode standard.

- Subsequent characters can also be a dash (-) or a digit.
- Names are case-sensitive, so the start and end tags must match exactly.
- Names cannot contain spaces
- Names beginning with the letters XML, either in uppercase- or lowercase, are reserved, and shouldn't be used (although many parsers allow them in practice).

NOTE *Just because names are case-sensitive doesn't mean it's sensible to have two elements that differ only by case, such as* `<myElement />` *and* `<MyElement />`*. Just as this would be poor practice for variable names in a case-sensitive programming language such as C#, you should not have elements with such similar names in XML.*

In theory you can also use a colon (:) as part of a name but this conflicts with the way XML Namespaces (covered in the next chapter) are handled, so in practice you should avoid using it. If you want to see the full range of element naming specifications, visit `www.w3.org/TR/2008/REC-xml-20081126/#NT-Name`.

Formatting your elements correctly is critical to creating well-formed XML. Table 2-1 provides some examples of correctly and incorrectly formed elements:

TABLE 2-1: Legal and illegal elements

LEGAL ELEMENT	REASON	ILLEGAL ELEMENT	REASON
`<myElement> </myElement>`	Spaces are allowed after a name.	`<my Element />`	Names cannot contain spaces.
`<my1stElement/>`	Digits can appear within a name.	`<1stName />`	Names cannot begin with a digit.
`<myElement />`	Spaces can appear between the name and the forward slash in a self-closing element.	`< myElement />`	Initial spaces are forbidden.
`<my-Element />`	A hyphen is allowed within a name.	`<-myElement/>`	A hyphen is not allowed as the first character.
`<όνομα />`	Non-roman characters are allowed if they are classified as letters by the Unicode specification. In this case the element name is *forename* in Greek.	`<myElement> </MyElement>`	Start and end tags must match case-sensitively.

Root Element

The next step after writing the prolog is creating the *root element*. All documents must have one and only one root element. Everything else in the document lies under this element to form a hierarchical tree. The rule stating that there can only be one root element is one of the keystones of XML, yet it has led to many complaints and a lot of people have put forward cases where having more than one "root" would be advantageous. One example is when using XML as a logging format. A typical log file might look like this:

```
<entry date="2012-03-03T10:09:53" type="audit">Failed logon attempt
 with username jfawcett</entry>
<entry date="2012-03-03T10:11:01" type="audit">Successful
 logon attempt with username jfawcett</entry>
<entry date="2012-03-03T10:12:11" type="information">Successful folder
 synchronisation for use jfawcett</entry>
```

This is an easy format to manage. Each time the machine wants to add a log entry it opens the relevant file and writes one line to the end of it, a standard task for any system. The problem with this format, though, is that there isn't a unique root element; you have to add one to make it well-formed:

```
<log>
  <entry date="2012-03-03T10:09:53" type="audit">Failed logon attempt
   with username jfawcett</entry>
  <entry date="2012-03-03T10:11:01" type="audit">Successful
   logon attempt with username jfawcett</entry>
  <entry date="2012-03-03T10:12:11" type="information">
  Successful folder synchronisation for use jfawcett</entry>
</log>
```

But now, with only one root element, it's difficult to add new entries. A simple file writer would have to open the file, find the closing log tag (`</log>`), and then add a line. Alternatively, the file could be opened by a parser, the root element (`<log>`) found, and a new child `<entry>` added at the end of all the other `<entry>` children. This task is much more process-heavy, and might prove to be a problem if dozens of entries need to be created every minute.

However the XML standards committees have stuck to their guns, deciding that the advantages of having a single, all-encompassing element, (the main one being easier parsing) outweigh the issues, such as the difficulty creating log files. They have, however, agreed that there is a need for such a construct and it is known as a *document fragment*. Document fragments do not need a single root element but they cannot be processed in isolation; they need to be nested inside a document that does have a single root. There are a number of ways that this can be done and some are covered in the "Entity Declarations" section of Chapter 4.

Other Elements

Underneath the root element can lie other elements that follow the same rules for naming and attributes and, as you saw earlier, there can also be free text. These nested elements can be used to show individual or repetitive items of data depending on what you are trying to represent. For example, your root element could be `<person>` and the elements underneath could show the person's

characteristics, such as <biography>and <address>. Alternatively, your main element could be <people> and underneath that you could have one or more <person> elements, each with its own children. You can add more elements and comments to the example document like so:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This is a comment that follows the XML declaration -->
<!DOCTYPE myElement [
  <!ENTITY nbsp  " ">
]>
<myElement myFirstAttribute="One"mySecondAttribute="Two">
Here is some text with a non-breaking space in it.
  <anotherElement>
    <aNestedElement anotherAttribute="Some data here">
Some more text</aNestedElement>
    <!-- a second comment -->
  </anotherElement>
</myElement>
```

Remember that all elements must be nested underneath the root element, so the following sort of markup, which you may have gotten away with in HTML, is not allowed:

```
<myElement>
  <elementA><elementB></elementA></elementB>
</myElement>
```

You can't have the end tag of an element before the end tag of one nested below it.

Attributes

Elements are one of the two main building blocks of XML — the other one is *attributes*. Attributes are name-value pairs associated with an element. You can add a couple of attributes to the example document like so:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!-- This is a comment that follows the XML declaration -->
  <myElement myFirstAttribute="One" mySecondAttribute="Two"></myElement>
```

The way you style your attribute names should be consistent with the one chosen for elements, so don't mix and match like this: <applicationUser first-name="Joe" />, where you have camel-casing for the element names and hyphenated attributes.

A number of rules also govern attributes exist:

- Attributes consist of a name and a value separated by an equals sign. The name, for example, myFirstAttribute, follows the same rules as element names.
- The attribute value must be in quotes. You can use either single or double quotes, the choice is entirely yours. You can use single on some attributes and double on others, but you can't mix them in a single attribute.
- There must be a value part, even if it's just empty quotes. You can't have something like <option selected> as you might in HTML.

- Attribute names must be unique per element.
- If you use double quotes as the delimiter you can't also use them as part of the value. The same applies for single quotes.

Table 2-2 provides some examples of correct and incorrect usage:

TABLE 2-2: Legal and illegal attributes

LEGAL ATTRIBUTE	REASON	ILLEGAL ATTRIBUTE	REASON
`<myElement value="Joe's attribute" />`	Single quote inside double quote delimiters.	`<myElement 1stAttribute= "value" />`	Attribute names cannot begin with a digit.
`<myElement value='"a quoted value"' />`	Double quotes inside single quote delimiters.	`<myElement value= 'Joe's attribute' />`	Single quote inside single quote delimiters.
		`<myElement name="Joe" name="Fawcett" />`	Two attributes with the same name is not allowed.
		`<myElement name='Joe" />`	Mismatching delimiters.

Element and Attribute Content

Attribute values and elements can both contain character data (called *text* in normal parlance). You've already seen examples of attributes in earlier code snippets. A similar example for an element would be:

```
<myElement>Here is some character content</myElement>
```

In addition to the rules described previously, there are only two more restrictions to follow regarding character content. Two characters cannot appear in attribute values or direct element content: the ampersand (&) and the left angle bracket (<). You cannot use the latter because it's used to delimit elements and it can confuse the parser. You cannot use the former because it's used to begin entity and character references.

Entity and Character References

There are two ways of inserting characters into a document that cannot be used directly, either because they are forbidden by the specification or because they don't exist in the encoding you have chosen. The first is *entity references*. There are five entity references in XML, shown in the Table 2-3.

TABLE 2-3: Entity References

CHARACTER	REFERENCE
&	&amp;
<	&lt;
>	&gt;
"	&quot;
'	&apos;

References start with an ampersand and finish with a semicolon. The actual reference appears as the middle part and is an abbreviation of the character; for instance, `lt` stands for less than. So instead of using `&` or `<` as characters for instance, you must use the reference instead. References `&apos;` and `&quot;` are especially useful if you need an attribute value to contain both types of quote marks.

The references in Table 2-3 are the only built-in entity references. You can declare your own if you want using a DTD — an example of this is shown shortly.

Character references take a similar form. They begin with `&#` and end with a semicolon, but instead of an abbreviation as the middle part they have a number representing the character's Unicode code point. The number can be in hexadecimal or decimal. For example, if you wanted to represent the Greek letter omega (Ω) as a reference it would be `&#x03A9;` in hexadecimal or `&937#;` in decimal.

A common question in XML forums is how to represent the non-breaking space—the character that has no visible output but prevents two words joined by it from breaking across a line. It's commonly used in web pages for formatting purposes where it's represented by the reference ` `. You have four ways to insert this into an XML document. The first is to simply insert it as a character; there's often no need to use a reference at all. For example, in Microsoft Word you can type the omega character by first typing 3A9 and then hitting Alt+X. Various other methods exist for different editors. The Unicode code point of the non-breaking space is xA0 so the same technique can be used. The second and third ways use the character reference ` ` in hexadecimal and ` ` in decimal. The fourth method requires that you create a DTD at the start of the document and declare the entity. You might want to do this if the character is used many times in the XML and you want the reader to recognize it more easily. In HTML, the reference ` ` is used to insert a non-breaking space, so to mimic this in an XML document you'd do this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This is a comment that follows the XML declaration -->
<!DOCTYPE myElement [
  <!ENTITY nbsp " ">
]>
<myElement myFirstAttribute="One" mySecondAttribute="Two">
Here is some text with a non-breaking space in it.
</myElement>
```

The DTD (covered in more detail in Chapter 4) declares that the root element is named `myElement`, and then there's one entity declaration; wherever ` ` appears in the document the parser will read it as the Unicode character A0, a non-breaking space.

You could also use this method to add references that refer to more than one character. For instance, you may want a reference named copyright that outputs © *Wrox 2012* wherever you put `&copyright,` that way you can just update the DTD in one place if you want to change all your references to read © *Wrox 2013*. This is achieved in exactly the same way as the preceding example, using the following:

```
<!DOCTYPE myElement [
  <!ENTITY copyright "© Wrox 2012">
]>
```

See Chapter 4 for more on these types of references.

WARNING *It's important to remember that you can't add the forbidden characters, such as null, to your document using either entity or character references.*

Elements Versus Attributes

On many occasions you will have a choice whether to represent data as an element or an attribute. For example, take the `appUsers.xml` file from Chapter 1 (shown in Listing 2-1):

Available for download on Wrox.com

LISTING 2-1: appUsers.xml

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

You could choose to represent the users' first names and last names as elements instead as shown in Listing 2-2:

Available for download on Wrox.com

LISTING 2-2: appUsers-elementCentric.xml

```
<applicationUsers>
  <user>
    <firstName>Joe</firstName>
    <lastName>Fawcett</lastName>
  </user>
  <user>
```

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

There are no fixed rules regarding whether you should use one form or the other, but the following are some things to consider when making your decision.

When to Use Attributes

Attributes are usually a good choice when there is only one piece of data to be shown. In Listing 2-1 a person can have only one first name so an attribute is the best choice. Attribute names cannot be repeated, though; putting a list into an attribute, perhaps by separating role names with a comma, makes the file difficult to work with when you want to extract and manipulate this data later. Therefore, if you need to show something like role names for a user multiple times, you would have to use elements.

Using attributes also results in a smaller file size because each element containing data needs an end tag as well as the small overhead of the angle brackets, which means more characters are being used to show the same amount of data. This might be a consideration if you know your files will often be sent across a network where bandwidth is an issue.

Typically, veering toward using attributes is a good idea unless there is a firm reason not to.

When to Use Elements

Elements are useful when the data is not a simple type—that is, some text or a date that can be easily represented as a string in an attribute value. So something like an address would be better split into its constituent parts and represented via elements rather than be represented as a delimited string and squashed into an attribute.

Therefore use this:

```
<person firstName="Joe" lastName="Fawcett">
  <address>
    <line1>Chapter House</line1>
    <line2>Crucifix Lane</line2>
    <city>London</city>
    <postCode>SE1 3JW</postCode>
    <country>England</country>
  </address>
</person>
```

Rather than this:

```
<person
   firstName="Joe"
   lastName="Fawcett"
   address="Chapter House, Crucifix Lane, London, SE1 3JW, England" />
```

Elements are also better when items may need to be repeated. To associate role names with a user in the file previously shown, a structure like this one would work best:

```
<applicationUsers>
  <user firstName="Joe" lastName="Fawcett">
    <roles>
      <role name="administrators" />
      <role name="general" />
    </roles
  <!-- other users here -->
</applicationUsers>
```

Notice how each `role` can have only one name, so an attribute rather than an element represents that portion.

Another plus for elements is that they can be ordered. Attributes are, by definition, unordered. You can place attributes in a special order in your document but the XML parser may ignore this order for processing purposes. If you need data items in a specific sequence, elements are the way to go.

The other major case for using an element is when you have a large amount of content that is just text. Technically, you could use an attribute in this instance, but that would mean you could get a file that looks like this:

```
<longDocument data="In here is a very long piece of text that goes on for many,
many,
many,
many,
lines" />
```

This can look very unusual. A file with a lot of text is normally easier to read if the content is within an element and possibly uses a CDATA section to avoid having to escape special characters.

Processing Instructions

Another common building block of an XML document is the *processing instruction*. You already saw one of these in Chapter 1 when you tried a browser-based XSL transformation. The processing instruction, or PI, is used to communicate with the application that is consuming the XML. It is not used directly by the XML parser at all.

A PI takes the form of a target that identifies which application should be carrying out the instruction and some data that is fed to the application. A common PI is the one that tells a browser to perform a transformation on the XML and looks like this:

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

The target in this example is `xml-stylesheet`, followed by two pseudo attributes: `type="text/xsl"` and `href="appUsers.xslt"`. They are not true attributes because they don't have to follow the rules of having a name and value and using quotes; they're just data that the target application will use. In this case a browser will recognize the target as saying that the XML should be transformed before

being shown; the first attribute states that the type of the transform is XSL and the second attribute points to its location. This particular processing instruction works only for a limited number of applications, mostly browsers; if you open the file in a standard text editor the PI will be ignored.

NOTE *Processing instructions and XML declarations look quite similar, but the declaration is not technically a processing instruction and therefore is not handled as such.*

CDATA Sections

One further construct you may need to use in a document is known as a *CDATA section* (*CDATA* stands for *character data* and means that no markup is present). These are used as a way to avoid repetitive escaping of characters. For example, suppose you have a simple document that contains information that makes use of the less than sign (<). Normally this is taken as part of the markup so it must be escaped using the entity reference `<`. So your document may look like this:

```
<conversionData>
  1 kilometer &lt; 1 mile
  1 pint &lt; 1 liter
  1 pound &lt; 1 kilogram
</conversionData>
```

If you'd prefer the text to use the readily recognizable < sign, which makes it easier for humans to read and write, you can mark the element's contents as a CDATA section:

```
<conversionData><![CDATA[
  1 kilometer < 1 mile
  1 pint < 1 liter
  1 pound < 1 kilogram
]]></conversionData>
```

The CDATA section starts with `<![CDATA[` and ends with `]]>`. Anything inside is considered text, not markup, and you can use any characters that normally need escaping such as the less than sign and the ampersand. If you need to represent the combination `]]>`, which marks the end of a section, you'll have to escape the final character of the sequence as so: `]]>`.

A common use of CDATA is in XHTML, the XML version of HTML. When you need to embed some JavaScript in an XHTML page, many of the characters often need escaping. Rather than doing this, which often then confuses the JavaScript parser, you can wrap the whole script section in a CDATA section such as the following example. This example tests whether a customer is trying to transfer more money from his account than he actually has:

```
<script type=text/javascript>
//<![CDATA[
function validateTransfer(currentBalance, transferAmount)
{
  if (currentBalance > 0 && transferAmount < currentBalance)
  {
    return true;
  }
  alert("Insufficient funds to transfer the requested amount.");
  return false;
}
//]]>
</script>
```

Because the text has been wrapped in a CDATA section, the JavaScript can be written in its standard form; otherwise, if the test for the transfer amount is less than the current balance you would have to escape the `&&` and the `<` sign as shown in the following code:

```
if (currentBalance > 0 &amp;&amp; transferAmount &lt; currentBalance)
```

This leaves a line that's difficult for both a human and a script parser to interpret.

Another noteworthy item is the JavaScript comments (`//`) before the CDATA section start and end markers. This is meant to help older browsers that don't know how to handle the construct.

Remember that a CDATA section is only a visual aid for human readers. The XML parser won't treat the two preceding examples differently, so once the XML has been parsed you won't be able to tell whether the character data was escaped using references or marked as CDATA. Some people use CDATA sections as a way to embed one XML document inside another, like this:

```
<myDocument>
  <someData>
    <myNestedDocument><![CDATA[
      <anotherDocument>This is bad practice</anotherDocument>
    ]]></myNestedDocument>
  </someData>
</myDocument>
```

This sort of XML is difficult to work with and should be avoided. The correct way to handle multiple documents—without mixing up what belongs where—is with namespaces, which are covered in the following chapter.

ADVANCED XML PARSING

You've now covered all the common building blocks of an XML document and can move on to more advanced matters. The next three major areas of discussion relating to advanced XML parsing include the following:

- **XML equivalence:** How documents that are written differently can still be treated as identical by the XML parser.

- **Whitespace handling:** How characters such as spaces and tabs receive special treatment.
- **Error handling:** What happens if your document contains an error.

XML Equivalence

XML equivalence refers to the idea that many documents, though having a different lexical representation, are considered equal by the XML parser. Once the document has been parsed it is impossible to tell if a particular style was used to create the XML. For example, the following two documents, Listing 2-3 and Listing 2-4, differ in three places:

LISTING 2-3: Document 1

```
<exampleData source="web">
  <section><![CDATA This is some example data ]]></section>
  <section>Here's some more data</section>
</exampleData>
```

LISTING 2-4: Document 2

```
<exampleData source='web'>
  <section>This is some example data</section>
  <section>Here&apos;s some more data</section>
</exampleData>
```

The three lexical differences are as follows:

- In the first file the attribute values are enclosed in double quotes whereas the second uses single quotes.
- The first file has a CDATA section for the first `<section>` element whereas the second doesn't.
- Finally, the second file uses an entity reference for the apostrophe in the second `<section>` element and the first does not.

There is nothing wrong with either of these two representations, it's purely a matter of personal preference but once either of these files is parsed it will be impossible to tell which one was the source. To the parser, how attribute values have been quoted and other differences have no bearing on the internal representation of the XML. Therefore, these documents have achieved XML equivalence.

The fact that more than one lexical version of an XML document can lead to the same in-memory representation has some negative effects. For example, if you are going to transform the file and want to treat data in a CDATA section differently than text that isn't in a CDATA section, you're out of luck and will need a different approach. You can handle variations like these differently if you prefer though, perhaps by preprocessing the file using a non-XML tool to add some markup identifying which elements need to be treated differently.

Similarly, a common request in the XML forums is that people want to create XML with various characters represented by references rather than the characters themselves, similar to Listing 2-4 where the apostrophe was shown as `&apos;`. The reason for this is that the software that processes

the XML "needs" this particular format. The fact that these two forms are XML equivalent indicates that the receiving application is not XML-compliant, otherwise it would accept either variation, and the best thing to do with this request is to throw it back and fix the relevant application. Obviously this isn't always going to be possible, but again there's no way to insist on how an apostrophe is represented. You'd have to find a non–XML-based workaround in order to fulfill this particular need if you couldn't get the problem fixed at source.

Whitespace Handling

Whitespace is text that is composed of a space, a tab, a newline, or a carriage return (the characters defined in Unicode as 0x20, 0x9, 0xA, and 0xD, respectively). Whitespace does not include the non-breaking space you saw earlier in the chapter that is in common use in web pages. Although the non-breaking space cannot be seen it is treated as though it can be, just like any other letter or punctuation mark:

Whitespace doesn't sound like it would cause problems; any human reading a document can usually cope with blank lines or occasionally two spaces together. But in XML it has to be treated carefully for the following two main reasons:

- First, some whitespace is significant. Take a standard English sentence; each word is separated by a space, and without these spaces the text would be difficult to read. On the other hand, some whitespace is insignificant — many books have blank pages at the front or back that wouldn't be missed and don't add to the content. In XML there is a similar situation; you will see a few examples shortly.
- The second reason whitespace is important in XML is that different operating systems use different conventions regarding such things as the line endings in a file which are constructed using the newline and carriage return whitespace characters. For example, a Windows-produced text file would normally have line endings that mimic the old typewriter action: a carriage return and a newline. On the contrary, files created on a UNIX environment will just have the newline character.

Therefore, whitespace handling rules exist to provide a consistent experience, and so that XML files can be portable data formats. For example, when an XML file is processed, the line endings (whether they started as a carriage return and newline, just newline, or just carriage return) will all be transformed into a single newline character.

Significant and Insignificant Whitespace

There is a big difference between how whitespace in XML documents is handled by XML parsers and how HTML is treated by browsers. If you are used to developing HTML you know that multiple consecutive whitespace characters are merged and that newlines and carriage returns are typically ignored. For example, the following HTML markup has lots of whitespace between the letters and two newlines in the middle.

```
<p>Here is some text     with    lots of   whitespace

             That won't show in a browser</p>
```

Figure 2-1 shows what you'll see in a browser.

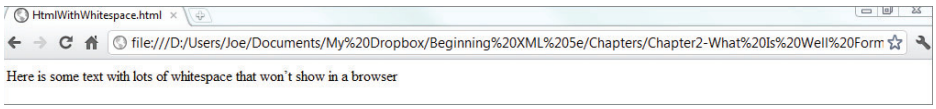

FIGURE 2-1

You can see that neither the long spaces nor the newline characters are shown. This is because HTML browsers normalize space by merging multiple spaces into one single space and ignoring newlines. If you do want a newline character to appear in HTML you need to use markup, specifically the `<br />` element.

XML isn't as strict as HTML; the parser preserves whitespace that's not part of markup. Take, for example, the following snippet:

```
<chapter   title="What is XML? ">
  <para> XML stands for Extensible Markup Language (presumably the original authors
 thought that sounded more exciting than EML) and has followed a path similar to
others in the software and IT world.</para>
</chapter>
```

This design is such that there will never be any text directly inside the `<chapter>` element, but it will just contain `<para>` elements. Due to this design, the whitespace at the end of the start tag and the general indentation are just there to make it easier for a human to read and see the document's structure. There are also some extra spaces between the `<chapter>` element's name and its attribute, `title`. All this whitespace is deemed insignificant. Any whitespace that appears inside the `<para>` element is known as significant.

An XML parser can choose to ignore insignificant whitespace — but how does it know that the element has no direct text content? It can only know the content type of an element if there is an XML schema or DTD associated with the document. Schemas and DTDs are not fully covered until Chapters 4 and 5; suffice it to say that insignificant whitespace does not have to be preserved by the parser. If you do want all of your document's whitespace to remain, there is a special attribute that you can add to either the root element or one lower down: `xml:space="preserve"`. This informs the parser to leave your document completely intact. So for the preceding example you'd add this to the `<chapter>` element:

```
<chapter xml:space="preserve"  title="What is XML? ">
  <para> XML stands for Extensible Markup Language (presumably the original authors
 thought that sounded more exciting than EML) and has followed a path similar to
others in the software and IT world.</para>
</chapter>
```

There is also an `xml:space="default"` if you need to reset whitespace handling back to its default.

WARNING *As stated previously, XML parsers are obliged to preserve whitespace unless they categorically know it to be insignificant. Unfortunately Microsoft's COM-based parser, known as MSXML Core Services, falls from grace here and ignores the standards by stripping what it considers to be insignificant space. This has caused many problems in the past for developers who were forced to use this parser (for example, when working inside Internet Explorer). You can overcome the problem in some scenarios by setting the* `preserveWhitespace` *property of the parser to* `true`.

Error Handling

As you've seen you have a few hurdles to overcome to make sure your document is classified as XML — matching tags, quoted attribute values, and escaped characters when necessary, to name a few. If an XML parser finds that one of the rules has been broken it has two main options, which depend on whether the specification states that it's an *error* or a *fatal error*.

You can recover from an error and continue document parsing if possible. A fatal error, as its name suggests, cannot be recovered from and the processor's only option is to report it. It can also report other errors if necessary, but it can't produce a parsed document at the end of the procedure. Any errors that are to do with well-formedness are considered fatal.

This strict view was deliberately taken after seeing how allowing HTML writers to be lax with their syntax has adversely affected the Web. Because browsers accept incorrect HTML and try to second guess the author's intention, there is inconsistency on how such pages are displayed — each browser has a different set of rules on how to cope with badly formed content. It also means that web pages cannot easily be processed by machines to extract meaningful information without first putting the content through a number of different algorithms. This was one of the problems that an XML-based version of HTML, namely XHTML, was meant to solve. Unfortunately it didn't work in practice because it was too difficult to learn for many. Additionally there was not enough toolset support and many browsers, Internet Explorer in particular, couldn't handle it properly.

At the end of the day you can compare XML strict error checking to standard programming languages. Some enforce strict type checking at compile time whereas others only fail at runtime. The XML view is that it's better to find errors earlier even if that means having the document rejected for only minor glitches.

Most browsers have good error reporting facilities and are often used to help find errors in a document that aren't immediately obvious. They are usually very strict and will terminate processing on errors even if they are not defined as fatal. This is common with nearly all parsers and is in line with the specification that only states that they *may* recover. In practice it's easier to just stop when an error is encountered than try to repair it by divining the author's original intention. This would also lead to discrepancies in how parsers handled documents and lead to a similar unwanted situation to that previously mentioned in regards to web pages. The following Try It Out deliberately creates a badly-formed file to demonstrate how error reporting is handled in a browser.

TRY IT OUT Using a Browser to Find Errors

To see how errors are reported in a browser use the following code file:

Available for download on Wrox.com

```
<?xml version="1.0" encoding="utf-8"?>
<pangrams createdOn="2012-01-04T10:19:45'>
  <!-- This file is designed to show
  how errors are reported in a browser -->
  <pangram>The quick brown fox jumps over the lazy dog.</pangram>
  <pangram>Pack my box with five dozen liquor jugs.</pangram>
  <pangram>Glib jocks quiz nymph to vex dwarf.</pangram>
  <pangram>The five boxing wizards jump quickly.</Pangram>
  <pangram>What you write deserves better than a jiggling, shaky,
     inexact & questionably fuzzy approximation of blur</pangram>
</pangrams>
```

XmlFileWithErrors.xml

This file contains three errors, which may or may not be immediately apparent depending on your familiarity with XML and your general proofreading abilities. However, these errors are reported differently in the browser in Figure 2-2 than how you might describe them yourself. In each case the browser stops after reporting the error and simply shows content up to that point where possible. Google's Chrome browser is used for this demo.

1. Browse to the file locally and you see the first error report, shown in Figure 2-2.

FIGURE 2-2

2. Look carefully at the file source you'll see that the `createdOn` attribute uses mismatched quotes — a double quote to start with and a single quote to finish. However, this isn't the error that's reported. Instead the parser complains that there is an unescaped < present. This is because it thinks the attribute hasn't been closed yet and suddenly it's found an illegal character. So far it hasn't read any textual content so there's nothing to display apart from the error message.
3. Correct the error by replacing the single quote with a double quote and reopen the file in Chrome. You get a new message as shown in Figure 2-3.

FIGURE 2-3

4. This time Chrome reports mismatching tag names. The fourth `<pangram>` element has `<pangram>` for a start tag but `</Pangram>` for the end tag and tags must match by case. This time, though, some content has been read and so the text of the first three `<pangram>` elements is shown. Correct the mismatch and reopen the file. The final error is reported, as shown in Figure 2-4.

FIGURE 2-4

5. This slightly confusing report claims you have an entity reference without a name. This is because in XML the ampersand (`&`) is used to begin a character or entity reference as discussed previously. In the file there is no name and no closing semicolon, so the parser thinks the reference is malformed rather than the simpler explanation, which is that you've forgotten to escape the `&` altogether. Correct this final error and you see the entire file in Figure 2-5.

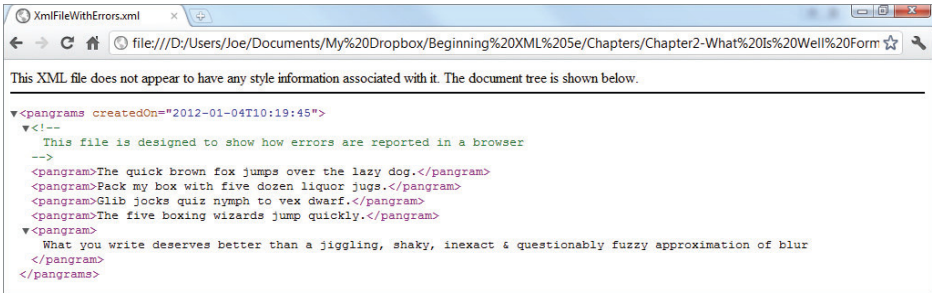

FIGURE 2-5

This time the file is displayed as a familiar tree and you can see the comment, plus you can expand and collapse various regions.

How It Works

XML parsers typically work in a sequential fashion when checking for errors. They start reading through the file and as soon as an error is encountered it is reported and any further parsing ceases. This explains how, on each occasion, only one error is displayed by Chrome and how, after fixing that error, the parser is able to get a little further in the parsing each time until, eventually, all three errors are removed and the browser can display the whole XML.

NOTE *If you're wondering about the file's contents, a* pangram *is a sentence in a language that uses each letter of the alphabet at least once.*

Read the XML specification carefully to learn more about determining which errors are defined as fatal and which are not. Fatal errors are clearly pointed out, and any other infringement is a recoverable error. As already noted, however, most parsers treat all errors as fatal and the best policy is to make sure any files you create are entirely free of them and that files received by you are perfect. If you do receive files that have errors your best choice, if you can't afford to simply reject them, is to correct them using non-XML means before moving them to your XML processing pipeline.

Now that you understand the building blocks of an XML file, the rules that need to be followed when creating them, and how errors are reported you can move on to the next stage. The next section deals with how a document is treated once it has been parsed.

THE XML INFOSET

Sometime after the W3C's XML recommendation was published a need arose for a common way to talk about the structure of an XML document after it had been parsed.

Up until this point there were many ways of describing a document depending on which technology it was being used with. The document object model (DOM) referred to elements and attributes as different types of nodes, as well as all the other building blocks such as comments and processing instructions. Other technologies (as you see in later chapters) had other terms, and part of the new model's job was to come up with a common vocabulary. This model was also meant to abstract away the individual differences in the way an original file had been written, such as whether it used single- or double–quotation marked attribute values. Additionally, the new model enables other XML-related applications, such as those used to transform XML, to work against an idealized picture of the document. The model was given the name the *XML Information Set* and is now commonly called the *XML Infoset*.

The XML Infoset consists of eleven components. This section takes a short look at each one to see how it relates to the underlying lexical representation. These components have the official title of *information items*.

The Document Information Item

Every XML document has one document information item. This item enables access to all the other items in the document as well as possessing a number of properties of its own. Some of these properties are those seen in the XML declaration such as character encoding and whether the document is standalone. Others are those such as the *Base URI*, which is essentially a pointer to the document's source, and the document element, which is the outermost element (what up until now has been referred to as the root element). In the older DOM terminology the document information item most closely resembles the root node of the entire XML.

To navigate from this item to other information items you can use the `Document Element` property, which points to the root element, and the `Children` property , which gives access to any comments or processing instructions that lie in the XML prolog (that is, before the root element).

Element Information Items

Element information items provide access to all element-related information, and each element has one associated item. The element information item has a number of properties, including:

- **Local Name:** The name of the element without any namespace prefixes (this is covered in the next chapter). For example, the local name of both `<pangram>` and `<ns:pangram>` would be `pangram`.
- **Children:** Any elements, comments, processing instructions, and references beneath this element.
- **Attributes:** An unordered list of all the attributes of this element. Note that attributes are not considered to be children of an element.
- **Parent:** The element, or in the case of the document element, the document, that has this element as its child.

Attribute Information Items

Attribute information items give access to each attribute found on an element. Properties include:

- **Local Name:** As for elements, this is the name without a namespace prefix.
- **Normalized Value:** The value of the attribute after the standard whitespace normalization, such as various line feeds, all being changed to a single newline character and all references being expanded.
- **Owner Element:** The element for which this attribute appears in its attributes property.

Processing Instruction Information Items

One processing instruction information item will be present for each processing instruction (PI) in the document. The properties include the target, which represents the target of the PI, and content, which is the rest of the text that appears. Quite often the content is split into what looks like attributes—set of name/value pairs—but that's not mandatory, so the information item does not parse the content any further.

Character Information Item

In theory each character that appears in the document, either literally as a character reference or within a CDATA section, will have an associated character information item. The properties of these include:

- **Character Code:** A value in the range of 0 to #x10FFFF indicating the character code. These codes are defined by the ISO 10646 standard which, for this interpretation, is the same as the Unicode one. Remember that some codes, such as 0, are not allowed in an XML document so you won't come across them if the XML is well-formed.
- **Element Content Whitespace:** This is a Boolean property indicating whether or not the character is whitespace within an element.
- **Parent:** The element that contains this item in its children property.

In practice, XML applications often group characters into strings of text because it's unlikely you'll want to process text one character at a time.

Comment Information Item

A comment information item refers to a comment in the source document. It has only two properties: `content`, which has the text of the comment, and `parent`.

Namespace Information Item

Each element in the document has one namespace information item for each namespace that is in scope. The wonderful world of namespaces is covered in the next chapter.

The Document Type Declaration Information Item

If a document has a document type declaration, this information item will have details about it. Properties include System Identifier and Public Identifier, which enable the XML to retrieve the DTD. This information item (as well as the following three: unexpected entity reference, unparsed entity, and notation) are only applicable when a document type definition is associated with the document.

Unexpanded Entity Reference Information Item

You're unlikely to come across these; they are placeholders for an external entity that has not been expanded. Most parsers will expand these references anyway, so they are quite rare.

Unparsed Entity Information Item

Again these are something declared in a DTD and you are unlikely to come across them.

Notation Information Item

One of these appears for each *notation* described in the DTD. Notations allow you to include references to non-XML content, such as images, in your XML document by declaring a reference to that content in the DTD.

In addition to the XML Infoset there is also a version known as the Post Schema Validation Infoset (PSVI), which brings additional information due to the fact that the XML has an associated schema and has been checked against that schema. You'll see some of this extra information in Chapter 5.

SUMMARY

- What exactly is meant by well-formed XML
- What characters are not allowed in an XML document
- What an encoding is and what is meant by Unicode
- The basic building blocks of XML, including elements and attributes
- How each of these blocks is formed and what rules need to be followed
- How some characters need to be escaped and how to represent characters that are not present in your chosen encoding
- How whitespace is handled
- How errors are handled
- What to think about when choosing between elements and attributes
- The XML Infoset and how it is an idealized model of an XML document

EXERCISES

Answers to exercises can be found in Appendix A.

1. What errors make the following file not well-formed?

```
<xmlLibrary>
  <play publicationYear=1898>
    <title>Arms & The Man</title>
    <author>George Bernard Shaw</author>
    <play description>Comedy dealing with the futility of war
 and the hypocrisy of human nature.</play description>
  <play>
  <play publicationYear=1950>
    <title>The Mousetrap</title>
    <author>Agatha Christie</author>
    <play description>A traditional whodunnit
 with an extraordinary twist.</play description>
  <play>
</xmlLibrary>
```

2. How would you declare an entity reference for your e-mail address so that you could easily change it in one place rather than many. Give a complete example.

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEYPOINTS
Why it's essential that your XML is well-formed	If XML is not well-formed then it's not XML and an XML parser won't be able to read it.
Certain characters are not allowed in XML documents	The null character (0x0) is forbidden in XML 1.0 and is the only forbidden Unicode character in XML 1.1. XML 1.0 also forbids the characters in the range 0x1 to 0x19 with the exception the whitespace characters tab (0x9), newline (0xa), and carriage return (0xd).
Some characters have special meanings	& and < are used for references and tags respectively so they need to be escaped as & and < if they are needed as text.
XML files can use a variety of characters encodings	If your file isn't in the UTF-8 encoding then the encoding must be declared in the XML declaration.
Basic structure	XML files are mainly built from elements and attributes. An XML document must have one all-encompassing root or document element.
Whitespace receives special treatment	In general, line endings consisting of newline and carriage return characters are compressed into a single newline and multiple spaces into a single space unless the parser considers them to be significant.
Elements or Attributes	In general, choose elements for complex structures or data that is repeated. Choose attributes for single atomic values.
The XML Infoset	The infoset is an idealized version of an XML document created after a document has been successfully parsed. Applications that consume XML should base their behavior on this structure so that do not rely on insignificant differences in how the document was written.