Extensible Markup Language

XML

Pocket Reference

XML Pocket Reference

Robert Eckstein

First Edition, October 1999

ISBN: 1-56592-709-5, 112 pages

The XML Pocket Reference is both a handy introduction to XML terminology and syntax, and a quick reference to XML instructions, attributes, entities, and datatypes.

This small book acts both as a perfect tutorial for learning the basics of XML and as a reference to the XML and XSL specifications.

XML Pocket Reference

XML, the Extensible Markup Language, is the next-generation markup language for the Web. It provides a more structured (and therefore more powerful) medium than HTML, allowing us to define new document types and stylesheets as needed. Although the generic tags of HTML are sufficient for everyday text, XML gives us a way to add rich, well-defined markup to electronic documents.

The XML Pocket Reference is both a handy introduction to XML terminology and syntax, and a quick reference to XML instructions, attributes, entities, and datatypes. It also covers XSL (Extensible Stylesheet Language), necessary to ensure that your XML documents have a consistent look and feel across platforms.

Although XML itself is complex, its basic concepts are simple. This small book acts both as a perfect tutorial for learning the basics of XML, and as a reference to the XML and XSL specifications.

Chapter 1. XML Pocket Reference

The Extensible Markup Language (XML) is a document processing standard proposed by the World Wide Web Consortium (W3C), the same group responsible for overseeing the HTML standard. Although the exact specifications have not been completed yet, many expect XML and its sibling technologies to replace HTML as the markup language of choice for dynamically generated content, including nonstatic web pages. Already several browser and word processor companies are integrating XML support into their products.

XML is actually a simplified form of Standard Generalized Markup Language (SGML), an international documentation standard that has existed since the 1980s. However, SGML is extremely bulky, especially for the Web. Much of the credit for XML's creation can be attributed to Jon Bosak of Sun Microsystems, Inc., who started the W3C working group responsbile for scaling down SGML to a form more suitable for the Internet.

Put succinctly, XML is a *meta-language* that allows you to create and format your own document markups. With HTML, existing markup is static: `<HEAD>` and `<BODY>`, for example, are tightly integrated into the HTML standard and cannot be changed or extended. XML, on the other hand, allows you to create your own markup tags and configure each to your liking: for example, `<HeadingA>`, `<Sidebar>`, `<Quote>`, or `<ReallyWildFont>`. Each of these elements can be defined through your own *document type definitions* and *stylesheets* and applied to one or more XML documents. Thus, it is important to realize that there are no "correct" tags for an XML document, except those you define yourself.

While many XML applications currently support Cascading Style Sheets (CSS), a more extensible stylesheet specification exists called the *Extensible Stylesheet Language* (XSL). By using XSL, you ensure that your XML documents are formatted the same no matter which application or platform they appear on.

Note: As you read this, the XSL specification is still in flux. There have been several rumors regarding XSL and the formatting object's portions of the specifications changing dramatically. (There is even one rumor about XSL becoming its own language in the future.) However, even if XSL changes dramatically in the future, the material presented here should give you a firm foundation and enough expertise to make any leap of knowledge much easier.

This book offers a quick overview of XML, as well as some sample applications that allow you to get started in coding. We won't cover everything about XML. In fact, much of XML is still in flux as this book goes to print. Consequently, creating a definitive reference at this point in XML's life seems a bit futile. However, after you read this book, we hope that the components that make up XML will seem a little less foreign.

1.1 XML Terminology

Before we move further, we need to standardize some terminology. An XML document consists of one or more *elements*. An element is marked with the following form:

```
<Body>
This is text formatted according to the Body element
</Body>
```

This element consists of two *tags*, an opening tag which places the name of the element between a less-than sign (`<`) and a greater-than sign (`>`), and a closing tag which is identical except for the forward slash (`/`) that appears before the element name. Like HTML, the text contained between the opening and closing tags is considered part of the element and is formatted according to the element's rules.

Elements can have *attributes* applied, such as the following:

```
<Price currency="Euro">25.43</Price>
```

Here, the attribute is specified inside of the opening tag and is called "currency." It is given a value of "Euro," which is expressed inside quotation marks. Attributes are often used to further refine or modify the default behavior of an element.

In addition to the standard elements, XML also supports *empty elements*. An empty element has no text appearing between the opening and closing tag. Hence, both tags can (optionally) be merged together, with a forward slash appearing before the closing marker. For example, these elements are identical:

```
<Picture src="blueball.gif"></Picture>
<Picture src="blueball.gif"/>
```

Empty elements are often used to add nontextual content to a document, or to provide additional information to the application that is parsing the XML. Note that while the closing slash may not be used in single-tag HTML elements, it is *mandatory* for single-tag XML empty elements.

1.1.1 Unlearning Bad Habits

Whereas HTML browsers often ignore simple errors in documents, XML applications are not nearly as forgiving. For the HTML reader, there are a few bad habits from which we should first dissuade you:

Attribute values must be in quotation marks.

You can't specify an attribute value such as `<picture src=/images/blueball.gif>`, an error that HTML browsers often overlooked. An attribute value must always be inside single or double quotation marks, or the XML parser will flag it as an error. Here is the correct way to specify such a tag:

```
<picture src="/images/blueball.gif">
```

A non-empty element must have an opening and closing tag.

Each element that specifies an opening tag must have a closing tag that matches it. If it does not, and it is not an empty element, the XML parser generates an error. In other words, you cannot do the following:

```
<Paragraph>
This is a paragraph.
<Paragraph>
This is another paragraph.
```

Instead, you must have an opening and closing tag for each paragraph element:

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

Tags must be nested correctly.

It is illegal to do the following:

```
<Italic><Bold>This is incorrect</Italic></Bold>
```

The closing tag for the `Bold` element should be inside the closing tag for the `Italic` element, to match the nearest opening tag and preserve the correct element nesting. It is essential for the application parsing your XML to process the hierarchy of the elements:

```
<Italic><Bold>This is correct</Bold></Italic>
```

These syntactic rules are the source of many common errors in XML, especially given that some of this behavior can be ignored by HTML browsers. An XML document that adheres to these rules (and a few others which we'll see later) is said to be *well-formed*.

1.1.2 An Overview of an XML Document

There are generally three files that are processed by an XML-compliant application to display XML content:

The XML document

This file contains the document data, typically tagged with meaningful XML elements, some of which may contain attributes.

A stylesheet

The stylesheet dictates how document elements should be formatted when they are displayed, whether it be in a word processor or a browser. Note that you can apply different stylesheets to the same document, depending on the environment, thus changing its appearance without affecting any of the underlying data. The separation between content and formatting is an important distinction in XML.

Document Type Definition (DTD)

This file specifies rules for how the XML document elements, attributes, and other data are defined and logically related in an XML-compliant document.

1.1.3 A Simple XML Document

Example 1-1 shows a simple XML document.

Example 1-1. simple.xml

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE OReilly:Books SYSTEM "sample.dtd">
<!-- Here begins the XML data -->
<OReilly:Books xmlns:OReilly=\(ashttp://www.oreilly.com/\(as>
  <OReilly:Product>XML Pocket Reference</OReilly:Product>
  <OReilly:Price>8.95</OReilly:Price>
</OReilly:Books>
```

Let's look at this example line by line.

In the first line, the code between the `<?xml` and the `?>` is called an *XML declaration*. This declaration contains special information for the XML processor (the program reading the XML) indicating that this document conforms to Version 1.0 of the XML standard. In addition, the `standalone="no"` attribute informs the program that an outside DTD is needed to correctly interpret the document. (In this case, the DTD will reside in a separate file called *sample.dtd*.) On a side note, it is possible to simply embed the stylesheet and the DTD in the same file as the XML document. However, this is not recommended for general use, as it hampers reuse of both DTDs and stylesheets.

The second line is as follows:

```
<!DOCTYPE OReilly:Books SYSTEM "sample.dtd">
```

This line points out the *root element* of the document, as well as the DTD that validates each of the document elements that appear inside the root element. The root element is the outermost element in the document that the DTD applies to; it typically denotes the document's starting and ending point. In this example, the `<OReilly:Books>` element serves as the root element of the document. The `SYSTEM` keyword denotes that the DTD of the document resides in a separate local file named *sample.dtd*.

Following that line is a comment. Comments always begin with `<!–` and end with `–>`. You can write whatever you want inside comments; they are ignored by the XML processor. Be aware that comments, however, cannot come before the XML declaration and cannot appear inside of an element tag. For example, this is illegal:

```
<OReilly:Books <!-- This is the tag for a book --!>>
```

Finally, `<OReilly:Product>`, `<OReilly:Price>`, and `<OReilly:Books>` are XML elements we invented. Like most elements in XML, they hold no special significance except for whatever document and style rules we define for them. Note that these elements look slightly different than those you may have seen previously because we are using *namespaces*. Each element tag can be divided into two parts. The portion before the colon (`:`) forms the tag's namespace; the portion after the colon identifies the name of the tag itself.

Let's discuss some XML terminology: the `<OReilly:Product>` and `<OReilly:Price>` elements would consider the `<OReilly:Books>` element their *parent*. In the same manner, elements can be grandparents and grandchildren of other elements. However, we typically abbreviate multiple levels by stating that an element is either an *ancestor* or a *descendant* of another element.

1.1.3.1 Namespaces

Namespaces are a recent addition to the XML specification. The use of namespaces is not mandatory in XML, but it's often wise. Namespaces were created to ensure uniqueness among XML elements.

For example, let's pretend that the `<OReilly:Books>` element was simply named `<Books>`. When you think about it, it's not out of the question that another publisher would create its own `<Books>` element in its own XML documents. If the two publishers combined their documents, resolving a single (correct) definition for the `<Books>` tag would be impossible. When two XML documents containing identical elements from different sources are merged, those elements are said to *collide*. Namespaces help to avoid element collisions by scoping each tag.

In Example 1-1 we scoped each tag with the `OReilly` namespace. Namespaces are declared using the `xmlns:`*`something`* attribute, where *`something`* defines the ID of the namespace. The attribute value is a unique identifier that differentiates it from all other namespaces; the use of a URI is recommended. In this case, we use the O'Reilly URI http://www.oreilly.com/ as the default namespace, which should guarantee uniqueness. A namespace declaration can appear as an attribute of any element, so long as the namespace's use remains inside that element's opening and closing tags. Here are some examples:

```
<OReilly:Books
    xmlns:OReilly=\(ashttp://www.oreilly.com/\(as>
<xsl:stylesheet xmlns:xsl=\(ashttp://www.w3.org/\(as>
```

You are allowed to define more than one namespace in the context of an element:

```
<OReilly:Books xmlns:OReilly=\(ashttp://www.oreilly.com/\(as
    xmlns:Songline=\(ashttp://www.songline.com/\(as>
...
</OReilly:Books>
```

If you do not specify a name after the `xmlns` prefix, the namespace is dubbed the *default namespace* and is applied to all elements inside the defining element that do not use a namespace prefix of their own. For example:

```
<Books xmlns=\(ashttp://www.oreilly.com/\(as
       xmlns:Songline=\(ashttp://www.songline.com/\(as>
    <Book>
        <Title>XML Pocket Reference</Title>
        <ISBN>1-56592-709-5</ISBN>
    </Book>
    <Songline:CD>18231</Songline:CD>
</Books>
```

Here, the default namespace (represented by the URI http://www.oreilly.com/) is applied to the elements `<Books>`, `<Book>`, `<Title>`, and `<ISBN>`. However, it is not applied to the `<Songline:CD>` element, which has its own namespace.

Finally, you can set the default namespace to an empty string to ensure that there is no default namespace in use within a specific element:

```
<header xmlns=\(as\(as
        xmlns:OReilly=\(ashttp://www.oreilly.com/\(as
        xmlns:Songline=\(ashttp://www.songline.com/\(as>
    <entry>Learn XML in a Week</entry>
    <price>10.00</price>
</header>
```

Here, the `<entry>` and `<price>` elements have no default namespace.

1.1.4 A Simple Document Type Definition (DTD)

Example 1-2 creates a simple DTD for our XML document.

Example 1-2. simple.dtd

```
<!-- DTD for sample document -->
<!ELEMENT OReilly:Books (OReilly:Product, OReilly:Price)>
<!ELEMENT OReilly:Product (#PCDATA)>
<!ELEMENT OReilly:Price (#PCDATA)>
```

The purpose of the this DTD is to declare each of the elements used in our XML document. All document-type data is placed inside a construct with the characters `<!`*something*`>`. Like the previous XML example, the first line is a comment because it begins with `<!–` and ends with `–>`.

The `<!ELEMENT>` construct declares each valid element for our XML document. With the second line, we've specified that the `OReilly:Books` element is valid:

```
<!ELEMENT OReilly:Books
    (OReilly:Product, OReilly:Price)>
```

The parentheses group required child elements for the element `<OReilly:Books>`. In this case, the element `<OReilly:Product>` and the element `<OReilly:Price>` *must* be included inside our `<OReilly:Books>` element tags, and they must appear in the order specified. The elements `<OReilly:Product>` and `<OReilly:Price>` are *children* of `<OReilly:Books>`.

Likewise, both the `<OReilly:Product>` element and the the `<OReilly:Price>` element are declared in our DTD:

```
<!ELEMENT OReilly:Product (#PCDATA)>
<!ELEMENT OReilly:Price (#PCDATA)>
```

Again, parentheses specify required elements. In this case, they both have a single requirement, which is represented by `#PCDATA`. This is shorthand for *parsed character data*, which means that any characters are allowed, so long as they do not include other element tags or contain the characters `<` or `&`, or the sequence `]]>`. These characters are forbidden because they could be interpreted as markup. (We'll see how to get around this shortly.)

The XML data shown in Example 1-2 adheres to the rules of this DTD: it contains an `<OReilly:Books>` element, which in turn contains an `<OReilly:Product>` element, followed by an `<OReilly:Price>` element inside it (in that order). Therefore, if this DTD is applied to it with a `<!DOCTYPE>` statement, the document is said to be *valid*.

So far, we've structured the data, but we haven't paid much attention to its formatting. Now let's move on and add some style to our XML document.

1.1.5 A Simple XSL Stylesheet

The Extensible Stylesheet Language consists of a series of markups that can be used to apply formatting rules to each of the elements inside an XML document. XSL works by applying various style rules to the contents of an XML document, based on the elements that it encounters.

(As we mentioned earlier, the XSL specification is changing as we speak, and will undoubtedly change after this book is printed. So while you can use the XSL information in this book to develop a conceptual overview of XSL, we recommend referring to http://www.w3.org for the latest XSL specification.)

Let's add a simple XSL stylesheet to the example:

```
<?xml version="1.0"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/TR/WD-xsl"
    xmlns:fo="http://www.w3.org/TR/WD-xsl/FO">
    <xsl:template match="/">
        <fo:block font-size="18pt">
          <xsl:apply-templates/>
        </fo:block>
    </xsl:template>
</xsl:stylesheet>
```

The first thing you might notice when you look at an XSL stylesheet is that it is formatted in the same way as a regular XML document. This is not a coincidence. In fact, by design XSL stylesheets are themselves XML documents, so they must adhere to the same rules as well-formed XML documents.

Breaking down the pieces, you should first note that all the XSL elements must be enclosed in the appropriate `<xsl:stylesheet>` tags. These tags tell the XSL processor that it is describing stylesheet information, not XML content itself. Between the `<xsl:stylesheet>` tags lie each of the rules that will be applied to our XML document. Each of these rules can be further broken down into two items: a *template pattern* and a *template action*.

Consider the line:

```
<xsl:template match="/">
```

This line forms the template pattern of the stylesheet rule. Here, the target pattern is the root element, as designated by `match="/"`. The `"/"` is shorthand to represent the XML document's root element (`<OReilly:Books>` in our case). Remember that if this stylesheet is applied to another XML document, the root element matched might be different.

The following lines:

```
<fo:block font-size="18pt">
  <xsl:apply-templates/>
</fo:block>
```

specify the template action that should be performed on the target. In this case, we see the empty element `<xsl:apply-templates/>` located inside a `<fo:block>` element. When the XSL processor formats the target element, every element that is inside the root element's opening and closing tags uses an 18-point font.

In our example, the `<OReilly:Product>` element and the `<OReilly:Price>` element are enclosed inside the `<OReilly:Books>` tags. Therefore, the font size will be applied to the contents of those tags.

Example 1-3 displays a more realistic example.

Example 1-3. simple.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl"
                xmlns:fo="http://www.w3.org/TR/WD-xsl/FO">
                xmlns:OReilly="http://www.oreilly.com/">
  <xsl:template match="/">
      <fo:display-sequence>
        <xsl:apply-templates/>
      </fo:display-sequence>
  </xsl:template>
  <xsl:template match="OReilly:Books">
      <fo:block font-size="18pt">
        <xsl:text>Books:</xsl:text>
        <xsl:apply-templates/>
      </fo:block>
  </xsl:template>
  <xsl:template match="OReilly:Product">
      <fo:block font-size="12pt">
        <xsl:apply-templates/>
      </fo:block>
  </xsl:template>
  <xsl:template match="OReilly:Price">
      <fo:block font-size="14pt">
        <xsl:text>Price: $</xsl:text>
        <xsl:apply-templates/>
        <xsl:text> + tax</xsl:text>
      </fo:block>
  </xsl:template>
</xsl:stylesheet>
```

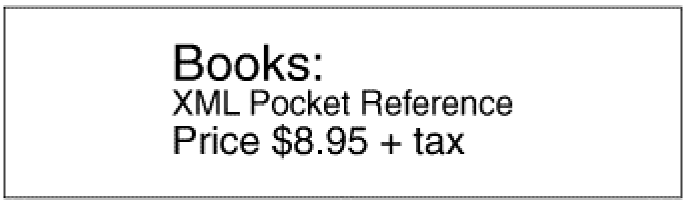
In this example, we're now targeting the `<OReilly:Books>` element, printing the word "Books:" before it in an 18-point font. In addition, the `<OReilly:Product>` element now applies a 12-point font to each of its children, and the `<OReilly:Price>` tag now uses a 14-point font to display its children, overriding the default 18-point font of its parent, `<OReilly:Books>`. (Of course, neither one has any children elements; they simply have text between their tags in the XML document.) The text "`Price: $`" will now precede each of `<OReilly:Price>`'s children, and the characters " `+ tax`" will now come after it, formatted accordingly.[1]

Here is the result after we pass *simple.xsl* through an XSL processor:

```
<?xml version="1.0"?>
<fo:display-sequence>
  <fo:block font-size="18pt">
Books:
    <fo:block font-size="12pt">
XML Pocket Reference
    </fo:block>
    <fo:block font-size="14pt">
Price $8.95 + tax
    </fo:block>
  </fo:block>
</fo:display-sequence>
```

And that's it: everything needed for a simple XML document! Running it through an XML processor, you should see something similar to Figure 1-1.

Figure 1-1. Sample XML output



[1] *You may have noticed that we are using the* `<fo:display-sequence>` *element instead of* `<fo:block>` *for the root element. This is primarily because the pattern that matches our root element really doesn't do anything anymore. However, you needn't be concerned with this here*

1.2 XML Reference

Now that you have had a quick taste of working with XML, here is an overview of the more common rules and constructs of the XML language.

1.2.1 Well-Formed XML

These are the rules for a well-formed XML document:

- The document must either use a DTD or contain an XML declaration with the `standalone` attribute set to "`no`". For example:

  ```
  <?xml version="1.0" standalone="no"?>
  ```

- All element attribute values must be in quotation marks.
- An element must have both an opening and closing tag, unless it is an empty element.
- If a tag is a standalone empty element, it must contain a closing slash (/) before the end of the tag.
- All opening and closing element tags must nest correctly.
- Isolated markup characters are not allowed in text: `<` or `&` must use entity references instead. In addition, the sequence `]]>` must be expressed as `]]&gt;` when used as regular text. (Entity references are discussed in further detail later.)
- Well-formed XML documents without a corresponding DTD must have all attributes of type CDATA by default.

1.2.2 XML Instructions

The following XML instructions are legal.

<?xml ... ?>	<?xml version= *number* [encoding= *encoding*] [standalone= `yes|no`] ?>
	
	Although they are not required to, XML documents typically begin with an XML declaration. An XML declaration must start with the characters `<?xml` and end with the characters `?>`.
	
	Attributes
	
	`version`
	
	The `version` attribute specifies the correct version of XML required to process the document, such as "`1.0`". This attribute cannot be omitted.
	
	`encoding`
	
	The `encoding` attribute specifies the character encoding used in the document (e.g., "`US-ASCII`" or "`iso- 8859-1`"). This attribute is optional.
	
	`standalone`
	
	The optional `standalone` attribute specifies whether a DTD is required to parse the document. The value must be either `yes` or `no`. If the value is `no`, a DTD must be declared with an XML `<!DOCTYPE>` instruction.

<!DOCTYPE>	<!DOCTYPE *root-element* SYSTEM|PUBLIC [*name*] *URI-of-DTD*>
	
	The <!DOCTYPE> instruction allows you to specify a DTD for an XML document. This instruction can currently take one of two forms:
	
	`<!DOCTYPE root-element SYSTEM "URI_of_DTD">`
	`<!DOCTYPE root-element PUBLIC "name" "URI_of_DTD">`
	
	Keywords
	
	SYSTEM
	
	The SYSTEM variant specifies the URI location of a DTD for private use in the document. The DTD is applied to all elements inside of *root-element*. For example:
	
	`<!DOCTYPE <Book> SYSTEM`
	` "http://mycompany.com/dtd/mydoctype.dtd">`
	
	PUBLIC
	
	The PUBLIC variant is used in situations where a DTD has been publicized for widespread use. In those cases, the DTD is assigned a unique name, which the XML processor may use by itself to attempt to retrieve the DTD. If that fails, the URI is used:
	
	`<!DOCTYPE <Book> PUBLIC`
	`  "-//O\(asReilly//DTD//EN"`
	`  "http://www.oreilly.com/dtd/xmlbk.dtd">`
	
	Public DTDs follow a specific naming convention. See the XML specification for details on naming public DTDs.

<?...?>	<?*target attribute1*= *value attribute2*= *value* ... ?> A processing instruction allows developers to place information specific to an outside application within the document. Processing instructions always begin with the characters <? and end with the characters ?>. For example: `<?works document="hello.doc" data="hello.wks"?>` You can create your own processing instructions if the XML application processing the document is aware of what the data means and acts accordingly.

CDATA	<![CDATA[...]]>
	
	You can define special marked sections of character data, or CDATA, which the XML processor will not attempt to interpret as markup. Anything that is included inside a CDATA marked section is treated as plain text. CDATA marked sections begin with the characters <![CDATA[and end with the characters]]>. For example:
	
	`<![CDATA[`
	`  I\(asm now discussing the <element> tag of documents`
	`  5 & 6: "Sales" and "Profit and Loss". Luckily,`
	`  the XML processor won\(ast apply rules of formatting`
	`  to these sentences!`
	`]]>`
	
	Note that you may not use entity references inside a CDATA marked section, as they will not be expanded.

-1	-1
	
	You can place comments anywhere in an XML document, except within element tags or before the initial XML processing instructions. Comments in an XML document always start with the characters <!– and end with the characters –>. In addition, they may not include double hyphens within the comment. The contents of the comment are ignored by the XML processor:
	
	`<!-- Sales Figures Start Here -->`
	`<Units>2000</Units>`
	`<Cost>49.95</Cost>`

1.2.3 Element and Attribute Rules

An element is either bound by its starting and ending tags, or is an empty element. Elements can contain text, other elements, or a combination of both. For example:

```
<para>Elements can contain text, other elements, or
a combination. For example, a chapter might contain
a title and multiple paragraphs, and a paragraph
might contain text and <emphasis>emphasis
elements</emphasis>:</para>
```

An element name must start with a letter or an underscore. It can then have any number of letters, numbers, hyphens, periods, or underscores in its name. Elements are case-sensitive: <Para>, <para>, and <pArA> are considered three different element types.

Element type names may not start with the string xml, in any variation of upper- or lowercase. Names beginning with xml are reserved for special uses by the W3C XML Working Group. Colons are permitted in element type names only for specifying namespaces; otherwise, colons are forbidden. For example:

<_Budget>	Legal
<Punch line>	Illegal: has a space
<205Para>	Illegal: starts with number
<repair\(atlog>	Illegal: contains \(at character
<xml>	Illegal: starts with xml

Element type names can also include accented Roman characters, letters from other alphabets (e.g., Cyrillic, Greek, Hebrew, Arabic, Thai, hiragana, katakana, or Devanagari), and ideograms from the Chinese, Japanese, and Korean languages. Valid element type names can therefore include <são>, <peut-être>, <più>, and <niño>, plus a number of others our publishing system isn't equipped to handle.

If you are using a DTD, the content of an element is constrained by its DTD declaration. Better XML applications inform you what elements and attributes can appear inside a specific element. Otherwise, you should check the element declaration in the DTD to determine the exact semantics.

Attributes describe additional information about an element. They always consist of a name and a value, as follows:

```
<price currency="Euro">
```

The attribute value is always quoted, using either single or double quotes. Attribute names are subject to the same restrictions as element type names.

1.2.4 XML Reserved Attributes

The following are reserved attributes in XML.

xml:lang	xml:lang= *iso_639_identifier* "
	
	 The `xml:lang` attribute can be used on any element. Its value indicates the language of that element. This is useful in a multilingual context. For example, you might have:
	
	 `<para xml:lang="en">Hello</para>`
	 `<para xml:lang="fr">Bonjour</para>`
	
	 This format allows you to display one or the other, depending on the user's language preference.
	
	 The syntax of the `xml:lang` value is defined by RFC 1766, available at http://ds0.internic.net/rfc/rfc1766.txt. A two-letter language code is optionally followed by a hyphen and a two-letter country code. The languages are defined by RFC 1766 and the countries are defined by ISO 3166. Traditionally, the language is given in lowercase and the country in uppercase (and for safety, this rule should be followed), but processors are expected to use the values in a case-insensitive manner.
	
	 In addition, RFC 1766 also provides extensions for nonstandardized languages or language variants. Valid `xml:lang` values include such notations as `en`, `en-US`, `en\(hyUK`, `en\(hycockney`, `i-navajo`, and `x-minbari`.

xml:space	xml:space= default|preserve "
	
	 The `xml:space` attribute indicates whether any whitespace inside the element is significant and should not be altered by the XML processor. The attribute can take one of two enumerated values:
	
	 `preserve`
	
	 The XML application honors all whitespace (newlines, spaces, and tabs) present within the element.
	
	 `default`
	
	 The XML processor is free to do whatever it wishes with the whitespace inside the element.
	
	 You should set `xml:space` to `preserve` only if you have an element you wish to behave similar to the HTML `<pre>` element, such as documenting source code.

xml:link	xml:link= *link_type* "
	
	The `xml:link` attribute signals an XLink processor that an element is a link element. It can take one of the following values:
	
	`simple`
	
	A one-way link, pointing to the area in the target document where the referenced element occurs.
	
	`document`
	
	A link that points to a member document of an extended link group.
	
	`extended`
	
	An extended link, which can point to more than one target through the use of multiple locators. Extended links can also support multidirectional and out-of-line links (a listing of links stored in a separate document).
	
	`group`
	
	A link that contains a group of document links.
	
	The `xml:link` attribute is always used with other attributes to form an XLink. See Section 1.5 for much more information on the `xml:link` attribute. This section also has more information on attribute remapping. (Note that this attribute may change to `xlink:form` in the future.)

xml:attribute	xml:attribute= *existing-attribute replacement-attribute* "
	
	The `xml:attribute` attribute allows you to remap attributes to prevent conflict with other potential uses of XLink attributes. For example:
	
	`<person title="Reverend" title-abbr="Rev."`
	`  given="Kirby" family="Hensley"`
	`  href="http://www.ulc.org/"`
	`  link-title="Universal Life Church"`
	`  xml:attributes="title link-title"/>`
	
	In this example, since the `title` attribute is already taken, the `xml:attributes` attribute remaps it to use `link-title` instead.
	
	See Section 1.5 for more information on attribute remapping. (Note that this attribute may change to `xlink:attribute` in the future.)

1.2.5 Entity References

Entity references are used as substitutions for specific characters in XML. A common use for entity references is to denote document symbols that might otherwise be mistaken for markup by an XML processor. XML predefines five entity references for you, which are substitutions for basic markup symbols. However, you can define as many entity references as you like in your own DTD. (See the next section.)

Entity references always begin with an ampersand (`&`) and end with a semicolon (`;`). They cannot appear inside `CDATA` sections, but can be used anywhere else. Predefined entities defined in XML are shown in Table 1-2.

Table 1-2, Predefined Entities in XML

Entity	Char	Notes
&	&	Do not use inside processing instructions
<	<	Use inside attribute values quoted with "
>	>	Use after]] in normal text and inside processing instructions
"	"	
'	'	Use inside attribute values quoted with '

In addition, you can provide character references for Unicode characters by using a *hexadecimal character reference*. This consists of the string `&#x` followed by the hexadecimal number representing the character, and finally a semicolon (`;`). For example, to represent the copyright character, you could use the following:

```
This document is &#xA9; 1999 by O\(asReilly and Assoc.
```

The entity reference is replaced with the “circled-C” (\(co) copyright character when the document is formatted.

1.3 Document Type Definitions

A DTD specifies how elements inside an XML document should relate to each other. It also provides grammar rules for the document and each of its elements. A document that adheres to the specifications outlined by its DTD is considered to be *valid*. (Don’t confuse this with a well-formed document, which adheres to the XML syntax rules outlined earlier.)

1.3.1 Element Declarations

You must declare each of the elements that appear inside your XML document within your DTD. You can do so with the `<!ELEMENT>` declaration, which uses the this format:

```
<!ELEMENT elementname rule>
```

This declares an XML element and an associated rule, which relates the element logically in the XML document. The element name should not include `<>` characters. An element name must start with a letter or an underscore. After that, it can have any number of letters, numbers, hyphens, periods, or underscores in its name. Element names may not start with the string `xml`, in any variation of upper- or lowercase. You can use a colon in element names only if you are using namespaces; otherwise, it is forbidden.

1.3.1.1 ANY and PCDATA

The simplest element declaration states that between the opening and closing tags of the element, anything can appear:

```
<!ELEMENT library ANY>
```

Using the ANY keyword allows you to include both other tags and general character data within the element. However, you may want to specify a situation where you want only general characters appearing. This type of data is better known as *parsed character data*, or PCDATA for short. You can specify that an element can contain only PCDATA with the following declaration:

```
<!ELEMENT title (#PCDATA)>
```

Remember, this declaration means that any character data that is *not* an element can appear between the element tags. Therefore, it's legal to write the following in your XML document:

```
<title></title>
<title>XML Pocket Reference</title>
<title>Java Network Programming</title>
```

However, the following is illegal with the previous PCDATA declaration:

```
<title>XML <emphasis>Pocket Reference</emphasis></title>
```

On the other hand, you may want to specify that another element *must* appear between the two tags specified. You can do this by placing the name of the element in the parentheses. The following two rules state that a <books> element must contain a <title> element and a <title> element must contain parsed character data (or null content) but not another element:

```
<!ELEMENT books (title)>
<!ELEMENT title (#PCDATA)>
```

1.3.1.2 Multiple elements

If you wish to dictate that multiple elements must appear in a specific order between the opening and closing tags of a specific element, you can use a comma (,) to separate the two instances:

```
<!ELEMENT books (title,authors)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT authors (#PCDATA)>
```

In the preceding declaration, the DTD states that within the opening <books> and closing </books> tags, there must first appear a <title> element consisting of parsed character data. It must be immediately followed by an <authors> element containing parsed character data. The <authors> element cannot precede the <title> element.

Here is a valid XML document for the DTD excerpt defined previously:

```
<books>
  <title>XML Pocket Reference</title>
  <authors>Robert Eckstein</authors>
</books>
```

The last example showed how to specify both elements in a declaration. You can just as easily specify that one or the other appear (but not both) by using the vertical bar (|):

```
<!ELEMENT books (title|authors)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT authors (#PCDATA)>
```

This DTD states that either a <title> element or an <author> element can appear inside the <books> element. Note that it must have one or the other. If you omit both elements, or include both elements, the XML document is not considered valid.

1.3.1.3 Grouping and recurrence

You can nest parentheses inside your declarations to give finer granularity to the syntax you're specifying. For example, the DTD below states that inside the `<books>` element, the XML document must contain either a `<description>` element or a `<title>` element immediately followed by an `<author>` element. All three elements must consist of parsed character data.

```
<!ELEMENT books ((title,author)|description)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT description (#PCDATA)>
```

Now for the fun part: you are allowed to dictate inside an element declaration whether a single element (or a grouping of elements contained inside parentheses) must appear zero or one times, one or more times, or zero or more times. The characters used for this appear immediately after the target element (or element grouping) that they refer to, and should be familiar to shell programmers. They are shown in Table 1-3.

Table 1-3, Occurrence Operators

Attribute	**Description**
`?`	Must appear once or not at all (0 or 1 times)
`+`	Must appear at least once (1 or more times)
`*`	May appear any number of times or not at all (0 or more times)

For example, if you want to provide finer granularity to the `<author>` element, you can redefine the following in the DTD:

```
<!ELEMENT author (authorname+)>
<!ELEMENT authorname (#PCDATA)>
```

This indicates that the `<author>` element must have at least one `<authorname>` element under it. It is allowed to have more than one as well. You can define more complex relationships with the use of parentheses:

```
<!ELEMENT reviews (rating, synopsis?, comments+)*>
<!ELEMENT rating ((tutorial|reference)*,overall)>
<!ELEMENT synopsis (#PCDATA)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT tutorial (#PCDATA)>
<!ELEMENT reference (#PCDATA)>
<!ELEMENT overall (#PCDATA)>
```

1.3.1.4 Empty elements

You must also declare each of the empty elements that can be used inside a valid XML document. This can be done with the `EMPTY` keyword:

```
<!ELEMENT elementname EMPTY>
```

For example, the following declaration defines an element in the XML document that can be used as `<statuscode/>` or `<statuscode></statuscode>`:

```
<!ELEMENT statuscode EMPTY>
```

1.3.2 Entities

Inside a DTD, you can declare an *entity*, which allows you to use an *entity reference* to substitute a series of characters for another character in an XML document, similar to macros.

1.3.2.1 General entities

A general entity is an entity that can substitute other characters inside the XML document. The declaration for a general entity uses the following format:

```
<!ENTITY name "replacement_characters">
```

We have already seen five general entity references, one each for the characters `<`, `>`, `&`, `'`, and `"`. Each of these can be used inside an XML document to prevent the XML processor from interpreting the characters as markup. (Incidentally, you do not need to declare these in your DTD; they are always provided for you.)

Earlier, we provided an entity reference for the copyright character. We could declare such an entity in the DTD with the following:

```
<!ENTITY copyright "&#xA9;">
```

Again, we have tied the `&copyright;` entity to Unicode value 169 (or hexadecimal 0xA9), which is the ″circled-C" (\(co) copyright character. Then you can use the following in your XML document:

```
<copyright>
&copyright; 1999 by MyCompany, Inc.
</copyright>
```

There are a couple of restrictions to declaring entities:

- You cannot make circular references in the declarations. For example, the following is invalid:

  ```
  <!ENTITY entitya "&entityb; is really neat!">
  <!ENTITY entityb "&entitya; is also really neat!">
  ```

- You cannot substitute nondocument text in a DTD with a general entity reference. The general entity reference is resolved only in an XML document, not a DTD document. (If you wish to have an entity reference resolved in the DTD, you must instead use a *parameter entity reference*.)

1.3.2.2 Parameter entities

Parameter entity references appear only in DTDs and are replaced by their entity definitions in the DTD. All parameter entity references begin with a percent sign, which denotes that they cannot be used in an XML document–only the DTD in which they are defined. Here is how to define a parameter entity:

```
<!ENTITY % name "replacement_characters">
```

Here are some examples using parameter entity references:

```
<!ENTITY % pcdata "(#PCDATA)">
<!ELEMENT authortitle %pcdata;>
```

As with general entity references, you cannot make circular references in declarations. In addition, parameter entity references cannot be used before they are declared.

1.3.2.3 External entities

XML allows you to declare an external entity with the following syntax:

```
<!ENTITY quotes SYSTEM
        "http://www.oreilly.com/stocks/quotes.xml">
```

This allows you to copy the XML content (located at the specified URI) into the current XML document using an external entity reference. For example:

```
<document>
  <heading>Current Stock Quotes</heading>
  &quotes;
</document>
```

This example copies the XML content located at the URI http://www.oreilly.com/stocks/quotes.xml into the document when it's run through the XML processor. As you might guess, this works quite well when dealing with dynamic data.

1.3.2.4 Unparsed entities

By the same token, you can use an *unparsed entity* to declare non-XML content in an XML document. For example, if you wanted to declare an outside image to be used inside an XML document, you could specify the following in the DTD:

```
<!ENTITY image1 SYSTEM
    "http://www.oreilly.com/ora.gif" NDATA GIF89a>
```

Note that we also specify the `NDATA` (notation data) keyword, which tells exactly what type of unparsed entity the XML processor is dealing with. You typically use an unparsed entity reference as the value of an element's attribute, one that is defined in the DTD with the type `ENTITY` or `ENTITIES`. Here is how you might use the unparsed entity declared previously:

```
<image src="&image1;"/>
```

1.3.2.5 Notations

Finally, notations are used in conjunction with unparsed entities. A notation declaration simply matches the value of an `NDATA` keyword with more specific information. The XML processor is free to use or ignore this information as it sees fit.

```
<!NOTATION GIF89a SYSTEM "-//CompuServe//NOTATION
        Graphics Interchange Format 89a//EN">
```

1.3.3 Attribute Declarations in the DTD

Attributes for various XML elements must be specified in the DTD. You can specify each of the attributes with the `<!ATTLIST>` declaration, which uses the following form:

<!ATTLIST *target_element attr_name attr_type default*>

The `<!ATTLIST>` declaration consists of the target element name, the name of the attribute, its datatype, and any default value you want to give it.

Here are some examples of legal <!ATTLIST> declarations:

```
<!ATTLIST box length CDATA "0">
<!ATTLIST box width CDATA "0">
<!ATTLIST frame visible (true|false) "true">
<!ATTLIST person marital
    (single | married | divorced | widowed) #IMPLIED>
```

In these examples, the first keyword after ATTLIST declares the name of the target element (i.e., box, frame, person). This is followed by the name of the attribute (i.e., length, width, visible, marital). This is generally followed by the datatype of the attribute and its default value.

1.3.3.1 Default values

Let's look at the default value first. You can specify any default value allowed by the specified datatype. If a default value is not appropriate, you can specify the keywords listed in Table 1-4 in its place.

Table 1-4, Default Modifiers in DTD Attributes

Attribute	Description
#REQUIRED	The attribute value must be specified with the element.
#IMPLIED	The attribute value can remain unspecified.
#FIXED	The attribute value is fixed and cannot be changed by the user.

A few notes: with the #IMPLIED keyword, if the value is not specified and none is given in the XML document, the XML parser must notify the application that no value has been specified. The application can take whatever action it deems appropriate at that point. With the #FIXED keyword, you must specify the default value immediately afterwards, as shown:

```
<!ATTLIST date year CDATA #FIXED "1999">
```

1.3.3.2 Datatypes

Datatypes in DTD Attributes lists legal datatypes to use in a DTD.

Table Datatypes in DTD Attributes

Attribute	Description
CDATA	Character data
enumerated	A series of values from which only one can be chosen
ENTITY	An entity declared in the DTD
ENTITIES	Multiple whitespace-separated entities declared in the DTD
ID	A unique element identifier
IDREF	The value of a unique ID type attribute
IDREFS	Multiple whitespace-separated IDREFs of elements
NMTOKEN	An XML name token
NMTOKENS	Multiple whitespace-separated XML name tokens
NOTATION	A notation declared in the DTD

The CDATA keyword simply declares that any character data can appear. Here are some examples of attribute declarations that use CDATA:

```
<!ATTLIST person name CDATA #REQUIRED>
<!ATTLIST person email CDATA #REQUIRED>
<!ATTLIST person company CDATA #FIXED "O\(asReilly">
```

Here is an enumerated datatype. In this case, there is no keyword specified. Instead, the enumeration itself is simply listed:

```
<!ATTLIST person marital
    (single | married | divorced | widowed) #IMPLIED>
<!ATTLIST person sex (male | female) #REQUIRED>
```

The ID, IDREF, and IDREFS datatypes allow you to define attributes as IDs and ID references. An ID is simply an attribute whose value distinguishes this element from all others in the current XML document. IDs are useful for linking to various sections of a document that contain an element with a uniquely tagged ID. IDREFs are attributes that reference other IDs. For example, consider the following XML document:

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE sector SYSTEM sector.dtd>
<sector>
  <employee empid="e1013">Jack Russell</employee>
  <employee empid="e1014">Samuel Tessen</employee>
  <employee empid="e1015" boss="e1013">Terri White
        </employee>
  <employee empid="e1016" boss="e1014">Steve McAlister
       </employee>
</sector>
```

And its DTD:

```
<!ELEMENT sector (employee*)>
<!ELEMENT employee (#PCDATA)>
<!ATTLIST employee empid ID #REQUIRED>
<!ATTLIST employee boss IDREF #IMPLIED>
```

Here, all employees have their own identification numbers (e1013, e1014, etc.), which we define in the DTD with the ID keyword using the empid attribute. This attribute then forms an ID for each <employee> element; no two <employee> elements can have the same ID.

Attributes that only reference other elements use the IDREF datatype. In this case, the boss attribute is an IDREF because it can use only the values of other IDs attributes as its values. IDs will come into play when we discuss XLink and XPointer.

The NMTOKEN and NMTOKENS attributes declare XML name tokens. An *XML name token* is simply a legal XML name that consists of letters, digits, underscores, hyphens, and periods. It can contain a colon if it is part of a namespace. However, an XML name token cannot contain spaces. These datatypes are useful in the event that you are enumerating tokens of languages or other keyword sets that match these restrictions in the DTD.

An ENTITY allows you to exploit an entity declared in the DTD. This includes unparsed entities. For example, to link to an image:

```
<!ELEMENT image EMPTY>
<!ATTLIST image src ENTITY #REQUIRED>
<!ENTITY chapterimage SYSTEM "chapimage.jpg" NDATA "jpg">
```

Which you can use as follows:

```
<image src="chapterimage">
```

The NOTATION keyword simply expects a notation that appears in the DTD with a <!NOTATION> declaration. Here, the player attribute of the <media> element can be either mpeg or jpeg:

```
<!NOTATION mpeg SYSTEM "mpegplay.exe">
<!NOTATION jpeg SYSTEM "netscape.exe">
<!ATTLIST media player
        NOTATION (mpeg | jpeg) #REQUIRED>
```

Note that you must enumerate each of the notations allowed in the attribute. For example, to dictate the possible values of the player attribute of the <media> element, use the following:

```
<!NOTATION mpeg SYSTEM "mpegplay.exe">
<!NOTATION jpeg SYSTEM "netscape.exe">
<!NOTATION mov SYSTEM "mplayer.exe">
<!NOTATION avi SYSTEM "mplayer.exe">
<!ATTLIST media player
        NOTATIONS (mpeg | jpeg | mov) #REQUIRED>
```

Note that by the rules of this DTD, the <media> element is not allowed to play AVI files.

Note that you can place all the declarations inside a single ATTLIST declaration, as long as you follow the rules of each datatype:

```
<!ATTLIST person
          name CDATA #REQUIRED
          number IDREF #REQUIRED
          company CDATA #FIXED "O\(asReilly">
```

1.3.4 Included and Ignored Marked Sections

Within a DTD, you can bundle together a group of declarations that should be ignored using the IGNORE directive:

```
<![ IGNORE [
   DTD content to be ignored
]]>
```

Conversely, if you wish to ensure that declarations are included in your DTD, you can use the INCLUDE directive, which has a similar syntax:

```
<![ INCLUDE [
   DTD content to be included
]]>
```

Why you would want to use either of these declarations is not obvious until you consider replacing the INCLUDE or IGNORE directives with a parameter entity reference you can change easily on the spot. For example, consider the following DTD:

```
<!ENTITY % ifstrict "INCLUDE">
<![ %ifstrict; [
  <!ELEMENT reviews (rating, synopsis?, comments+)*>
  <!ELEMENT rating ((tutorial|reference)*,overall)>
  <!ELEMENT synopsis (#PCDATA)>
  <!ELEMENT comments (#PCDATA)>
  <!ELEMENT tutorial (#PCDATA)>
  <!ELEMENT reference (#PCDATA)>
  <!ELEMENT overall (#PCDATA)>
]]>
```

You can either include or remove the enclosed declarations in this DTD by simply setting the parameter entity %ifstrict; to either the characters "IGNORE" or "INCLUDE", instead of commenting out each one when it is not needed.

1.3.5 Internal Subsets

As mentioned earlier, you can place parts of your DTD declarations inside the DOCTYPE declaration of the XML document, as shown:

```
<!DOCTYPE boilerplate SYSTEM "generic-inc.dtd" [
  <!ENTITY corpname "Acme, Inc.">
]>
```

The region between brackets is called the DTD's *internal subset*. When a parser reads the DTD, the internal subset is read first, followed by the *external subset*, which is the file referenced by the `DOCTYPE` declaration.

There are restrictions on the complexity of the internal subset, as well as processing expectations that affect how you should structure it:

- First, conditional marked sections (such as `INCLUDE` or `IGNORE`) are not permitted in an internal subset.
- Second, any parameter entity reference in the internal subset must expand to zero or more declarations. For example, specifying the following parameter entity reference is legal:

  ```
  %paradecl;
  ```

  as long as `%paradecl;` expands to the following:

  ```
  <!ELEMENT para CDATA>
  ```

  However, if you simply wrote the following in the internal subset, it would be considered illegal, as it does not expand to a whole declaration:

  ```
  <!ELEMENT para (%paracont;)>
  ```

A nonvalidating parser (one that doesn't check the external subset) is still expected to process any attribute defaults and entity declarations in the internal subset. However, a parameter entity can change the meaning of those declarations in an unresolvable way. Therefore, a parser must stop processing the internal subset when it comes to the first external parameter entity reference that it does not process. If it's an internal reference, it can expand it, and if it chooses to fetch the entity, it can continue processing. If it does not process the entity's replacement, it must not process the attribute list or entity declarations in the internal subset.

Why use this? Since some entity declarations are often relevant only to a single document (for example, declarations of chapter entities or other content files), the internal subset is a good place to put them. Similarly, if a particular document needs to override or alter the DTD values it uses, you can place a new definition in the internal subset. Finally, in the event that an XML processor is nonvalidating (as we mentioned previously), the internal subset is the best place to put certain DTD-related information, such as the identification of `ID` and `IDREF` attributes, attribute defaults, and entity declarations.

1.4 The Extensible Stylesheet Language

The Extensible Stylesheet Language (XSL) is one of the most intricate parts of the XML specification. It's also a bit of a moving target right now: as we write this, the XSL specification is moving in a completely new direction, possibly becoming its own language in the future. Much of the information in the following pages will be out-of-date very soon, as it is based on the current XSL specification in early 1999. For the very latest information on XSL, visit the home page for the W3C XSL working group at http://www.w3.org/Style/XSL/. This section will still provide you with a firm understanding of how XSL is meant to be used.

As we mentioned, XSL works by applying element-formatting rules that you define to each XML document it encounters. In reality, XSL simply transforms each XML document from one series of element types to another. For example, XSL can be used to apply HTML formatting to an XML document, which would transform it from:

```
<?xml version="1.0"?>
<OReilly:Book title="XML Comments">
 <OReilly:Chapter title="Working with XML">
   <OReilly:Image src="http://www.oreilly.com/1.gif"/>
   <OReilly:HeadA>Starting XML</OReilly:HeadA>
   <OReilly:Body>If you haven\(ast used XML, then ...
     </OReilly:Body>
 </OReilly:Chapter>
</OReilly:Book>
```

to the following HTML:

```
<HTML>
  <HEAD>
  <TITLE>XML Comments</TITLE>
  </HEAD>
  <BODY>
    <H1>Working with XML</H1>
    <img src="http://www.oreilly.com/1.gif"/>
    <H2>Starting XML</H2>
    <P>If you haven\(ast used XML, then ...</P>
  </BODY>
</HTML>
```

If you look carefully, you can see a predefined hierarchy that remains from the source content to the resulting content. To venture a guess, the `<OReilly:Book>` element probably maps to the `<HTML>`, `<HEAD>`, `<TITLE>`, and `<BODY>` elements in HTML. The `<OReilly:Chapter>` element maps to the HTML `<H1>` element, the `<OReilly:Image>` element maps to the `<img>` element, and so on.

This demonstrates an essential aspect of XML: each document contains a hierarchy of elements that can be organized in a tree-like fashion. (If the document uses a DTD, that hierarchy is well-defined.) In the previous XML example, the `<OReilly:Chapter>` element would be a leaf of the `<OReilly:Book>` element, while in the HTML document, the `<BODY>` and `<HEAD>` elements are leaves of the `<HTML>` element. XSL's primary purpose is to apply formatting rules to a *source tree*, rendering its results to a *result tree*, as we've just done.

1.4.1 Formatting Objects

One area of the XSL specification that is gaining steam is the idea of *formatting objects*. These objects serve as universal formatting tags that can be applied to virtually any arena, including both video and print. However, this (rather large) area of the specification is still in its infancy, so while we use formatting objects on occasion, we will not delve into their definitions. Where we do employ formatting objects, their meaning should be obvious.

Formatting objects do not use the `xsl` namespace, but instead employ the `fo` namespace, as shown here:

```
<fo:block font-size="10pt">
  Formatted text using 10 point font
</fo:block>
```

The `fo` namespace is mapped to the URI http://www.w3.org/ TR/WD-xsl/FO. If you wish to learn more about formatting objects, see the W3C XSL home page at http://www.w3.org/ Style/XSL/.

1.4.2 General Formatting

The general order for elements in an XSL stylesheet is as follows:

```
<?xml version="1.0"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/TR/WD-xsl">
    <xsl:import/>
    <xsl:include/>
    <xsl:id/>
    <xsl:strip-space/>
    <xsl:preserve-space/>
    <xsl:macro/>
    <xsl:attribute-set/>
    <xsl:constant/>
    <xsl:template match="pattern">
        template action
    </xsl:template>
    <xsl:template match="pattern">
        template action
    </xsl:template>
    ...
</xsl:stylesheet>
```

Essentially, this ordering boils down to a few simple rules. First, all XSL stylesheets must be well-formed XML documents, and each XSL element must use the `xsl:` namespace. Second, all XSL stylesheets must begin with the XSL root element tag `<xsl:stylesheet>` and close with a corresponding tag `</xsl:stylesheet>`. Within the opening tag, the XSL namespace must be defined:

```
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/TR/WD-xsl">
```

After the root element, you can import or include other XSL data using the `<xsl:include>` and `<xsl:import>` elements. Following that, you can use any of these optional elements: `<xsl:preserve-space>`, `<xsl:strip-space>`, `<xsl:attribute-set>`, `<xsl:id>`, `<xsl:macro>`, or `<xsl:constant>`. Each of these must be performed before defining any rules of your own using the `<xsl:template>` element.

1.4.3 Pattern Matching

Before going further, we should discuss the concept of pattern matching in XSL. XSL requires you to match elements in an XML document based on various characteristics. The most common of these XSL elements is the `<xsl:template>` element, which uses the `match` attribute to determine which elements to format.

With `<xsl:template>` and other elements, you can apply several pattern-matching strategies. These patterns match an element that has a relationship to the current *node*. A node defines the element that is at the current level of the hierarchy tree you are matching from. For example, consider this XML document:

```
<?xml version="1.0"?>
<book>
  <chapter>
    <image src="ch1.gif"/>
    <head>Learning XML</head>
    <body>
    XML is a <emphasis>wonderful</emphasis> tool
    </body>
  </chapter>
</book>
```

The stylesheet we wish to apply to this document is:

```
<?xml version="1.0"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template match="book/chapter/head">
      <xsl:text>The heading is: </xsl:text>
      <xsl:apply-templates/>
      <xsl:text>.</xsl:text>
  </xsl:template>
  <xsl:template match="book/chapter/body">
      <xsl:text>The body is: </xsl:text>
      <xsl:apply-templates/>
  </xsl:template>
</xsl:stylesheet>
```

Consider the line `<xsl:template match="book/chapter/head">`. Its current node is the `<book>` element that has the `<chapter>` as its child even though the `<head>` element is the element that is matched.

1.4.3.1 Matching on ancestry

As we hinted, you can ensure that a template formatting rule is applied only to, say, a `<body>` element that has a `<chapter>` parent with the following syntax:

```
<xsl:template match="chapter/body">
```

This matches the `<body>` element in following XML fragment:

```
<chapter>
  <body>Woe as I looked upon the raging sea.</body>
</chapter>
```

However, it will not match this <body> element:

```
<book>
  <body>In this day and age, we must wonder.</body>
</book>
```

Nor will it match this <body>:

```
<chapter>
  <section>
    <body>Woe as I looked upon the raging sea.</body>
  </section>
</chapter>
```

The latter example failed to match because the <chapter> element was not the direct parent of the <body> element. If you want to expand your matching capability to include *any* ancestor of a specific element, specify the following:

```
<xsl:template match="chapter//body">
```

This allows any number of elements (including zero) to appear between the <chapter> element and the <body> element in a qualifying match. As you might expect, you can get quite specific when matching on ancestry:

```
<xsl:template match="book//chapter/section//body">
```

You can select the first ancestor of the current node using the ancestor() keyword. For example:

```
<xsl:template match="ancestor(chapter)/head">
```

This selects the <head> elements of the first <chapter> ancestor of the current node. Note that the <chapter> element doesn't have to be the immediate parent of the current node. Now let's add an interesting twist:

```
<xsl:template match="*/body">
```

As you might expect, the asterisk performs the role of a wildcard. The previous example matches instances where the <body> element is the direct child of any element. (In fact, the only case it will not match is where the <body> element is the root element.)

By the same token, consider this:

```
<xsl:template match="section/*">
```

This template will match any element that has the <section> element as its immediate parent. Note that it will not match any element that has the <section> element as its grandparent.

You can select the current node with the period (.) operator. For example, to ensure that the <section> element is a child of the current node, use:

```
<xsl:template match="./section">
```

In the same manner, you can select the parent of the current node with a double period (..):

```
<xsl:template match="../section">
```

This matches a <section> element that is a sibling (parent's child) of the current node. Now let's suppose you want to choose one of two possible elements to match upon. These will work:

```
<xsl:template match="head|body">
<xsl:template match="chapter/head|chapter/body">
```

The first example matches either the <head> element or the <body> element from the current node. The second one does the same thing, ensuring that both elements are the child of a <chapter> element. (Note that a | has the lowest order of precedence.)

1.4.3.2 Tests

Let's look at some other pattern-matching rules:

```
<xsl:template match="book[index]/chapter">
```

This matches a `<chapter>` element that has a `<book>` element as its immediate parent. In addition, the `<book>` element is tested to ensure it has an `<index>` element as its immediate child. (This means that the `<index>` and `<chapter>` elements are siblings.) In other words, for this template to match, the following `XML` document elements must be in place:

```
<book>
  <chapter>
  Chapter 1: The Opening of Pandora's Box
  </chapter>
  <index>An index element</index>
</book>
```

The square brackets (`[]`) indicate that a test occurs to see if a particular element matched. Here, each `<chapter>` element that has a `<book>` element as its parent is tested to see if there is an `<index>` sibling.

If you wish to match an element that has a specific attribute defined, you can do so with the following:

```
<xsl:template match="book[@title]">
```

This matches any `<book>` element that has a `title` attribute defined, such as the following:

```
<book title="XML Pocket Reference">...</book>
```

However, it doesn't match the XML fragment shown here, as the book has no `title` attribute defined:

```
<book>
  <index>An index element</index>
</book>
```

To take this a step further, if you wish to match against an element attribute set to a known value, you can augment the rule by adding a test using the following pattern:

```
<xsl:template
    match="book[@title=\(asXML Pocket Reference\(as]">
```

This matches only if the attribute for the XML element is defined and has the matching value (in this case, "XML Pocket Reference").

You can negate a test using `not()`. For example, if you want to match every `<book>` element without a `title` attribute defined:

```
<xsl:template match="book[not(@title)]">
```

The `not()` keyword negates the result of any test inside its parentheses.

In addition, XSL allows you to match a pattern based on an element's position compared to other siblings, using the qualifiers shown in Table 1-5.

Table 1-5, Position Matching Qualifiers in XSL

Qualifier	Description
first-of-type()	Matches a single element that is the first sibling of its type
last-of-type()	Matches a single element that is the last sibling of its type
first-of-any()	Matches the first sibling element of any type
last-of-any()	Matches the last sibling element of any type

For example, if you want to place the text ”First Index:‟ before the first <index> element in the document and ”Next Index:‟ before the other <index> elements, you could use the following:

```
<xsl:template match="index[first-of-type()]">
    <xsl:text>First Index:</xsl:text>
    <xsl:apply-templates/>
</xsl:template>
<xsl:template match="index[not(first-of-type())]">
    <xsl:text>Next Index:</xsl:text>
    <xsl:apply-templates/>
</xsl:template>
```

You can use the or and and logical operators in tests as well:

```
<xsl:template
  match="chapter[first-of-type() or last-of-type()]">
<xsl:template
  match="chapter[first-of-any() and last-of-any()]">
```

1.4.3.3 Other matchings

To match any comment children of the current node, you can use the comment() keyword:

```
<xsl:template match="book/comment()">
```

This matches:

```
<book>
   <!--This comment will be matched-->
</book>
```

This is useful in the event that you want to style any of the comments in the XML document. We should warn you, however, that this is not always guaranteed to work, as many XML parsers throw comments away. By the same token, to match any processing instruction children of the current node, use the pi() keyword:

```
<xsl:template match="book/pi(works)">
```

This matches any processing instructions that are children of the <book> element and use the works application, such as:

```
<book>
   <?works version="4.0" document="rihol.xmp"?>
</book>
```

If you wish to match against the value of an element’s ID attribute, use the following code:

```
<xsl:template match="id(\(asdefinition\(as)">
```

This matches the following XML:

```
<book code="definition">...</book>
```

where the code attribute has been defined as a special ID attribute in the DTD.

1.4.3.4 Attribute value templates

Finally, you can use the {} characters as a template to match attribute values in result-tree elements. This is a little different than what we’ve seen, as the expression inside the curly braces is evaluated while processing, and both the expression and the curly braces are replaced immediately by the result. For example:

```
<xsl:constant name="codebase" value="/src/code"/>
<xsl:template match="applet">
<APPLET CODE="{code/@class}"
    CODEBASE="{constant(codebase)}/java"/>
</xsl:template>
```

This could be evaluated on the following XML:

```
<applet>
   <code class="myapplet"/>
</applet>
```

The result after processing is:

```
<APPLET CODE="myapplet" CODEBASE="/src/code/java"/>
```

Note that the `<xsl:constant>` element is still considered to be on the result tree of the XSL processor, even though an analogous element isn't present in the output XML.

You are not allowed to use curly braces inside an attribute value template.

1.4.3.5 Numbering elements

Let's assume we want to provide an autonumber for each element matched. For this, we can use the `<xsl:number>` element, which helps us assign a number based on the attributes we pass into it. For example, consider the following XML document:

```
<?xml version="1.0"?>
<book>
    <chapter>
        <title>A Mystery Unfolds</title>
        <paragraph>
        It was a dark and stormy...
        </paragraph>
    </chapter>
    <chapter>
        <title>A Sudden Visit</title>
        <paragraph>
        He awoke to a horrible sound...
        </paragraph>
    </chapter>
</book>
```

Here is how we would autonumber the `<chapter>` elements in the previous example:

```
<xsl:template match="book">
  <ol>
    <xsl:for-each select="chapter">
      <li>
      <xsl:text>Chapter </xsl:text>
      <xsl:number level="single" count="chapter"
          format="1"/>
      <xsl:text>:</xsl:text>
      <xsl:apply-templates select="title"/>
      </li>
    </xsl:for-each>
  </ol>
</xsl:template>
```

This creates the following:

```
<ol>
<li>Chapter 1: A Mystery Unfolds</li>
<li>Chapter 2: A Sudden Visit</li>
</ol>
```

In this case, the `<xsl:number>` element replaces itself with a numeral based on the number of `<chapter>` elements it has encountered so far.

You can use the `<xsl:number>` element to come up with much more extensive numbering schemes. The `<xsl:number>` element has three primary attributes that can be set: `count`, `level`, and `format`. In addition, there are three other attributes you can set: `letter-value`, `digit-group-sep`, and `n-digits-per-group`.

`count`

The `count` attribute decides which elements should be counted. You can use the standard pattern matching described previously to determine which elements will be numbered, such as:

```
<xsl:number count="chapter[attribute(number)=\(as4\(as]">
```

This selects `<chapter>` elements with a number attribute of 4. The following example accepts any of three types of elements:

```
<xsl:number count="chapter|section|paragraph">
```

`level`

Next, the `level` attribute specifies what levels of the source tree should be considered for counting. The `level` attribute always climbs the hierarchy of the source tree, searching for elements to pattern match against. Note that the `<xsl:number>` element never travels lower in the hierarchy than the position of the current element. The `level` attribute can take one of three string values:

`single`

Number each of the elements matched by the `count` attribute that are siblings of the current element. This is the default value.

`multi`

Number each of the elements matched by the `count` attribute that are children of any of the current element’s ancestors, so long as they do not travel deeper than the current element.

`any`

Number each of the elements matched by the `count` attribute that are anywhere in the document, so long as they do not travel deeper than the current element.

`from`

You can set a limit to the level of ancestry that is searched using the `from` attribute. For example, let’s assume you wish to number all the `<paragraph>` elements in the following document example:

```
<book>
  <paragraph>
  This book is copyright 1999 by O\(asReilly
  </paragraph>
  <chapter number="1">
    <title>A Mystery Unfolds</title>
    <paragraph>
    It was a dark and stormy night...
    </paragraph>
    <paragraph>
    Luckily, as Marcus downed a steaming java...
    </paragraph>
    <paragraph>
    "Are you sure that's what it was?"...
    </paragraph>
  </chapter>
  <chapter number="2">
    <title>A Sudden Visit</title>
    <paragraph>
    Marcus found himself sleeping...
    </paragraph>
    <paragraph>
    "How could you come back to me?"...
    </paragraph>
    <paragraph>
    Marcus had no idea how to answer this...
    </paragraph>
  </chapter>
</book>
```

However, anything not inside a <chapter> element should be ignored. The following XSL does the trick:

```
<xsl:template match="chapter">
  <xsl:number level="any" from="chapter"
      count="paragraph">
  <xsl:text>. </xsl:text>
</xsl:template>
```

This catches each of the <paragraph> elements inside the chapters, without picking up the copyright statement.

format

The format attribute tells XSL how you would like the numbers formatted. The default is a standard cardinal number. However, the XML specification lists several choices with <xsl:number>, including several Unicode values with numerical properties as shown in Table 1-6.

Table 1-6, <xsl:number> Format Values

Attribute	Description
1	Use standard numbers (1, 2, 3, 4…10, 11…)
A	Use standard capital letters (A, B, C…AA, BB…)
a	Use standard lowercase leters (a, b, c…aa, bb…)
i	Use lowercase Roman numerals (i, ii, iii, iv…)
I	Use capital Roman numerals (I, II, III, IV…)
&#x30A2;	Use katakana numbering
&#x30A4;	Use katakana numbering in *iroha* order
&#x0E51;	Use Thai digits for numbering
&#x05D0;	Use traditional Hebrew (letter-value=”other“)
&#x10D0;	Use Gregorian (letter-value=”other“)
&#x03B1;	Use classical Greek (letter-value=”other“)
&#x0430;	Use Old Slavic (letter-value=”other“)

All other characters appear as standard text. Consider, for example, the following format:

```
<xsl:number count="chapter|paragraph"
    format="1.1">
```

Assuming that <paragraph> elements are always contained inside of <chapter> elements, the first number corresponds to the chapter number, while the second number corresponds to the paragraph number. This yields a numbering system such as:

```
1.1
1.2
1.3
2.1
2.2
2.3
```

letter-value

When using other languages, you can use the letter-value attribute of the <number> element to differentiate numbering schemes that order letters from those that don’t. The value alphabetic specifies that the ordering should take place using an increment of the character codes of the language (e.g., a, b, c, d, e, f,…), while other specifies that an alternate rule should be used (e.g., i, ii, iii, iv, v, vi,…). In some alphabets, there can be two or more numbering systems that start with the same letter; the letter-value attribute can clarify those cases.

`digit-group-sep` and `n-digits-per-group`

Finally, the `digit-group-sep` attribute specifies the separator characters used between a succession of digits. For example, to place a comma between each of three digits in a number, use the following:

```
<xsl:number ...  digit-group-sep=","
    n-digits-per-group="3">
```

The `n-digits-per-group` attribute specifies the maximum amount of digits that should come between the `digit-group-sep` separator. For example, the United States uses a maximum of three digits between each comma when representing large numbers.

There are many other interesting possibilities you can use with the `<xsl:number>` element. See the W3C specification for more details.

1.4.3.6 Template matching rules

In the event that more than one template matches a given element in the XML document, the following rules apply:

- Template rules that have greater *importance* are chosen over those with lesser importance. This applies in a case where one stylesheet has been imported into another, in which case its contents are considered less important than the contents of the stylesheet containing the import statement.
- The template rule with the highest priority (if there is more than one remaining) is then chosen, as specified by the `priority` attribute of the `<xsl:template>` element.

If there is no template match, the XML processor generally performs a default format, which processes the children of elements and renders any text in the current context.

1.4.3.7 Linking in stylesheets

You can link stylesheets from your XML documents using a processor directive that points to the stylesheet:

```
<?xml-stylesheet
    href="http://www.oreilly.com/stylesheet1.xsl"
    type="text/xsl"?>
```

The directive points to the address of the XSL document, as well as the type of stylesheet used. This directive must come after the initial XML processor directive but before any DTD directives.

Although we do not cover it here, CSS is also a legitimate value for the type pseudo-attribute, indicating a cascading stylesheet:

```
<?xml-stylesheet
    href="http://www.oreilly.com/stylesheet2.css"
    type="text/css"?>
```

1.4.4 XSL Elements

The following list is an enumeration of XSL elements.

<xsl:apply-imports>	<xsl:apply-imports/> Styles the current node and each of its children, using the imported stylesheet rules, ignoring those in the stylesheet that performed the import. Note that the rules aren't applied to the current node's siblings or ancestors.

<xsl:apply-templates>	<xsl:apply-templates [select= *pattern* ”]/>
	
	Specifies that the immediate children of the source element should be processed further. For example:
	
	`<xsl:template match="section"/>`
	`  <B><xsl:apply-templates/><B>`
	`</xsl:template>`
	
	This example processes the children of the selected `<section>` element after applying a bold tag. You can optionally use the `select` attribute to determine which children should be processed.
	
	`<xsl:template match="section"/>`
	`  <HR>`
	`  <xsl:apply-templates`
	`    select="paragraph(indent)//sidebar"/>`
	`  <HR>`
	`  <xsl:apply-templates`
	`    select="paragraph(indent)/quote"/>`
	`  <HR>`
	`</xsl:template>`
	
	This example processes only specific children of the selected `<section>` element. In this case, the first target is a `<sidebar>` element that is a descendant of a `<paragraph>` element that has defined an `indent` attribute. The second target is a `<quote>` element that is the direct child of a `<paragraph>` element that has defined an `indent` attribute.

<xsl:arg>	<xsl:arg name= *string* default= *value* “/>
	
	Defines an argument and default value, which can be used when invoking a macro. For example:
	
	`<xsl:macro name="mymacro">`
	`  <xsl:macro-arg name="format" default="A. "/>`
	`  <xsl:number format="{arg(format)}"/>`
	`  <xsl:contents/>`
	`</xsl:macro>`
	`<xsl:template match="toc/entry">`
	`  <xsl:invoke macro="mymacro">`
	`    <xsl:arg name="format" value="1. "/>`
	`    <xsl:apply-templates/>`
	`  </xsl:invoke>`
	`</xsl:template>`
	
	This passes the value `”1. <“>` into the XSL macro `mymacro` when invoking it.

<xsl:attribute>	<xsl:attribute name= *name* > … </xsl:attribute>
	
	Adds an attribute with the given name to an element in the result tree. There can only be one attribute with a given name added to a specific element. The contents of the `<xsl:attribute>` element form the value of the attribute:
	
	`<xsl:element name="book">`
	`<xsl:attribute name="title">Moby Dick</xsl:attribute>`
	`<xsl:text>This is about a whale</xsl:text>`
	`</xsl:element>`
	
	This creates the following element in the result tree:
	
	`<book title="Moby Dick">This is about a whale</book>`

<xsl:attribute-set>	<xsl:attribute-set name= *value* ... />
	
	Allows the naming of a collection of formatting attributes, which can be applied using a formatting object. For example, the following will assign a bold 24-point font to the name `”heading-style“`, which can be used with the `xsl:use` attribute:
	
	`<xsl:attribute-set name="heading-style"`
	`                   font-size="24pt"`
	`                   font-weight="bold"/>`
	`<xsl:template match="heading">`
	`  <fo:block xsl:use="heading-style">`
	`    <xsl:apply-templates/>`
	`  </fo:block>`
	`</xsl:template>`

<xsl:choose>	<xsl:choose> ... </xsl:choose>
	
	The `<xsl:choose>` element, in conjunction with the elements `<xsl:when>` and `<xsl:otherwise>`, offers the ability to perform multiple condition tests. For example:
	
	`<xsl:template match="chapter/title">`
	`  <xsl:choose>`
	`    <xsl:when test=".[first-of-type()]">`
	`        Start Here:`
	`    </xsl:when>`
	`    <xsl:otherwise>`
	`        Then Read:`
	`    </xsl:otherwise>`
	`  </xsl:choose>`
	`  <xsl:apply-templates/>`
	`</xsl:template>`
	
	This example matches against each of the qualifying `<title>` elements, but it must test each `<title>` element to determine how to format it. Here, the formatting used depends on whether the element is the first of its type or not. If it is, it applies the letters ”Start Here:“ before the first `<title>` element. Otherwise, the letters ”Then Read:“ are placed before the others.

<xsl:comment>	<xsl:comment>...</xsl:comment>
	
	Inserts a comment into the XML document. For example, the following:
	
	`<xsl:comment>English material below</xsl:comment>`
	
	is translated into a comment in the XML document when it is processed:
	
	`<!-- English material below -->`

<xsl:constant>	<xsl:constant name= *name* value= *value* />
	
	Allows the definition of a named constant that can be substituted in XSL documents. For example:
	
	`<xsl:constant name="size" value="24pt"/>`
	`<xsl:template match="copyright">`
	`  <fo:block font-size="{constant(size)}">`
	`    <xsl:apply-templates/>`
	`  </fo:block>`
	`</xsl:template>`

<xsl:contents>	<xsl:contents/> Used inside `<xsl:macro>`. When the macro is invoked, the contents of the `<xsl:invoke>` invocation element are inserted at this point. (See `<xsl:macro>` for an example.)
<xsl:copy>	<xsl:copy>...</xsl:copy> Copies all nodes matched inside the opening and closing tags.

<xsl:counter>

<xsl:counter name= *string* />

Provides a named counter that appears in the result tree. The counter is initialized using the `<xsl:counter-reset/>` element and incremented using the `<xsl:counter-increment/>` element. Consider the following XML document:

```
<book>
  <chapter>Introduction</chapter>
  <chapter>Learning to Fly</chapter>
  <chapter>Simple Aerobatics</chapter>
</book>
```

The following XSL numbers and lists each of the chapters in the result tree:

```
<xsl:template match="book">
  <xsl:counter-reset name="chaps"/>
  <xsl:apply-templates>
</xsl:template>
<xsl:template match="book/chapter">
  <xsl:text>Chapter </xsl:text>
  <xsl:counter name="chaps"/>
  <xsl:counter-increment name="chaps"/>
  <xsl:text>:</xsl:text>
  <xsl:apply-templates>
</xsl:template>
```

<xsl:counters>

<xsl:counters name= *string* format= *format* />

Provides a named counter that appears in the result tree. The counter is formatted according to the string given using the `format` attribute. It can be initialized using the element `<xsl:counter-reset>` and incremented using the element `<xsl:counter-increment/>`. This differs from the `<xsl:counter>` element in that it maintains a counter for each level of ancestry it encounters. For example:

```
<xsl:template match="header">
  <fo:block>
    <xsl:counter-increment name="head"/>
    <xsl:counters name="head" format="1.1. "/>
    <xsl:apply-templates/>
  </fo:block>
  <xsl:counter-reset name="head"/>
</xsl:template>
```

Here, if we have a document as follows:

```
<header>The Long Road Home</header>
<header>Joyful Reunions</header>
```

the counter would return the following result: `(1, 2)`. However, if we had nested levels of the `<header>` element, such as the following:

```
<header>The Long Road Home
  <header>Starting the Long Road</header>
  <header>Halfway There</header>
</header>
<header>Joyful Reunions
  <header>Starting the Long Road</header>
  <header>Halfway There</header>
</header>
```

the counter returns the result `(1, 1.1, 1.2, 2, 2.1, 2.2)` for the `<header>` elements that it encounters.

<xsl:counter-increment>

<xsl:counter-increment name= *string* />

Increments the named counter by a value of one.

<xsl:counter-reset>

<xsl:counter-reset name= *string* [value= *value*]/>

Resets the counter identified by the `name` attribute to the value specified, or zero (0) if none is given. If the counter is not part of the set of named counters for this element, it is added.

<xsl:counter-scope>

<xsl:counter-scope/>

This element marks the scope of a set of counters but otherwise does nothing.

<xsl:element>	<xsl:element name= *name*|*URI#name* > ... </xsl:element>
	
	Inserts the element pointed to by the attribute *name* into the result document. For example:
	
	`<xsl:element name="book">`
	`  <xsl:element name="chapter">`
	`    <xsl:text>The Opening of Pandora's Box</xsl:text>`
	`  </xsl:element>`
	`</xsl:element>`
	
	This creates the following in the result node:
	
	`<book>`
	`  <chapter>The Opening of Pandora's Box</chapter>`
	`</book>`
	
	Elements without explicit namespaces use the default namespace of their current context. Also, you can create a namespace for the element yourself:
	
	`<xsl:element name="http://www.oreilly.com/#book">`
	
	This employs the namespace associated with the URI http://www.oreilly.com with the element. If no namespaces are associated with the URI, it becomes the default namespace.

<xsl:for-each>	<xsl:for-each select= *pattern* />
	
	The `<xsl:for-each>` directive allows you to select any number of identical siblings in an XML document. For example, consider the following XML document:
	
	`<book>`
	`  <chapter>`
	`    <title>A Mystery Unfolds</title>`
	`    <paragraph>`
	`    It was a dark and stormy night...`
	`    </paragraph>`
	`  </chapter>`
	`  <chapter>`
	`    <title>A Sudden Visit</title>`
	`    <paragraph>`
	`    Marcus found himself sleeping...`
	`    </paragraph>`
	`  </chapter>`
	`</book>`
	
	Note there are two `<chapter>` siblings in the document. Let's assume we want to provide an HTML numbered list for each `<title>` element that is the direct child of a `<chapter>` element, which in turn has a `<book>` element as a parent. The following XSL performs the task:
	
	`<xsl:template match="book>`
	`  <ol>`
	`  <xsl:for-each select="chapter">`
	`    <li><xsl:process select="title"></li>`
	`  </xsl:for-each>`
	`  </ol>`
	`</xsl:template>`
	
	After formatting, here is what the result looks like:
	
	`<ol>`
	`<li>A Mystery Unfolds</li>`
	`<li>A Sudden Visit</li>`
	`</ol>`
	
	The XSL processor processes a `<title>` element in each `<chapter>` element that is the child of a `<book>` element. The result is a list of each chapter that could be used for a table of contents.

<xsl:id>	<xsl:id attribute= *value* [element= *element*]/>
	
	Identifies a specific attribute that appears in an XML element as an XML ID attribute. This is useful in the event that an XML document without a corresponding DTD needs to be formatted. If `element` is used, only attributes localized to that element are marked as ID attributes. If `element` is omitted, the XML processor assumes that all element attributes matching the `value` specified, no matter what the element, should be treated as ID attributes:
	
	`<xsl:id attribute="id"/>`
	`<xsl:id attribute="marker" element="Body"/>`
	
	You are allowed to use multiple `<xsl:id>` elements. However, remember that the same ID cannot be assigned to more than one XML element at a time. If this instruction identifies several XML attributes as ID attributes such that any of those attributes have identical values, the XML processor could generate an error.
	
	This element may change in the future.

<xsl:if>	<xsl:if test= *pattern* >...</xsl:if>
	
	You can use the `<xsl:if>` conditional to select a specific element while inside a template. The `<xsl:if>` element uses the `test` attribute to determine which elements should be selected. The `test` attribute takes a standard pattern matching string. For example:
	
	`<xsl:template match="chapter/title">`
	`  <xsl:apply-templates/>`
	`  <xsl:if test=".not([last-of-type()])">, </xsl:if>`
	`</xsl:template>`
	
	This template matches each qualifying `<title>` element but inserts commas after those that are not the last `<title>` element. The result is a standard comma-separated list.

<xsl:import>	<xsl:import href= *address* /> Specifies the URI of an XSL stylesheet whose rules should be imported into this stylesheet. The import statement must occur before any other elements in the stylesheet. If a conflict arises between matching rules, imported stylesheets are of lesser importance to those rules in the XSL stylesheet performing the import. In addition, if more than one stylesheet is imported into this document, the more recently imported stylesheet will be of greater importance than stylesheets imported before it. `<xsl:import href="webpage.xsl"/>` This example imports the stylesheet included in the file *webpage.xsl*.
<xsl:include>	<xsl:include href= *address* /> Specifies the name of an XSL stylesheet to be included in the document. The include process will replace the `<xsl:include>` statement with the contents of the file (moving any other `<xsl:import>` elements before it). Because the included document has been inserted in the referring stylesheet, any included rules will be of equal importance to those in the referring stylesheet. (Compare to `<xsl:import>`.) `<xsl:include href="chapterFormats.xsl"/>`

<xsl:macro>	<xsl:macro>...</xsl:macro> Creates a reusable XSL fragment that can be inserted verbatim at given points by invoking its macro name. A macro is defined using the `<xsl:macro>` element and can be inserted into an XSL document using the `<xsl:invoke>` element, as shown: `<xsl:macro name="warning-header">` `<B>WARNING! </B>` `<xsl:contents/>` `</xsl:macro>` `<xsl:template match="warning">` `<xsl:invoke macro="warning-header">` `<xsl:apply-templates/>` `</xsl:invoke>` `</xsl:template>` This example places the HTML bold letters ”WARNING!“ in front of any child elements of the `<warning>` element in the target document. When the macro is invoked, the contents of the `<xsl:macro>` element replace the `<xsl:invoke>` element. In addition, the `<xsl:contents/>` element, seen in the macro itself, are replaced with the contents of the `<xsl:invoke>` element. XSL macros are allowed to take arguments using the `<xsl:macro-arg>` element.
<xsl:macro-arg>	<xsl:macro-arg name= *string* default= *value* /> Used to declare an argument that should be passed to an XSL macro, as defined with the `<xsl:macro>` command. The macro argument can take a default value, specified by the `default` attribute, in the event that the programmer does not specify one. `<xsl:macro name="function">` `<xsl:macro-arg name="x" default="0"/>` `...` `<xsl:contents/>` `</xsl:macro>`
<xsl:number>	<xsl:number level= *level* count= *section* format= *format* /> This element can autonumber elements within the XML document. See Section 1.4.3.5 earlier in this section.
<xsl:otherwise>	<xsl:otherwise>...</xsl:otherwise> Default conditional for testing in an `<xsl:choose>` element. See `<xsl:choose>`.
<xsl:pi>	<xsl:pi name= *name* />...</xsl:pi> Creates a processing instruction in the XML document. The `name` attribute is mandatory; it specifies the name of the application this instruction will be processed by. Any other attributes should appear between the opening and closing tags. For example: `<xsl:pi name="works">applevel="A" version="3.0"</xsl:pi>` This is translated in the output XML document as: `<?works applevel="A" version="3.0"?>`
<xsl:preserve-space>	<xsl:preserve-space element= *element_name* /> Declares an XML element in which all whitespace located between its opening and closing tags is preserved; hence, the XML processor will not remove it. `<xsl:preserve-space element="title"/>` This is similar to the `xml:space=”preserve“` attribute.

<xsl:strip-space>	<xsl:strip-space element= *element_name* /> Declares an XML element in which all whitespace located between its opening and closing tags is insignificant and should be removed by the XML processor. `<xsl:strip-space element="title"/>` Note that this is not necessarily the same as the `xml:space="default"` attribute, which allows the XSL processor more freedom to decide how to handle whitespace.

<xsl:value-of>	<xsl:value-of select= *pattern* />
	
	Extracts a specific value from a source tree. The only attribute to the `<xsl:value>` element is a single pattern-matching expression that resolves to the value of a string, an element, or an attribute.
	
	`<xsl:template match="index">`
	`   This index is <xsl:value-of select="@(type)">`
	`   <xsl:apply-templates/>`
	`</xsl:template>`
	
	The `select` attribute extracts the value of an element or attribute in the source tree and prints it verbatim in the result tree.

<xsl:template>	<xsl:template match= *pattern* > ... </xsl:template>
	
	The XSL template directive localizes various elements from which stylesheet rules can be applied. Each element is targeted with the `match` attribute. Formatting directives are located inside the opening and closing `<xsl:template>` tags.
	
	`<xsl:template match="para">`
	`  <fo:block font-size="12pt">`
	`    <xsl:apply-templates/>`
	`  </fo:block>`
	`</xsl:template>`
	
	See the earlier Section 1.4.3 for more information on matching elements.

<xsl:text>	<xsl:text>...</xsl:text>
	
	Inserts text verbatim into the document. For example:
	
	`<xsl:text>The price is $20.00.</xsl:text>`
	
	is inserted into the XML document as:
	
	`The price is $20.00.`

<xsl:use>	<xsl:use attribute-set= *name* /> Uses a specific attribute set. See `<xsl:attribute-set>`.
<xsl:when>	<xsl:when test= *pattern* >...</xsl:when> Conditional for testing in an `<xsl:choose>` element. See `<xsl:choose>`.

1.5 XLink and XPointer

The final piece of XML we cover is XLink and XPointer. These two creations fall under the Extensible Linking Language (XLL), a separate portion of the XML standard dedicated to working with XML links. Before we delve into this, however, we should warn you that the standard described here is subject to change at any time.

It's important to remember that an XML link is only an *assertion* of a relationship between pieces of documents; how the link is actually presented to a user depends on a number of factors, including the application processing the XML document.

1.5.1 Unique Identifiers

In order to create a link, we must first have a labeling scheme for XML elements. We do this by assigning an identifier to specific elements we want to reference using an ID attribute:

```
<paragraph id="attack">Suddenly the skies were filled
with aircraft.</paragraph>
```

You can think of IDs in XML documents as street addresses: they provide a unique identifier for an element within a document. However, just as there might be an identical address in a different city, an element in a different document might have the same ID. Consequently, you can tie together an ID with the document's URI, as shown below:

```
http://www.oreilly.com/documents/story.xml#attack
```

The combination of a document's URI and an element's ID should uniquely identify that element throughout the universe. Remember that an ID attribute does not need to be named "id," as we showed in the first example. You can name it anything you want, as long as you define it as an XML ID in the document's DTD. (However, using "id" is preferred in the event that the XML processor does not read the DTD.)

Should you give an ID to every element in your documents? No: the odds are that most elements will never be referenced. It's best to place IDs on items that a reader would want to refer to later, such as chapter and section divisions, as well as important items, such as term definitions.

1.5.2 ID References

The easiest way to refer to an ID attribute is with an ID reference, or IDREF. Consider this example:

```
<?xml version="1.0" standalone="yes"?>
<DOCTYPE document [
   <!ELEMENT document (employee*)>
   <!ELEMENT employee (#PCDATA)>
   <!ATTLIST person empnumber ID #REQUIRED>
   <!ATTLIST person boss IDREF #IMPLIED>
]>
<employee empnumber="emp123">Jay</employee>
<employee empnumber="emp124">Kay</employee>
<employee empnumber="emp125"
    boss="emp123">Frank</employee>
<employee empnumber="emp126"
    boss="emp124">Hank</employee>
```

As with ID attributes, an IDREF can be declared in the DTD. However, if you're in an environment where the processor might not read the DTD, you might want to call your ID references "idref."

The chief benefit of using an IDREF is that a validating parser can ensure that every one points to an actual element; unlike other forms of linking, an IDREF is guaranteed to refer to something within the current document.

As we mentioned before, the IDREF only asserts a relationship of some sort; the stylesheet and the browser will determine what is to be done with it. If the referring element has some content, it might become a link to the target. But if the referring element is empty, the stylesheet might instruct the browser to perform some other action.

As for the linking behavior, remember that in HTML a link can point to an entire document (which the browser will download and display, positioned at the top), or it can point to a specific location in a document (which the browser will display, usually positioned with that point at the top of the screen). However, linking changes drastically in XML. What does it mean to have a link to an entire element, which might be a paragraph (or smaller) or an entire group of chapters? The XML application will attempt some kind of guess, but the display is best controlled by the stylesheet. For now, it's best to simply make a link as meaningful as you can.

1.5.3 XPointers

ID references provide a convenient and efficient way to refer to elements within the same document. However, what happens if you want to refer to a point (or even a range) in a different document, especially if the point doesn't have a convenient ID to anchor to and you don't have permission to change the document? Luckily, XPointer can help.

XPointer uses the URI-augmented addressing scheme we introduced earlier. A typical XPointer looks like this:

```
http://www.oreilly.com/documents/xmlpocket/XLink#XPtr
```

There are two parts, separated by a hash mark (#). The first portion locates a resource (usually a document); the other forms the XPointer that locates something within that resource. The behavior and syntax of the XPointer portion is defined by the kind of document to which it is applied; in HTML, it finds an <a> element with a name attribute equal to the specified string. In XML, the syntax is defined by XPointer.

1.5.3.1 ID references with XPointer

The easiest way to use an XPointer is to place an ID on the element you wish to point to. A simple string after the hash mark with no other qualifiers is assumed to refer to the element with that ID. For example, to find the element with an ID equal to "chapter01" in a document *book.xml*, the partial URL would look like this:

```
book.xml#chapter01
```

This would locate an element in a file *book.xml* such as:

```
<chapter id="chapter01"> ... </chapter>
```

With XPointers, it's best to use references to IDs whenever possible. That way, if the document is edited and your target section is moved, XPointer will still find the correct location so long as you refer directly to an element's ID. On the other hand, if the element is deleted, the XPointer will break. This is better than being taken to the approximate place where the information was and having to search fruitlessly for it.

1.5.3.2 Absolute location terms

The XPointer ID reference we saw previously is actually shorthand for a slightly more verbose syntax. The same element as above could be identified as:

```
book.xml#id(chapter01)
```

The syntax of a keyword followed by parentheses is common to all the pieces of XPointer. Some location terms, such as id(), locate a specific element anywhere in a document without help from anything else. These terms are called *absolute location terms*. When used, they immediately follow the hash mark. You can use only one absolute location term per XPointer.

Besides id(), there are three other absolute location terms we should discuss: root(), html(), and origin().

root()

root() locates the root element of the document to which the base URI points. For instance, our examples have a root element of <OReilly:Books>. Therefore, if we name that document simpledoc.xml, simpledoc.xml#root() points to the <OReilly:Books> element.

html()

html() is intended as a transitional term for use with XML documents that are primarily HTML. The html() term finds the element <a> whose name attribute has the value given between parentheses. For example:

```
http://www.oreilly.com/docs/xmlpr#html(authors)
```

points to the following HTML:

```
<A NAME="authors">Robert Eckstein</A>
```

The html() term works the same way that the fragment locator does when pointing to an HTML document, with one difference: the html() term always finds the first match, unlike the HTML fragment locator, whose behavior varies depending on the browser in use.

origin()

The origin() keyword is similar to root(), but it points to the document element from which the user began the link. origin() is useful only when used in association with a relative location term, which we discuss next.

1.5.3.3 Relative location terms

Relative location terms locate an element relative to the position given by another. The preceding terms (often an absolute location term) provide a *location source* for the relative term to work from. Relative location terms are concatenated onto the end of other location terms with a period (.) character.

There are seven relative location terms: child(), ancestor(), descendant(), preceding(), following(), psibling(), and fsibling(). Each term can take up to four arguments. The arguments of a relative location term hold the same meaning from one keyword to the next. The first argument can be either a number or the word all. The second argument is the type of object being located, and the third and fourth arguments filter the results based on attributes and their values.

Here are some examples using relative location terms:

```
root().child(6, para)
root().descendant(3, emphasis)
root().child(1, employee, boss, #IMPLIED)
```

1.5.3.4 Arguments

Let's start with the arguments and cover each one by one.

The first term selects the number, or instance, of the match from a set of candidates. For example, child(all,para) selects all the <para> elements that are children of the location source. Alternatively, child(2,para) selects the second <para> element. The word all simply selects all the eligible elements. A positive integer selects an element by counting forward in the list of candidates, while a negative integer selects an element by counting backward in the list.

The second argument is the *node type*. For most purposes, this will be the name of an element type. For instance, as you've probably already guessed, `child(all,para)` will locate all of the `<para>` elements that are children of the current location source. In addition, you can use the keywords `#element` and `#text` as the second argument. For example, `child(all,#element)` finds all of the elements (of any type) that are children of the location source. If the second argument is omitted, it is the same as if `#element` had been specified. On the other hand, the `#text` keyword finds chunks of text that are directly contained within the location source; that is, they're not in any other element besides the current one. For example, given this XML:

```
<para>
This is a <literal>#text</literal> example.
</para>
```

the XPointer term `child(all,#text)` would locate two text chunks: the one containing "This is a " and the one containing " example." (including the spaces after "a" and before "example"). The other chunk of text could be found with the term `child(1,literal).(all,#text)`.

The third argument forms the name of an attribute, and the fourth is its value. These can be used to locate elements with specific attribute values. The location term `child(all, #element,type,"zarknab")` will find any and all child elements of the location source that have `type="zarknab"` given as an attribute. The attribute name can also be given as `*`, meaning that any attribute with the value is acceptable: `child(all,#element,*,"zarknab")` will find any child element with any attribute with a value of `"zarknab"`.

The fourth argument, instead of a value, can also be * or `#IMPLIED`. The first requires that some value was specified for the attribute, but its exact value doesn't matter. The value might have been given in the document directly, or it might have been provided as a default value in the DTD. `#IMPLIED` means the opposite, that no value at all was specified, nor was any default given. So `child(all,#element,type,*)` finds all the children who specified any value at all for the `type` attribute, while `child(all,#element,type, #IMPLIED)` finds all of the other children, the ones that did not specify any value for `type`.

You can chain together keywords of the same type by placing periods between the sets of arguments and omitting the keywords. For example:

```
child(1,para).(1,emph)
```

is equivalent to:

```
child(1,para).child(1,emph)
```

1.5.3.5 Terms

Here is an example of the relative location terms. For the following, consider this simple XML document:

```
<?xml version="1.0"?>
<!DOCTYPE simpledoc [
  <!ELEMENT simpledoc (title, para+)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT para (#PCDATA | emph)*>
  <!ATTLIST para
            type CDATA #IMPLIED>
  <!ELEMENT emph (#PCDATA)>
]>
<simpledoc>
<title>A Simple Document</title>
<para>This is a <emph>very</emph> simple document.
</para>
<para type="sarcastic">Yeah, like there's
<emph>so</emph> much going on here.</para>
</simpledoc>
```

child()

The child() keyword locates elements that are immediate children of the location source. For example, root().child(1,para) finds the first <para> in the previous example, while root().child(1,*, type,"sarcastic") finds the second <para> in the example. Both root().child(all,title) and #root().child(1, *) find the <title>.

Negative numbers for child() simply count backward from the end of the list of candidates. In this example, root().child(2,para) and root().child(-1, para) find the second <para> and the last <para>, respectively–which are the same element.

descendant()

The descendant() keyword works like the child() keyword, except that it is not limited to the immediate children. In the example above, both <emph> elements are descendants of <simpledoc>. For example:

```
root().descendant(2,emph)
root().descendant(-1,emph)
root().descendant(2,para).(1,emph)
root().child(2,para).(1,emph)
```

These XPointers all locate the same element, the final <emph> in the document.

Negative numbers are more complicated for descendants than for children. Since an element's descendants can contain other descendants, it matters whether you count the beginnings or the ends of the elements. The XPointer specification states that it is the ends that matter when using negative numbers, not the beginnings (which count with positive numbers). In the example, the first <para> starts before its child <emph>, but it ends after the <emph>.

Table 1-7 shows a set of positive-numbered descendants and the elements they locate.

Table 1-7, Negative XPointers with descendant()

Positive XPointer and Negative XPointer	**Element**
root().descendant(1,*)	<title>
root().descendant(-5,*)	
root().descendant(2,*)	The first <para>
root().descendant(-3,*)	
root().descendant(3,*)	The first <emph>
root().descendant(-4,*)	
root().descendant(4,*)	The second <para>
root().descendant(-1,*)	
root().descendant(5,*)	The second <emph>
root().descendant(-2,*)	

Note that this isn't a strict reversal. The <emph> elements always come after their containing <para> elements, whether you count forward or backward.

You may be wondering, since every child is a descendant, why have the child() keyword at all? The first reason is that child() is slightly more robust; a change in the document's structure is more likely to change the descendants of an element than its children, as there are (usually) more descendants than children. The second reason is that it is faster for a computer program to look through the children than through the descendants, so you may get better performance from your XPointers using child().

ancestor()

The ancestor() keyword works up the tree to locate an element that contains the location source. For example, root().descendant(2, emph).ancestor(1, simpledoc) finds the root element in the previous example. So do root().descendant(2, emph).ancestor(2,*) and root().descendant(2,emph).ancestor(-1,*).

Negative numbers are simple once again; positive numbers count upward from the location source towards the root; negative numbers count downward along the same path, with the root element (<simpledoc> in this case) being –1.

preceding()

The preceding() keyword finds elements that start before the location source. From the second <emph>, everything is a preceding element; from the <title>, only the <simpledoc> precedes it.

Positive and negative numbers work in a way that might seem backward at first for preceding elements. Positive numbers count backward in the document; that is, from the second <emph>, preceding(1,*) locates the <para> that contains it. Negative numbers count forward from the root element: from that same <emph>, preceding(-1,*) finds the <simpledoc>, and preceding(-2,*) finds the <title>.

following()

The following() keyword picks from among the elements that end after the current one; the second <para> follows the second <emph>, as does <simpledoc>.

Positive numbers count forward for the following(). Beware, however, that anything that ends after the current element is considered to follow it. So, from the second <emph>, following(1,*) and following(-2,*) both find the second <para> element, while following(2,*) and following(-1,*) both find the <simpledoc> element.

psibling()

The psibling() keyword is to preceding() as child() is to descendant(). It can locate only those elements that both precede it and have the same parent. In other words, the second <para> element has two previous siblings, the first <para> has one, and the <title> has none at all.

Counting works the same way as it does with preceding(). From the second <para>, psibling(2,*) and psibling(-1,*) both find the <title> element.

fsibling()

This keyword is like psibling(), only it locates elements after it instead of before it that share the same parent. From the <title>, both fsibling(1,*) and fsibling(-2,*) find the first <para>.

Now let's take a peek at how these might be used in an actual XML link:

```
<?xml version="1.0"?>
<linkdoc>
<link xml:link="extended" inline="false"
    title="origin() link">
 <start xml:link="locator" role="start" actuate="user"
   href="simp.xml#id(target-section).child(1,para)"/>
 <end xml:link="locator" role="end" actuate="none"
   href="#origin().fsibling(1,*)"/>
</link>
</linkdoc>
```

The xml:link attribute helps us form the appropriate link (<link>, <linkdoc>, <start>, and <end> are our own elements). When the XML link is activated, it links to the following element of any type, from wherever the link was activated. (In this case, that simply means the following paragraph.) In the previous example, id(target-section) found the element that has an ID of "target-section"; then child(1,para) finds the first <para> element that is its child.

Note that the origin() location term in the end link does not follow a filename; an error will occur if a filename is given that differs from the one in which the link traversal actually began. Also, note that neither root() nor origin() have anything between their parentheses. They are not permitted to; in reality, the parentheses are there for consistency with other location terms, and to avoid a potential clash with an ID called root or origin.

1.5.3.6 Strings

Once you've found an element, you may not want the whole thing. You may only want the third word in the paragraph, or the first sentence, or the last three letters. XPointer provides the string() location term to select specific text within an element. Like the relative location term keywords, string() takes up to four arguments. But don't let that fool you; the arguments have little to do with those of the other keywords.

The simplest form of string() takes two arguments. The first argument is the keyword all or a number, the second is a string to look for: string(2,<">fnord<">) finds the second occurrence of the string "fnord" within the location source. (Note that there is no restriction on matching only words; this example finds a part of "Medfnord" as well as the whole word "fnord.") Also, the match is case-sensitive.

To simply count characters, you can give an empty string for the second argument: string(23,<"><">) will find the point immediately before the 23rd character in the location source.

Having found the string, you're not necessarily done. You may want to move relative to the string you found; for example, you may want to find the string "Medfnord", and then move relative to its location. To do that, use the following:

```
string(all,Medfnord,3,5)
```

The first two arguments locate all occurrences of the string "Medfnord," and the next two select the string beginning three characters from the beginning of the targeted string (just before the "f") and continuing for five characters –ding just after the "d").

The offsets don't have to locate something within the string you found; for example, string(1,Medfnord,8,0) locates the point immediately after the word. Also, the located object doesn't need to actually contain any characters; it can just be a point. That may be a difficult link for a user to click on with a mouse, but it is a perfectly acceptable destination for a link or an insertion point for a block text from another page.

When counting text within an element, all of the text of that element and its descendants are counted. So for the paragraph:

```
<para id="somnolence">Some sought <emph>so</emph>
sonorous sounds <emph>south</emph>ward.</para>
```

The XPointer id(somnolence).string(all,"so") finds five occurrences. (Remember, the match is case-sensitive. "So" doesn't count.) id(somnolence).string(1,"southward") would locate the last word, even though it is partly in one element and partly outside of it.

1.5.3.7 Spans

Not everything you want to locate is going to lend itself to neat packaging as an element or a bit of text entirely within one element. For this reason, XPointer gives you a way to locate two objects and everything in between using the *span* keyword.

The syntax is very simple:

```
span(XPointer,XPointer)
```

For instance, in our <simpledoc> example earlier you could locate the range from the emphasized word "very" to the emphasized word "so" with root().span (descendant(1,emph),descendant(2,emph)).

1.5.3.8 Arcana

XPointers can actually do a bit more. These things aren't really of interest to someone reading a document in a browser or preparing a document for viewing in a browser. They're mostly of interest to programmers who are processing XML documents. However, we have included them here for completeness.

1.5.3.9 Additional node types

XPointers can locate more than elements and text. For example, everywhere `#element` is allowed, you can also use these keywords:

`#pi`

> Finds processing instructions; for example, the XPointer `id(foo).child(5,#pi)` finds the fifth processing instruction in the element with ID "foo."

`#comment`

> Finds comments; `id(fnord).child(1,#comment)` locates the first comment in the element with ID "fnord".

`#all`

> Finds all children. In the example at the beginning of Section 1.5.3.5 the XPointer `root().child(1, para).child(all,#all)` finds the first text chunk (`"This is a <">`), the `<emph>` element, and the last text chunk (`"simple document.<">`).

1.5.3.10 Locating attributes

You can locate elements with specific attributes using `attr()`. As with the special node types described earlier, this is really only of interest to programmers processing XML documents. Since attributes aren't displayed (though they can be used as the source for text created by a stylesheet), there's really nothing for a browser to link to. This is more of a programmer's abstraction, providing a uniform interface to all of the pieces of a document.

The `attr()` keyword doesn't have a number like `child()` because attributes aren't ordered; `<para type="sarcastic<"> clearance="low<">>` is the same as `<para clearance="low<"> type="sarcastic">`.

1.5.3.11 When XPointers fail

There are two uses for XPointers: finding where a link can begin and finding where it can end. In both cases, we would expect a link-checking program to tell you that it can't find the end of the XPointer you provided. However, for a browser, things can fail silently. If the XPointer is supposed to find where a link can begin and doesn't find any matching objects in a document, the browser simply won't create any linking points. The user may never know that the link was intended to exist, although a browser might provide an option to tell the user.

If the beginning of a link was successfully located, but the end is broken, the browser may not notice until the user requests that the browser follow the link. In that case, it would be much like if you linked to an HTML page that no longer existed; the browser would return an error message stating that it simply couldn't find the end of the link.

This is largely speculation since the XPointer draft is not final and does not describe behavior in the face of error. Moreover, as of this writing, there are only a few small test implementations of XPointer.

1.5.4 XLink

Now that we know about XPointers, let's take a look at some inline links:

```
<?xml version="1.0"?>
<simpledoc>
<title>An XLink Demonstration</title>
<section id="target-section">
<para>This is a paragraph in the first section.</para>
<para>More information about XLink can be found at
   <reference xml:link="simple"
       href="http://www.w3.org/">
     the W3C
   </reference>.
</para>
</section>
<section id="origin-section">
<para>
This is a paragraph in the second section.
</para>
<para>
You should go read
  <reference xml:link="simple" href="#target-section">
     the first section
   </reference>
first.
</para>
</section>
</simpledoc>
```

The first link states that the text "the W3C" is linked to the URL http://www.w3.org/. How does the browser know? Simple. An HTML browser knows that every `<a>` element is a link because the browser has to handle only one document type. In XML, you can make up your own element type names, so the browser needs some way of identifying links.

XLink provides the `xml:link` attribute for link identification. A browser knows that it has found a simple link when any element sets the `xml:link` attribute to a value of "simple." A simple link is like the links in HTML: one-way, beginning at the point in the document where it occurs. (In fact, HTML links can be recast as XLinks with minimal effort.) In other words, the content of the link element can be selected for traversal at the other end. Returning to the source document is left to the browser.

Once an XLink processor has found a simple link, it looks for other attributes that it knows:

`href`

This attribute is deliberately named to be familiar to anyone who's used the Web before. Its value is the URI of the other end of the link; it can refer to an entire document, or to a point or element within that document. If the target is in an XML document, the fragment part of the URI is an XPointer.

This attribute must be specified, since without it, the link is meaningless. It is an error not to include it.

`role`

This is the nature of the object at the other end of the link. XLink doesn't predefine any roles; you might use a small set to distinguish different types of links in your documents, such as cross-references, additional reading, and contact information. The stylesheet might take a different action (such as presenting the link in a different color) based on the role, but the application won't do anything automatically. It is possible that a set of roles will be standardized, similar to the `rel` and `rev` attributes in HTML, and that a browser would automatically recognize those values, but primarily this attribute is for your own use.

`title`

A title for the resource at the other end of the link can be provided, identical to HTML's `title` attribute for the `<a>` element. A GUI browser might display the title as a tool tip; an aural browser might read the title when the user pauses at the link before selecting it. A stylesheet might also make use of the information, perhaps to build a list of references for a document.

`show`

This attribute is a bit anachronistic. In a world without stylesheets, there's no need for information like this in a link, but since XLink will be complete before XSL, it includes some hints for a user agent on what to do with the link. This attribute suggests what to do when the link is traversed. It can take three values:

`embed`

The content at the other end of the link should be retrieved and displayed where the link is. An example of this behavior in HTML is the `<img>` element, whose target is usually displayed within the document.

`replace`

When the link is activated, the browser should replace the current view with a view of the resource targeted by the link. This is what happens with the `<a>` element in HTML: the new page replaces the current one.

`new`

The browser should somehow create a new context, if possible. This is similar to the hackery done in HTML to convince Netscape to open a new window.

You do not need to give a value for this attribute. Remember that a link primarily *asserts* a relationship between data; behavior is best left to a stylesheet. So unless the behavior is paramount (as it might be in some cases of `embed`), it is best not to use this attribute.

`actuate`

The second of the behavioral attributes specifies when the link should be activated. It can take the following values:

`user`

The program waits until the user requests that the link be followed, as the `<a>` element in HTML does.

`auto`

The link should be followed immediately by the program; this is what most HTML browsers do with `<img>` elements, unless the user has turned off image loading.

`behavior`

The last of the behavioral attributes is a free-form one. If you are writing documents for an intranet and can make certain assumptions about the software used to process your documents, you can put instructions in this attribute specific to that software. Similarly, your stylesheet can take advantage of this information; you might use this as a place to hide special scripts that only apply to this one link and aren't sufficiently general to live in a stylesheet. Do not expect any browser to automatically understand the value you specify for this attribute, if any.

`content-role`

This is similar to the `role` attribute above; it specifies the role the content of the link plays. For instance, a link containing a name might specify that the role is "bibliography," and the `content-role` is "author," indicating that this is a link to the works of the author named in the link.

`content-title`

Analogously, this is the title of the content of the link. It might be used by a browser displaying a navigational diagram of a web, or as a "tool tip" on the "Back" button of a browser.

1.5.4.1 Attribute remapping

As if that weren't complicated enough, XLink provides a way to call all of those attributes something else. For instance, you may have existing data that uses the "title" attribute for something else; perhaps the honorific for a person, or the name of a book. But you still want the function provided by XLink's title; what do you do?

The `xml:attributes` attribute lets you rename, or remap, the attributes used. Its value is a series of pairs, all separated by spaces. The first item in each pair is the name of an attribute that an XLink processor expects to find, such as "title" or "role"; the second is the name of the attribute that you actually used.

In this example, since "title" is already taken, you might want to use the attribute "link-title" instead. Your element might look like this:

```
<person title="Reverend" title-abbr="Rev."
  given="Kirby" family="Hensley"
  href="http://www.ulc.org/"
  link-title="Universal Life Church"
  xml:attributes="title link-title"/>
```

In this case, the `xml:attributes` attribute says that the title of the link is in the "link-title" attribute; the browser might display "Universal Life Church" when the mouse hovers over the link, instead of "Reverend" (which wouldn't be helpful).

The remapping attribute can take more than one pair of attribute names; this is a more degenerate example:

```
<job title="Managerial Analyst" role="counting beans"
  show="Dog and Pony" actuate="self" behavior="bland"
  href="http://www.oreilly.com/oreilly/jobs/"
  href-role="job description"
  href-title="Full Description" link-show="replace"
  link-actuate="user"
  xml:attributes="title href-title role href-role
  show link-show actuate link-actuate behavior
  link-behavior"/>
```

Notice that there doesn't need to be any system to the new names, and that even unused attributes (link-behavior, in this case) can be referred to.

1.5.5 Building Extended Links

XLink has much more to offer, including links similar to simple ones but that can travel to multiple destinations, and links that aren't even in the same document as any of the information that they link to.

1.5.5.1 Inline multiended links

Consider the following:

```
<para>
More information about XLink can be found at
these sites
<references xml:link="extended" inline="true">
<resource xml:link="locator"
  href="http://www.w3.org/" title="W3C"/>
<resource xml:link="locator"
  href="http://www.ucc.ie/xml/" title="The XML FAQ"/>
<resource xml:link="locator"
  href="http://www.xml.com/" title="XML.com"/>
</reference>.</para>
```

This somewhat complicated example creates a link with four ends:

- The text "these sites"
- http://www.w3.org/
- http://www.ucc.ie/xml/
- http://www.xml.com/

The text is part of the link because `inline="true"`. The other ends of the link are identified by a locator; in this case, the `<resource>` elements (identified as locators by its `xml:link` attributes).

An extended link is an element that might contain text and also contains locators. Each locator identifies one end of the link; the content of the element itself might also be part of the link. The extended link element is identified to an XLink processor by the `xml:link` attribute having a value of `extended`, similar to how simple links are identified. Extended links also share a number of attributes with simple links, but now those attributes are split between the extended link and the locators themselves. Links on the extended link element itself are:

`inline`

> This states whether or not (`true` or `false`) the content of the extended link is one end of the link. In the previous example, this attribute is set to `ftrue`, so the text (or other content) of the link element acts as one end, selectable by the user.

`role`

> In slight contrast to the same attribute on a simple link, `role` is a free-form attribute describing the nature of the whole link. You need not specify this attribute, but it can be used by your stylesheet to distinguish between different kinds of links that you might use, or by future browsers or specialized applications to understand certain predefined kinds of links.

`content-role`

> This is identical to the `content-role` of the simple link; it identifies the role of the content, as a link end, if inline is true.

`content-title`

> Like `content-role`, this is the title for the content as a link end, if indeed it is one.

Note that link group itself does not have an `href` attribute; rather, each locator child identifies a separate end of the link, with the content of the link group perhaps acting as one end. Each link locator can have attributes that further describe it. For any of these, a value can be specified on the extended link, which will act as a default for any links contained therein that don't specify an explicit value of their own.

`href`

> As with a simple link, this is the URI of one of the other ends of the link. Each link end locator can have a different `href`.

`role`

> This is also like the role of the simple link but different from the role of the extended link itself. This role is the role of the information at the other end of the link. There are no prescribed values; use whatever seems appropriate, and take advantage of the information in a stylesheet.

`title`

> The same as the simple link's title, this is a descriptive string that might be shown by a browser before the link is activated; for instance, in a pop-up menu of link destinations.

`show`

> This is the same as in a simple link. It is a hint to a browser in the absence of a stylesheet, to suggest behavior for the browser to take regarding the link.

`actuate`

> This is the same as in a simple link.

`behavior`

> This is the same as in a simple link.

1.5.5.2 Out-of-line links

A multiended link already links multiple resources. That means that the link could choose not to participate, and the link would still make sense. In other words, one document can create a link between two or more completely separate documents. The creator of the link doesn't even need to be able to change those other documents!

Out-of-line links are just as easy as inline multiended ones. Just use `inline="false"` and add locators for the portions of the other documents that you want to link. You'll probably want to use `role` attributes to denote which end of the link should be the beginning point for a user to activate, and which should be the end. Right now, there's no formalized mechanism for specifying this, but we expect that there will be shortly, as the Working Group is aware of the need.

Naturally, a browser won't know about the link you've added to a document unless it's also seen the document containing the link itself. XLink provides a feature called *extended link groups* (not to be confused with the similarly named extended links). A group contains links to multiple documents. When encountering a group, a browser is expected to read all the referenced documents; for the following political commentary, the hub document would direct the browser to the opponent's web site, and would either contain links to be applied to it or direct the browser to the document that did.

```
<critique xml:link="group">
  <hogwash xml:link="document"
      href="http://www.bobroberts.org/"/>
  <sanity xml:link="document"
      href="http://www.ourside.org/hooray.xml"/>
</critique>
```

A group takes one attribute: `steps`, which is a sort of sanity check an author can apply. Its value is a number; if a linked document contains a group that in turn links to other documents, there is a potential for a browser to get caught in a perfidious descent. The `steps` attribute says how far to go. Its value is not binding on a processor; one would hope that any XLink-enabled browser would have some reasonable error detection and recovery behavior built-in, rendering this attribute unnecessary.

The document only takes an `href` attribute that points to the document, or part thereof, in question. Obviously, on both the document and the group, any other attributes can be applied as you see necessary for your particular use, and a document with a group in it can have other things as well.