2

Well-Formed XML

We've discussed some of the reasons why XML makes sense for communicating data, so now let's get our hands dirty and learn how to create our own XML documents. This chapter will cover all you need to know to create "well-formed" XML.

> **Well-formed XML is XML that meets certain grammatical rules outlined in the XML 1.0 specification.**

You will learn:

- How to create XML **elements** using **start-** and **end-tags**
- How to further describe elements with **attributes**
- How to declare your document as being XML
- How to send instructions to applications that are processing the XML document
- Which characters aren't allowed in XML, and how to put them in anyway

Because XML and HTML appear so similar, and because you're probably already familiar with HTML, we'll be making comparisons between the two languages in this chapter. However, if you don't have any knowledge of HTML, you shouldn't find it too hard to follow along.

If you have Internet Explorer 5, you may find it useful to save some of the examples in this chapter on your hard drive, and view the results in the browser. If you don't have IE5, some of the examples will have screenshots to show what the end results look like.

Tags and Text and Elements, Oh My!

It's time to stop calling things just "items" and "text"; we need some names for the pieces that make up an XML document. To get cracking, let's break down the simple `<name>` document we created in Chapter 1:

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

The words between the < and > characters are XML **tags**. The information in our document (our data) is contained within the various tags that constitute the markup of the document. This makes it easy to distinguish the *information* in the document from the *markup*.

As you can see, the tags are paired together, so that any opening tag also has a closing tag. In XML parlance, these are called **start-tags** and **end-tags**. The end-tags are the same as the start-tags, except that they have a "/" right after the opening < character.

In this regard, XML tags work the same as start-tags and end-tags do in HTML. For example, you would create an HTML paragraph like this:

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

As you can see, there is a `<P>` start-tag, and a `</P>` end-tag, just like we use for XML.

All of the information from the start of a start-tag to the end of an end-tag, and including everything in between, is called an **element**. So:

- `<first>` is a start-tag
- `</first>` is an end-tag
- `<first>John</first>` is an element

The text between the start-tag and end-tag of an element is called the **element content**. The content between our tags will often just be data (as opposed to other elements). In this case, the element content is referred to as **Parsed Character DATA**, which is almost always referred to using its acronym, **PCDATA**.

> *Whenever you come across a strange-looking term like PCDATA, it's usually a good bet the term is inherited from SGML. Because XML is a subset of SGML, there are a lot of these inherited terms.*

The whole document, starting at `<name>` and ending at `</name>`, is also an element, which happens to include other elements. (And, in this case, the element is called the **root element**, which we'll be talking about later.)

To put this new-found knowledge into action, let's create an example that contains more information than just a name.

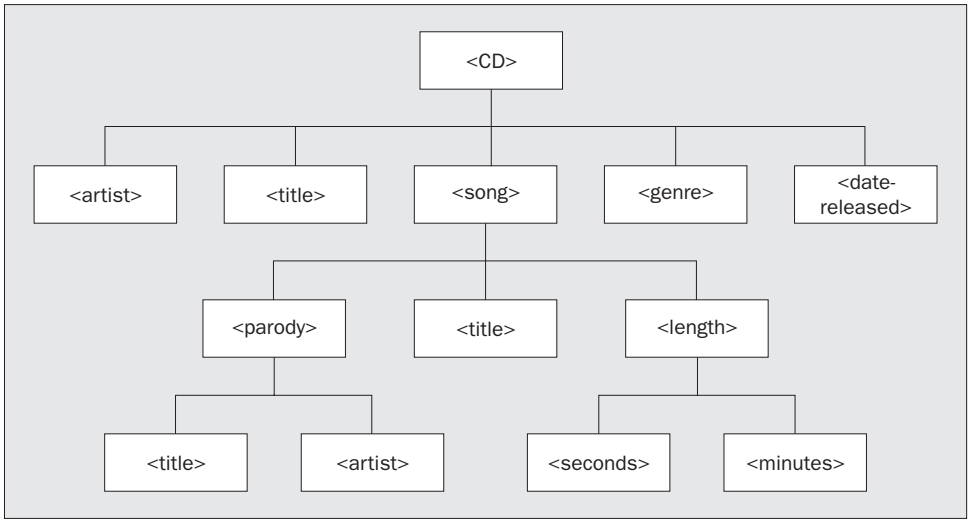
Try It Out – Describing Weirdness

We're going to build an XML document to describe one of the greatest CDs ever produced, *Dare to be Stupid*, by Weird Al Yankovic. But before we break out Notepad and start typing, we need to know what information we're capturing.

In Chapter 1, we learned that XML is hierarchical in nature; information is structured like a tree, with parent/child relationships. This means that we'll have to arrange our CD information in a tree structure as well.

1. Since this is a CD, we'll need to capture information like the artist, title, and date released, as well as the genre of music. We'll also need information about each song on the CD, such as the title and length. And, since Weird Al is famous for his parodies, we'll include information about what song (if any) this one is a parody of.

   Here's the hierarchy we'll be creating:



   Some of these elements, like `<artist>`, will appear only once; others, like `<song>`, will appear multiple times in the document. Also, some will have PCDATA only, while some will include their information as child elements instead. For example, the `<artist>` element will contain PCDATA for the title, whereas the `<song>` element won't contain any PCDATA of its own, but will contain child elements that further break down the information.

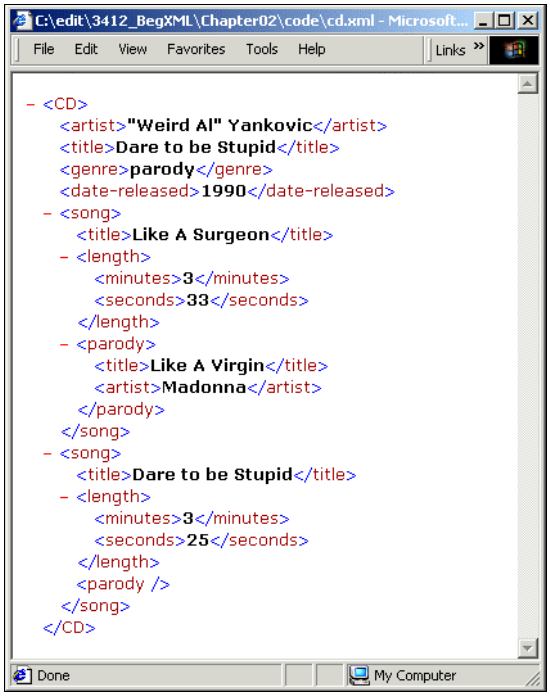
2. With this in mind, we're now ready to start entering XML. If you have Internet Explorer 5 installed on your machine, type the following into Notepad, and save it to your hard drive as `cd.xml`:

```
<CD>
  <artist>"Weird Al" Yankovic</artist>
  <title>Dare to be Stupid</title>
  <genre>parody</genre>
```

```
    <date-released>1990</date-released>
    <song>
      <title>Like A Surgeon</title>
      <length>
        <minutes>3</minutes>
        <seconds>33</seconds>
      </length>
      <parody>
        <title>Like A Virgin</title>
        <artist>Madonna</artist>
      </parody>
    </song>
    <song>
      <title>Dare to be Stupid</title>
      <length>
        <minutes>3</minutes>
        <seconds>25</seconds>
      </length>
      <parody></parody>
    </song>
</CD>
```

For the sake of brevity, we'll only enter two of the songs on the CD, but the idea is there nonetheless.

3. Now, open the file in IE5. (Navigate to the file in Explorer and double click on it, or open up the browser and type the path in the URL bar.) If you have typed in the tags exactly as shown, the `cd.xml` file will look something like this:



How It Works

Here we've created a hierarchy of information about a CD, so we've named the root element accordingly.

The `<CD>` element has children for the artist, title, genre, and date, as well as one child for each song on the disc. The `<song>` element has children for the title, length, and, since this is Weird Al we're talking about, what song (if any) this is a parody of. Again, for the sake of this example, the `<length>` element was broken down still further, to have children for minutes and seconds, and the `<parody>` element broken down to have the title and artist of the parodied song.

You may have noticed that the IE5 browser changed `<parody></parody>` into `<parody/>`. We'll talk about this shorthand syntax a little bit later, but don't worry: it's perfectly legal.

If we were to write a CD Player application, we could make use of this information to create a play-list for our CD. It could read the information under our `<song>` element to get the name and length of each song to display to the user, display the genre of the CD in the title bar, etc. Basically, it could make use of any information contained in our XML document.

Rules for Elements

Obviously, if we could just create elements in any old way we wanted, we wouldn't be any further along than our text file examples from the previous chapter. There must be some rules for elements, which are fundamental to the understanding of XML.

> **XML documents must adhere to these rules to be well-formed.**

We'll list them, briefly, before getting down to details:

- Every start-tag must have a matching end-tag
- Tags can't overlap
- XML documents can have only one root element
- Element names must obey XML naming conventions
- XML is case-sensitive
- XML will keep white space in your text

Every Start-tag Must Have an End-tag

One of the problems with parsing SGML documents is that not every element requires a start-tag and an end-tag. Take the following HTML for example:

```
<HTML>
<BODY>
<P>Here is some text in an HTML paragraph.
<BR>
Here is some more text in the same paragraph.
<P>And here is some text in another HTML paragraph.</p>
</BODY>
</HTML>
```

Notice that the first <P> tag has no closing </P> tag. This is allowed – and sometimes even encouraged – in HTML, because most web browsers can detect automatically where the end of the paragraph should be. In this case, when the browser comes across the second <P> tag, it knows to end the first paragraph. Then there's the <BR> tag (line break), which by definition has no closing tag.

Also, notice that the second <P> start-tag is matched by a </p> end-tag, in lower case. HTML browsers have to be smart enough to realize that both of these tags delimit the same element, but as we'll see soon, this would cause a problem for an XML parser.

The problem is that this makes HTML parsers much harder to write. Code has to be included to take into account all of these factors, which often makes the parsers much larger, and much harder to debug. What's more, the way that files are parsed is not standardized – different browsers do it differently, leading to incompatibilities.

For now, just remember that in XML the end-tag is required, and has to exactly match the start-tag.

Tags Can Not Overlap

Because XML is strictly hierarchical, you have to be careful to close your child elements before you close your parents. (This is called **properly nesting** your tags.) Let's look at another HTML example to demonstrate this:

```
<P>Some <STRONG>formatted <EM>text</STRONG>, but</EM> no grammar no good!</P>
```

This would produce the following output on a web browser:

Some **formatted *text***, *but* no grammar no good!

As you can see, the <STRONG> tags cover the text `formatted text`, while the <EM> tags cover the text `text, but`.

But is <EM> a child of <STRONG>, or is <STRONG> a child of <EM>? Or are they both siblings, and children of <P>? According to our stricter XML rules, the answer is none of the above. The HTML code, as written, can't be arranged as a proper hierarchy, and could therefore not be well-formed XML.

If ever you're in doubt as to whether your XML tags are overlapping, try to rearrange them visually to be hierarchical. If the tree makes sense, then you're okay. Otherwise, you'll have to rework your markup.

For example, we could get the same effect as above by doing the following:

```
<P>Some <STRONG>formatted <EM>text</EM></STRONG><EM>, but</EM> no grammar no
good!</P>
```

Which can be properly formatted in a tree, like this:

```
<P>
  Some
  <STRONG>
    formatted
    <EM>
```

```
        text
      </EM>
    </STRONG>
    <EM>
      , but
    </EM>
    no grammar no good!
  </P>
```

An XML Document Can Have Only One Root Element

In our <name> document, the <name> element is called the **root element**. This is the top-level element in the document, and all the other elements are its children or descendents. An XML document must have one and only one root element: in fact, it must have a root element even if it has no content.

For example, the following XML is not well-formed, because it has a number of root elements:

```
<name>John</name>
<name>Jane</name>
```

To make this well-formed, we'd need to add a top-level element, like this:

```
<names>
  <name>John</name>
  <name>Jane</name>
</names>
```

So while it may seem a bit of an inconvenience, it turns out that it's incredibly easy to follow this rule. If you have a document structure with multiple root-like elements, simply create a higher-level element to contain them.

Element Names

If we're going to be creating elements we're going to have to give them names, and XML is very generous in the names we're allowed to use. For example, there aren't any reserved words to avoid in XML, as there are in most programming languages, so we have a lot flexibility in this regard.

However, there are some rules that we must follow:

- Names can start with letters (including non-Latin characters) or the "_" character, but not numbers or other punctuation characters.
- After the first character, numbers are allowed, as are the characters "-" and ".".
- Names can't contain spaces.
- Names can't contain the ":" character. Strictly speaking, this character is allowed, but the XML specification says that it's "reserved". You should avoid using it in your documents, unless you are working with namespaces (which are covered in Chapter 8).
- Names can't start with the letters "xml", in uppercase, lowercase, or mixed – you can't start a name with "xml", "XML", "XmL", or any other combination.
- There can't be a space after the opening "<" character; the name of the element must come immediately after it. However, there can be space before the closing ">"character, if desired.

Here are some examples of valid names:

```
<first.name>
<résumé>
```

And here are some examples of invalid names:

```
<xml-tag>
```

which starts with xml,

```
<123>
```

which starts with a number,

```
<fun=xml>
```

because the "=" sign is illegal, and:

```
<my tag>
```

which contains a space.

Remember these rules for element names – they also apply to naming other things in XML.

Case-Sensitivity

Another important point to keep in mind is that the tags in XML are **case-sensitive**. (This is a big difference from HTML, which is case-insensitive.) This means that `<first>` is different from `<FIRST>`, which is different from `<First>`.

> *This sometimes seems odd to English-speaking users of XML, since English words can easily be converted to upper- or lower-case with no loss of meaning. But in almost every other language in the world, the concept of case is either not applicable (in other words, what's the uppercase of β? Or the lowercase, for that matter?), or is extremely important (what's the uppercase of é? The answer may be different, depending on the context). To put intelligent rules into the XML specification for case-folding would probably have doubled or trebled its size, and still only benefited the English-speaking section of the population. Luckily, it doesn't take long to get used to having case-sensitive names.*

This is the reason that our previous `<P></p>` HTML example would not work in XML; since the tags are case-sensitive, an XML parser would not be able to match the `</p>` end-tag with any start-tags, and neither would it be able to match the `<P>` start-tag with any end-tags.

> **Warning! Because XML is case-sensitive, you could legally create an XML document which has both `<first>` and `<First>` elements, which have different meanings. This is a bad idea, and will cause nothing but confusion! You should always try to give your elements distinct names, for your sanity, and for the sanity of those to come after you.**

To help combat these kinds of problems, it's a good idea to pick a naming style and stick to it. Some examples of common styles are:

- `<first_name>`
- `<firstName>`
- `<first-name>` (some people don't like this convention, because the "-" character is used for subtraction in so many programming languages, but it is legal)
- `<FirstName>`

Which style you choose isn't important; what is important is that you stick to it. A naming convention only helps when it's used consistently. For this book, I'll usually use the `<FirstName>` convention, because that's what I've grown used to.

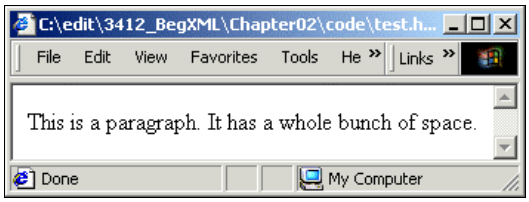
White Space in PCDATA

There is a special category of characters, called **white space**. This includes things like the space character, new lines (what you get when you hit the *Enter* key), and tabs. White space is used to separate words, as well as to make text more readable.

Those familiar with HTML are probably quite aware of the practice of white space stripping. In HTML, any white space considered insignificant is stripped out of the document when it is processed. For example, take the following HTML:

```
<P>This is a paragraph.       It has a whole bunch
  of space.</P>
```

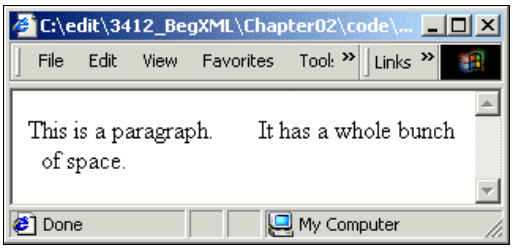
As far as HTML is concerned, anything more than a single space between the words in a `<P>` is insignificant. So all of the spaces between the first period and the word `It` would be stripped, except for one. Also, the line feed after the word `bunch` and the spaces before `of` would be stripped down to one space. As a result, the previous HTML would be rendered in a browser as:



In order to get the results as they appear in the HTML above, we'd have to add special HTML markup to the source, like the following:

```
<P>This is a paragraph.       It has a whole
bunch<BR>
  of space.</P>
```

` ` specifies that we should insert a space (`nbsp` stands for **Non-Breaking SPace**), and the `<BR>` tag specifies that there should be a line feed. This would format the output as:



Alternatively, if we wanted to have the text displayed exactly as it is in the source file, we could use the `<PRE>` tag. This specifically tells the HTML parser not to strip the white space, so we could write the following and also get the desired results:

```
<PRE>This is a paragraph.       It has a whole bunch
  of space.</PRE>
```

However, in most web browsers, the `<PRE>` tag also has the added effect that the text is rendered in a fixed-width font, like the courier font we use for code in this book.

White space stripping is very advantageous for a language like HTML, which has become primarily a means for displaying information. It allows the source for an HTML document to be formatted in a readable way for the person writing the HTML, while displaying it formatted in a readable, and possibly quite different, way for the user.

In XML, however, no white space stripping takes place for PCDATA. This means that for the following XML tag:

```
<tag>This is a paragraph.       It has a whole bunch
  of space.</tag>
```

the PCDATA is:

```
This is a paragraph.       It has a whole bunch
  of space.
```

Just like our second HTML example, none of the white space has been stripped out. As far as white space stripping goes, all XML elements are treated just as the HTML `<PRE>` tag. This makes the rules much easier to understand for XML than they are for HTML**:**

> **In XML, the white space stays.**

Unfortunately, if you view the above XML in IE5 the white space will *be stripped out – or will seem to be. This is because IE5 is not actually showing you the XML directly; it uses a technology called XSL to transform the XML to HTML, and it displays the HTML. Then, because IE5 is an HTML browser, it strips out the white space.*

End-of-Line White Space

However, there is one form of white space stripping that XML performs on PCDATA, which is the handling of **new line** characters. The problem is that there are two characters that are used for new lines – the **line feed** character and the **carriage return** – and computers running Windows, computers running Unix, and Macintosh computers all use these characters differently.

For example, to get a new line in Windows, an application would use both the line feed and the carriage return character together, whereas on Unix only the line feed would be used. This could prove to be very troublesome when creating XML documents, because Unix machines would treat the new lines in a document differently than the Windows boxes, which would treat them differently than the Macintosh boxes, and our XML interoperability would be lost.

For this reason, it was decided that XML parsers would change all new lines to a single line feed character before processing. This means that any XML application will know, no matter which operating system it's running under, that a new line will be represented by a single line feed character. This makes data exchange between multiple computers running different operating systems that much easier, since programmers don't have to deal with the (sometimes annoying) end-of-line logic.

White Space in Markup

As well as the white space in our data, there could also be white space in an XML document that's not actually part of the document. For example:

```
<tag>
  <another-tag>This is some XML</another-tag>
</tag>
```

While any white space contained within `<another-tag>`'s PCDATA is part of the data, there is also a new line after `<tag>`, and some spaces before `<another-tag>`. These spaces could be there just to make the document easier to read, while not actually being part of its data. This "readability" white space is called **extraneous white space**.

While an XML parser must pass all white space through to the application, it can also inform the application which white space is not actually part of an element's PCDATA, but is just extraneous white space.

So how does the parser decide whether this is extraneous white space or not? That depends on what kind of data we specify `<tag>` should contain. If `<tag>` can only contain other elements (and no PCDATA) then the white space will be considered extraneous. However, if `<tag>` is allowed to contain PCDATA, then the white space will be considered part of that PCDATA, so it will be retained.

Unfortunately, from this document alone an XML parser would have no way to tell whether `<tag>` is supposed to contain PCDATA or not, which means that it has to assume none of the white space is extraneous. We'll see how we can get the parser to recognize this as extraneous white space in Chapter 9 when we discuss content models.

Attributes

In addition to tags and elements, XML documents can also include **attributes**.

> **Attributes are simple name/value pairs associated with an element.**

They are attached to the start-tag, as shown below, but not to the end-tag:

```
<name nickname='Shiny John'>
  <first>John</first>
  <middle>Fitzgerald Johansen</middle>
  <last>Doe</last>
</name>
```

Attributes must have values – even if that value is just an empty string (like `""`) – and those values must be in quotes. So the following, which is part of a common HTML tag, is not legal in XML:

```
<INPUT checked>
```

and neither is this:

```
<INPUT checked=true>
```

Either single quotes or double quotes are fine, but they have to match. For example, to make this well-formed XML, you can use one of these:

```
<INPUT checked='true'>
<INPUT checked="true">
```

but you can't use:

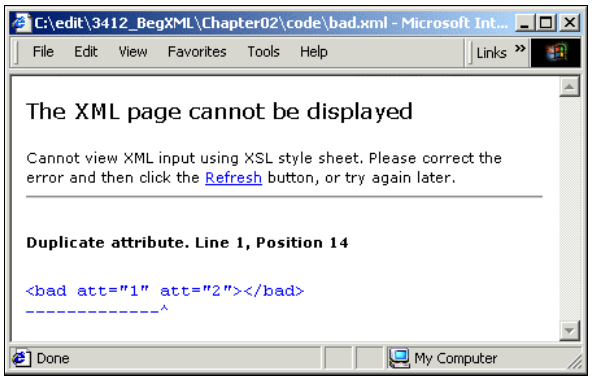
```
<INPUT checked="true'>
```

> *Because either single or double quotes are allowed, it's easy to include quote characters in your attribute values, like `"John's nickname"` or `'I said "hi" to him'`. You just have to be careful not to accidentally close your attribute, like `'John's nickname'`; if an XML parser sees an attribute value like this, it will think you're closing the value at the second single quote, and will raise an error when it sees the `"s"` which comes right after it.*

The same rules apply to naming attributes as apply to naming elements: names are case sensitive, can't start with "xml", and so on. Also, you can't have more than one attribute with the same name on an element. So if we create an XML document like this:

```
<bad att="1" att="2"></bad>
```

we will get the following error in IE5:



Try It Out – Adding Attributes to Al's CD

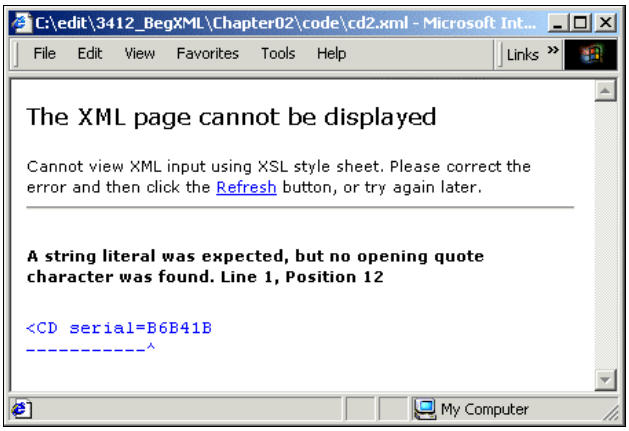
With all of the information we recorded about our CD in our earlier Try It Out, we forgot to include the CD's serial number, or the length of the disc. Let's add some attributes, so that our hypothetical CD Player application can easily find this information out.

1. Open your `cd.xml` file created earlier, and resave it to your hard drive as `cd2.xml`.

2. With our new-found attributes knowledge, add two attributes to the `<CD>` element, like this:

```
<CD serial=B6B41B
    disc-length='36:55'>
  <artist>"Weird Al" Yankovic</artist>
  <title>Dare to be Stupid</title>
  <genre>parody</genre>
  <date-released>1990</date-released>
  <song>
    <title>Like A Surgeon</title>
    <length>
      <minutes>3</minutes>
      <seconds>33</seconds>
    </length>
    <parody>
      <title>Like A Virgin</title>
      <artist>Madonna</artist>
    </parody>
  </song>
  <song>
    <title>Dare to be Stupid</title>
    <length>
      <minutes>3</minutes>
      <seconds>25</seconds>
    </length>
    <parody></parody>
  </song>
</CD>
```

3. If you typed in exactly what's written above, when you display it in IE5 it should look something like this:



4. Now edit the first attribute, like this:

```
<CD serial='B6B41B'
    disc-length='36:55'>
```

5. Re-save the file, and view it in IE5. It will look something like this:

How It Works

Using attributes, we added some information about the CD's serial number and length to our document:

```
<CD serial=B6B41B
    disc-length='36:55'>
```

When the XML parser got to the "=" character after the `serial` attribute, it expected an opening quotation mark, but instead it found a `B`. This is an error, and it caused the parser to stop and raise the error to the user.

So we changed our `serial` attribute declaration:

```
<CD serial='B6B41B'
```

and this time the browser displayed our XML correctly.

The information we added might be useful, for example, in the CD Player application we considered earlier. We could write our CD Player to use the serial number of a CD to load any previous settings the user may have previously saved (such as a custom play list).

Why Use Attributes?

There have been many debates in the XML community about whether attributes are really necessary, and if so, where they should be used. Here are some of the main points in that debate:

Attributes Can Provide Metadata that May Not be Relevant to Most Applications Dealing with Our XML

For example, if we know that some applications may care about a CD's serial number, but most won't, it may make sense to make it an attribute. This logically separates the data most applications will need from the data that most applications won't need.

In reality, there is no such thing as "pure metadata" – all information is "data" to *some* application. Think about HTML; you could break the information in HTML into two types of data: the data to be shown to a human, and the data to be used by the web browser to format the human-readable data. From one standpoint, the data used to format the data would be metadata, but to the browser or the person writing the HTML, the metadata *is* the data. Therefore, attributes can make sense when we're separating one type of information from another.

What Do Attributes Buy Me that Elements Don't?

Can't elements do anything attributes can do?

In other words, on the face of it there's really no difference between:

```
<name nickname='Shiny John'></name>
```

and:

```
<name>
  <nickname>Shiny John</nickname>
</name>
```

So why bother to pollute the language with two ways of doing the same thing?

The main reason that XML was invented was that SGML could do some great things, but it was too massively difficult to use without a fully-fledged SGML expert on hand. So one concept behind XML is a simpler, kinder, gentler SGML. For this reason, many people don't like attributes, because they add a complexity to the language that they feel isn't needed.

On the other hand, some people find attributes easier to use – for example, they don't require nesting and you don't have to worry about crossed tags.

Why Use Elements, if Attributes Take Up So Much Less Space?

Wouldn't it save bandwidth to use attributes instead?

For example, if we were to rewrite our `<name>` document to use only attributes, it might look like this:

```
<name nickname='Shiny John' first='John' middle='Fitzgerald Johansen'
last='Doe'></name>
```

Which takes up much less space than our earlier code using elements.

However, in systems where size is really an issue, it turns out that simple compression techniques would work much better than trying to optimize the XML. And because of the way compression works, you end up with almost the same file sizes regardless of whether attributes or elements are used.

Besides, when you try to optimize XML this way, you lose many of the benefits XML offers, such as readability and descriptive tag names. And there are cases where using elements allows more flexibility and scope for extension. For example, if we decided that `first` needed additional metadata in the future, it would be much simpler to modify our code if we'd used elements rather than attributes.

Why Use Attributes when Elements Look So Much Better? I Mean, Why Use Elements when Attributes Look So Much Better?

Many people have different opinions as to whether attributes or child elements "look better". In this case, it comes down to a matter of personal preference and style.

In fact, *much* of the attributes versus elements debate comes from personal preference. Many, but not all, of the arguments boil down to "I like the one better than the other". But since XML has both elements and attributes, and neither one is going to go away, you're free to use both. Choose whichever works best for your application, whichever looks better to you, or whichever you're most comfortable with.

Comments

Comments provide a way to insert into an XML document text that isn't really part of the document, but rather is intended for people who are reading the XML source itself.

Anyone who has used a programming language will be familiar with the idea of comments: you want to be able to annotate your code (or your XML), so that those coming after you will be able to figure out what you were doing. (And remember: the one who comes after you may be you! Code you wrote six months ago might be as foreign to you as code someone else wrote.)

Of course, comments may not be as relevant to XML as they are to programming languages; after all, this is just data, and it's self-describing to boot. But you never know when they're going to come in handy, and there are cases where comments can be very useful, even in data.

Comments start with the string `<!--` and end with the string `-->`, as shown here:

```
<name nickname='Shiny John'>
  <first>John</first>
  <!--John lost his middle name in a fire-->
  <middle></middle>
  <last>Doe</last>
</name>
```

There are a couple of points that we need to note about comments. First, you can't have a comment inside a tag, so the following is illegal:

```
    <middle></middle <!--John lost his middle name in a fire--> >
```

Second, you can't use the string `--` inside a comment, so the following is also illegal:

```
  <!--John lost his middle name -- in a fire-->
```

The XML specification states that an XML parser doesn't need to pass these comments on to the application, meaning that you should never count on being able to use the information inside a comment from your application.

> **HTML programmers have often used the trick of inserting scripting code in comments, to protect users with older browsers that didn't support the `<SCRIPT>` tag. That kind of trick can't be done in XML, since comments won't necessarily be available to the application. Therefore, if you have text that you need to get at later, *put it in an element or an attribute*!**

Try It Out – Some Comments On Al's CD

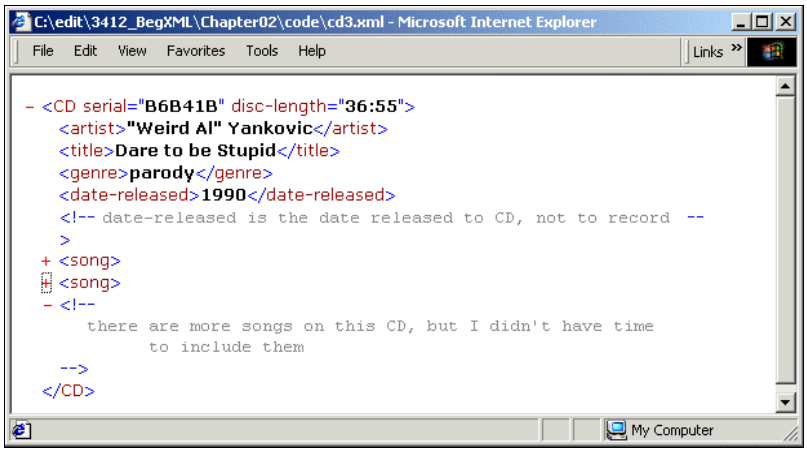
Since we've only included a couple of the songs from this fine album in our document, perhaps we should inform others that this is the case. That way some kind soul may finish the job for us!

1. Open up your `cd2.xml` file, make the following changes, and save the modified XML file as `cd3.xml`:

```
<CD serial='B6B41B'
    disc-length='36:55'>
  <artist>"Weird Al" Yankovic</artist>
  <title>Dare to be Stupid</title>
  <genre>parody</genre>
  <date-released>1990</date-released>
  <!--date-released is the date released to CD, not to record-->
  <song>
```

```
    <title>Like A Surgeon</title>
    <length>
      <minutes>3</minutes>
      <seconds>33</seconds>
    </length>
    <parody>
      <title>Like A Virgin</title>
      <artist>Madonna</artist>
    </parody>
  </song>
  <song>
    <title>Dare to be Stupid</title>
    <length>
      <minutes>3</minutes>
      <seconds>25</seconds>
    </length>
    <parody></parody>
  </song>
  <!--there are more songs on this CD, but I didn't have time to include
      them-->
</CD>
```

2. View this in IE5:



How It Works

With the new comments, anyone who reads the source for our XML document will be able to see that there are actually more than two songs on "Dare To Be Stupid". Furthermore, they can see some information regarding the `<date-released>` element, which may help them in writing applications that work with this information.

In this example, the XML parser included with IE5 *does* pass comments up to the application, so IE5 has displayed our comments. But remember that a lot of the time, for all intents and purposes this information is only available to people reading the source file. The information in comments *may or may not* be passed up to our application, depending on which parser we're using. We can't count on it, unless we specifically choose a parser that does pass them through. This means that the application has no way to know whether or not the list of songs included here is comprehensive.

Try It Out – Making Sure Comments Get Seen

If we really need this information, we should add in some real markup to indicate it.

1. Modify `cd3.xml` like this, and save it as `cd4.xml`:

```
<CD><!--our attributes used to be here-->
  <songs>11</songs>
  <!--the rest of our XML...-->
  <artist>"Weird Al" Yankovic</artist>
  <title>Dare to be Stupid</title>
  <genre>parody</genre>
  <date-released>1990</date-released>
  <song>
    <title>Like A Surgeon</title>
    <length>
      <minutes>3</minutes>
      <seconds>33</seconds>
    </length>
    <parody>
      <title>Like A Virgin</title>
      <artist>Madonna</artist>
    </parody>
  </song>
  <song>
    <title>Dare to be Stupid</title>
    <length>
      <minutes>3</minutes>
      <seconds>25</seconds>
    </length>
    <parody></parody>
  </song>
</CD>
```

2. This XML is formatted like this in IE5:

This way, the application could be coded such that if it only finds two `<song>` elements, but it finds a `<songs>` element which contains the text `"11"`, it can deduce that there are 9 songs missing.

Empty Elements

Sometimes an element has no data. Recall our earlier example, where the `middle` element contained no name:

```
<name nickname='Shiny John'>
  <first>John</first>
  <!--John lost his middle name in a fire-->
  <middle></middle>
  <last>Doe</last>
</name>
```

In this case, you also have the option of writing this element using the special **empty element** syntax:

```
<middle/>
```

This is the one case where a start-tag doesn't need a separate end-tag, because they are both combined together into this one tag. In all other cases, they do.

Recall from our discussion of element names that the only place we can have a space within the tag is before the closing ">". This rule is slightly different when it comes to empty elements. The "/" and ">" characters always have to be together, so you can create an empty element like this:

```
<middle />
```

but not like these:

```
<middle/ >
<middle / >
```

Empty elements really don't buy you anything – except that they take less typing – so you can use them, or not, at your discretion. Keep in mind, however, that as far as XML is concerned `<middle></middle>` is *exactly* the same as `<middle/>`; for this reason, XML parsers will sometimes change your XML from one form to the other. You should never count on your empty elements being in one form or the other, but since they're syntactically exactly the same, it doesn't matter. (This is the reason that IE5 felt free to change our earlier `<parody></parody>` syntax to just `<parody/>`.

> *Interestingly, nobody in the XML community seems to mind the empty element syntax, even though it doesn't add anything to the language. This is especially interesting considering the passionate debates that have taken place on whether attributes are really necessary.*

One place where empty elements are very often used is for elements that have no (or optional) PCDATA, but instead have all of their information stored in attributes. So if we rewrote our `<name>` example without child elements, instead of a start-tag and end-tag we would probably use an empty element, like this:

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

Another common example is the case where just the element name is enough; for example, the HTML `<BR>` tag might be converted to an XML empty element, such as the XHTML `<br/>` tag. (XHTML is the latest "XML-compliant" version of HTML.)

XML Declaration

It is often very handy to be able to identify a document as being a certain type. XML provides the **XML declaration** for us to label documents as being XML, along with giving the parsers a few other pieces of information. You don't need to have an XML declaration, but you should include it anyway.

A typical XML declaration looks like this:

```
<?xml version='1.0' encoding='UTF-16' standalone='yes'?>
<name nickname='Shiny John'>
  <first>John</first>
  <!--John lost his middle name in a fire-->
  <middle/>
  <last>Doe</last>
</name>
```

Some things to note about the XML declaration:

- The XML declaration starts with the characters `<?xml`, and ends with the characters `?>`.
- If you include it, you must include the `version`, but the `encoding` and `standalone` attributes are optional.
- The `version`, `encoding`, and `standalone` attributes must be in that order.
- Currently, the version should be `1.0`. If you use a number other than `1.0`, XML parsers that were written for the version 1.0 specification should reject the document. (As of yet, there have been no plans announced for any other version of the XML specification. If there ever is one, the version number in the XML declaration will be used to signal which version of the specification your document claims to support.)
- The XML declaration must be right at the beginning of the file. That is, the first character in the file should be that <; no line breaks or spaces. Some parsers are more forgiving about this than others.

So an XML declaration can be as full as the one above, or as simple as:

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

The next two sections will describe more fully the `encoding` and `standalone` attributes of the XML declaration.

Encoding

It should come as no surprise to us that text is stored in computers using numbers, since numbers are all that computers really understand.

> **A** character code **is a one-to-one mapping between a set of characters and the corresponding numbers to represent those characters. A** character encoding **is the method used to represent the numbers in a character code digitally, (in other words how many bytes should be used for each number, etc.)**

One character code/encoding that you might have come across is the **American Standard Code for Information Interchange** (**ASCII**). For example, in ASCII the character "a" is represented by the number 97, and the character "A" is represented by the number 65.

There are seven-bit and eight-bit ASCII encoding schemes. 8-bit ASCII uses one byte (8 bits) for each character, which can only store 256 different values, so that limits ASCII to 256 characters. That's enough to easily handle all of the characters needed for English, which is why ASCII was the predominant character encoding used on personal computers in the English-speaking world for many years. But there are way more than 256 characters in all of the world's languages, so obviously ASCII can only handle a small subset of these. This is reason that **Unicode** was invented.

Unicode

Unicode is a character code designed from the ground up with internationalization in mind, aiming to have enough possible characters to cover all of the characters in any human language. There are two major character encodings for Unicode: **UTF-16** and **UTF-8**. UTF-16 takes the easy way, and simply uses two bytes for every character (two bytes = 16 bits = 65,356 possible values).

UTF-8 is more clever: it uses one byte for the characters covered by 7-bit ASCII, and then uses some tricks so that any other characters may be represented by two or more bytes. This means that ASCII text can actually be considered a subset of UTF-8, and processed as such. For text written in English, where most of the characters would fit into the ASCII character encoding, UTF-8 can result in smaller file sizes, but for text in other languages, UTF-16 should usually be smaller.

Because of the work done with Unicode to make it international, the XML specification states that all XML processors must use Unicode internally. Unfortunately, very few of the documents in the world are encoded in Unicode. Most are encoded in **ISO-8859-1**, or **windows-1252**, or **EBCDIC**, or one of a large number of other character encodings. (Many of these encodings, such as ISO-8859-1 and windows-1252, are actually variants of ASCII. They are not, however, subsets of UTF-8 in the same way that "pure" ASCII is.)

Specifying Character Encoding for XML

This is where the `encoding` attribute in our XML declaration comes in. It allows us to specify, to the XML parser, what character encoding our text is in. The XML parser can then read the document in the proper encoding, and translate it into Unicode internally. If no encoding is specified, UTF-8 or UTF-16 is assumed (parsers must support at least UTF-8 and UTF-16). If no encoding is specified, and the document is not UTF-8 or UTF-16, it results in an error.

Sometimes an XML processor is allowed to ignore the encoding specified in the XML declaration. If the document is being sent via a network protocol such as HTTP, there may be protocol-specific headers which specify a different encoding than the one specified in the document. In such a case, the HTTP header would take precedence over the encoding specified in the XML declaration. However, if there are no external sources for the encoding, and the encoding specified is different from the actual encoding of the document, it results in an error.

If you're creating XML documents in Notepad on a machine running a Microsoft Windows operating system, the character encoding you are using by default is windows-1252. So the XML declarations in your documents should look like this:

```
<?xml version="1.0" encoding="windows-1252"?>
```

However, not all XML parsers understand the windows-1252 character set. If that's the case, try substituting ISO-8859-1, which happens to be very similar. Or, if your document doesn't contain any special characters (like accented characters, for example), you could use ASCII instead, or leave the `encoding` attribute out, and let the XML parser treat the document as UTF-8.

If you're running Windows NT or Windows 2000, Notepad also gives you the option of saving your text files in Unicode, in which case you can leave out the `encoding` attribute in your XML declarations.

Standalone

If the `standalone` attribute is included in the XML declaration, it must be either `yes` or `no`.

- `yes` specifies that this document exists entirely on its own, without depending on any other files
- `no` indicates that the document may depend on other files

This little attribute actually has its own name: the **Standalone Document Declaration**, or **SDD**. The XML specification doesn't actually require a parser to do anything with the SDD. It is considered more of a hint to the parser than anything else.

This is only a partial description of the SDD. If it has whetted your appetite for more, you'll have to be patient until Chapter 11, when all will be made clear.

It's time to take a look at how the XML declaration works in practice.

Try It Out – Declaring Al's CD to the World

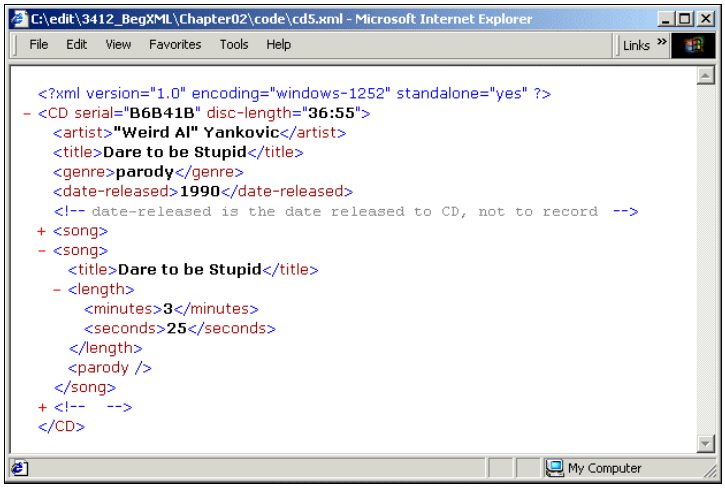
Let's declare our XML document, so that any parsers will be able to tell right away what it is. And, while we're at it, let's take care of that second `<parody>` element, which doesn't have any content.

1. Open up the file `cd3.xml`, and make the following changes:

```
<?xml version='1.0' encoding='windows-1252' standalone='yes'?>
<CD serial='B6B41B'
    disc-length='36:55'>
  <artist>"Weird Al" Yankovic</artist>
  <title>Dare to be Stupid</title>
  <genre>parody</genre>
  <date-released>1990</date-released>
  <!--date-released is the date released to CD, not to record-->
  <song>
    <title>Like A Surgeon</title>
    <length>
      <minutes>3</minutes>
      <seconds>33</seconds>
    </length>
    <parody>
      <title>Like A Virgin</title>
      <artist>Madonna</artist>
    </parody>
```

```
    </song>
    <song>
      <title>Dare to be Stupid</title>
      <length>
        <minutes>3</minutes>
        <seconds>25</seconds>
      </length>
      <parody/>
    </song>
    <!--There are more songs on this CD, but I didn't have time
        to include them!-->
</CD>
```

2. Save the file as `cd5.xml`, and view it in IE5:



How It Works

With our new XML declaration, any XML parser can tell right away that it is indeed dealing with an XML document, and that document is claiming to conform to version 1.0 of the XML specification.

Furthermore, the document indicates that it is encoded using the windows-1252 character encoding. Again many XML parsers don't understand windows-1252, so you may have to play around with the encoding. Luckily, the parser used by Internet Explorer 5 does understand windows-1252, so if you're viewing the examples in IE5 you can leave the XML declaration as it is here.

In addition, because the Standalone Document Declaration declares that this is a standalone document, the parser knows that this one file is all that it needs to fully process the information.

And finally, because "Dare to be Stupid" is not a parody of any particular song, the `<parody>` element has been changed to an empty element. That way we can visually emphasize the fact that there is no information there. Remember, though, that to the parser `<parody/>` is exactly the same as `<parody></parody>`, which is why this part of our document looks the same as it did in our earlier screenshots.

Processing Instructions

Although it isn't all that common, sometimes you need to embed application-specific instructions into your information, to affect how it will be processed. XML provides a mechanism to allow this, called **processing instructions** or, more commonly, **PIs**. These allow you to enter instructions into your XML which are not part of the actual document, but which are passed up to the application.

```
<?xml version='1.0' encoding='UTF-16' standalone='yes'?>
<name nickname='Shiny John'>
  <first>John</first>
  <!--John lost his middle name in a fire-->
  <middle/>
  <?nameprocessor SELECT * FROM blah?>
  <last>Doe</last>
</name>
```

There aren't really a lot of rules on PIs. They're basically just a "`<?`", the name of the application that is supposed to receive the PI (the **PITarget**), and the rest up until the ending "`?>`" is whatever you want the instruction to be. The PITarget is bound by the same naming rules as elements and attributes. So, in this example, the PITarget is `nameprocessor`, and the actual text of the PI is `SELECT * FROM blah`.

PIs are pretty rare, and are often frowned upon in the XML community, especially when used frivolously. But if you have a valid reason to use them, go for it. For example, PIs can be an excellent place for putting the kind of information (such as scripting code) that gets put in comments in HTML. While you can't assume that comments will be passed on to the application, PIs always are.

Is the XML Declaration a Processing Instruction?

At first glance, you might think that the XML declaration is a PI that starts with `xml`. It uses the same "`<? ?>`" notation, and provides instructions to the parser (but not the application). So is it a PI?

Actually, no: the XML declaration isn't a PI. But in most cases it really doesn't make any difference whether it is or not, so feel free to look at it as one if you wish. The only places where you'll get into trouble are the following:

- Trying to get the text of the XML declaration from an XML parser. Some parsers erroneously treat the XML declaration as a PI, and will pass it on as if it were, but many will not. The truth is, in most cases your application will never need the information in the XML declaration; that information is only for the parser. One notable exception might be an application that wants to display an XML document to a user, in the way that we're using IE5 to display the documents in this book.
- Including an XML declaration somewhere other than at the beginning of an XML document. Although you can put a PI anywhere you want, an XML declaration must come at the beginning of a file.

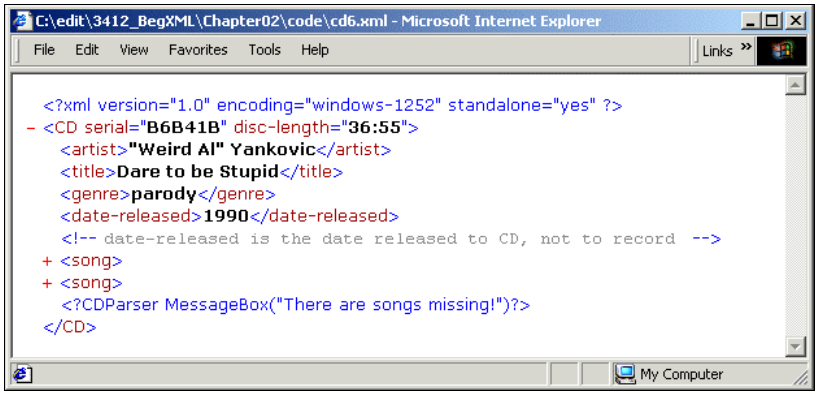
Try It Out – Dare to be Processed

Just to see what it looks like, let's add a processing instruction to our Weird Al XML:

1. Make the following changes to `cd5.xml` and save the file as `cd6.xml`:

```
<?xml version='1.0' encoding='windows-1252' standalone='yes'?>
<CD serial='B6B41B'
    disc-length='36:55'>
  <artist>"Weird Al" Yankovic</artist>
  <title>Dare to be Stupid</title>
  <genre>parody</genre>
  <date-released>1990</date-released>
  <!--date-released is the date released to CD, not to record-->
  <song>
    <title>Like A Surgeon</title>
    <length>
      <minutes>3</minutes>
      <seconds>33</seconds>
    </length>
    <parody>
      <title>Like A Virgin</title>
      <artist>Madonna</artist>
    </parody>
  </song>
  <song>
    <title>Dare to be Stupid</title>
    <length>
      <minutes>3</minutes>
      <seconds>25</seconds>
    </length>
    <parody/>
  </song>
  <?CDParser MessageBox("There are songs missing!")?>
</CD>
```

2. In IE5, it looks like this:



How It Works

For our example, we are targeting a *fictional* application called `CDParser`, and giving it the instruction `MessageBox("There are songs missing!")`. The instruction we gave it has no meaning in the context of XML itself, but only to our `CDParser` application, so it's up to `CDParser` to do something meaningful with it.

Illegal PCDATA Characters

There are some reserved characters that you can't include in your PCDATA because they are used in XML syntax.

For example, the "<" and "&" characters:

```
<!--This is not well-formed XML!-->
<comparison>6 is < 7 & 7 > 6</comparison>
```

Viewing the above XML in IE5 would give the following error:

This means that the XML parser comes across the "<" character, and expects a tag name, instead of a space. (Even if it had got past this, the same error would have occurred at the "`&`" character.)

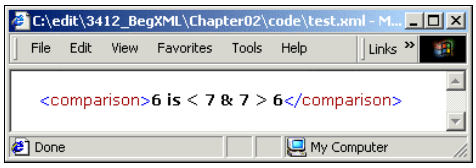
There are two ways you can get around this: **escaping characters**, or enclosing text in a **CDATA section**.

Escaping Characters

To escape these two characters, you simply replace any "<" characters with `<` and any "&" characters with `&`. The above XML could be made well-formed by doing the following:

```
<comparison>6 is &lt; 7 &amp; 7 &gt; 6 </comparison>
```

Which displays properly in the browser:



Notice that IE5 automatically un-escapes the characters for you when it displays the document, in other words it replaces the `<` and `&` strings with < and & characters.

`<` and `&` are known as **entity references**. The following entities are defined in XML:

- `&` – the & character
- `<` – the < character
- `>` – the > character
- `&apos;` – the ' character
- `&quot;` – the " character

Other characters can also be escaped by using **character references**. These are strings such as `&#`*nnn*`;`, where "*nnn*" would be replaced by the Unicode number of the character you want to insert. (Or `&#x`*nnn*`;` with an "x" preceding the number, where "*nnn*" is a hexadecimal representation of the Unicode character you want to insert. All of the characters in the Unicode specification are specified using hexadecimal, so allowing the hexadecimal numbers in XML means that XML authors don't have to convert back and forth between hexadecimal and decimal.)

Escaping characters in this way can be quite handy if you are authoring documents in XML that use characters your XML editor doesn't understand, or can't output, because the characters escaped are *always* Unicode characters, regardless of the encoding being used for the document. As an example, you could include the copyright symbol (©) in an XML document by inserting `&#169;` or `&#xA9;`.

CDATA Sections

If you have a lot of "<" and "&" characters that need escaping, you may find that your document quickly becomes very ugly and unreadable. Luckily, there are also **CDATA sections.**

> **CDATA is another inherited term from SGML. It stands for** Character DATA**.**

Using CDATA sections, we can tell the XML parser not to parse the text, but to let it all go by until it gets to the end of the section. CDATA sections look like this:

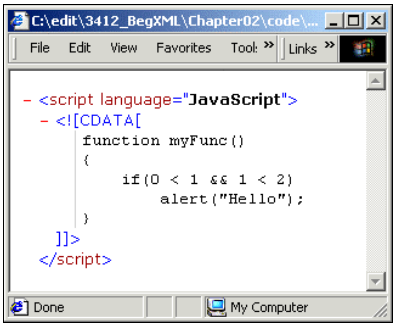
```
<comparison><![CDATA[6 is < 7 & 7 > 6]]></comparison>
```

Everything starting after the `<![CDATA[` and ending at the `]]>` is ignored by the parser, and passed through to the application as is. In this trivial case, CDATA sections may look more confusing than the escaping did, but in other cases it can turn out to be more readable. For example, consider the following example, which uses a CDATA section to keep an XML parser from parsing a section of JavaScript:

```
<script language='JavaScript'><![CDATA[
function myFunc()
{
    if(0 < 1 && 1 < 2)
        alert("Hello");
}
]]></script>
```

If you aren't familiar with JavaScript and want to know what the above script does, take a look at the tutorial in Appendix D.

This displays in the IE5 browser as:



Notice the vertical line at the left hand side of the CDATA section. This is indicating that although the CDATA section is indented for readability, the actual data itself starts at that vertical line. This is so we can visually see what white space is included in the CDATA section.

If you're familiar with JavaScript, you'll probably find the `if` statement much easier to read than:

```
if(0 &lt; 1 &amp;&amp; 1 &lt; 2)
```

Try It Out – Talking about HTML in XML

Suppose that we want to create XML documentation, to describe some of the various HTML tags in existence.

1. We might develop a simple document type such as the following:

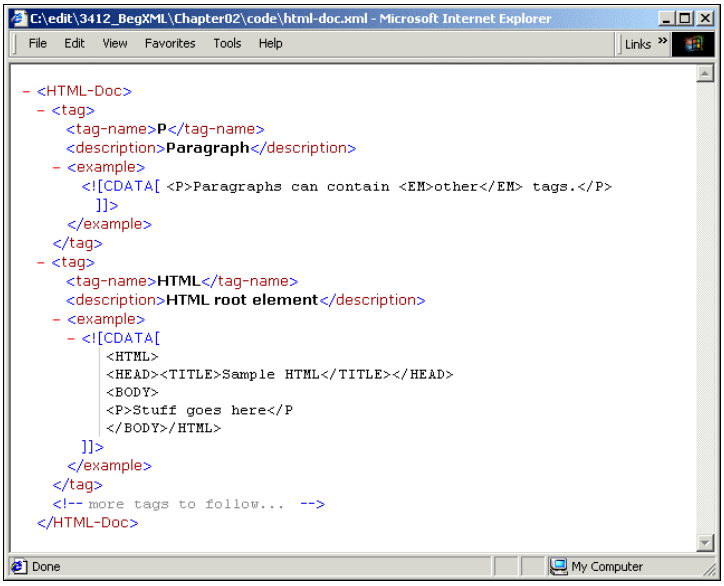
```
<HTML-Doc>
  <tag>
    <tag-name></tag-name>
    <description></description>
    <example></example>
  </tag>
</HTML-Doc>
```

In this case, we know for sure that our `<example>` element is going to need to include HTML syntax, meaning that there are going to be a lot of "<" characters included. This makes `<example>` the perfect place to use a CDATA section, meaning that we don't have to search through all of our HTML code looking for illegal characters. To demonstrate, lets document a couple of HTML tags.

2. Create a new file and type this code:

```
<HTML-Doc>
  <tag>
    <tag-name>P</tag-name>
    <description>Paragraph</description>
    <example><![CDATA[
<P>Paragraphs can contain <EM>other</EM> tags.</P>
]]></example>
  </tag>
  <tag>
    <tag-name>HTML</tag-name>
    <description>HTML root element</description>
    <example><![CDATA[
<HTML>
<HEAD><TITLE>Sample HTML</TITLE></HEAD>
<BODY>
<P>Stuff goes here</P
</BODY></HTML>
]]></example>
  </tag>
  <!--more tags to follow...-->
</HTML-Doc>
```

3. Save this document as `html-doc.xml` and view it in IE5:



How It Works

Because of our CDATA sections, we can put whatever we want into the `<example>` elements, and don't have to worry about the text being mixed up with the actual XML markup of the document. This means that even though there are typos in the second `<example>` element (the `</P` is missing the `>` and `/HTML>` is missing a `<`), our XML is not affected.

Parsing XML

The main reason for creating all of these rules about writing well-formed XML documents is so that we can create a computer program to read in the data, and easily tell markup from information.

> *According to the XML specification (http://www.w3.org/TR/1998/REC-xml-19980210#sec-intro): "A software module called an **XML processor** is used to read XML documents and provide access to their content and structure. It is assumed that an XML processor is doing its work on behalf of another module, called the **application**."*

An XML processor is more commonly called a **parser**, since it simply parses XML and provides the application with any information it needs. There are quite a number of XML parsers available, many of which are free. Some of the better known ones are listed below.

Microsoft Internet Explorer Parser

Microsoft's first XML parser shipped with Internet Explorer 4 and implemented an early draft of the XML specification. With the release of IE5, the XML implementation was upgraded to reflect the XML version 1 specification. The latest version of the parser (March 2000 Technology Preview Release) is available for download from http://msdn.microsoft.com/downloads/webtechnology/xml/msxml.asp. In this book we'll be mainly using the IE5 version.

James Clark's Expat

Expat is an XML 1.0 parser toolkit written in C. More information can be found at http://www.jclark.com/xml/expat.html and Expat can be downloaded from ftp://ftp.jclark.com/pub/xml/expat.zip. It is free for both private and commercial use.

Vivid Creations ActiveDOM

Vivid Creations (http://www.vivid-creations.com) offers several XML tools, including ActiveDOM. ActiveDOM contains a parser similar to the Microsoft parser and, although it is a commercial product, a demonstration version may be downloaded from the Vivid Creations web site.

DataChannel XJ Parser

DataChannel, a business solutions software company, worked with Microsoft to produce an early XML parser written in Java. Their website (http://xdev.datachannel.com/directory/xml_parser.html) provides a link to get their most recent version. However, they are no longer doing parser development. They have opted instead to use the xml4j parser from IBM.

IBM xml4j

IBM's AlphaWorks site (http://www.alphaworks.ibm.com) offers a number of XML tools and applications, including the xml4j parser. This is another parser written in Java, available for free, though there are some licensing restrictions regarding its use.

Apache Xerces

The Apache Software Foundation's Xerces sub-project of the Apache XML Project (http://xml.apache.org/) has resulted in XML parsers in Java and C++, plus a Perl wrapper for the C++ parser. These tools are in beta, they are free, and the distribution of the code is controlled by the GNU Public License.

Errors in XML

As well as specifying how a parser should get the information out of an XML document, it is also specified how a parser should deal with errors in XML. There are two types of errors in the XML specification: **errors** and **fatal errors**.

- An error is simply a violation of the rules in the specification, where the results are undefined; the XML processor is allowed to recover from the error and continue processing.
- Fatal errors are more serious: according to the specification a parser is *not allowed to continue as normal* when it encounters a fatal error. (It may, however, keep processing the XML document to search for further errors.) Any error which causes an XML document to cease being well-formed is a fatal error.

The reason for this drastic handling of non-well-formed XML is simple: it would be extremely hard for parser writers to try and handle "well-formedness" errors, and it is extremely simple to make XML well-formed. (HTML does not force documents to be as strict as XML does, but this is one of the reasons why web browsers are so incompatible; they must deal with *all* of the errors they may encounter, and try to figure out what the person who wrote the document was really trying to code.)

But draconian error handling doesn't just benefit the parser writers; it also benefits us when we're creating XML documents. If I write an XML document that doesn't properly follow XML's syntax, I can find out right away and fix my mistake. On the other hand, if the XML parser tried to recover from these errors, it may misinterpret what I was trying to do, but I wouldn't know about it because no error would be raised. In this case, bugs in my software would be much harder to track down, instead of being caught right at the beginning when I was creating my data.

Summary

This chapter has provided you with the basic syntax for writing well-formed XML documents.

We've seen:

- Elements and empty elements
- How to deal with white space in XML
- Attributes
- How to include comments
- XML declarations and encodings
- Processing instructions
- Entity references, character references and CDATA sections

We've also learned why the strict rules of XML grammar actually benefit us, in the long run, and how some of the rules for authoring HTML are different from the rules for authoring well-formed XML.

Unfortunately – or perhaps fortunately – you probably won't spend much of your time just authoring XML documents. But once you have the data in XML form, you still have to be able to use that data. In the chapters that follow we'll learn some of the other technologies surrounding XML, which will help you to make use of your data, starting with one of the most common: display.