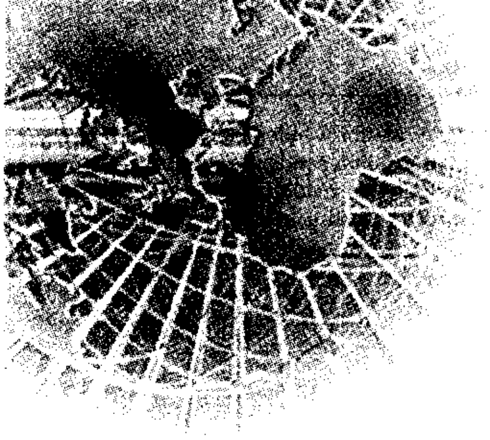


27 Working with XML

Extensible Markup Language (XML) is a markup language based on simple, platform-independent rules for processing and displaying textual information in a structured way. The platform-independent nature of XML makes an XML document an ideal format for exchanging structured textual information among different applications. In the recent years, XML has evolved as the de facto standard for document markup. Since its advent, XML has been used to implement various operations, such as:

- Configuration information
- Publishing
- Electronic Data Interchange (EDI)
- Voicemail systems
- Vector graphics
- Remote Method Invocation (RMI)
- Object serialization

XML provides customized tags to format and display textual information. What makes XML an integral element of enterprise computing is that XML documents are simple text documents that represent data in a platform-neutral manner. For example, an XML document generated by an application running on Microsoft Windows can be easily consumed by an application running on Sun Solaris.

This chapter describes XML and its related specifications. It also describes how to write XML data, so that you can become familiar with the XML syntax.

Introduction to XML

Markup Languages (MLs) such as SGML, HTML and Extensible Hypertext Markup Language (XHTML), are used to format text. In early times, the set of instructions given to printers to print a page in a specified format was called markup, and the language based on these instructions was called a Markup Language. This is how the concept of MLs originated. These instructions were written in a format that was different from the main text so that they would be easily differentiated from the main text. This format was later transformed in the form of tags that are currently used with markup languages.

Introduced in 1969, Generalized Markup Language (GML) was the first markup language. GML usᵉd tags to format text in a document. However, different documents created using GML required different compilers to compile them. In addition, standards were not available for compiling the documents written in GML. This led to the evolution of SGML.

SGML, introduced in 1986, is the markup language used to define another markup language. SGML allows you to define and create documents that are platform independent and can be used to exchange data over a network. Despite the advantages that SGML offers, developers felt a need for another markup language because the authoring software used to create SGML documents was complex and expensive.

Unfortunately, SGML is such a complicated language that it is not well suited for data interchange over the web. Another problem related with SGML is that it is computer-specific. Therefore, the developers felt a need for another markup language, which is not computer-specific. This led to the evolution of HTML.

HTML was invented by Tim Berners-Lee in 1990. It is based on Tim's own protocol, which is known as HTTP. Although HTML is incredibly successful, it is also limited in its scope; it is only intended for displaying documents in a browser. The tags it makes available do not provide any information about the content they encompass, it only has instructions on how to display that content. This means that you could create an HTML document which displays information about a person, but you couldn't write a program to figure out from the HTML document which piece of information relates to the person's first name. In fact, that program would not even know that the document was about a person at all. This led to the evolution of XML.

XML Basics

In this section, we'll go through the basics of XML and understand what XML is all about, and how it is used. Let's know about the major components that make XML great for information storage and interchange. An XML document contains many components, such as syntax, elements, attributes, and declarations. In this section, we'll cover the following components:

- XML syntax
- XML declaration
- XML elements
- XML attributes
- Valid XML documents
- Viewing XML
- XML Parser
- Document Type Definition
- Parameter Entities and Conditional Sections

XML Syntax

Every XML document abides by some syntax rules that specify how a document is created. The declaration statements or markup tags define the syntax for creating XML documents. The syntax used to create an XML document is called markup syntax. It is used to define the structure of data in the document. The following rules are associated with the markup syntax:

- XML documents must have a starting tag and closing tag
- XML tags are case-sensitive
- XML elements must be properly nested
- XML documents must have a Root Element and only one Root Element.
- XML attributes values must be quoted
- In XML, white space is preserved

Let's consider an XML file as shown in the following code snippet:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<Employee>
   <h1><ul><FirstName>Ambrish</FirstName></ul></h1>
   <MiddleName>Kumar</MiddleName>
   <LastName>Singh</LastName>
   <Age>25</Age>
   <EMPID id="A24" </EMPID>
</Employee>
```

In the preceding code snippet, the document contains numerous starting and closing tags. These tags are case-sensitive. It means if you write `<FirstName>` as a starting tag and close it with `</firstname>`, then it would generate an error. In the code, given previously, all elements are properly nested. For example, consider the following line of code:

```
<h1><ul><FirstName>Ambrish</FirstName></ul></h1>
```

In this code line, the first tag is `<h1>` and the line closes with the same tag in the same order. You cannot write like this:

```
<h1><ul><FirstName>Ambrish</FirstName></h1></ul>
```

In this case, it will give an error.

XML Declaration

The XML declaration statement is used to indicate that the specified document is an XML document. Although it is not required to have an XML declaration, it is considered a good practice to include it. If XML declaration statement is present, then it will be the first line in the document which defines the XML version and character encoding. An XML declaration looks like this:

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

Some important points to note about XML declaration are as follows:

- XML declaration starts with `<? xml`, and ends with `?>`
- If it is included then it must include the `version` attribute as it is required, but the encoding and standalone attributes are optional
- The XML declaration must be at the beginning of the file
- The version, encoding, and standalone attributes must be in that order

XML Elements

An XML element is the basic syntactic construct of an XML document. You can say that the building blocks of any XML document are its elements. A start tag and an end tag delimit an element in an XML document. An example of an XML element is as follows:

```
<Employees></Employees>
```

A start tag within an element is delimited by the < and > characters and has a tag name. In the previous start tag, the name is `Employees`. However, it is useful to keep in mind that a tag name must begin with a letter and can contain hyphen (-) and underscore (_) characters. An end tag is delimited by the </ and > character sequences and also contains a tag name. A document must have a single root element, which is also known as the document element. If you assume that the `Employees` element is your root element, then your document would be as follows:

```
<? xml version='1.0' encoding='UTF-8' standalone='yes'?>
<Employees></Employees>
```

This is an example of a well-formed XML document, where, of course, the XML declaration on the first line is optional; omitting the XML declaration would still leave you with a well-formed document.

However, there are some rules associated with elements. These rules are as follows:

- Element names can start with letters or the underscore (_) character, but not numbers or other punctuation characters.
- After the first character, numbers are allowed, as are the characters `-` and `.`.
- Element names cannot contain spaces.
- Element names can't contain the : character. Strictly speaking, this character is allowed, but the XML specification says that it is reserved. You should avoid using it in your documents, unless you are working with namespaces, which we'll discuss later in this chapter.
- Element names cannot start with the letters `xml`, in uppercase, lowercase, or mixed (i.e. `xml`, `XML`, `XmL`, or any other combination)
- There cannot be a space after the opening < character; the name of the element must come immediately after it. However, there can be space before the closing > character, if desired.

Nested Elements

An XML element can contain other nested elements. For example, the root element may contain a nested element having the text content "Ambrish":

```
<? xml version='1.0' encoding='UTF-8' standalone='yes'?>
<Employees>
  <FirstName>Ambrish</FirstName>
</Employees>
```

Here, the `<FirstName>` element is nested in the `<Employees>` element. The `<FirstName>` element contains the text `Ambrish`.

Empty Elements

An element may have no nested element or content. Such an element is termed an empty element, and it can be written with a special start tag that has no end tag. For example, the `<MiddleName/>` is an empty element. If you include this empty element within your document, the document looks like this:

```
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
<Employees>
  <FirstName>Ambrish</FirstName>
  <MiddleName/>
</Employees>
```

Elements can have attributes, which are specified in the start tag. An example of an attribute is `<LastName title="Singh"></LastName>`. An attribute is defined as a name-value pair. In the previous example, the name of the attribute is, of course, `title`, and the value of the attribute is `Singh`. With an attribute added, the example document looks like the one shown here:

```
<? xml version='1.0' encoding='UTF-8' standalone='yes' ?>
<Employees>
  <FirstName>Ambrish</FirstName>
  <MiddleName/>
  <LastName title="Singh"></LastName>
</Employees>
```

Escaping Delimiter Characters

XML tags use angle brackets (> and < symbols) for enclosing tag names. Thus, these characters cannot appear in the element data. If you want to include these symbols in your data, you must use the replacement character sequences. Table 27.1 lists all the special characters and their corresponding replacement character sequences in XML:

Table 27.1: Special characters and replacement character sequences in XML

[illegible]	[illegible]
&	&amp;
‘	&apos;
“	&quot;
<	&lt;
>	&gt;

Now let's assume that you want to add another element `Birthday` having an attribute name, `date`, with the value `<24/01/1982>`. As already mentioned under the heading "XML Elements", you are not allowed to include delimiter characters within an attribute value. However, you can use the `& lt;` character sequence to escape <, and the `&gt;` character sequence to escape >. So, with that in place, the document now looks as follows:

```
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
<Employees>
  <FirstName>Ambrish</FirstName>
  <MiddleName/>
  <LastName title="Singh"></LastName>
  <Birthday date="&lt; 24/01/1982&gt;"> < /Birthday>
</Employees>
```

Another mechanism for including delimiter characters within the body of a construct is to use escaped numeric references. For example, the numeric American Standard Code for Information Interchange (ASCII) value for the > character is `62`. So you can use the `&#62;` character sequence instead of `&gt;`. Using escaped numeric references is, of course, the most general mechanism for including delimiter characters within a construct's body.

XML Attributes

XML attributes provide additional information about elements. Let's again consider the following code snippet:

```
<? xml version='1.0' encoding='UTF-8' standalone='yes' ?>
<Employees>
  <FirstName>Ambrish</FirstName>
```

```
        <Middlename/>
        <LastName title="Singh"></LastName>
        <Birthday date="&lt;24/01/1982&gt;"></Birthday>
    </Employees>
```

Here, `date` is an attribute for the element `<Birthday>` as the `title` is for the element `<LastName>`. Attributes values are always given in single or double quotes. In case the value itself contains double quotes then give it in single quote, otherwise use the double quote syntax. Attributes are used for storing data. However, the data can also be stored in child elements. There is no rule associated with XML documents as to when to use child elements or attributes for storing data. The best practice says that you should avoid using attributes. Some problems with using attributes are as follows:

- Attributes have no multiple values, whereas child elements have.
- You cannot expand attributes easily.
- Attributes cannot describe structures, whereas child elements can.
- Attributes are difficult to maintain programmatically.
- It is difficult to test attributes values against DTD.

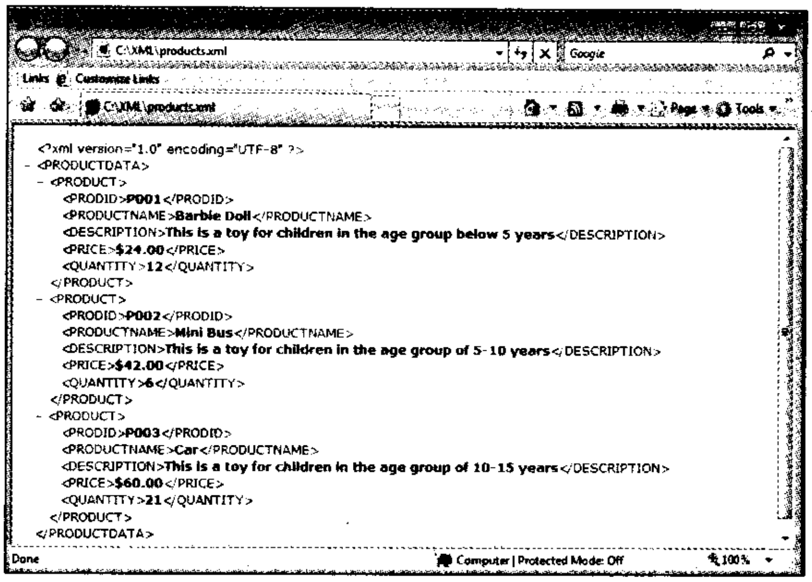
Valid XML Documents

An XML document is said to be valid if it is validated against a Document Type Definition (DTD). Valid XML documents have correct XML syntax. An XML document is said to be well formed, if it conforms the following:

- XML documents must have a starting tag and closing tag.
- XML tags are case-sensitive.
- XML elements must be properly nested.
- XML documents must have a Root Element and only one Root Element.
- XML attributes values must be quoted.
- In XML, white space is preserved.
- XML declaration starts with `<? Xml`, and ends with`?>`.
- If it is included then you must include the `version` attribute as it is required, but the encoding and standalone attributes are optional.
- The XML declaration must be at the beginning of the file.
- The version, encoding, and standalone attributes must be in that order.
- Element names can start with letters or the _ character, but not numbers or other punctuation characters.
- After the first character, numbers are allowed, as are the characters - and **.**.
- Element names can't contain spaces.
- Element names can't contain the : character. Strictly speaking, this character is allowed but the XML specification says that it is reserved. You should avoid using it in your documents, unless you are working with namespaces, which we discuss later in this chapter.
- Element names can't start with the letters `xml`, in uppercase, lowercase, or mixed, i.e. `xml`, `XML`, `XmL`, or any other combination.
- There can't be a space after the opening < character; the name of the element must come immediately after it. However, there can be space before the closing > character, if desired.

Viewing XML

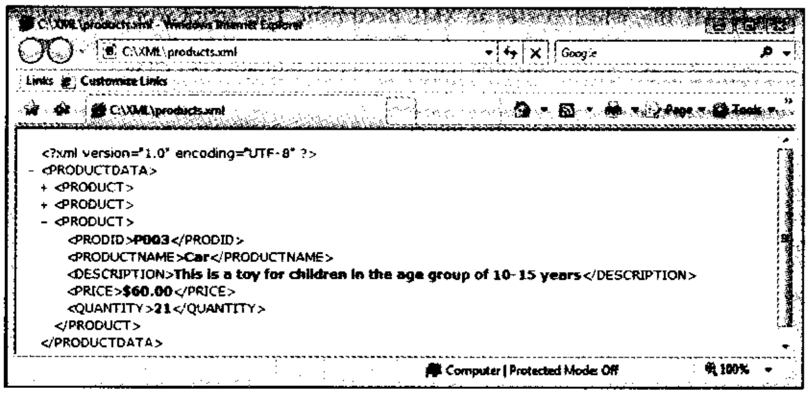
Let's now understand how XML documents are viewed in your browsers. You can use XML for any browser. Each XML file contains a plus (+) or minus (-) sign in the left of the elements. When you click on the minus sign the elements collapse, and when you click on the plus sign the elements expand. For example, when you open an XML document in IE, it will look like the one shown in Figure 27.1(you can find the products.xml file in the `Code/XML/Chapter 27/XML` folder on the CD):



```
  <?xml version="1.0" encoding="UTF-8" ?>
- <PRODUCTDATA>
  - <PRODUCT>
      <PRODID>P001</PRODID>
      <PRODUCTNAME>Barbie Doll</PRODUCTNAME>
      <DESCRIPTION>This is a toy for children in the age group below 5 years</DESCRIPTION>
      <PRICE>$24.00</PRICE>
      <QUANTITY>12</QUANTITY>
    </PRODUCT>
  - <PRODUCT>
      <PRODID>P002</PRODID>
      <PRODUCTNAME>Mini Bus</PRODUCTNAME>
      <DESCRIPTION>This is a toy for children in the age group of 5-10 years</DESCRIPTION>
      <PRICE>$42.00</PRICE>
      <QUANTITY>6</QUANTITY>
    </PRODUCT>
  - <PRODUCT>
      <PRODID>P003</PRODID>
      <PRODUCTNAME>Car</PRODUCTNAME>
      <DESCRIPTION>This is a toy for children in the age group of 10-15 years</DESCRIPTION>
      <PRICE>$60.00</PRICE>
      <QUANTITY>21</QUANTITY>
    </PRODUCT>
  </PRODUCTDATA>
```

Figure 27.1: Viewing XML in IE

Suppose you click on the second and third minus (-) sign from the top, then the display will look like the one shown in Figure 27.2:


C:\XML\products.xml - Windows Internet Explorer
C:\XML\products.xml
Google
Links
Customize Links
C:\XML\products.xml
Page
Tools
Computer | Protected Mode: Off
100%

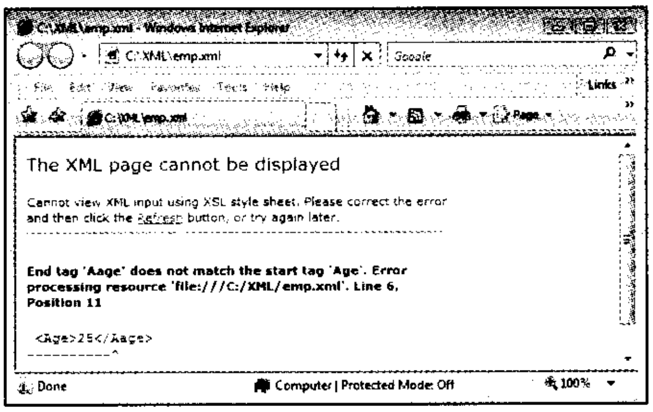
```
  <?xml version="1.0" encoding="UTF-8" ?>
- <PRODUCTDATA>
  + <PRODUCT>
  + <PRODUCT>
  - <PRODUCT>
      <PRODID>P003</PRODID>
      <PRODUCTNAME>Car</PRODUCTNAME>
      <DESCRIPTION>This is a toy for children in the age group of 10-15 years</DESCRIPTION>
      <PRICE>$60.00</PRICE>
      <QUANTITY>21</QUANTITY>
    </PRODUCT>
  </PRODUCTDATA>
```

Figure 27.2: Collapsed XML File in IE

However, when your XML file contains any error it will display it in the browser. In this case the XML file that contains an error because the start tag and end tag of Age elements do not match:

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

<Employee>
  <FirstName>Ambrish</FirstName>
  <MiddleName>Kumar</MiddleName>
  <LastName>Singh</LastName>
  <Age>25</Aage>

</Employee>
```

When you open this file in IE, it displays an error message like the one shown in Figure 27.3:


C:\XML\emp.xml - Windows Internet Explorer
C:\XML\emp.xml
Google
File Edit View Favorites Tools Help
Links
C:\XML\emp.xml
Page
Done
Computer | Protected Mode: Off
100%

The XML page cannot be displayed

Cannot view XML input using XSL style sheet. Please correct the error and then click the Refresh button, or try again later.

End tag 'Aage' does not match the start tag 'Age'. Error processing resource 'file:///C:/XML/emp.xml'. Line 6, Position 11

```
 <Age>25</Aage>
----------^
```

Figure 27.3: An XML File Showing Error in IE

XML Parser

The XML Parser is used to read, update, create, and manipulate the XML document. For manipulating the XML document, the XML parser loads the document into the computer's memory and then the data is manipulated using the DOM node-tree structure. The XML parser is a part of the software, which reads the XML files and tests whether the XML document is well-formed against the given DTD or the XML schema. Moreover, the XML Parser also makes the XML files available to the application with the use of the DOM. There are some differences between Microsoft's XML parser and Mozilla's XML parser.

Let's consider an HTML file, which parses an XML document. This script works for Internet Explorer, Mozilla, Opera, etc. The code, given in Listing 27.1, for the HTML file required for this application (you can find this file named xmlparser2.html in the `Code/XML/Chapter 27/XML` folder on the CD):

Listing 27.1: xmlparser2.html

```
<html>
<head>
<script type="text/javascript">
var xmlParseDoc;
function loadParseXML()
{
  if (window.ActiveXObject) {
        xmlParseDoc=new ActiveXObject("Microsoft.XMLDOM");
        xmlParseDoc.async=false;
        xmlParseDoc.load("emp.xml");
        getmessage();
  }
  else if (document.implementation &&
  document.implementation.createDocument) {
        xmlParseDoc=document.implementation.createDocument("","",null);
        xmlParseDoc.load("emp.xml");
        xmlParseDoc.onload=getmessage;
  } else {
        alert('The Script is not Compatible with your Browser');
  }
}
function getmessage()
{
  document.getElementById("FirstName").innerHTML=
  xmlParseDoc.getElementsByTagName("FirstName")[0].childNodes[0].nodeValue;
  document.getElementById("MiddleName").innerHTML=
  xmlParseDoc.getElementsByTagName("MiddleName")[0].childNodes[0].nodeValue;
```

```
        document.getElementById("LastName").innerHTML=
        xmlParseDoc.getElementsByTagName("LastName")[0].childNodes[0].nodeValue;
        document.getElementById("Age").innerHTML=
        xmlParseDoc.getElementsByTagName("Age")[0].childNodes[0].nodeValue;
    }
    </script>
    </head><body onload="loadParseXML()">
    <h1>Employee Details</h1>
    <p><b>First Name:</b> <span id="FirstName"></span><br />
    <b>Middle Name:</b> <span id="MiddleName"></span><br />
    <b>Last Name:</b> <span id="LastName"></span><br />
    <b>Age:</b> <span id="Age"></span>
    </p>
    </body>
    </html>
```

Consider the following lines:

```
    xmlParseDoc=new ActiveXObject("Microsoft.XMLDOM");
      xmlParseDoc.async=false;
      xmlParseDoc.load("emp.xml");
```

In the preceding code snippet, the first line creates an instance of Microsoft's XML parser. The script will not start execution before the document is properly loaded. This can be done by turning off asynchronized loading. The second line does the same. The third line simply loads the XML documents.

The code will slightly change when performing the same things in Mozilla, Firebox, and Opera. It looks like the one shown here:

```
    xmlParseDoc=document.implementation.createDocument("","",null);
      xmlParseDoc.load("emp.xml");
      xmlParseDoc.onload=getmessage;
```

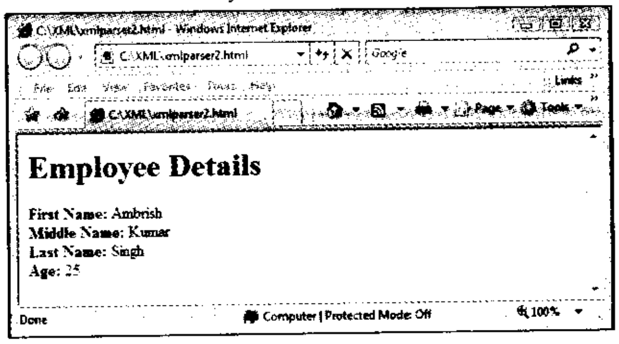
In the first line, the first parameter tells the XML namespace, the second parameter specifies the root of XML document, and the third is always null as it is not implemented yet.

Here's the XML document, given in Listing 27.2, which this file parses (you can find the emp.xml file in the `Code/XML/Chapter 27/XML` folder on the CD):

Listing 27.2: emp.xml

```
    <?xml version="1.0" encoding="ISO-8859-1" ?>
    <Employee>
      <FirstName>Ambrish</FirstName>
      <MiddleName>Kumar</MiddleName>
      <LastName>Singh</LastName>
      <Age>25</Age>
    </Employee>
```

When you run the `xmlparser2.html` file on your browser, a Web page, as shown in Figure 27.4, is displayed:


C:\XML\xmlparser2.html - Windows Internet Explorer
C:\XML\xmlparser2.html
Google
File Edit View Favorites Tools Help
Links
Page Tools
Done
Computer | Protected Mode: Off
100%

Employee Details

First Name: Ambrish
Middle Name: Kumar
Last Name: Singh
Age: 25

Figure 27.4: Parsing an XML File

Document Type Definition

The purpose of a Document Type Definition (DTD) is to define the rules and attributes for using the tags in an XML document. A DTD defines the XML document structure with a list of legal elements and attributes. A DTD can be declared inline in your XML document, or as an external reference.

Defining DTD for a Single Element

Let's learn about writing DTD definitions. Initially, we define DTD for a single element, which does not contain any text or other element. This DTD tells the parser that the XML document contains elements of the type defined in DTD, and these elements are also not designed to contain text. The following code snippet shows the portion of DTD required for the product.xml file:

```
<?xml version='1.0' encoding='UTF-8'?>
<!--
DTD for a list of toys
-->
<!ELEMENT PRODUCTDATA (PRODUCT+)>
```

In this code snippet, the ELEMENT tag prefixes with characters <! and is followed by the name of the element (PRODUCTDATA) and PRODUCT+ expression inside parenthesis. This indicates that the PRODUCTDATA element contains one or more elements of type <PRODUCT>. If we do not use the plus sign, the DTD definition conveys to the parser that the PRODUCTDATA element only contains a single PRODUCT element.

Table 27.2 lists all qualifiers that we can use in DTD definition.

Table 27.2: Qualifiers of a DTD File

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

In the previous code snippet, you saw a DTD definition where one element can contain several elements of only one type. You can define DTDs for complex and hierarchical XML documents where the root element contains many instances of different types of elements specified in some definite order. For this, you need to include multiple elements, appended with qualifiers, inside the parentheses in a comma-separated list. The comma-separated list specifies the validity and the order of occurrence of each element.

You can also nest parentheses to group multiple items. For example, to include an image element for every title element, you need to include the ((image, `title`)+) expression inside parentheses.

Defining Text and Nested Elements

Now we are ready to include text inside the elements and define nested elements. We now create a DTD definition that tells the parser that which elements can contain nested elements or text. The following code snippet shows how to create the elements of the product.xml file to contain text or nested elements:

```
<!ELEMENT PRODUCTDATA (PRODUCT+)>
<!ELEMENT PRODUCT (PRODUCTNAME,DESCRIPTION)>
<!ELEMENT PRODUCTNAME (#PCDATA)>
<!ELEMENT DESCRIPTION(#PCDATA)>
```

This code snippet tells the parser that the <PRODUCT> element must consist of the <PRODUCTNAME> and <DESCRIPTION> elements. The <PRODUCTNAME> and <DESCRIPTION> elements are defined to contain the data of type PCDATA. PCDATA stands for parsed character data. The difference between PCDATA and CDATA is that PCDATA text is parsed while the text inside the CDATA section is ignored by the parser. The # symbol in the preceding code snippet indicates that the keyword used after this symbol is a special word rather than an element name.

You can define DTD using the or condition. You can do this using the vertical bar (|). Consider the following code snippet:

```
<!ELEMENT item (#PCDATA | item)* >
```

In the preceding code snippet, the element item can contain either PCDATA or an item. The asterisk at the end of the code snippet signifies that either element can occur zero or more number of times in a succession. Therefore, item element can contain mixed content (both text and element).

Referencing the DTD

You can include DTD in two ways, directly and indirectly. Including a DTD directly involves including the DTD in the parent XML document. Including a DTD indirectly involves providing a DTD as a separate file outside the parent document, with a reference of the external DTD in the parent document. This process of providing a reference to the external DTD in the parent XML document is called referencing the DTD. Referencing a DTD in a XML document is required when size of the DTD is large.

Let's suppose our DTD name is drink.dtd and it is located in the `http://kogentindia.com/lifestyle/drink.dtd`. Listing 27.3 shows the drink.xml document:

Listing 27.3: drink.xml document

```
<!-- A SAMPLE set of -->
<!DOCTYPE  collection SYSTEM "http://kogentindia.com/lifestyle/drink.dtd">
<collection>
<desc>This XML document contains the procedure for creating Fruit Delight drink.</desc>
<drink>
  <name>Fruit Delight</name>
  <ingredient name="Chopped Fresh Pineapple" amount="1 cup" unit="gram" />
  <ingredient name="Mashed Ripe Banana" amount="3/4 cup"/>
  <ingredient name="Sugar" amount="10 spoons" unit=gram />
  <ingredient name="Ice" amount="12" unit="cube"/>
  <ingredient name="Lemon Juice" amount="1/2 cup" unit="cup" />
  <ingredient name="Lemon Drink" amount="3/2 bottle" unit="ml" />
  <ingredient name="Cherries" amount="1/2 spoon" unit="mg" />
  <prep>
        <step>Mix all finely chopped fruits.</step>
        <step>Blend these finely chopped fruits with sugar in a blender.</step>
        <step>Add lemon juice and crushed ice cubes to the fruit mix.</step>
        <step>Add lemon drink to the mixture to thin down to required liquidity.</step>
        <step>Blend all the ingredients to a smooth puree.</step>
        <step>Pour the drink in a tall glass and granish with pineapple wedges and
        cherries.</step>
  </prep>
  <nutri calories="126.05" fat="0.31g" carbohydrates="32.28g" protein="1.07g" />
</drink>
</collection>
```

The preceding Listing 27ontains an arbitrary reference to drink.dtd (as a placeholder). You have to manually include drink.dtd to make this example work correctly.

The DOCTYPE tag shows the location of drink.dtd as `http://kogentindia.com/lifestyle/drink.dtd` and also shows the name of the root element (that is collection) of drink.xml . You can provide a reference to the external DTD in the following two ways:

- **<!DOCTYPE collection SYSTEM "drink.dtd"**—This syntax is used when both drink.dtd and the XML document are residing in same directory.
- **<!DOCTYPE collection SYSTEM "c://drink.dtd">**—This syntax is used when drink.dtd and the parent XML document reside in separate directories.

The DOCTYPE specification can also contain DTD definitions within the XML document, rather than referring an external DTD file. Such definitions are specified in square brackets, as shown in the following code snippet:

```
<!DOCTYPE collection SYSTEM "drink.dtd" [
...local subset definitions here...
]>
```

Defining Attributes in the DTD

A non-validating parser is required while defining simple and nested elements. However, while working with validating parser, DTD must specify valid attributes for the different elements. Let's define the attributes for the elements in the drink.xml document. Listing 27.4 shows drink.dtd file used in drink.xml:

Listing 27.4: drink.dtd file

```
<!ELEMENT collection (desc,drink*)>

<!ELEMENT desc ANY>

<!ELEMENT drink (name,ingre*,prep,nutri)>

<!ELEMENT name (#PCDATA)>

<!ELEMENT ingre (ingre*,prep)?>
<!ATTLIST ingre name CDATA #REQUIRED
                quant CDATA #IMPLIED
                unit CDATA #IMPLIED>

<!ELEMENT prep(step*)>
<!ELEMENT step (#PCDATA)>

<!ELEMENT nutri EMPTY>
<!ATTLIST nutri protein CDATA #REQUIRED
                carbohydrates CDATA #REQUIRED
                fat CDATA #REQUIRED
                calories CDATA #REQUIRED>
```

The drink.dtd defines that drink.xml must have one root element (collection), which consists of one desc element and zero or more elements named drink. The element named drink must contain one name element, zero or more elements named ingre, one prep element, and one nutria element. Each element is defined using the ELEMENT standard DTD tag.

The `ATTLIST` DTD tag is used to define the attribute definitions of elements named ingre and nutri. The name parameter, which follows the `ATTLIST` tag, specifies the element for which the attributes are being defined.

Each attribute is defined by a series of three space-separated values. The first value in each line is the name of the attribute: in this case it is `name`, `quant`, or `unit`. The second element indicates the type of the data, such as `CDATA`. Table 27.3 lists the types of attributes of a DTD:

Table 27.3: Attributes of a DTD

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

Table 27.3: Attributes of a DTD

Attribute Type	Description
`NMTOKEN`	Specifies a valid XML name composed of letters, numbers, hyphens, underscores, and colons
`NMTOKENS`	Specifies a space-separated list of names
`NOTATION`	Specifies the name of a DTD-specified notation, which is used for non-XML data formats. For example, image files are specified in the non-XML data format

An attribute type may consist of a parenthesized list of choices, separated by vertical bars. In this case, the attribute must use one of the specified values. The following code snippet illustrates such a case:

```
<!ELEMENT shop (name, product*)>
<!ATTLIST product
type (cassette | cd ) #REQUIRED
>
<!ELEMENT name (#PCDATA)>
<!ATTLIST product name #PCDATA #REQUIRED
                  singer #PCDATA #REQUIRED
                  price  #PCDATA #REQUIRED>
```

This code snippet specifies that the `product` element's type attribute must be given as type="`cd`" or type="`cassette`". The type attribute cannot take any other value.

The last value in the attribute definition determines the attribute's default value, if any; and specifies whether or not the attribute is required. Table 27.4 lists the specifications of a DTD file:

Table 27.4: Specifications of a DTD file

[illegible]	[illegible]
`#FIXED "fixedValue"`	The attribute must take the value specified in double quotes after the #FIXED keyword
`#IMPLIED`	The attribute value need not be specified in the document
`#REQUIRED`	The attribute must take the value of the type specified in the attribute definition of the document
`"defaultValue"`	The default value must be used if a value is not specified in the document

Defining Entities in the DTD

You have seen predefined entities, such as `&amp`, in the section Substituting and Inserting Text. Let's now learn how to define entities of your own. Entities can be defined internally in an XML document or in an external DTD. Entities defined in an external DTD can be referenced using the following syntax:

```
<!ENTITY entity-name SYSTEM "URI/URL">
```

The `ENTITY` tag name specifies the referencing of a user-defined entity. The entity-name parameter specifies the name of the entity and the URL parameter represents the URL of the DTD or XML document that contains the definition of the entity.

Referencing external entities has been described in detail later. For now, let's define an entity internally, as shown in the following code snippet:

```
<!ENTITY entity-name "entity definition ">
```

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

Based on the preceding syntax, let's write the following code snippet defining two entities, writer and copyright:

```
<!ENTITY writer "Ken Greenwood">
<!ENTITY copyright "Copyright 1999 by sams">
```

Now, add the following code snippet in an XML document to use these entities:

```
<author> &writer;&copyright;</author>
```

In the preceding two code snippets, we have defined the copyright information of certain published materials created by an organization, Kogent. Suppose at a later point of time, another organization acquires the copyright of the book, the copyright information needs to be changed only at one place, i.e. in the copyright entity defined in the first code snippet. The updated copyright information would then be reflected in all the XML documents using the copyright entity (defined in the first code snippet),

Note that user defined entities are also referenced with the same syntax (`&entityName;`) that you use for predefined entities, and the entity can be referenced in an attribute value as well as in an element's contents.

Referencing External Entities

To reference external entities defined in a DTD or XML document, you need to use the `SYSTEM` or `PUBLIC` identifier. Let's reference an external entity, which is a copyright message contained in the copyright.xml file. The following code snippet shows how to specify a name for an external entity:

```
<!DOCTYPE books SYSTEM "books.dtd" [
<!ENTITY writer "Ken Greenwood">
<!ENTITY copyright SYSTEM "copyright.xml">
]>
```

The copyright.xml contains the following copyright message:

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

This book may not be duplicated in any way without the express written consent of the publisher, except in the form of brief excerpts or quotations for the purposes of review.

We can use external entities similar to internal entities, as shown in the following code snippet:

```
<author> &writer;&copyright;</author>
```

You can also use an external entity declaration to access a Servlet that produces the current `date` by referencing an external entity, as shown in the following code snippet:

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

You would then reference this entity similar to any other entity:

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

Referencing Binary Entities

You can reference unparsed entities, such as image files and multimedia data files, in XML documents in the following two ways:

- Using a MIME Data Type
- Using Entity References

Let's discuss each method in detail.

Using a MIME Data Type

XML namespaces standard, in conjunction with the `MIME` data types defined for electronic messaging attachments, provide a very useful, understandable, and extensible mechanism for referencing unparsed external entities. The following code snippet defines a DTD for XML documents that refer to image files:

```
<!ELEMENT placesindelhi (image?, title, description)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT image EMPTY>
<!ATTLIST image
alt CDATA #IMPLIED
src CDATA #REQUIRED
type CDATA "image/gif"
>
```

This code snippet declares `image` as an optional element in the parent element, `placesindelhi`. The code snippet also defines image as an `empty` element, and then defines the attributes it requires. The definition of the `image` tag is similar to that of the HTML 4.0 `img` tag, except the type attribute. The image tag attributes are defined by the `ATTLIST` entry. The `alt` attribute, which defines the alternative text to display in case the image cannot be found, accepts character data (`CDATA`). It has an implied value, which means that it is optional and that the program processing the data can replace the image with an alternative text, such as Image not found. On the other hand, the `src` attribute, which specifies the name of the image to be display, is required. The `type` attribute is intended for the specification of a MIME data type, as defined at `http://www.iana.org/assignments/media-types/`. It has a default value named `image/gif`. In the document, a reference to an image named `Intro-pic` can be declared as shown in the following code snippet:

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

Using Entity References

The second way is to create an external `ENTITY` reference is by using the notation mechanism. However, in this case, you need to specify DTD `NOTATION` elements for `JPEG` and `GIF` data. These notation elements can be obtained from a database. In the notation mechanism, you need to define a different `ENTITY` element for each image that you intend to reference. If you include new images in the XML document, you need to specify both a new entity definition in the `DTD` and a reference to it in the XML document. As a result, this mechanism is not preferred.

Limitations of DTDs

This section explains about the limitations of DTDs. Suppose you need to define an entity definition to contain either text, or text followed by one or more list items. However, this specification turns out to be hard to achieve in a DTD.

For example, you might be tempted to define an item as shown in the following code snippet:

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

However, part of the definition does not conform to mixed content model. As a result, the parser raises the following error:

```
Illegal mixed content model for 'item'. Found &#x28; ...
```

Let's now redefine the `item` element twice in a DTD definition:

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

Above DTD definition raises a duplicate definition warning during parsing and ignores the second definition.

In such cases, we can define item elements in a DTD to support mixed content model. Further, we cannot perform tests such as determining the kind of text inside PCDATA, determining whether text of PCDATA contains numbers or corresponds to appropriate date format.

All these limitations are fundamental motivations behind the development of schema-specification standards.

Parameter Entities and Conditional Sections

In this section, you'll see how to define and use parameter entities. You'll also learn how to use parameter entities with conditional sections in a DTD.

Creating and Referencing a Parameter Entities

If the drink.xml document uses HTML tags, it cannot be validated using entities and attribute definitions defined in drink.dtd. To validate these HTML-style tags in the XML document, you need to include the definitions of HTML style tags defined in xhtml.dtd. A parameter entity is required in such cases.

In the following code snippet, the <i> tag is used to display the text in italized format:

```
<prep>
  <step><i>Mix all finely chopped fruits.</i></step>
  <step><i>Blend these finely chopped fruits with sugar in a blender.</i></step>
```

```
    <step><i>Add lemon juice and crushed ice cubes to the fruit mix. </i></step>
    <step><i>Add lemon drink to the mixture to thin down to required liquidity. </i></step>
    <step><i>Blend all the ingredients to a smooth puree. </i></step>
    <step><i>Pour the drink in a tall glass and garnish with pineapple wedges and cherries.
    </i></step>
</prep>
```

Add the following code snippet to drink.dtd, so that the drink.xml document can contain the <i> HTML tag:

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

NOTE

To see all tags of xhtml.dtd (in HTML format), visit http://java.sun.com/j2ee/1.4/docs/tutorial/examples/xml/samples/xhtml-dtd.html.

In the preceding code snippet, you use an `<!ENTITY>` tag to define a parameter entity. The definition of the parameter entity is similar to a general entity, but you use a somewhat different syntax. You include a percent sign (`%`) before the entity name when you define the entity. In addition, you use the percent sign instead of an ampersand when you reference it in the parent XML document. Note that there are always two steps to be followed while using a parameter entity:

- The first is to define the Entity name.
- The second is to reference the entity name, which actually includes the external definitions in the current DTD.

The following code snippet defines the definition of an entity, which includes breaks and whose text is in the bold format:

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

The following code snippet shows how we can use this entity in element definitions:

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

The preceding code snippet allows the title element in the XML document to use the <b> and <em> tags.

Conditional Sections

You can use parameter entities to control conditional sections. As you know, you cannot check or verify the contents of an XML document. The verification of contents of an XML document can be done by defining conditional sections in a DTD, which become part of the DTD by specifying the include keyword. On the other hand, if you specify the ignore keyword; the conditional section is not included. For example, you need to define different versions of a DTD according to whether you were treating the document as an XML document or as a SGML document. You can do this by using DTD definitions, as show in the following code snippet:

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

The above code snippet shows how to include a conditional section in a DTD. A conditional section begins with `<![`, followed by the `INCLUDE` or `IGNORE` keyword and another `[ character; and finally the contents` of the conditional section are defined within [] followed by the terminator:] >.

The above code snippet specifies that the XML definitions are included, and the SGML definitions are excluded. We can include DTD definitions for both XML and SGML documents using parameter entities in place of the `INCLUDE` and `IGNORE` keywords, as shown in the following code snippet:

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

When using parameter entities, each document that uses the DTD can set up the following appropriate entity definitions:

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

This procedure verifies each document (XML or SGML) on the basis of the entity definitions provided in the DTD. It also replaces the INCLUDE and IGNORE keywords with variable names that more accurately reflect the purpose of the conditional section, producing a more readable, self-documenting version of the DTD.

Advanced XML

So far we discussed the basics of XML. Now it is time to discuss some advanced features of XML. The advanced features of XML include the following:

- XML namespaces
- XML CDATA
- XML encoding
- XML on the server
- XML application
- XMLHttpRequest object
- Saving Data to XML file

Let's explore more about these features.

XML Namespaces

We are now able to create a well-formed XML document. However, what happens if our application becomes more complex? Then we need to combine elements from various document types into one XML document. It is possible that two document types have elements with the same name, but with different meanings and semantics. This section will introduce XML namespaces, the means by which we can differentiate elements and attributes of different XML document types from each other when combining them together into other documents, or even when processing multiple documents simultaneously.

It is not required that every XML document have namespaces. Namespaces are optional components of basic XML documents. However, namespace declarations are recommended if your XML document is going to be shared with other XML documents that may share the same element names. Also, newer XML-based technologies, such as XML Schemas, SOAP, and WSDL, make heavy use of XML namespaces to identify data encoding types and important elements of their structure.

Let's consider a case in which one company believes that a <product> should contain a certain set of information and another company believes that it should contain a different set of information. Both these companies will create different document types to describe that information.

Let's consider an example in which we're going to combine various XML elements from different document types into one XML document. For example, we might create an XML document type containing information about an employee, including the employee's title, but also containing the employee's resume in XHTML form. Such a document may look similar to the one as shown here:

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

```
    <name>
      <title>Sir</title>
      <first>Ambrish</first>
      <middle>Kumar</middle>
      <last>Singh</last>
    </name>
    <position>Technical Writer</position>
    <resume>
      <html>
      <head><title>Resume of Ambrish Kumar Singh</title></head>
      <body>
      <h1>Ambrish</h1>
      <p>Ambrish is working in Kogent Solutions Inc</p>
      </body> </html>
    </resume>
    </employees>
```

To an XML parser, there isn't any difference between the two `<title>` elements in this document. If we do a simple search of the document to find Ambrish's title, we might accidentally get the 'Resume of Ambrish Kumar Singh', instead of 'Sir'. Even in our application, we cannot know which elements are XHTML elements and which aren't without knowing in advance the structure of the document. That is, we'd have to know that there is a `<resume>` element, which is a direct child of `<employees>`, and that all of the descendents of `<resume>` are a separate type of element from the others in our document. If our structure ever changed, all of our assumptions would be lost. In the document above, it looks like anything inside the `<resume>` element is XHTML, but in other documents it might not be so obvious, and to an XML parser it isn't obvious at all. Such types of problem are solved by using prefixes, which is discussed next.

Using Prefixes

The best way with which an XML parser can identify an element is by giving every element in a XML document a completely distinct name. For example, we might come up with a naming convention whereby every element for a proprietary XML document type gets its own prefix, and every XHTML element gets another prefix. We could rewrite our previous XML document something like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<emp:employees>
<emp:name>
  <emp:title>Sir</emp:title>
  <emp:first>Ambrish</emp:first>
  <emp:middle>Kumar</emp:middle>
  <emp:last>Singh</emp:last>
</emp:name>
<emp:position>Technical Writer</position>
<emp:resume>
<xhtml:html>
<xhtml:head><xhtml:title>Resume of Ambrish Kumar Singh</xhtml:title></xhtml:head>
<xhtml:body>
<xhtml:h1>Ambrish</xhtml:h1>
<xhtml:p>Ambrish is a Java Programmer</xhtml:p>
</xhtml:body>
</xhtml:html>
</emp:resume>
</emp:employees>
```

In this case, our XML parser can immediately tell what kind of title we're talking about—an `<emp:title>` or an `<xhtml:title>`. Doing a search for `<emp:title>` will always return 'Sir'. We can always tell which elements are XHTML elements, without having to know in advance the structure of our document.

By separating these elements using a prefix, we have effectively created two kinds of elements in our document—`emp` types of elements, and `xhtml` types of elements. So any element with the `emp` prefix belong in the same 'category' as each other, just as any elements with the xhtml prefix belong in another 'category'. These 'categories' are called 'namespaces.'

Unfortunately, there is a drawback to the prefix approach to namespaces used in the previous XML—who will monitor the prefixes? The whole reason for using them is to distinguish names from different document types, but if it is going to work, the prefixes themselves also have to be unique. If one company chose the prefix pers and another company also chose the same prefix, the original problem would still exist.

In fact, this prefix administration would have to work a lot like it works now for domain names on the Internet. A company or individual would go to the 'prefix administrators' with the prefix they would like to use; if that prefix wasn't already being used, they could use it, otherwise they would have to pick another one.

In order to solve this problem, we could take the advantage of the already unambiguous Internet domain names in existence, and specify that URIs must be used for the prefix names.

For example, if you work for a company called Kogent India Inc., which owns the domain name kogentindia.com, then you could incorporate that into your prefix. Perhaps the document might end up looking like this:

```
<?xml version="1.0" encoding="windows-1252 ISO-8859-1"?>

<{http://kogentindia.com/emp}employees>
  <{http://kogentindia.com/emp}name>
  <{http://kogentindia.com/emp}title>
  Sir
  </{http://kogentindia.com/emp}title>
  <{http://kogentindia.com/emp}first>
  Ambrish
  </{http://kogentindia.com/emp}first>
  ......
  ......
  ......

</{http://kogentindia.com/emp}employees>
```

Since the company owns the kogentindia.com domain name, we know that nobody else will be using the http://kogentindia.com/emp prefix in their XML documents, and if we want to create any additional document types, we can just keep using our domain name, and add the new namespace name to the end, such as http://kogentindia.com/other-namespace.

It is important to note that we need more than just the kogentindia.com part of the URI; we need the whole thing. Otherwise, there would be further problems—different people could have control of different sections on that domain, and they might all want to create namespaces. For example, the company's HR department could be in charge of http://kogentindia.com/hr, and might need to create a namespace for names (of employees), and the sales department could be in charge of http://kogentindia.com/sales, and need to create a namespace for names (of customers). As long as we're using the whole URI, we're fine; we can both create our namespaces (in this case http://kogentindia.com/hr/names and http://kogentindia.com/sales /names, respectively). We also need the protocol (http) there because there could be yet another department, which is in charge of, e.g. ftp://kogentindia.com/hr and ftp://kogentindia.com/sales.

The only drawback to this solution is that our XML is no longer well-formed. Our names can now include a myriad of characters that are allowed in URIs, but not in XML names, '/ characters' for example, and for the sake of this example we used { } character to separate the URL from the name, neither of which is allowed in XML element or attribute name.

To solve all of our namespace-related problems, we create a two-part names in XML. The first part would be the name we are giving this element, and the second part would be an arbitrarily chosen prefix that refers to a URI. This URI specifies the namespace which belongs to this element. And, in fact, this is what XML namespaces provide.

XML namespaces are used in XML documents by giving qualified names for the elements. These qualified names consist of two parts – the local part, which is the same as the names we are giving elements all along, and the namespace prefix, which specifies to which namespace this name belongs.

For example, to declare a namespace called `http://kogentindia.com/emp`, and associate a `<employees>` element with that namespace, we would do something like the following:

```
<emp:employees xmlns:emp=http://kogentindia.com/emp/>
```

The key is the `xmlns:emp` attribute (`xmlns` stands for XML namespace). Here, we are declaring the `emp` namespace prefix and the URI of the namespace, which it represents (`http://kogentindia.com/emp`). We can then use the namespace prefix with our elements, as in `emp:employees`. As opposed to our previous prefixed version, the prefix itself (emp) doesn't have any meaning. Its only purpose is to point to the namespace name. For this reason, we could replace our prefix (emp) with any other prefix, and this document would have exactly the same meaning.

This prefix can be used for any descendants of the `<emp:employees>` element to denote that they also belong to the `http://kogentindia.com/emp` namespace. For example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<emp:employees xmlns:emp="http://kogentindia.com/emp"
  xmlns:html="http://www.w3.org/1999/xhtml">
<emp:name>
  <emp:title>Sir</emp:title>
  <emp:first>Ambrish</emp:first>
  <emp:middle>Kumar</emp:middle>
  <emp:last>Singh</emp:last>
</emp:name>
<emp:position>Technical Writer</position>
<emp:resume>
<xhtml:html>
<xhtml:head><xhtml:title>Resume of Ambrish Kumar Singh</xhtml:title></xhtml:head>
<xhtml:body>
<xhtml:h1>Ambrish</xhtml:h1>
<xhtml:p>Ambrish is a Java Programmer</xhtml:p>
</xhtml:body>
</xhtml:html>
</emp:resume>
</emp:employees>
```

Here, we declare the emp prefix, which will be used to specify elements that belong to the emp namespace, and the `html` prefix, which will be used to specify elements that belong to the XHTML namespace.

Default Namespaces

Although the previous document solves all our namespace-related problems, it is just a little bit complex. We have to give every element in the document a prefix to specify which namespace this element belongs to that makes the document look similar to our first prefixed version. Luckily, we have the option of creating default namespaces and it looks like this:

```
<employees xmlns=" http://kogentindia.com/emp ">
<name>
  <title>Sir</title>
</name>
.....
....
</employees>
```

Notice that the `xmlns` attribute no longer specifies a prefix name to use for this namespace. As this is a default namespace, this element and any elements descended from it belong to this namespace, unless they explicitly specify another namespace. So the `<name>` and `<title>` elements both belong to this namespace.

You can declare more than one namespace for an element, but only one can be the default. This allows us to write XML like this:

```
<employees xmlns=" http://kogentindia.com/emp "
xmlns:html="http://www.w3.org/1999/xhtml">
<name>
  <title>Sir</title>
</name>
.....
...
<xhtml:p>Ambrish is a Java Programmer</xhtml:p>
....
</employees>
```

In this case, all of the elements belong to the `http://kogentindia.com/emp` namespace, except for the `<p>` element which is part of the `xhtml` namespace. We've declared the namespaces and their prefixes, if applicable, in the root element so that all the elements in the document can use these prefixes. However, we can't write XML like this:

```
<employees xmlns=" http://kogentindia.com/emp "
  xmlns="http://www.w3.org/1999/xhtml">
```

This tries to declare two default namespaces. In this case, the XML parser wouldn't be able to figure out what namespace the `<employees>` element belongs to.

Defining a Namespace in a DTD

In a DTD, you can specify that an element belongs to a specific namespace by adding an attribute to the element's definition, where the attribute name is `xmlns` ("xml namespace"). The following code snippet defines the namespace for the title element in a DTD:

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

The parser understands the xmlns attribute; therefore, declaring the attribute as FIXED makes the element title unique among elements of the same name defined in other DTDs.

Referencing a Namespace

You must provide reference to the namespace of an element when an element with the same name exists in multiple DTDs. You qualify a reference to an element name by specifying the `xmlns` attribute, as shown here:

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

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

XML CDATA

XML uses some characters so you cannot include these characters in your Parsed Character Data (PCDATA) because they are used in XML syntax – the < and & characters – like this here:

```
<!--This is not well-formed XML! -->
<compare>5 is < 7 & 7 > 5</compare>
```

This means that the XML parser comes across the < character, and expects a tag name, instead of a space. Then it assumes it as an element and searches for its closing tag, which it will not get and consequently give an error message. There are two ways you can get around this – escaping characters, or enclosing text in a CDATA section.

Escaping Characters

To escape these two characters, you simply replace any < characters with `&lt;` and any & characters with `&amp;`. The preceding <compare> element could be made well-formed by doing the following:

```
<compare>5 is &lt; 7 &amp; 7 &gt; 5 </compare>
```

Notice that IE 5 automatically un-escapes the characters for you when it displays the document. In other words, it replaces the `&lt;`, `&amp;` and `&gt;` strings with <, &, and, > characters. The strings `&lt;` and `&amp;` are known as entity references. The following entities are defined in XML:

❑ `&amp;`	-the & character
❑ `&lt;`	-the < character
❑ `&gt;`	-the > character
❑ `&apos;`	-the ' character
❑ `&quot;`	-the " character

Other characters can also be escaped by using character references. These are strings, such as `&#nnn;`, where nnn would be replaced by the Unicode number of the character you want to insert. We have already discussed this in the section "XML Elements".

CDATA Sections

What happens if your data contains many <, >, &, and "characters? In that case, if you use escaping character tricks then you may find that your document quickly becomes very complex and unreadable. To overcome this problem the other technique is used which is a CDATA section.

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

```
<compare><![CDATA[5 is < 7 & 7 > 5]]></compare>
```

Everything starting after the `<![CDATA[` and ending at the `]]>` is ignored by the parser, and passed through to the application as is. The only character sequence that can't occur within a CDATA section is `]]>`, since the XML parser would think that you were closing the CDATA section.

In these trivial cases, CDATA sections may look more confusing than the escaping did, but in other cases it can turn out to be more readable. Consider the following example that uses a CDATA section to keep an XML parser from parsing a section of JavaScript:

```
<script language='JavaScript'><![CDATA[
Function myFund() {
  If(0 < 1 && 1< 2)
  Alert("Correct");
}
]]></script>
```

XML Encoding

As we know that text is stored in computers using numbers, since numbers are all that computers really understand. You are already familiar with one character code, the American Standard Code for Information Interchange (ASCII). For example, in ASCII the character a is represented by the number 97, and the character A is represented by the number 65.

There are seven-bit and eight-bit ASCII encoding schemes. 7-bit ASCII uses 7 bits for each character, which limits it to 128 different values, while 8-bit ASCII uses one byte (8 bits) for each character, which limits it to 256 different values. 7-bit ASCII is a much more universal standard for text, while there are a number of 8-bit ASCII character codes, which were created to add additional characters not covered by ASCII, such as ISO-8859-1. Each 8-bit ASCII encoding scheme might have slightly different sets of characters represented, and those characters might map to a different number. However, the first 128 characters are always the same as the 7-bit ASCII character code.

ASCII can easily handle all the characters needed for English, which is why it is the predominant character encoding used on personal computers in the English-speaking world for many years. But there are more than 256 characters in all of the world's languages, so obviously ASCII (or any other 8-bit encoding limited to 256 characters) can only handle a small subset of these. This is why Unicode was invented.

Unicode

Unicode is a character code designed from the ground up with internationalization in mind, aiming to have enough possible characters to cover all the characters in any human language. There are two major character encodings for Unicode—UTF-16 and UTF-8. UTF stands for Universal Character Set Transformation Format, and the number 8 or 16 refers to the number of bits that the character is stored in. UTF-16 takes the easy way, and simply uses two bytes for every character (two bytes = 16 bits = 65,356 possible values).

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

Because of the work done with Unicode to make it international, the XML specification states that all XML processors must use Unicode internally. Unfortunately, very few of the documents in the world are encoded in Unicode. Most are encoded in ISO-8859-1, or windows-1252, or EBCDIC, or one of a large number of other character codes.

Unicode is managed and developed by a non-profit group called the Unicode Consortium. For more information on encoding and a listing of encoding types for XML, the Unicode consortium and the W3C has published a joint report, available at the Unicode Consortium site, `http://www.unicode.org/unicode/reports/tr20`.

Apart from UTF declarations for XML document encoding, any ISO registered `charset` name that is registered by the Internet Assigned Numbers Authority (IANA) is an acceptable substitute. For example, an XML 1.0 document encoded in Macedonian would look like this in XML declaration: `<?xml version="1.0" encoding="JUS_I.B1.003-mac"?>`

A list of currently registered names can be found at: `http://www.iana.org/assignments/character-sets`.

XML on the Server

You can store the XML file on the Internet server as you store HTML files. Write any XML file code and save it with `.xml` extension on the server. For an example, open the Notepad and type the following lines:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<Employee>
  <FirstName>Ambrish</FirstName>
  <MiddleName>Kumar</MiddleName>
  <LastName>Singh</LastName>
  <Age>25</Age>
</Employee>
```

Now save this file with the name `employee.xml` on the server.

XML Application

Let's now understand how we can load an XML file directly into an HTML page. This section also explains how you can display the data stored in an XML file in table format. The application works in a manner similar to the one given in section "XML Parser", except that here we are not using any parsing techniques. We simply load the XML file directly.

The code, given in Listing 27.5, shows the xmlparser.html file (you can find the xmlparser.html file in the `Code/XML/Chapter 27/XML` folder on the CD):

Listing 27.5: xmlparser.html

```
<html>
<head>
</head>
  <body>
        <xml src="emp.xml" id="empid" async="false"> </xml>
        <h1>Employee Details</h1>
```

```
            <table datasrc="#empid" width="100%" border="1">
                <thead>
                    <th>FirstName</th>
                    <th>MiddleName</th>
                    <th>LastName</th>
                    <th>Age</th>
                </thead>
                <tr align="left">
                    <td><span datafld="FirstName"></span></td>
                    <td><span datafld="MiddleName"></span></td>
                    <td><span datafld="LastName"></span></td>
                    <td><span datafld="Age"></span></td>
                </tr>
            </table>
            </p>
    </body>
  </html>
```

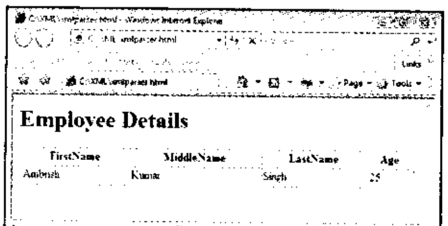
This file loads an XML file, `emp.xml`, by the code line:

```
  <xml src="emp.xml" id="empid" async="false"> </xml>
```

The `id` attributes here specifies the key for accessing this XML file. Now we require that the script will not start executing before the document is not properly loaded. This can be done by turning off the asynchronized loading. So we define `async="false"`. Then we access the XML elements by using the `datafld` attribute, as shown in following code snippet:

```
  <table datasrc="#empid" width="100%" border="1">
  <td><span datafld="FirstName"></span></td>
  <td><span datafld="MiddleName"></span></td>

  <td><span datafld="LastName"></span></td>
  <td><span datafld="Age"></span></td>
```

The XML file used for this application is the same as the one given in Listing 27.2. When you run the `xmlparser.html` file in the Internet Explorer browser, it displays the XML data in the form of a table, as shown in Figure 27.5:



Employee Details

FirstName	MiddleName	LastName	Age
Ambrish	Kumar	Singh	25

Figure 27.5: Showing XML Data in Tabular Form

Here's another HTML file, given in Listing 27.6, which simply displays the data of an XML file (you can find the xmlparser3.html file in the `Code/XML/Chapter 27/XML` folder on the CD):

Listing 27.6: xmlparser3.html

```
  <html>
  <body>
    <xml src="emp.xml" id="empid" async="false"></xml>
    <h3>Employee Details</h3>
    <br /><b>FirstName:</b>
    <span datasrc="#empid" datafld="FirstName"></span>
    <br /><b>MiddleName:</b>
    <span datasrc="#empid" datafld="MiddleName"></span>
    <br /><b>LastName:</b>
    <span datasrc="#empid" datafld="LastName"></span>
    <br /><b>Age:</b>
    <span datasrc="#empid" datafld="Age"></span>
  </body>
  </html>
```

This file simply loads an XML file in a similar manner as the previous one. Its output in the IE will look like the one shown in Figure 27.4.

XMLHttpRequest Object

The `XMLHttpRequest` object allows scripts to communicate with the server, outside the normal HTTP request-response scenario. The main purpose of the `XMLHttpRequest` object is to be able to use just HTML and script to connect directly with the data layer that is stored on the server. It allows you to bypass the need for the server-side scripting in many instances. The benefit of using the `XMLHttpRequest` is that you don't have to send the page or refresh it because changes to the underlying data are immediately reflected in the web page displayed by the browser.

Here we are going to discuss how you can create the `XMLHttpRequest` object in both FireFox and IE. In IE 7, Firefox, Safari, and Opera, the syntax needed to create an `XMLHttpRequest` object is as follows:

```
var xmlHttpRequest = new XMLHttpRequest();
```

In IE 5 and IE 6, the code looks like this:

```
var xmlHttpRequest = new ActiveXObject("Microsoft.XMLHTTP");
```

You can use the `XMLHttpRequest` in two ways—synchronously or asynchronously. When you use it synchronously, you first create an object, create a request, send it to the server, and then wait for a response. This is what you normally do in a traditional web model. But, if you are using it asynchronously, then it behaves in a different way. In this case, you must call the object using the `onreadystatechange` event. It happens in the following ways:

- First, you create an object and then set the `onreadystatechange` event to trigger a specific function.
- The function checks the `readyState` property.
- If the data is ready, it opens the request and sends it.
- After sending the request you can continue your processing. You need not wait for a response and you'll interrupt only when you get the response from the server.

XML Technologies

So far we discussed about XML and its components. Now we are going to discuss about the technologies related to XML. Some of the commonly used XML-related technologies are as follows:

- Extensible HTML (XHTML)
- DOM
- SAX
- XSLT
- Xpath

Now let's explore more about these technologies.

Extensible HTML (XHTML)

XHTML refers to the eXtensible HyperText Markup Language whose main focus of existence is to replace HTML. XHTML is the HTML defined in the form of the XML application. XHTML is very similar to HTML 4.01. From 26 January 2000, the W3C defined XHTML as the latest version of HTML. All browsers support XHTML. Although HTML is a markup language, we know that HTML documents are not well formed like the XML documents. XML is used to describe data, while HTML is used to display data. Therefore, when both HTML and XML are combined, we get a more powerful markup language—XHTML. XHTML documents must also follow some rules. These rules are as follows:

- Elements should be properly nested

- Elements should have end tag or closing tag
- Elements should be in lower-case
- Like XML, XHTML documents should have one root element

The following is the code snippet providing the minimum XHTML document template:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head>
<title>simple document</title>
</head>
<body>
<p>a simple paragraph</p>
</body>
</html>
```

As provided in the code snippet, the XHTML document comprises of three indispensable parts:

- DOCTYPE
- Head
- Body

The DOCTYPE declaration defines the document type as follows:

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

The rest of the document remains the same as the HTML document having the `<html> </html>` tag containing the `<title>`, `<head>`, and the `<body>` as the essentials and various other tags can be added as and when required for the document.

In other words, we can say that XHTML is a markup language used to define well-formatted documents with the help of XML and HTML.

Converting HTML to XML

While editing HTML code, it's very common to make mistakes. Now, with the help of HTML TIDY utility, you can fix up these mistakes automatically. Dave Raggett's HTML TIDY is a free utility, it also works very well on the complex to read markup generated by HTML editors and conversion tools. HTML TIDY also helps you to identify the areas on your Web page, which makes your pages more accessible to people with disabilities. In HTML TIDY utility each item found is properly organized. For example, items are listed with the line number and column so that you can see where the problem lies in your markup. The most common Web browsers such as Internet Explorer and Mozilla Firefox understand the following code snippet (HTML document), but a number of things prevent it from being well-formed XML:

```
<HTML>
<body>
<h1>Kogent</H1>
<p>This is the first line.
<P>This is the second line.<br>
This topic gives you the brief of global warming
<IMG SRC=image.jpg>
</BODY>
```

In the preceding code snippet:

- The html start-tag does not contain a corresponding end-tag.
- The tags enclosing the h1 and body elements are not in a consistent case.
- The value of the IMG element's SRC attribute isn't quoted.
- The br and IMG elements' tags have no closing tags.

HTML TIDY utility maps this code to:

```
<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/strict.dtd">
<html xmlns="http://www.w3.org/TR/xhtml1">
<head>
<title></title>
</head>
<body>
<h1>Kogent</h1>
<p>This is the first line.</p>
<P>This is the second line.</p><br/>
This topic gives you the brief of global warming
<IMG SRC="image.jpg">
</body>
<html
```

Let's move further with JAXP, which is Java API for XML processing.

Java API for XML Processing

Java API for XML Processing (JAXP) provides a high-level API for writing vendor neutral applications that process XML. JAXP is used to process XML data by using applications built on the Java platform. JAXP provides an extra layer of adaptor around the vendor-specific parser and transformer implementations. With JAXP API, you can choose either Simple API for XML Parsing (SAX) parser or Document Object Model (DOM) parser to `parse` an XML document using a stream of events or using DOM object representation.

JAXP also supports the Extensible Stylesheet Language Transformations (XSLT) standard, giving you control over the presentation of the data and enabling you to convert the data to other XML documents or other formats, such as HTML. JAXP also provides namespace support, allowing you to work with DTDs that might otherwise have naming conflicts. The Table 27.5 lists the description of the packages used in JAXP:

Table 27.5: Packages in JAXP

Package	Description
`javax.xml.transform`	Defines the `factory` class that you use to get a `Transformer` object. You then configure the transformer with input (source) and output (result) objects, and invoke its `transform()` method to transform the objects. The source and resultant objects are created using classes from one of the other three packages
`javax.xml.transform.dom`	Defines the `DOMSource` and `DOMResult` classes, which let you use a DOM as an input to or output from a transformation
`javax.xml.transform.sax`	Defines the `SAXSource` and `SAXResult` classes, which let you use a SAX event generator as input to a transformation, or deliver SAX events as output to a SAX event processor
`javax.xml.transform.stream`	Defines the `StreamSource` and `StreamResult` classes, which let you use an I/O stream as an input to or output from a transformation

JAXP allows the following sources for the XML document to be transformed by using XSL transformation:

- A DOM node
- A SAX XML reader or input source
- A file, input stream, or reader

The following implementations are provided for the sinks, to which the result tree of transformation is written:

- A DOM node
- A SAX content handler
- A file, output stream, or writer

The JAXP APIs

In Java, the `javax.xml.parsers` package contains the JAXP APIs. This package provides two vendor-neutral factory classes, SAXParserFactory and DocumentBuilderFactory, which are used to create instances of the SAXParser and DocumentBuilder classes.

The factory classes allow you to plug-in an XML implementation offered by another vendor, without changing your source code. The implementation you get depends on the setting of the properties of the javax.xml.parsers.SAXParserFactory and javax.xml.parsers.DocumentBuilderFactory classes.

An Overview of the Packages

The XML-DEV and World Wide Web Consortium (W3C) groups define SAX and DOM APIs, respectively. As discussed earlier, the javax.xml.parsers package provides common interface for SAX and DOM parsers. Specific classes and interfaces of DOM and SAX APIs are contained in the org.w3c.dom and org.xml.sax package respectively. Another package, javax.xml.transform, provides the classes of the XSLT API, which allows you transform XML into other forms.

`SAX` provides a mechanism to access XML documents in the event-driven, serial-access manner. The SAX API reads and writes XML data to a data repository or the Web.

The `DOM API` considers XML document as a tree structure of objects. You can use the `DOM API` to manipulate the hierarchy of objects of an XML application. The `DOM API` is mostly used for applications where the data needs to be retained, since DOM representation of the XML document resides in memory. This representation of XML document can be accessed and manipulated by the user. To create the DOM representation of an XML document, the DOM API first reads the entire `XML` structure and holds the object tree in memory.The `SAX API` does not require an in-memory representation of the data; and is therefore, preferred for use in server-side applications.

The `XSLT` API defined in the `javax.xml.transform` package helps you to transform the XML documents into other document formats. In addition, you can even use it in conjunction with the `SAX` APIs to convert legacy data to XML.

Document Object Model (DOM)

Document Object Model (DOM) presents the XML document as the tree-structure having the root node as the parent element and the elements, attributes, and text defined as the child nodes. So, XML DOM defines the standard way for accessing and manipulating XML documents. With the help of the DOM tree, the elements containing the text and the attributes can be manipulated and accessed. The contents of these elements can be modified, new elements can be created or the unwanted elements can be removed from the DOM tree. The most important thing to be noted is that all the elements, their text, and their attributes are known as the nodes.

In the DOM structure, the entire document is considered as the Document node. The XML tag or the XML element is recognized as the Element node. The text in XML elements is referred to as the Text node, the XML attributes are considered as the Attribute nodes and the comments are considered as the Comment node. In the DOM tree structure, the nodes have an hierarchical relationship with each other. The terms 'parent' and 'child' are used to describe the relationships between the nodes.

Let's consider an example of an XML file and look at its DOM tree-structure. Here's the code, given in Listing 27.7, for the `product.xml` file containing the data related to the various products (you can find the product.xml file in the `Code/XML/Chapter 27/XML` folder on the CD):

Listing 27.7: product.xml;

```
<? xml version="1.0" encoding="UTF-8"?>
<PRODUCTDATA>
  <PRODUCT PRODID="P001">
       <PRODUCTNAME>Barbie Doll</PRODUCTNAME>
       <DESCRIPTION>This is a toy for children in the age group below 5 years
       </DESCRIPTION>
       <PRICE>$24.00</PRICE>
```

```
        <QUANTITY>12</QUANTITY>
    </PRODUCT>
    <PRODUCT PRODID="P002">
        <PRODUCTNAME>Mini Bus</PRODUCTNAME>
        <DESCRIPTION>This is a toy for children in the age group of 5-10 years
        </DESCRIPTION>
        <PRICE>$42.00</PRICE>
        <QUANTITY>6</QUANTITY>
    </PRODUCT>
    <PRODUCT PRODID="P003">
        <PRODUCTNAME>Car</PRODUCTNAME>
        <DESCRIPTION>This is a toy for children in the age group of 10-15 years
        </DESCRIPTION>
        <PRICE>$60.00</PRICE>
        <QUANTITY>21</QUANTITY>
    </PRODUCT>
</PRODUCTDATA>
```

In Listing 27.7, the <PRODUCTDATA> is the root element of the document. Since all other elements are within the <PRODUCTDATA> element, it is considered as the root element. The root element has three <PRODUCT> nodes and an Attribute node named, PRODID. Each of the Element nodes has the Text node as well.

Figure 27.6 shows the DOM tree-structure for products.xml:

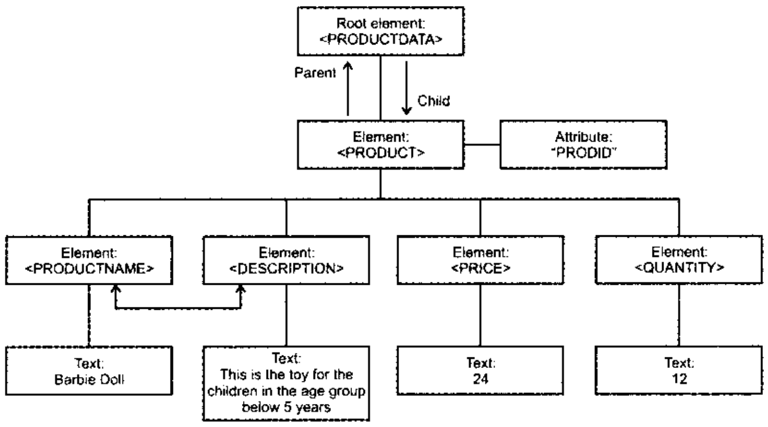


Figure 27.6: Displaying DOM Node Tree Structure

The DOM node tree structure, shown in Figure 27.6, has the root node <PRODUCTDATA> containing the child node named <PRODUCT>. The <PRODUCT> child node has four element nodes and an Attribute node. Each Element node has the respective Text node, as shown in the Figure 27.6.

The DOM implementation for Java is defined in the packages, which are listed in Table 27.6:

Table 27.6: DOM Packages

Package	Description
javax.xml.parsers	Contains the DocumentBuilderFactory and DocumentBuilder classes, which collectively return an object that implements the W3C Document interface. You can change the implementation of DocumentBuilderFactory by changing the System property either from the command-line or override it when invoking the new Instance method. This package also provides the ParserConfigurationException class to report errors
org.w3c.dom	Defines the DOM programming interfaces for XML (and, optionally, HTML) documents, as specified by the W3C

The javax.xml.parsers.DocumentBuilderFactory class provides the DocumentBuilder instance based on System property. The System property of this class chooses one factory implementation to generate the DocumentBuilder instance. You can then create a Document object either by invoking the new Document() method of DocumentBuilder instance or by invoking the parse methods of the DocumentBuilder instance. The Document object is internally represented as a DOM tree in the memory.

Other standards, such as JDOM and dom4j, can perform minute tasks used to parse XML documents. This is because these models consider each node in the hierarchy as an object. Usually, these models are used with XML documents that have elements that either contain text or element nodes. However, these models are not designed to handle XML documents that have mixed content. Mixed content means that an XML element contains both text and other sub elements.

The following code snippet shows the use of JDOM and dom4j standards:

```
<addressbook>
<entry>
<name>Sams</name>
<email>Sams@domain</email>
</entry>
...
</addressbook>
```

If you use the JDOM or `dom4j model API`, you need to invoke the `text()` method to get the XML document content after navigating to an element that contains text. If you use the DOM API during parsing, you must verify the list of sub-elements and the text of the node even if that list contains only one item (a `TEXT` node).

Let us modify the previous code snippet to contain mixed content:

```
<addressbook>
<entry>Sams
<email>Sams@domain</email>
</entry>
...
</addressbook>
```

In this code snippet, each <entry> tag contains text data and some other elements. Here, you first need to navigate to an entry and then invoke the `text()` method to find the parent element, and finally process the `<email>` sub-element. Let's move further and learn about using the DOM API.

Parsing XML Documents Using DOM

The code, given in Listing 27.8, shows how we are going to parse an XML file and create a word file from it (you can find the ParsingUsingDOM.java file in the `Code/XML/Chapter 27/XML` folder on the CD):

Listing 27.8: ParsingUsingDOM.java

```
import java.io.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
public class ParsingUsingDOM{
static public void main(String[] arg)throws
IOException,SAXException,TransformerException {
        BufferedReader bf = new BufferedReader(new
        InputStreamReader(System.in));
        System.out.print("Enter XML File Name: ");
        String xmlFile = bf.readLine();
        File file = new File(xmlFile);
        if(file.exists()){
try {
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.parse(xmlFile);
TransformerFactory tranFact = TransformerFactory.newInstance();
Transformer transfor = tranFact.newTransformer();
Node node =doc.getDocumentElement();
Source src = new DOMSource(node);
Result dest = new StreamResult(new File("ambrish.doc"));
transfor.transform(src, dest);
System.out.println("File successfully created!");
}
catch (ParserConfigurationException e) {
  System.err.println(e);
  System.exit(1);
  }
}
else{
  System.out.print("File not found!");
   }
}
}
```

The program prompts the user to enter an XML file name. Then the program parses it and creates a word file.

Tthe XML file, given in Listing 27.9, is used in this program (you can find the products.xml file in the `Code/XML/Chapter 27/XML` folder on the CD):

Listing 27.9: products.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<PRODUCTDATA>
  <PRODUCT>
        <PRODID>P001</PRODID>
        <PRODUCTNAME>Barbie Doll</PRODUCTNAME>
        <DESCRIPTION>This is a toy for children in the age group below 5
        years </DESCRIPTION>
        <PRICE>$24.00</PRICE>
        <QUANTITY>12</QUANTITY>
  </PRODUCT>
  <PRODUCT>
        <PRODID>P002</PRODID>
        <PRODUCTNAME>Mini Bus</PRODUCTNAME>
        <DESCRIPTION>This is a toy for children in the age group of 5-10
  years </DESCRIPTION>
        <PRICE>$42.00</PRICE>
        <QUANTITY>6</QUANTITY>
  </PRODUCT>
  <PRODUCT>
        <PRODID>P003</PRODID>
        <PRODUCTNAME>Car</PRODUCTNAME>
        <DESCRIPTION>This is a toy for children in the age group of 10-15
  years </DESCRIPTION>
        <PRICE>$60.00</PRICE>
        <QUANTITY>21</QUANTITY>
  </PRODUCT>
</PRODUCTDATA>
```

Let's start discussing how the application works. First we create a `BufferReader` object for taking the name of the XML file from the standard input device as keyboard. This is done as shown here:

```
BufferedReader bf = new BufferedReader(new InputStreamReader(System.in));
System.out.print("Enter XML File Name: ");
String xmlFile = bf.readLine();
File file = new File (xmlFile);
```

After this we use the DOM API for parsing XML documents. We create an instance of `DocumentBuilderFactory` class, which is provided by DOM API. The `DocumentBuilderFactory` object provides you a parser for parsing an XML document and produces a DOM tree structure. The `DocumentBuilderFactory` object calls its `newInstance` static method to obtain a reference to itself. This is done as follows:

```
DocumentBuilderFactory documentBuilderFactory =
DocumentBuilderFactory.newInstance();
```

The next step is to create an instance of `DocumentBuilder` class. This class parses the XML document. An instance of this class can be accessed by calling its `newDocumentBuilder` method on the `DocumentBuilderFactory` object:

```
DocumentBuilder documentBuilder = documentBuilderFactory.newDocumentBuilder();
```

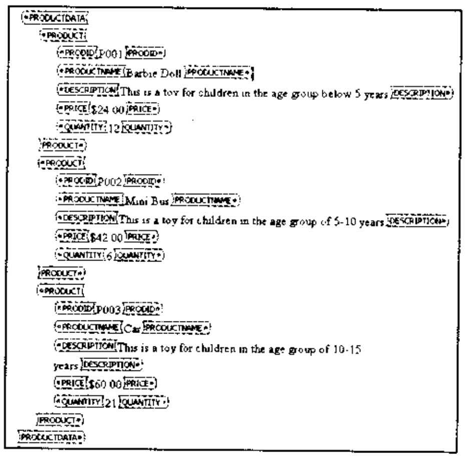
The next step obtains a DOM Document instance. The `Document` interface represents the whole XML document. It refers the document root and has several methods for creating nodes, attributes, etc. Document object is accessed by calling the parse method of `DocumentBuilder` class and passing the XML file to be parsed as argument. After the document is parsed, a Document object is returned to the caller. This is done as follows:

```
Document doc = builder.parse(xmlFile);
```

The next steps create an instance of `TransformerFactory` class provided by JAXP. This class is used for transforming a document to other formats. Here we are using it to create a word file. You can use this class as follows:

```
Transformer transfor = tranFact.newTransformer();
Node node =doc.getDocumentElement();
Source src = new DOMSource(node);
Result dest = new StreamResult(new File("ambrish.doc"));
transfor.transform(src, dest);
```

When you run this program it will parse the specified XML file and create a word file. The word file created by this program is shown in Figure 27.7:



```
PRODUCTDATA
    PRODUCT
        PRODID P001 PRODID
        PRODUCTNAME Barbie Doll PRODUCTNAME
        DESCRIPTION This is a toy for children in the age group below 5 years DESCRIPTION
        PRICE $24 00 PRICE
        QUANTITY 12 QUANTITY
    PRODUCT
    PRODUCT
        PRODID P002 PRODID
        PRODUCTNAME Mini Bus PRODUCTNAME
        DESCRIPTION This is a toy for children in the age group of 5-10 years DESCRIPTION
        PRICE $42 00 PRICE
        QUANTITY 6 QUANTITY
    PRODUCT
    PRODUCT
        PRODID P003 PRODID
        PRODUCTNAME Car PRODUCTNAME
        DESCRIPTION This is a toy for children in the age group of 10-15
        years DESCRIPTION
        PRICE $60 00 PRICE
        QUANTITY 21 QUANTITY
    PRODUCT
PRODUCTDATA
```

Figure 27.7: Document Showing XML file in Word Format

After discussing about the DOM APIs for XML parsing, let's move on to other APIs used for parsing XML documents, i.e. is SAX.

Simple API for XML (SAX)

The SAX refers to the Simple API for XML, which provides a mechanism to read data from an XML document. As discussed previously, DOM is also used to read data from an XML document. Thus, SAX parser proves to be an alternative to the Document Object Model (DOM). In technical terms, SAX is the serial access parser API for XML.

The SAX parser is the Event Driven parser. The user defines the number of callback methods which would be called when an event occurs during parsing. The following is the list of SAX events:

- XML Text nodes
- XML Element nodes
- XML processing instructions
- XML Comments

These events are fired at the start and end of each of these XML node, instruction or comments whenever they are encountered. For example, at the start and end of a Text node, the XML Text nodes event will be fired; or XML Comments event will be fired at the start and end of comments. The most important thing to note is that SAX parsing is unidirectional. Unidirectional means the previously parsed data cannot be read again, until the parsing operation is started again.

Following code snippet shows an example `prodetail.xml` that shows how parsing is done by the SAX parser:

```
<? Xml version="1.0" encoding="UTF-8"?>
<PRODUCTDATA>
  <PRODUCT PRODUCTID="P001">
       <PRODUCTNAME>Gone With the Wind</PRODUCTNAME>
       <DESCRIPTION> The backdrop of this book is the American Civil
  War</DESCRIPTION>
       <PRICE>25.00</PRICE>
       <QUANTITY>35</QUANTITY>
  </PRODUCT>
</PRODUCTDATA>
```

The `prodetail.xml` is the XML document providing the details of the products. The SAX parser fires off a series a XML events as it reads the `prodetail.xml` document. These events are fired in following sequence:

- XML Processing Instruction, with attributes version equal to '1.0'and encoding equal to 'UTF-8'.
- XML Element start named `PRODUCTDATA`.
- XML Element start named `PRODUCT`, with an attribute `PRODUCTID` equal to the value `P001`
- XML Element start named `PRODUCTNAME`.
- XML Text node with data equal to 'Gone with the Wind' (note that text processing with regard to spaces can be changed).
- XML Element end named `PRODUCTNAME`
- XML Element start named `DESCRIPTION`.
- XML Text node with data equal to 'The backdrop of this book is the American Civil War'.
- XML Element end named `DESCRIPTION`.
- XML Element start named `PRICE`.
- XML Text node with data equal to '25.00'.
- XML Element end named `PRICE`.
- XML Element start named `QUANTITY`.
- XML Text node with data equal to '35'.
- XML Element end named `QUANTITY`.
- XML Element end named `PRODUCT`.
- XML Element end named `PRODUCTDATA`.

In this way, the SAX Parser generates the sequence of events while parsing. DOM has the formal specification, but there is no formal specification for SAX. Since SAX is based on the nature of event-driven processing, the processing of the XML document would be done faster as compared to DOM-style parsers.

The quality of memory which a SAX parser uses in functioning is smaller as compared to that of the DOM parser. In other words, the DOM parser needs the entire DOM-tree structure of the document into the memory, so that the amount of memory used by the DOM parser depends upon the size of the input data.

The major drawback of the SAX parser is that, unlike XSLT or XPath, SAX parser cannot access any node at anytime. Instead, the SAX parser can read the parsed data again only when the parsing is done again.

The Table 27.7 lists a summary of the key SAX APIs:

Table 27.7: The SAX API

SAX API	Description
`ContentHandler`	This interface provides methods, such as `startDocument()`, `endDocument()`, `startElement()`, and `endElement()` which are invoked to receive the notifications of beginning, end of document and beginning and end of element respectively
`DefaultHandler`	This class is default class for SAX2 event handlers such as `ContentHandler`, `ErrorHandler`, `DTDHandler`, and `EntityResolver` interfaces
`DTDHandler`	The `DTDHandler` interface allows you to receive notifications of basic DTD related events
`EntityResolver`	The EntityResolver interface is a basic interface for resolving entities. It has `resolveEntity()` method which resolves external entities
`ErrorHandler`	This interface provides error(), `fatalError`, and warning methods that are invoked in response to occurrence of recoverable error, non-recoverable error, and warning respectively. To ensure the correct handling, you'll sometimes need to supply your own Error handler to the parser
`SAXParser`	The `SAXParser` class provides several types of `parse()` methods. In general, you pass an XML data source and a `DefaultHandler` object to these parse() methods , which process the XML and invokes the appropriate methods on the `handler` object
`SAXParserFactory`	A `SAXParserFactory` instance creates an instance of the SAXParser class based on System property of SAXParserFactory class
`SAXReader`	The `SAXParser` wraps a `SAXReader`. It is the `SAXReader` that carries on the conversation with the SAX event handlers you define

The SAX parser is defined in the packages listed in the Table 27.8:

Table 27.8: Packages of SAX API

Package	Description
`javax.xml.parsers`	This package provides `SAXParserFactory` class, which returns the `SAXParser instance` and exception classes for reporting errors
`org.xml.sax`	This package provides SAX interfaces
`org.xml.sax.ext`	This package provides SAX extensions that are used for doing more sophisticated SAX processing—for example, to process a Document Type Definition (DTD) for a file
`org.xml.sax.helpers`	This package provides the helper classes that make SAXParser instance easier to use SAX API

Let's take an example of how to parse XML document using SAX.

Parsing XML Documents Using SAX

Let's parse an XML file and then display its contents. The XML file used in this case is `employees.xml`. Here's the code, given in Listing 27.10, for ParsingUsingSAX program (you can find the files, named employees.xml and ParsingUsingSAX.java in the `Code/XML/Chapter 27/XML` folder on the CD):

Listing 27.10: ParsingUsingSAX.java

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
public class ParsingUsingSAX extends DefaultHandler
{
  List myEmpls;
  private String tempVal;
  private Employee tempEmp;
  public ParsingUsingSAX(){
      myEmpls = new ArrayList();
  }
  public void runExample() {
      parseDocument();
      printData();
  }
  private void parseDocument() {
      SAXParserFactory spf = SAXParserFactory.newInstance();
      try {

              SAXParser sp = spf.newSAXParser();
              sp.parse("employees.xml", this);

      }catch(SAXException se) {
              se.printStackTrace();
      }catch(ParserConfigurationException pce) {
              pce.printStackTrace();
      }catch (IOException ie) {
              ie.printStackTrace();
      }
  }

  private void printData()
  {
      System.out.println("No of Employees '" + myEmpls.size() + "'.");
      Iterator it = myEmpls.iterator();
      while(it.hasNext()) {
              System.out.println(it.next().toString());
      }
  }
  public void startElement(String uri, String localName, String qName,
  Attributes attributes) throws SAXException {
      tempVal = "";
      if(qName.equalsIgnoreCase("Employee")) {
              tempEmp = new Employee();
              tempEmp.setType(attributes.getValue("type"));
      }
  }
  public void characters(char[] ch, int start, int length) throws
  SAXException {
      tempVal = new String(ch,start,length);
  }
  public void endElement(String uri, String localName, String qName) throws
  SAXException {
      if(qName.equalsIgnoreCase("Employee")) {
```

```
                myEmpls.add(tempEmp);
            }else if (qName.equalsIgnoreCase("Name")) {
                tempEmp.setName(tempVal);
            }else if (qName.equalsIgnoreCase("Id")) {
                tempEmp.setId(Integer.parseInt(tempVal));
            }else if (qName.equalsIgnoreCase("Age")) {
                tempEmp.setAge(Integer.parseInt(tempVal));
            }
        }
        public static void main(String[] args){
            ParsingUsingSAX spe = new ParsingUsingSAX();
            spe.runExample();
        }
    }
```

Listing 27.10 also uses a Java Bean `employee.java` for storing the result. This file contains getter and setter methods for employee's property. These properties are specified in the XML file.

Here's the code, given in Listing 27.11, for `Employee.java` (you can find the Employee.java file in the `Code/XML/Chapter 27/XML` folder on the CD):

Listing 27.11: Employee.java

```
    public class Employee
    {
        private String name;
        private int age;
        private int id;
        private String type;
        public Employee(){
        }
        public Employee(String name, int id, int age,String type) {
            this.name = name;
            this.age = age;
            this.id  = id;
            this.type = type;
        }
        public int getAge() {
            return age;
        }
        public void setAge(int age) {
            this.age = age;
        }
        public int getId() {
            return id;
        }
        public void setId(int id) {
            this.id = id;
        }
        public String getName() {
            return name;
        }
        public void setName(String name) {
            this.name = name;
        }
        public String getType() {
            return type;
        }
        public void setType(String type) {
            this.type = type;
        }
    public String toString() {
            StringBuffer sb = new StringBuffer();
            sb.append("Employee Details - \n");
            sb.append("Name:" + getName());
            sb.append(", ");
            sb.append("Type:" + getType());
            sb.append(", ");
            sb.append("Id:" + getId());
            sb.append(", ");
            sb.append("Age:" + getAge());
            sb.append(".");
            return sb.toString();
    }
}
```

The XML file for employees.xml is given in Listing 27.12 (you can find the employees.xml file in the Code/XML/Chapter 27/XML folder on the CD):

Listing 27.12: employees.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Employees>
<Employee type="permanent">
   <Name>John</Name>
   <Id>3674</Id>
   <Age>34</Age>
</Employee>
<Employee type="contract">
   <Name>Jonny</Name>
   <Id>3675</Id>
   <Age>25</Age>
</Employee>
<Employee type="permanent">
   <Name>Abhishek</Name>
   <Id>3676</Id>
   <Age>28</Age>
</Employee>
</Employees>
```

Now set the xerces.jar and xml-apis.jar files in your classpath and compile all the preceding java files. After compilation of all these Java files, run the ParsingUsingSAX file. The detail of employees stored in the XML file gets displayed.

While parsing XML documents using SAX, we know that SAX provides several default event handlers. Therefore, we need to extend our class from the DafaultHandler to provide event handling code in our application:

```
public class ParsingUsingSAX extends DefaultHandler{
```

Now create a SAXParserFactory object by calling its static newInstance method. Next, create a SAX parser instance by calling newSAXParser method on the factory object. This is done in the following manner:

```
SAXParserFactory spf = SAXParserFactory.newInstance();
SAXParser sp = spf.newSAXParser();
```

Now pass the XML document to this parser instance. When the parsing starts, the parser calls the startElement method, whenever it encounters an element in the XML document. The method retrieves the namespaces—simple name, qualified names and list of attributes—as parameters as shown in the following code snippet:

```
public void startElement(String uri, String localName, String qName,
   Attributes attributes) throws SAXException {
         tempVal = "";
         if(qName.equalsIgnoreCase("Employee")) {
                tempEmp = new Employee();
```

```
                                tempEmp.setType(attributes.getValue("type"));
                    }
        }
```

Here, the qualified names are checked with `Employee`. If it matches, then the new `Employee` object is created and its type is set by the value of type attribute parsed from the XML.

The parser now calls the `endElement` method. This method is called whenever it encounters an element's end. The qualified names are again checked as shown here:

```
public void endElement(String uri, String localName, String qName) throws
    SAXException {
            if(qName.equalsIgnoreCase("Employee")) {
                    myEmpls.add(tempEmp);

            }else if (qName.equalsIgnoreCase("Name")) {
                    tempEmp.setName(tempVal);
            }else if (qName.equalsIgnoreCase("Id")) {
                    tempEmp.setId(Integer.parseInt(tempVal));

            }else if (qName.equalsIgnoreCase("Age")) {
                    tempEmp.setAge(Integer.parseInt(tempVal));
            }
}
```

Now consider the following function:

```
public void characters(char[] ch, int start, int length) throws
    SAXException {
            tempVal = new String(ch,start,length);
}
```

The characters method is an event handler and it is called by the parser whenever it encounters an element data. Because this data is in character format, the method accepts character array, offset, and length as its parameters.

Extensible Stylesheet Language Transformation (XSLT)

XSLT stands for extensible Stylesheet Language Transformations and it is used for transforming the structure and content of XML document into the required output. XSLT is used to transform XML documents into other XML documents. XSLT processors parse the input XML document, as well as the XSLT stylesheet, and then process the instructions found in the XSLT stylesheet, using the elements from the input XML document. During the processing of the XSLT instructions, a structured XML output is created. XSLT instructions are in the form of XML elements, and use XML attributes to access and process the content of the elements in the XML input document.

The main purpose of using XSLT is to convert the XML data into human readable format. It means that XSLT is used for displaying XML data in other formats, such as HTML, PDF, etc. The XSLT includes two steps for transforming an XML document into the required outputs. These steps are as follows:

- Data is converted from the structure of an XML document to the desired output structure.
- The new structure is displayed in the required format, such as HTML, PDF, etc.

XSLT is rarely discussed without a reference to XPath. XPath is a separate recommendation from the W3C that uses a simple path language to address parts of an XML document. Although XPath is used by other W3C recommendations, there is hardly a use for XSLT that does not involve XPath. Generally speaking, XSLT provides a series of operations and manipulators, while XPath provides precision of selection and addressing.

The XSLT APIs for Java are defined in the packages listed in Table 27.9:

Table 27.9: Packages of the XSLT API

Package	Description
`javax.xml.transform`	It defines the `TransformerFactory` and `Transformer` classes, which you use to get an object required for transformations. After creating a `transformer` object, you invoke its `transform()` method, passing an input (source) and

Table 27.9: Packages of the XSLT API

Package	Description
	output (result) s parameter
`javax.xml.transform.dom`	These are classes to create input (source) and output (result) objects from a DOM
`javax.xml.transform.sax`	These are classes to create input (source) objects from a SAX parser and output (result) objects from a SAX event handler
`javax.xml.transform.stream`	These are classes to create input (source) objects and output (result) objects from an I/O stream

Let's now get ready to take a look at an example of using XSLT to transform a very simple XML document.

The code for this example is given in Listing 27.13, (you can find the product.xml file in the `Code/XML/Chapter 27/XSLT` folder on the CD:

Listing 27.13: product.xml

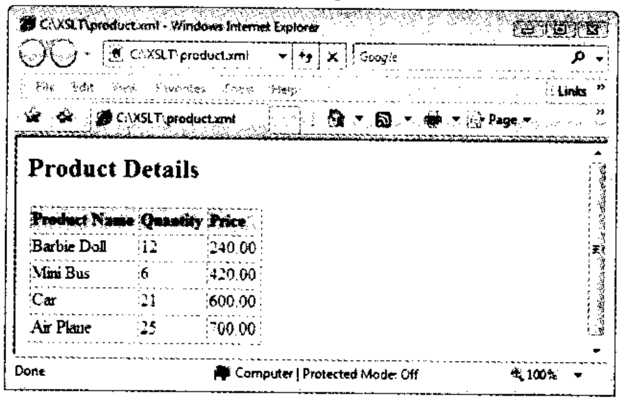
```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="product.xsl"?>
<PRODUCTDATA>
 <PRODUCT>
      <PRODID id="P001" ></PRODID>
      <PRODUCTNAME>Barbie Doll</PRODUCTNAME>
      <DESCRIPTION>This is a toy for children in the age group below 5
 years </DESCRIPTION>
      <PRICE>240.00</PRICE>
      <QUANTITY>12</QUANTITY>
 </PRODUCT>

 <PRODUCT>
      <PRODID id="P002" ></PRODID>
      <PRODUCTNAME>Mini Bus</PRODUCTNAME>
      <DESCRIPTION>This is a toy for children in the age group of 5-10
 years </DESCRIPTION>
      <PRICE>420.00</PRICE>
      <QUANTITY>6</QUANTITY>
 </PRODUCT>

 <PRODUCT>
      <PRODID id="P003" ></PRODID>
      <PRODUCTNAME>Car</PRODUCTNAME>
      <DESCRIPTION>This is a toy for children in the age group of 10-15
 years </DESCRIPTION>
      <PRICE>600.00</PRICE>
      <QUANTITY>21</QUANTITY>
 </PRODUCT>

 <PRODUCT>
      <PRODID id="P004" ></PRODID>
      <PRODUCTNAME>Air Plane</PRODUCTNAME>
      <DESCRIPTION>This is a toy for children in the age group of 08-15
       years </DESCRIPTION>
      <PRICE>700.00</PRICE>
      <QUANTITY>25</QUANTITY>
 </PRODUCT>
 </PRODUCTDATA>
```

The code given in Listing 27.14 shows the XSLT stylesheet used in this example to perform transformation (you can find the product.xsl file in the `Code/XML/Chapter 27/XSLT` folder on the CD):

Listing 27.14: product.xsl

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<body>
<h2>Product Details</h2>
  <table border="1">
  <tr bgcolor="#CCCFFF">
  <th align="left">Product Name</th>
  <th align="left">Quantity</th>
  <th align="left">Price</th>
  </tr>
  <xsl:for-each select="PRODUCTDATA/PRODUCT">
  <tr>
  <td><xsl:value-of select="PRODUCTNAME"/></td>
  <td><xsl:value-of select="QUANTITY"/></td>
  <td><xsl:value-of select="PRICE"/></td>
  </tr>
  </xsl:for-each>
  </table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

When you open the `product.xml` file in the Internet Explorer, it looks like the one shown in Figure 27.8:


C:\XSLT\product.xml - Windows Internet Explorer
C:\XSLT\product.xml
Google
File Edit View Favorites Tools Help
Links
C:\XSLT\product.xml
Page
Done
Computer | Protected Mode: Off
100%

Product Details

Product Name	Quantity	Price
Barbie Doll	12	240.00
Mini Bus	6	420.00
Car	21	600.00
Air Plane	25	700.00

Figure 27.8: The product.xml Displayed in IE

In this example when you load the `XML` document (`product.xml`) in your browser, the browser will recognize the following line:

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

This line simply tells the browser that this `XML` file is using an `XSLT` stylesheet named `product.xsl`. Let's now discuss more about `XSLT` stylesheet.

XSLT Stylesheet

An XSL Stylesheet is also an `XML` document (Listing 27.14 shows you an example of simple XSL Stylesheet). An XSL Stylesheet contains many `XSLT` elements and `XSLT` functions. An XSL Stylesheet begins with either the `stylesheet` elements or with `transform`. Both these elements do the same thing. The most important element in the XSL Stylesheet is the `template` element. Let's move ahead to discuss more about XSL elements and functions, which are generally used.

The XSLT elements define the structure of stylesheet. The most commonly used XSLT elements are as follows:

- <xsl:template>
- <xsl:apply-templates>
- <xsl:import>
- <xsl:apply-imports>
- <xsl:call-template>
- <xsl:stylesheet> or <xsl:transform>
- <xsl:include>
- <xsl:element>
- <xsl:attribute>
- <xsl:attribute-set>
- <xsl:value-of>

The <xsl:template> Element

This element defines a template for producing output. It is a top-level element. It contains rules to apply when a specified node is matched against a pattern or explicitly by name. It uses a match attribute for defining the pattern. Matching against a pattern makes use of the match attribute of the `<xsl:template>` element. For example, match ="/" defines the whole document as it matches the root element.

The pattern mentioned in the match attribute may not include a variable reference. Therefore, circular references are avoided. Similarly, if the `<xsl:template>` element has a name attribute like the one that follows:

```
<xsl:template name="PRODUCTNAME" >
```

then it can be accessed in the following manner:

```
<xsl:call-template name="PRODUCTNAME" >
```

Notice that the `name` attribute of both `<xsl:template>` and `<xsl:call-template>` must match exactly. An `<xsl:template>` element must have either a `match` attribute or a `name` attribute or both. When both the `match` and the `name` attributes exist, then, in that case, `<xsl:template>` element can be called by either `<xsl:apply-templates>` or `<xsl:call-template>`.

Syntax: <xsl:template>

The syntax of `<xsl:template>` is as follows:

```
<xsl:template name="name" match="pattern" mode="mode" priority="number" >
<! -- --- Content: (xsl:param*, template) -- --- >
</xsl:template>
```

Attributes

The attributes used by `<xsl:template>` element are listed in Table 27.10.

Table 27.10: Attributes of <xsl:template>

Name	Value	Meaning
match (optional)	Pattern	It is a pattern that defines which nodes are eligible to be processed. If this attribute is absent, then the `name` attribute must be present
name (optional)	Name	It specifies a name for the template. If this attribute is absent, then there must be a `match` attribute
priority (optional)	Number	It is a number (positive or negative, integer or decimal) that denotes the priority of this template, and is used when several templates match the same node
mode (optional)	List of mode names	It specifies the mode or modes to which this template rule applies. When `<xsl:apply-templates>` is used to process a set of nodes, the only templates considered are those with a matching mode

The <xsl:apply-templates> Element

This element defines a set of nodes to be processed and causes the system to process them by selecting an appropriate template rule for each one. The `<xsl:apply-templates>` element is an instruction, which is used within a template. It selects a set of nodes and processes each of them by finding a matching template. An `<xsl:sort>` element may be nested within an `<xsl:apply-templates>` element. If `<xsl:sort>` element is nested within `<xsl:apply-templates>`, then it determines the order in which the nodes are processed, otherwise the nodes are processed in document order.

Syntax: <xsl:apply-templates>

The syntax of `<xsl:apply-templates>` is as follows:

```
<xsl:apply-templates  select="expression" mode="mode" >
<!-- --- Content: (xsl:sort|xsl:with-param*) -- --- >
</xsl:apply-templates>
```

Attributes

The attribute used by the `<xsl:apply-templates>` element is listed in Table 27.11.

Table 27.11: Attributes of <xsl:apply-templates>

Name	Value	Meaning
select (optional)	expression	It specifies the nodes to be processed. If omitted, all the children of the current node are processed
mode (optional)	name	Template rules used to process the selected nodes must have a matching mode. If omitted, the default (unnamed) mode is used

Using <xsl:template> and <xsl:apply-templates>

In this example, we'll give you an idea about how you can use these two elements—`<xsl:template>` and `<xsl:apply-templates>`—in your applications. Let's consider the XML document given in Listing 27.13.

The code, given in Listing 27.15 shows how the XSLT stylesheet is applied to perform transformation (you can find the xslapplytemplates.xsl file on the CD in the `Code/XML/Chapter 27/XSLT` folder on the CD):

Listing 27.15: xslapplytemplates.xsl

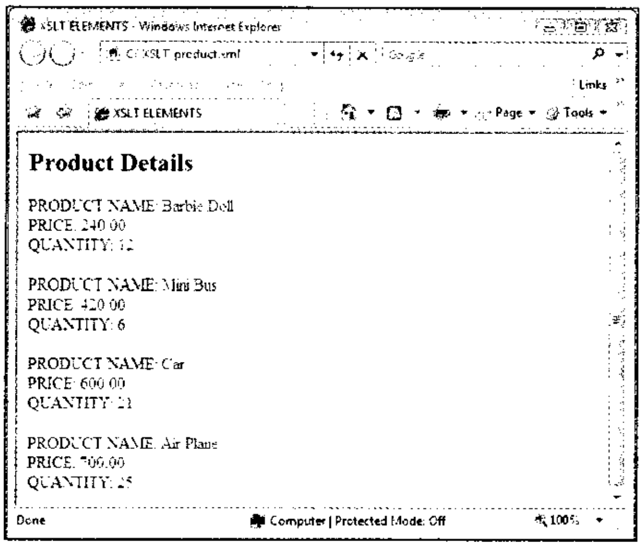
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<title>XSLT ELEMENTS</title>
<body>
<h2>Product Details</h2>
<xsl:apply-templates/>
</body>
</html>
</xsl:template>

<xsl:template match="PRODUCT">
<p>
  <xsl:apply-templates select="PRODUCTNAME"/>
  <xsl:apply-templates select="PRICE"/>
  <xsl:apply-templates select="QUANTITY"/>
</p>
</xsl:template>

<xsl:template match="PRODUCTNAME">
PRODUCT NAME: <span style="color:#ff0000">
```

```
<xsl:value-of select="."/></span>
<br />
</xsl:template>

<xsl:template match="PRICE">
PRICE: <span style="color:#0000FF">
<xsl:value-of select="."/></span>
<br />
</xsl:template>

<xsl:template match="QUANTITY">
QUANTITY: <span style="color:#006600">
<xsl:value-of select="."/></span>
<br />
</xsl:template>
</xsl:stylesheet>
```

You have to change the `<xml-stylesheet>` in the product.xml file so that it points to this `XSLT` stylesheet, as shown in the following code line:

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

In Listing 27.15, the `<xsl:template>` element matches the root element and calls the `<xsl:apply-templates>` element for each node set. The `<xsl:apply-templates>` selects the 'PRODUCTNAME', 'PRICE', and 'QUANTITY' for the matching 'PRODUCT' node specified in `<xsl:template>` element. When you open the `product.xml` file in IE, it will be transformed in HTML, as shown in Figure 27.9:


XSLT ELEMENTS - Windows Internet Explorer
C:\XSLT product.xml
Google
Links
XSLT ELEMENTS
Page
Tools
Done
Computer | Protected Mode: Off
100%

Product Details

PRODUCT NAME: Barbie Doll
PRICE: 240.00
QUANTITY: 12

PRODUCT NAME: Mini Bus
PRICE: 420.00
QUANTITY: 6

PRODUCT NAME: Car
PRICE: 600.00
QUANTITY: 21

PRODUCT NAME: Air Plane
PRICE: 700.00
QUANTITY: 25

Figure 27.9: Using <xsl:template> and <xsl:apply-templates>

The <xsl:import> Element

This element is the top-level element used to import the contents of one stylesheet into another stylesheet. The importing stylesheet has higher precedence than the imported stylesheet. It means that the `XML` document is first transformed using the importing stylesheet, but if importing stylesheet is unable to perform a transformation then the imported stylesheet will perform the transformation, if it can. The `<xsl:import>` element is the first element among the top-level elements. If it is present in a stylesheet, then it needs to proceed first from any other top-level element. The `<xsl:import>` element causes the stylesheet at the location specified in its `href` attribute to be imported. Thus, to import the `InsertStylesheet.xsl` stylesheet we could use the `<xsl:import>` like this (assuming that `InsertStylesheet.xsl` was situated in the same directory as the importing stylesheet):

```
<xsl:import href="InsertStylesheet.xsl"/>
```

The value of the `href` attribute may be a relative URI, as shown in the previous example, or an absolute URI. If the URI is relative then it is interpreted in the light of the base URI of the importing stylesheet. The file reference in the value of the `href` attribute must be a valid `XSLT` stylesheet. The top-level `<xsl:stylesheet>` element of the imported stylesheet is, in effect, discarded and the top-level children of the discarded element are inserted into the importing stylesheet in place of the `<xsl:import>` element.

Syntax: <xsl:import>

The syntax of `<xsl:import>` is as follows:

```
<xsl:import href="URI"/ >
```

Attributes

Table 27.12 lists the attribute used by this element:

Table 27.12: Attribute of <xsl:import>

Name	Value	Meaning
`href`	URI	It specifies the URI of the imported stylesheet

The <xsl:call-template> Element

This element calls a named template. We mentioned earlier that the `<xsl:template>` element may have a `name` attribute and that such a template can be called by name. The `<xsl:call-template>` element is used to do that. The `<xsl:call-template>` element has only one attribute, the `name` attribute, which is a required attribute. The `name` attribute of the `<xsl:call-template>` element and the `name` attribute of the `<xsl:template>` element must match. An `<xsl:call-template>` element may have one or more nested `<xsl:with-param>` elements.

If you have a template like this:

```
<xsl:template name="CalledTemplate">
<!-- The works of the template goes here. -->
</xsl:template>
```

then you can call that template using the following code:

```
<xsl:call-template name="CalledTemplate"/>
```

Or, if you also wish to pass a parameter, it would take this general form:

```
<xsl:call-template name="CalledTemplate">
<xsl:with-param name="ID" select="StudentID"/>
</xsl:call-template>
```

Since the parameter is passed to be evaluated, there would need to be a corresponding `<xsl:param>` element within the `<xsl:template>` element, like this:

```
<xsl:template name="CalledTemplate">
<xsl:param name="ID"/>
<!--The rest of the works of the template goes here. -->
</xsl:template>
```

The `<xsl:call-template>` element can be used recursively to process a list, either a node set or a list of separated strings.

Syntax: <xsl:call-template>

The syntax of `<xsl:call-template>` is as follows:

```
<xsl:call-template name ="templatename" >
<!-- --- Content: xsl:param* -- --- >
</xsl:call-template>
```

Attributes

Table 27.13 lists the attribute used by this element.

Table 27.13: Attribute of <xsl:call-template>

Name	Value	Meaning
name	templatename	It specifies the name of the template to be called

Using <xsl:call-template>

For an example, let's consider the XML document given in Listing 27.13. Here's the code, given in Listing 27.16, for applying the XSLT stylesheet to perform transformation (you can find the xslcalltemplates.xsl file in the Code/XML/Chapter 27/XSLT folder on the CD):

Listing 27.16: xslcalltemplates.xsl

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">

<html>
<title>XSLT ELEMENTS</title>
<body>
<h2>Product Details</h2>

<xsl:call-template name="PRODUCT"/>
</body>
</html>
</xsl:template>
<xsl:template name="PRODUCT">
<table border="1">
  <tr bgcolor="#CCCFFF">
  <th align="left">Product Name</th>
  <th align="left">Quantity</th>
  <th align="left">Price</th>
    </tr>
    <xsl:for-each select="PRODUCTDATA/PRODUCT">
    <tr>
  <td><xsl:value-of select="PRODUCTNAME"/></td>
  <td><xsl:value-of select="QUANTITY"/></td>
  <td><xsl:value-of select="PRICE"/></td></tr>
  </xsl:for-each>
</table>
</xsl:template>
</xsl:stylesheet>
```

You have to change <xml-stylesheet> in product.xml file so that it points to this XSLT stylesheet, as shown in the following code:

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

In this stylesheet, the <xsl:template> element matches the root element and calls the <xsl:call-template> element with template name 'PRODUCT' for each node set. The <xsl:call-template> calls the template 'PRODUCT' that selects the 'PRODUCTNAME', 'PRICE', and 'QUANTITY' for the matching 'PRODUCT' node specified in <xsl:for-each> element. When you open the XML file in IE, it will be transformed into HTML as shown in Figure 27. 8.

The <xsl:stylesheet> and <xsl:transform> Elements

These elements are the outermost elements of the stylesheet. Both of them are used to define the root element of the stylesheet.

Syntax: <xsl:stylesheet>

The syntax of `<xsl:stylesheet>` is as follows:

```
<xsl:   stylesheet   id   ="xmlname"   version   ="number"   extension-element-prefixes
   = "namespacelists exclude-result-prefixes = "namespacelist" >
<! -- --- Content: xsl:param* -- --- >
</xsl:stylesheet>
```

Syntax: <xsl:transform>

The syntax of `<xsl:transform>` is as follows:

```
<xsl:   transform   id   ="xmlname"   version   ="number"   extension-element-prefixes
   = "namespacelists exclude-result-prefixes = "namespacelist" >
<! -- --- Content: xsl:param* -- --- >
</xsl: transform >
```

Attributes

The attributes used by these elements are same and are listed in Table 27.14:

Table 27.14: Attributes of <xsl:stylesheet> or <xsl:transform>

Name	Value	Meaning
`id` (optional)	XML Name	It is an identifier used to identify this `<xsl:stylesheet>` element when it is embedded in another XML document
`version` (mandatory)	Number	It defines the version of XSLT required by this stylesheet. Use 2.0 for a stylesheet that requires XSLT 2.0 features or 1.0, if you want the stylesheet to be portable between XSLT 1.0 and XSLT 2.0 processors
`extension-element-prefixes` (optional)	Whitespace-separated list of Namespaces	It defines any namespaces used in this stylesheet to identify extension elements
`exclude-result-prefixes` (optional)	Whitespace-separated list of Namespaces	It defines namespaces used in this stylesheet that should not be copied to the output destination, unless they are actually used in the result document

The <xsl:include> Element

This element is a top level element, i.e. an element that comes in the starting of XML stylesheet, which is used for including the contents of one stylesheet within another. The `<xsl:include>` element has only one attribute, the `href` attribute, which is a required attribute. Thus, if we want to include a module called `Included.xsl` then we can do so by using the following code, provided the `ModuleToBeIncluded.xsl` is in the same directory as the stylesheet within which the `<xsl:include>` element exists:

```
<xsl:include href="Included.xsl"/>
```

The difference between `<xsl:import>` and `<xsl:include>` is that, with `<xsl:include>`, the semantics of the included stylesheet are not changed by the process, whereas an imported module will be wholly or partly overridden by similarly named templates in the importing stylesheet, if such templates are present.

Syntax: <xsl:include>

The syntax of `<xsl:include>` is as follows:

```
<xsl:include  href="URI"/ >
```

Attributes

Table 27.15 lists the attribute used by this element:

Table 27.15: Attribute of <xsl:include>

Name	Value	Meaning
href	URI	It specifies the URI of the included stylesheet

The <xsl:element> Element

This element is used to create an Element node. The name of the element that is created is determined by its name and namespace attributes. The name attribute gives the qualified name for the element and the namespace attribute gives the namespace URI for the element. The purpose of the <xsl:element> element is to cause an Element node to be created in the result tree.

There are several techniques available, which can be used to add attributes to such a new Element node. For example, the use-attribute-sets attribute of the <xsl:element> element itself can be used for this purpose. Alternatively, you can use the <xsl:attribute>, <xsl:copy>, or an <xsl:copy-of> element to add attribute to a new Element node.

Syntax: <xsl:element>

The syntax of <xsl:element> is as follows:

```
<xsl:element name = "name" namespace ="URI" use-attribute-sets = "namelist" >
< ! -- -- Content:template -- -- >
</xsl:element>
```

Attributes

Table 27.16 lists the attributes used by <xsl:element> element:

Table 27.16: Attributes of <xsl:element>

Name	Value	Meaning
name (mandatory)	Name	It is the name of the element to be generated
namespace (optional)	URI of namespace	It is the namespace URI of the generated element
use-attribute-sets (optional)	Whitespace-separated list of attribute-sets	It is the list of named attribute sets containing attributes to be added to this output element

Using <xsl:element> Element

A typical use of the <xsl:element> element would be in an XML to XML transformation. It is used to create an element in the output document for an attribute in the source document. Here's the code, given in Listing 27.17, that shows the XSLT stylesheet, if we consider the source document given in Listing 27.13 (you can find the xslelement.xsl file in the Code/XML/Chapter 27/XSLT folder on the CD):

Listing 27.17: xslelement.xsl

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" indent="yes"/>
<xsl:template match="/">
<PRODUCTDATA>
<PRODUCT>
<xsl:for-each select="PRODUCTDATA/PRODUCT/PRODID/@*">
<xsl:element name="{name()}">
<xsl:value-of select="."/>
</xsl:element>
</xsl:for-each>
</PRODUCT>
</PRODUCTDATA>
</xsl:template>
</xsl:stylesheet>
```

You have to change the `<xml-stylesheet>` in the product.xml file so that it points to this XSLT stylesheet, as follows:

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

Listing 27.17 will produce a restructured XML document with elements in the output replacing attributes in the source document:

```
<?xml version="1.0" encoding="UTF-8"?>
<PRODUCTDATA>
<PRODUCT>
   <id>P001</id>
   <id>P002</id>
   <id>P003</id>
   <id>P004</id>
</PRODUCT>
</PRODUCTDATA>
```

Basically, what the template does is that it takes each attribute on the `<PRODID>` element in the source document and creates a correspondingly named element in the output document, while inserting the value of the former attribute as the content of the newly created element.

The <xsl:attribute> Element

The `<xsl:attribute>` element is used to create attribute node and add it to the element.

Syntax: <xsl:attribute>

The syntax of `<xsl:attribute>` is as follows:

```
<xsl:attribute name = "nameofattribute" namespace ="URI" >
< ! -- -- Content:template -- -- >
</xsl:attribute>
```

Attributes

Table 27.17 lists the attributes used by this element:

Table 27.17: Attributes of <xsl:attribute>

Name	Value	Meaning
`name (mandatory)`	Name	It is the name of the attribute to be generated
`namespace (optional)`	URI of namespace	It is the namespace URI of the generated attribute

The <xsl:attribute-set> Element

This element is one of the starting elements of an XSL stylesheet and is used to define a named set of attributes names and values. The resulting attribute can be applied as a whole to any output element.

Syntax: <xsl: attribute-set>

The syntax of `<xsl: attribute-set>` is as follows:

```
<xsl:attribut-set name = "name" use-attribute-sets = "namelist" >
< ! -- -- Content:template -- -- >
</xsl:attribute-set>
```

Attributes

Table 27.18 lists the attributes used by this element:

Table 27.18: Attributes of <xsl:attribute-set>

Name	Value	Meaning
`name (mandatory)`	Name	It is the name of the element to be generated
`use-attribute-sets (optional)`	Whitespace-separated list of	It is the list of named attribute sets containing

Table 27.18: Attributes of <xsl:attribute-set>

Name	Value	Meaning
	attribute-sets	attributes to be added to this output element

The <xsl:value-of> Element

This element is mostly used for writing text to a result tree. It is used for constructing a Text node and extracting the value of a node.

Syntax: <xsl: value-of>

The syntax of `<xsl: value-of>` is as follows:

```
<xsl:value-of
select = "expression"
disable-output-escaping = "yes|no"/ >
```

Attributes

Table 27.19 lists the attributes used by this element:

Table 27.19: Attributes of <xsl:value-of>

Name	Value	Meaning
`select` (required)	expression	It is the expression given in XPath expression language form that is used to specify which nodes to extract the value from
`disable-output-escaping` (optional)	Yes No	The 'Yes' value specifies that the special characters (such as <) will be displayed as it is. The 'No' value specifies that the special characters (such as <) will be displayed as `&lt;` default value is No

XSLT Functions

There are over 100 built-in functions in XSLT. These functions also include XPath functions. However, in case of XSLT, there are the following built-in functions:

- `current()` —This function is used for accessing the current node.
- `document()` —This function is used for accessing external XML document.
- `element-available()` —This function checks whether a particular element is present under the XSLT Elements.
- `format-number()` —This function formats the specified number in a specified format.
- `function-available()` —This function checks whether a particular function is present under the XSLT functions or not.
- `generate-id()` —This function generates a string that uniquely identifies a node.
- `key()` —This function is used to find the nodes with a given value for a named key.
- `system-property()` —This function returns the information about the processing environment, like XSLT version, XSLT Processor's vendor name, and their URI.

Using XSLT Elements and Functions

Let's build a small application using XSLT functions. In this application, we are using two XML files—`product.xml` file (given in Listing 27.13) and `emp.xml`. The `emp.xml` file is as shown here:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<Employee>
  <FirstName>Ambrish</FirstName>
  <MiddleName>Kumar</MiddleName>
  <LastName>Singh</LastName>
```

```
        <Age>25</Age>
    </Employee>
```

The XSLT stylesheet used for this application is divided into three files. The first stylesheet explains the following functions:

- current()
- document()
- function-available()
- element-available()
- format-number()

Tthe code for products2.xsl is given in Listing 27.18, (you can find the product2.xsl file in the Code/XML/Chapter 27/XSLT folder on the CD):

Listing 27.18: products2.xsl

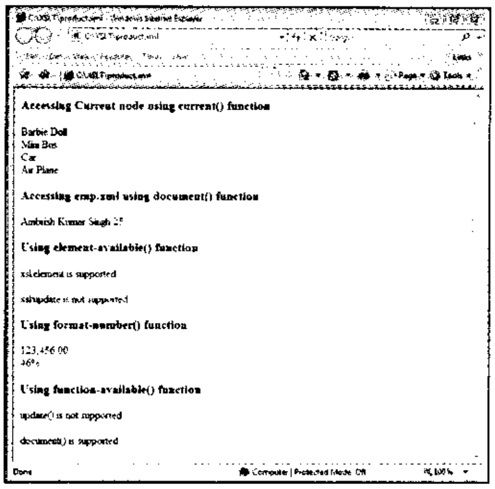
```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:key name="productlist" match="PRODUCT" use="PRODUCTNAME" />
<xsl:template match="/">
<html>
<body>
<h3>Accessing Current node using current() function</h3>
<xsl:for-each select="PRODUCTDATA/PRODUCT/PRODUCTNAME">
<xsl:value-of select="current()" /><br/>
</xsl:for-each>
<h3>Accessing emp.xml using document() function</h3>
<xsl:value-of select="document('emp.xml')"/>
<h3>Using element-available() function</h3>
<xsl:choose>
<xsl:when test="element-available('xsl:element')">
<p>xsl:element is supported.</p>
</xsl:when>
<xsl:otherwise>
<p>xsl:element is not supported.</p>
</xsl:otherwise>
</xsl:choose>
<xsl:choose>
<xsl:when test="element-available('xsl:update')">
<p>xsl:update is supported.</p>
</xsl:when>
<xsl:otherwise>
<p>xsl:update is not supported.</p>
</xsl:otherwise>
</xsl:choose>
<h3>Using format-number() function</h3>
<xsl:value-of select='format-number(123456, "###,###.00")' /><br/>
<xsl:value-of select='format-number(0.456789, "##%")' />
<h3>Using function-available() function</h3>
<xsl:choose>
<xsl:when test="function-available('update')">
<p>update() is supported.</p>
</xsl:when>
<xsl:otherwise>
<p>update() is not supported.</p>
</xsl:otherwise>
</xsl:choose>
<xsl:choose>
<xsl:when test="function-available('document')">
<p>document() is supported.</p>
</xsl:when>
<xsl:otherwise>
<p>document() is not supported.</p>
</xsl:otherwise>
```

```
</xsl:choose>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

You have to change the `<xml-stylesheet>` in the product.xml file so that it points to this `XSLT` stylesheet, as shown here:

```
<?xml-stylesheet type="text/xsl" href="products2.xsl"?>
```

Now open the product.xml file in the browser. This will look like the one shown in Figure 27.10:



Accessing Current node using current() function

Barbie Doll
Mini Bus
Car
Air Plane

Accessing emp.xml using document() function

Ambrish Kumar Singh 25

Using element-available() function

xsl:element is supported

xsl:update is not supported

Using format-number() function

123,456.00
46%

Using function-available() function

update() is not supported

document() is supported

Figure 27.10: Showing Various XSLT Functions

The second stylesheet explains the `generate-id()` function. The code for `products3.xsl` is given in Listing 27.19, (you can find the products3.xsl file in the `Code/XML/Chapter 27/XSLT` folder on the CD):

Listing 27.19: products3.xsl

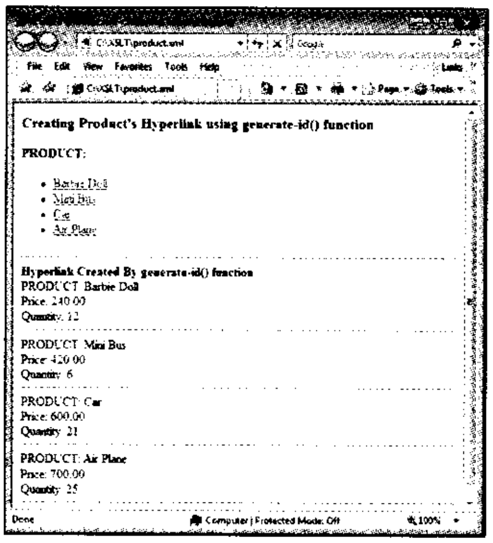
```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:key name="productlist" match="PRODUCT" use="PRODUCTNAME" />
<xsl:template match="/">
<html>
<body>
<h3>Creating Product's Hyperlink using generate-id() function</h3>
<b>PRODUCT:</b>
<ul>
<xsl:for-each select="PRODUCTDATA/PRODUCT">
<li>
<a href="#{generate-id(PRODUCTNAME)}">
<xsl:value-of select="PRODUCTNAME" /></a>
</li>
</xsl:for-each>
</ul>
<hr />
<b>Hyperlink Created By generate-id() function</b><br/>
<xsl:for-each select="PRODUCTDATA/PRODUCT">
PRODUCT: <a name="{generate-id(PRODUCTNAME)}">
<xsl:value-of select="PRODUCTNAME" /></a>
<br />
Price: <xsl:value-of select="PRICE" />
<br />
```

```
Quantity: <xsl:value-of select="QUANTITY" />
<hr />
</xsl:for-each>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Now let's change `<xml-stylesheet> in the product.xml file` so that it points to the `XSLT` stylesheet, as shown in the following code line:

```
<?xml-stylesheet type="text/xsl" href="products3.xsl"?>
```

Next open the product.xml file in browser. This will look like the one shown in Figure 27.11:



Creating Product's Hyperlink using generate-id() function

PRODUCT:

- Barbie Doll
- Mini Bus
- Car
- Air Plane

Hyperlink Created By generate-id() function
PRODUCT Barbie Doll
Price: 240.00
Quantity: 12

PRODUCT Mini Bus
Price 420.00
Quantity: 6

PRODUCT: Car
Price: 600.00
Quantity: 21

PRODUCT: Air Plane
Price: 700.00
Quantity: 25

Figure 27.11: Showing XSLT generate-id () Function

The third and the last stylesheet explains the functions, `key()` and `system-property()`. The code for `products4.xsl` is given in Listing 27.20, (you can find this file named products4.xsl in the `Code/XML/Chapter 27/XSLT` folder on the CD):

Listing 27.20: products4.xsl

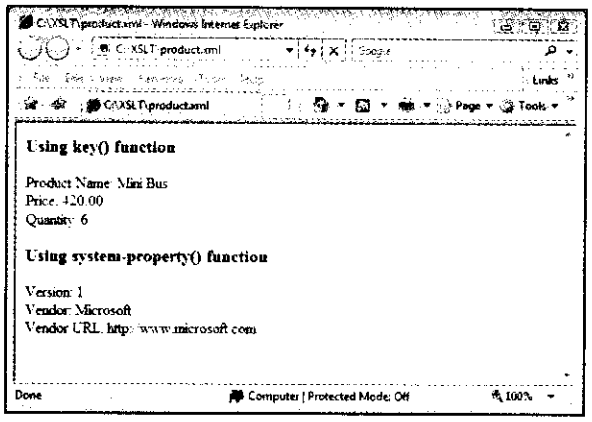
```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:key name="productlist" match="PRODUCT" use="PRODUCTNAME" />
<xsl:template match="/">
<html>
<body>
<h3>Using key() function</h3>
<xsl:for-each select="key('productlist', 'Mini Bus')">
<p>
Product Name: <xsl:value-of select="PRODUCTNAME" />
<br />
Price: <xsl:value-of select="PRICE" />
<br />
Quantity: <xsl:value-of select="QUANTITY" />
</p>
</xsl:for-each>
<h3>Using system-property() function</h3>
<p>
Version:
```

```
<xsl:value-of select="system-property('xsl:version')" />
<br />
Vendor:
<xsl:value-of select="system-property('xsl:vendor')" />
<br />
Vendor URL:
<xsl:value-of select="system-property('xsl:vendor-url')" />
</p>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Now let's change the `<xml-stylesheet>` in the product.xml file so that it points to this `XSLT` stylesheet, as shown here:

```
<?xml-stylesheet type="text/xsl" href="products4.xsl"?>
```

Next open the product.xml file in the browser. This will look like the one shown in Figure 27.12:



C:\XSLT\product.xml - Windows Internet Explorer

C:\XSLT\product.xml

Using key() function

Product Name: Mini Bus
Price: 420.00
Quantity: 6

Using system-property() function

Version: 1
Vendor: Microsoft
Vendor URL: http://www.microsoft.com

Done | Computer | Protected Mode: Off | 100%

Figure 27.12: Using XSLT Functions and Elements

After discussing about XSLT in detail, let's describe Xpath which is an expression language used by XSLT.

Transforming an XML Document Using XSLT

An XML document can be transformed into HTML, WML or other markup languages. Listing 27.21 shows an XML document product.xml, when you open the document in a browser supporting XML, such as Internet Explorer:

Listing 27.21: product.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
- <PRODUCTDATA>
- <PRODUCT>
    <PRODID>P001</PRODID>
    <PRODUCTNAME>Barbie Doll</PRODUCTNAME>
    <DESCRIPTION>This is a toy for children in the age group below 5
    years</DESCRIPTION>
    <PRICE>240.00</PRICE>
    <QUANTITY>12</QUANTITY>
</PRODUCT>
- <PRODUCT>
    <PRODID>P002</PRODID>
    <PRODUCTNAME>Mini Bus</PRODUCTNAME>
<DESCRIPTION>This is a toy for children in the age group of 5-10 years</DESCRIPTION>
    <PRICE>420.00</PRICE>
    <QUANTITY>6</QUANTITY>
</PRODUCT>
```

```
- <PRODUCT>
   <PRODID>P003</PRODID>
   <PRODUCTNAME>Car</PRODUCTNAME>
   <DESCRIPTION>This is a toy for children in the age group of 10-15
   years</DESCRIPTION>
   <PRICE>600.00</PRICE>
   <QUANTITY>21</QUANTITY>
  </PRODUCT>
- <PRODUCT>
   <PRODID>P004</PRODID>
   <PRODUCTNAME>Air Plane</PRODUCTNAME>
   <DESCRIPTION>This is a toy for children in the age group of 08-15
   years</DESCRIPTION>
   <PRICE>700.00</PRICE>
   <QUANTITY>25</QUANTITY>
 </PRODUCT>
 </PRODUCTDATA>
```

Notice that the data is displayed within tags. However, this format of display is not required by the users, who would appreciate data being displayed in an easy-to-navigate manner. To do this, you need to transform the XML document into an HTML document using a transformation language such as XSLT, which is part of XSL family. The XSL family contains two main parts: XSLT, which is a transformation language; and XSL, which is a formatting language.

Listing 27.22 shows the product.xsl document, which is used to transform the product.xml document into an HTML document:

Listing 27.22: product.xsl document

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

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
  <html>
  <body>
        <xsl:apply-templates/>
  </body>
  </html>
</xsl:template>

<xsl:template match="PRODUCTDATA">
  <table border="2" width="50%">
        <xsl:for-each select="PRODUCT">
        <tr>
                <td>
                        <xsl:value-of select=
                        "PRODID"/>
                </td>
                <td>
                        <xsl:value-of select="PRODUCTNAME"/>
                </td>
                <td>
                        <xsl:value-of select="DESCRIPTION"/>
                </td>
                <td>
                        <xsl:value-of select="PRICE"/>
                </td>
                <td>
                        <xsl:value-of select="QUANTITY"/>
                </td>
        </tr>
```

```
            </xsl:for-each>
    </table>
  </xsl:template>
  </xsl:stylesheet>
```

The Listing 27contains many standard XSLT elements of the XSL stylesheet. The XSLT elements define the structure of a stylesheet. An XSL Stylesheet begins with either the stylesheet elements or with transform. Both these elements do the same thing. The most important element in the XSL Stylesheet is the template element. The XSLT elements used in this listing are as follows:

- <xsl:template> —This element defines a template for producing output. It is a top-level element. It contains rules to apply when a specified node is matched against a pattern or explicitly by name. It uses a match attribute for defining the pattern. Matching against a pattern makes use of the match attribute of the <xsl:template> element. For example, match ="/" defines the whole document as it matches the root element.
- <xsl:apply-templates>—This element defines a set of nodes to be processed by selecting an appropriate template rule for each. The <xsl:apply-templates> element is an instruction, which is used within a template. It selects a set of nodes and processes each of them by finding a matching template.
- <xsl:stylesheet> or <xsl:transform>—These elements are the outermost elements of the stylesheet. Both of them are used to define the root element of the stylesheet.
- <xsl:value-of> —This element is mostly used for writing text to a result tree. It is used for constructing a Text node and extracting the value of a node.
- <xsl:for-each> —The <xsl:for-each> element allows you to implement looping in XSL.
- <xsl:output> —This element specifies the format of the output document.

Let us see XML document product.xml on which we are applying product.xsl stylesheet. Listing 27.23 shows the product.xml document. The product.xml document shown in this listing is little different from the previous product.xml document. In this listing, the product.xml document has no attributes under the <PRODUCT> tag.

Listing 27.23: products.xml

```
  <?xml version="1.0" encoding="UTF-8"?>
  <PRODUCTDATA>
    <PRODUCT>
          <PRODID>P001</PRODID>
          <PRODUCTNAME>Barbie Doll</PRODUCTNAME>
          <DESCRIPTION>This is a toy for children in the age group below 5 years
          </DESCRIPTION>
          <PRICE>240.00</PRICE>
          <QUANTITY>12</QUANTITY>
    </PRODUCT>
    <PRODUCT>
          <PRODID>P002</PRODID>
          <PRODUCTNAME>Mini Bus</PRODUCTNAME>
          <DESCRIPTION>This is a toy for children in the age group of 5-10 years
          </DESCRIPTION>
          <PRICE>420.00</PRICE>
          <QUANTITY>5</QUANTITY>
    </PRODUCT>
    <PRODUCT>
          <PRODID>P003</PRODID>
          <PRODUCTNAME>Car</PRODUCTNAME>
          <DESCRIPTION>This is a toy for children in the age group of 10-15 years
          </DESCRIPTION>
          <PRICE>600.00</PRICE>
          <QUANTITY>21</QUANTITY>
    </PRODUCT>
    <PRODUCT>
```

```
            <PRODID>P004</PRODID>
            <PRODUCTNAME>Air Plane</PRODUCTNAME>
            <DESCRIPTION>This is a toy for children in the age group of 08-15 years
            </DESCRIPTION>
            <PRICE>700.00</PRICE>
            <QUANTITY>25</QUANTITY>
      </PRODUCT>
  </PRODUCTDATA>
```

Listing 27.24 shows the code of the XML2HTMLUsingXSLT.java file, which uses the XSLT API to transform the XML document shown in the previous listing:

Listing 27.24: XML2HTMLUsingXSLT.java

```
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import java.io.*;
public class XML2HTMLUsingXSLT {
  public static void main(String[] args)
   throws TransformerException,
   TransformerConfigurationException,
   FileNotFoundException, IOException
  {
        TransformerFactory tfact =
        TransformerFactory.newInstance();
        Transformer t =
        tfact.newTransformer(
        new StreamSource("product.xsl"));
        t.transform(
        new StreamSource(args[0]),
        new StreamResult(new FileOutputStream(
        args[1])));
        System.out.println
        (" The output is written in "+args[1]);
  }
}
```

Listing 27.24 applies the product.xsl stylesheet on the product.xml document. In this listing, when we create the Transformer instance, the product.xsl document is passed as StreamSource to the newTransformer() method. Finally, the listing invokes the transform method on the Transformer instance (t) to apply the product.xsl stylesheet on the product.xml document.

Figure 27.13 shows the compilation and execution of XML2HTMLUsingXSLT.java file:

Figure 27.13: Showing product.html

In this Figure, first argument passed to main method of XML2HTMLUsingXSLT.java file is product.xsl and second is name of HTML file which resides the output of this file.

Let's move further and learn about XPath.

XPath

XPath is an expression language used for finding information in XML documents. XPath lets you address specific parts of XML documents. One of the fundamental things to realize about XPath is that it is not used alone. It is used in conjunction with other XML technologies, such as XSLT. Another important thing about XPath is that it is used with XML applications, but it is not written using XML syntax.

The most important concept related to XPath is that it represents an XML document in the form of a tree. This is known as XPath Data Model. According to this model, an XPath document has seven nodes. These nodes are as follows:

- Root node
- Element node
- Attributes node
- Namespaces node
- Processing-instruction node
- Comment node
- Text node

In XPath Data Model, each tree has only one Root node. This node cannot occur anywhere else other than the root of the tree. The Element node represents an element in the source XML document. An Element node may contain an ordered list of child Element nodes. The ordered list is useful when we want to access nodes according to their positions. The Attribute node represents an attribute in the source XML file. The parent node of Attribute node is Element node. The Namespace node represents a namespace that is in scope on the element in the source XML document represented by the parent Element node of the namespace node. The Processing-instruction node represents a processing-instruction in the source XML document. The Comment node represents a comment in the XML document, and the Text node represents the text content of an element.

Functionality of XPath

In this section, we'll discuss how XPath works with XSLT. Before that we'll explain how XPath expressions are written for accessing nodes and elements from the XML documents. But first, let's look at some commonly used path expressions.

Location Paths

Location path is one of the path expressions. The location paths are generally used for accessing node-sets. Suppose you want to access the root element of the XML document given in Listing 27.13. This can be accomplished by any of the following path statements:

- /PRODUCTDATA
- /*

Suppose you want to access the product element. This can be accomplished by any of the following path statements:

- /PRODUCTDATA/PRODUCT
- /*/PRODUCT
- /PRODUCTDATA/*
- /*/*
- //PRODUCT

Selection of Nodes

You can select the nodes in an XML document using XPath expressions. Table 27.20 lists the most important path expressions syntax for accessing nodes.

Table 27.20: XPath expression syntax

[illegible]	[illegible]
node	This expression selects all the child nodes of the specified nodes
/	This expression shows that the selection will start from the Root node
//element	This expression selects all the elements no matter where they are in the XML document
.	This expression selects the current node
..	This expression selects the parent of the current node
//@ id	This expression selects the attributes named id

Let's consider an XML document product.xml in Listing 27.13. When the path expressions (mentioned in Table 27.21) are applied on the XML document (product.xml), the corresponding results produced as listed in Table 27.21.

Table 27.21: Xpath expression with their results

[illegible]	[illegible]
PRODUCTDATA	This expression selects all the child nodes of the PRODUCTDATA element
/PRODUCTDATA	This expression selects the Root node
//PRODUCT	This expression selects all the PRODUCT elements, no matter where they are in the XML document
.	This expression selects the current node
..	This expression selects the parent of the current node
/PRODUCTDATA/PRODUCT/PRODUCTNAME	This expression selects all the PRODUCTNAME elements under the PRODUCT element

Using Predicates

Predicates are used with path expressions to find a specific node or a node with specific value. The predicates are enclosed in square brackets in path expressions. Table 27.22 lists some path expressions with predicates.

Table 27.22: Xpath expression using predicates

[illegible]	[illegible]
/PRODUCTDATA/PRODUCT[0]	This expression selects the first PRODUCT element, i.e. the child of the PRODUCTDATA element
/PRODUCTDATA/PRODUCT[last()]	This expression selects the last PRODUCT element, i.e. the child of the PRODUCTDATA element
/PRODUCTDATA/PRODUCT[last() -1]	This expression selects the last but one PRODUCT element, i.e. the child of the PRODUCTDATA element
/PRODUCTDATA/PRODUCT[position() < 3]	This expression selects the first two PRODUCT elements that are the children of the PRODUCTDATA element
//PRODID[@id]	This expression selects all the PRODID elements that have an attribute named id

Table 27.22: Xpath expression using predicates

[illegible]	[illegible]
/PRODUCTDATA/PRODUCT[PRICE > $60]	This expression selects all the PRODUCT elements of the PRODUCTDATA element that have a price element with a value greater than $60

XPath Functions

This section will introduce you to XPath functions. XPath functions return the following values:

- Node set
- Number
- String
- Boolean

We'll cover all these four categories here. So let's start with node set functions.

Node Set Functions

XPath provides various functions that allow us to access a selection from the node set and return another node set. These functions are generally used with those path expressions which use predicates. XPath contains the following node set functions:

- count()
- last()
- local-name()
- name()
- namespace-uri()
- position()

The count() Function

The `count` function returns the number of nodes in an argument node-list. For example, consider the XML document given in Listing 27.13. When we apply the stylesheet given in Listing 27.25, it returns the number of <PRODUCT> element present in the source XML document.

Here's the code, given in Listing 27.25, for `count.xsl` (you can find this file in the `Code/XML/Chapter 27/Xpath_Nodeset_Functions` folder on the CD):

Listing 27.25: count.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<head>
<title>Using the XPath Node Set functions</title>
</head>
<body>
<h4>Using count() Function</h4>
<p>In the source document there are <xsl:value-of
select="count(PRODUCTDATA/PRODUCT)"/> &lt;PRODUCT&gt; elements.</p>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

When you are using XPath functions, it is necessary that you use an XSLT processor. In this case, we are using a SAXON processor.

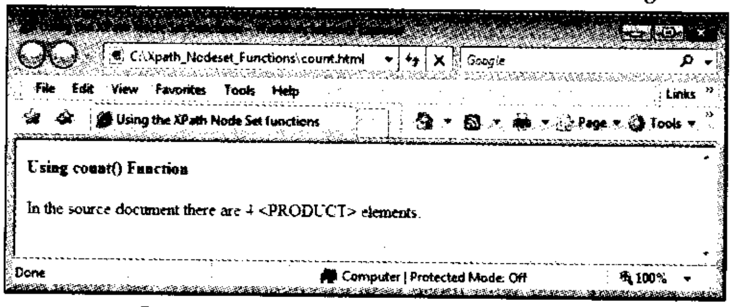
NOTE

You can download the SAXON processor from http://saxon.sourceforge.net/ and unzip this processor in a directory, e.g. in c:\saxon.

Type the following at the command line for running the stylesheet `count.xsl` over `product.xml`:

```
C:\>java -jar c:\saxon\saxon8.jar product.xml count.xsl > count.html
```

After running this command, a `count.html` file is generated which contains the actual transformation. When you open the `count.html` file in the browser, it will look like the one shown in Figure 27.14:



Using count() Function

In the source document there are 4 <PRODUCT> elements.

C:\Xpath_Nodeset_Functions\count.html
Google
File Edit View Favorites Tools Help
Links
Using the XPath Node Set functions
Page Tools
Done
Computer | Protected Mode: Off
100%

Figure 27.14: Using Xpath's count() Function

The last() Function

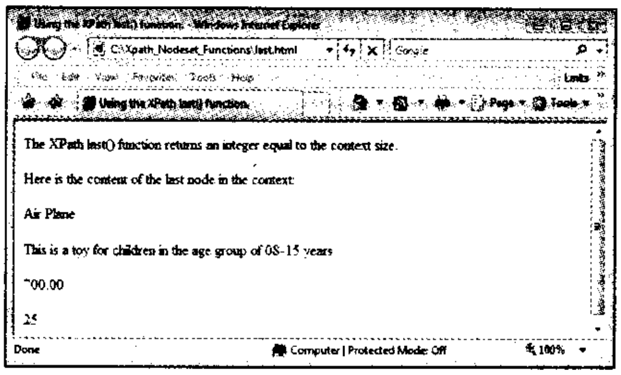
The `last()` function returns an integer equal to the context size. This function is used to access the last node in a context. Let's apply the stylesheet, `last.xsl` (given in Listing 27.26) on the XML document `product.xml` in Listing 27.11.

Here's the code, given in Listing 27.26, for applying the stylesheet (you can find this file in the `Code/XML/Chapter 27/Xpath_Nodeset_Functions` folder on the CD):

Listing 27.26: last.xsl

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<head>
<title>Using the XPath last() function.</title>
</head>
<body>
<p>The XPath last() function returns an integer equal to the context
size.</p>
<p>Here is the content of the last node in the context:</p>
<xsl:apply-templates select="PRODUCTDATA/PRODUCT[last()]"/>
</body>
</html>
</xsl:template>
<xsl:template match="PRODUCT">
<p><xsl:value-of select="PRODUCTNAME"/></p>
<p><xsl:value-of select="DESCRIPTION"/></p>
<p><xsl:value-of select="PRICE"/></p>
<p><xsl:value-of select="QUANTITY"/></p>
</xsl:template>
</xsl:stylesheet>
```

After processing the stylesheet by SAXON processor, when you open the resultant HTML file in your browser it looks like the one shown in Figure 27.15:



The XPath last() function returns an integer equal to the context size.

Here is the content of the last node in the context:

Air Plane

This is a toy for children in the age group of 05-15 years

~00.00

25

Figure 27.15: Using XPath's last() Function

Consider the following XML document, given in Listing 27.27 (you can find this file in the Code/XML/Chapter 27/Xpath_Nodeset_Functions folder on the CD):

Listing 27.27: emp_namespace.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<emp:employees xmlns:emp="http://kogentindia.com/emp"
xmlns:xhtml="http://www.w3.org/1999/xhtml">
<emp:name>
  <emp:title>Sir</emp:title>
  <emp:first>Ambrish</emp:first>
  <emp:middle>Kumar</emp:middle>
  <emp:last>Singh</emp:last>
</emp:name>
<emp:position>Technical Writer</emp:position>
<emp:resume>
  <xhtml:html>
  <xhtml:head><xhtml:title>Resume of Ambrish Kumar
Singh</xhtml:title></xhtml:head>
<xhtml:body>
  <xhtml:h1>Ambrish</xhtml:h1>
  <xhtml:p>Ambrish is a Java Programmer</xhtml:p>
  </xhtml:body>
  </xhtml:html>
</emp:resume>
</emp:employees>
```

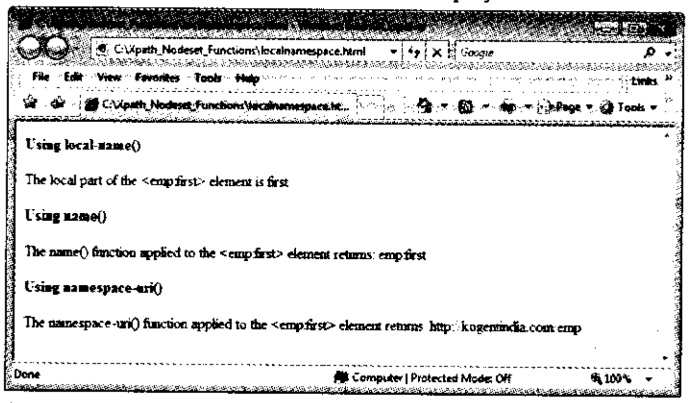
Here's the code, given in Listing 27.28, which apply the stylesheet (you can find this file in the Code/XML/Chapter 27/Xpath_Nodeset_Functions folder on the CD):

Listing 27.28: localnamespace.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:emp="http://kogentindia.com/emp">
<xsl:template match="/">
<html>
<head>
<title></title>
</head>
<body>
<h4>Using local-name()</h4>
<p>The local part of the &lt;emp:first&gt; element is <xsl:value-of
select="local-name(emp:employees/emp:name/emp:first)"/></p>
<h4>Using name()</h4>
```

```
<p>The name() function applied to the &lt;emp:first&gt; element returns:
<xsl:value-of select="name(emp:employees/emp:name/emp:first)"/></p>
<h4>Using namespace-uri()</h4>
<p>The namespace-uri() function applied to the &lt;emp:first&gt; element
 returns:
<xsl:value-of select="namespace-uri(emp:employees/emp:name/emp:first)"/></p>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

After processing with the SAXON processor, the browser will display the resultant document in Figure 27.16:



C:\Xpath_Nodeset_Functions\localnamespace.html

Using local-name()

The local part of the <emp:first> element is first

Using name()

The name() function applied to the <emp:first> element returns: emp:first

Using namespace-uri()

The namespace-uri() function applied to the <emp:first> element returns http://kogentindia.com/emp

Figure 27.16: Using Xpath's local-name(), name(), and namespace-uri() Functions

The local-name() Function

This function is part of the QName after the colon. So in the element `<emp:name>`, the string `name` is the local part. The `local-name()` function returns the local part of the expanded name of the node in the argument node-set that is first in the document order.

The name() Function

The `name()` function returns the string containing a `Qname` representing the expanded name of the node in the argument node-set that is first in the document order. An example of using `name()` function is given in Listing 27.28.

The namespace-uri() Function

The `namespace-uri()` function returns the namespace URI of the nodes in the argument node-set. Its example is given in Listing 27.28.

The position() Function

The `position()` function returns a number that reflects the context position of the context node. The example of this function is given when we discuss String functions.

Number Functions

In this section, we'll describe the XPath number functions. These functions are used for manipulating numbers. XPath has the following number functions:

- ceiling()
- floor()
- number()
- round()
- sum()

The ceiling() Function

The ceiling function is a type of round-up function. It returns the smallest integer which is greater than the argument for the function. Here's the code, given in Listing 27.29 that shows a XML document (you can find this file in the `Code/XML/Chapter 27/Xpath_Number_Functions` folder on the CD):

Listing 27.29: temperature.xml

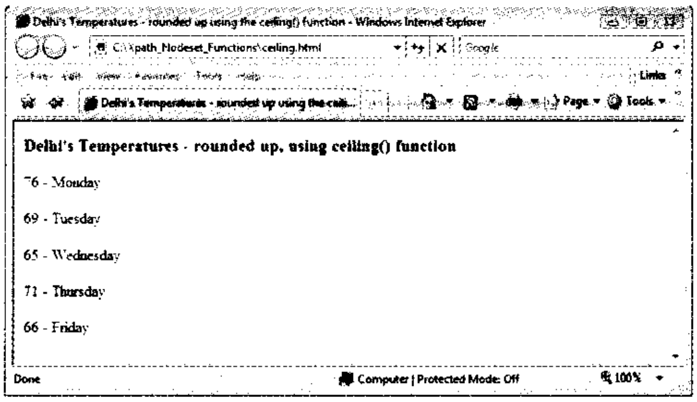
```
<?xml version='1.0'?>
<DelhiTemperature Units="Fahrenheit">
<Monday>75.3</Monday>
<Tuesday>68.1</Tuesday>
<Wednesday>64.9</Wednesday>
<Thursday>71.0</Thursday>
<Friday>65.4</Friday>
</DelhiTemperature>
```

Here's the code, given in Listing 27.30, for applying the stylesheet (you can find this file in the `Code/XML/Chapter 27/Xpath_Number_Functions` folder on the CD):

Listing 27.30: `ceiling.xsl`

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<head>
<title>Delhi's Temperatures - rounded up using the ceiling()
function</title>
</head>
<body>
<h3>Delhi's Temperatures - rounded up, using ceiling() function</h3>
<xsl:apply-templates select="DelhiTemperature/*"/>
</body>
</html>
</xsl:template>

<xsl:template match="Monday | Tuesday | Wednesday | Thursday | Friday">
<p><xsl:value-of select="ceiling(.)"/> - <xsl:value-of select="name()"
/></p>
</xsl:template>
</xsl:stylesheet>
```

After processing with the SAXON processor, the browser will display the resultant document like the one shown in Figure 27.17:


Delhi's Temperatures - rounded up using the ceiling() function - Windows Internet Explorer
C:\Xpath_Nodeset_Functions\ceiling.html
Google
Links
Page
Tools
Done
Computer | Protected Mode: Off
100%

Delhi's Temperatures - rounded up, using ceiling() function

76 - Monday

69 - Tuesday

65 - Wednesday

71 - Thursday

66 - Friday

Figure 27.17: Using Xpath's ceiling() Function

The floor() Function

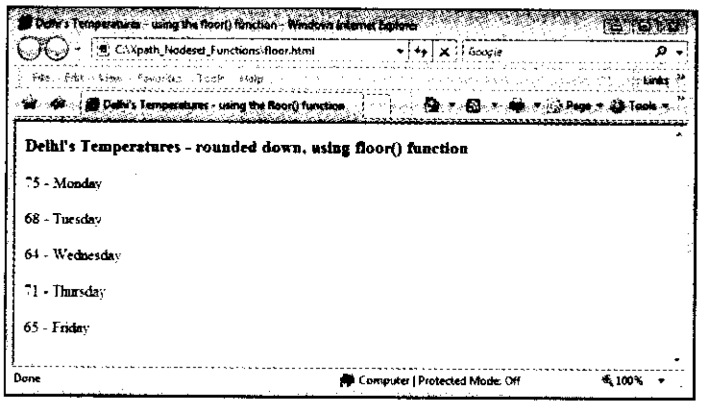
The `floor()` function is a type of round-down function. It returns the greatest integer which is not greater than the argument for the function. Let's take an example.

Here's the shylesheet, given in Listing 27.31, for applying it on the XML document of Listing 27.26 (you can find this file in the `Code/XML/Chapter 27/Xpath_Number_Functions` folder on the CD):

Listing 27.31: `floor.xsl`

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<head>
<title>Delhi's Temperatures - using the floor() function</title>
</head>
<body>
<h3>Delhi's Temperatures - rounded down, using floor() function</h3>
<xsl:apply-templates select="DelhiTemperature/*"/>
</body>
</html>
</xsl:template>
<xsl:template match="Monday | Tuesday | Wednesday | Thursday | Friday">
<p><xsl:value-of select="floor(.)"/> - <xsl:value-of
select="name()"/></p>
</xsl:template>
</xsl:stylesheet>
```

After processing with the SAXON processor, the browser will display the resultant document like the one shown in Figure 27.18:


Delhi's Temperatures - using the floor() function - Windows Internet Explorer
C:\Xpath_Nodeset_Functions\floor.html
Done
Computer | Protected Mode: Off
100%

Delhi's Temperatures - rounded down, using floor() function

75 - Monday

68 - Tuesday

64 - Wednesday

71 - Thursday

65 - Friday

Figure 27.18: Using Xpath's floor () Function

The number () Function

The `number()` function converts its argument to a number. It returns a NaN (Not a Number), if the specified argument is not a number. Let's change the `temperature.xml` file (Listing 27.29) so that the element `<Monday>` now contains some text value.

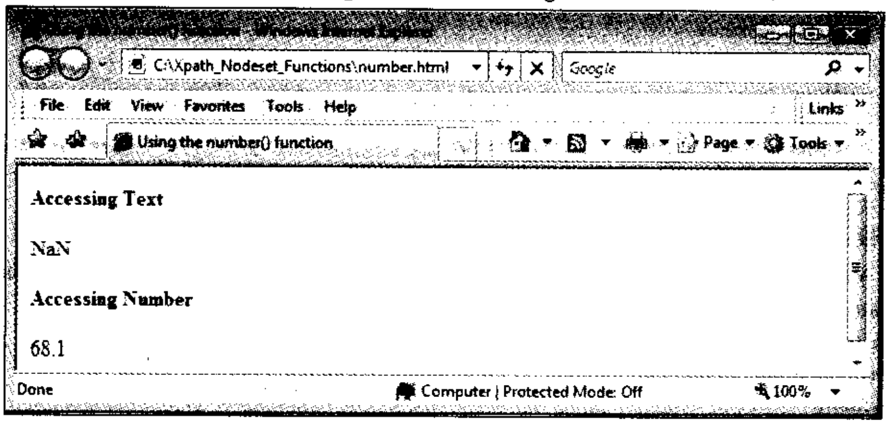
Here's the stylesheet, given in Listing 27.32, for applying it on the changed XML document of Listing 27.29 (you can find this file in the `Code/XML/Chapter 27/Xpath_Number_Functions` folder on the CD):

Listing 27.32: `number.xsl`

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

```
<xsl:template match="/">
<html>
<head>
<title>Using the number() function</title>
</head>
<body>
<h4>Accessing Text</h4>
<p><xsl:value-of select="number(DelhiTemperature/Monday)"/></p>
<h4>Accessing Number</h4>
<p><xsl:value-of select="number(DelhiTemperature/Tuesday)"/></p>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

After processing with the SAXON processor, the browser will display the resultant document like the one shown in Figure 27.19 (Observe that, while accessing the text value using `number ()` function, it returns NaN):


C:\Xpath_Nodeset_Functions\number.html
Google
File Edit View Favorites Tools Help
Links
Using the number() function
Page Tools
Done
Computer | Protected Mode: Off
100%

Accessing Text

NaN

Accessing Number

68.1

Figure 27.19: Using Xpath's number () Function

The round() Function

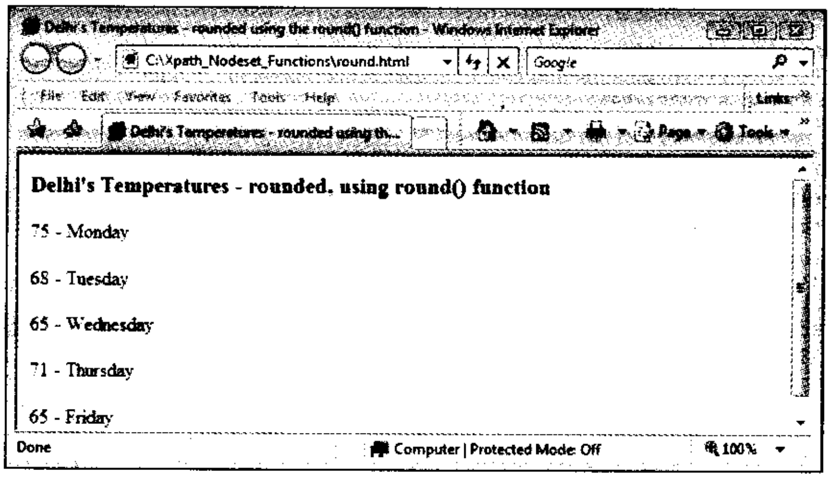
The `round()` function rounds a real number to the nearest integer. Consider the XML document, given in Listing 27.29, and apply the stylesheet given in Listing 27.33 (you can find this file in the `Code/XML/Chapter 27/Xpath_Number_Functions` folder on the CD):

Listing 27.33: `round.xsl`

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<head>
<title>Delhi's Temperatures - rounded up using the round()
function</title>
</head>
<body>
<h3>Delhi's Temperatures - rounded, using round() function</h3>
<xsl:apply-templates select="DelhiTemperature/*"/>
</body>
</html>
</xsl:template>
<xsl:template match="Monday | Tuesday | Wednesday | Thursday | Friday">
<p><xsl:value-of select="round(.)"/> - <xsl:value-of select="name()"
/></p>
```

```
</xsl:template>
</xsl:stylesheet>
```

After processing with the SAXON processor, the browser will display the resultant document like the one shown in Figure 27.20:



Delhi's Temperatures - rounded, using round() function

75 - Monday

68 - Tuesday

65 - Wednesday

71 - Thursday

65 - Friday

Delhi's Temperatures - rounded using the round() function - Windows Internet Explorer
C:\Xpath_Nodeset_Functions\round.html
Google
File Edit View Favorites Tools Help
Links
Delhi's Temperatures - rounded using th...
Page
Tools
Done
Computer | Protected Mode: Off
100%

Figure 27.20: Using Xpath's round() Function

String Functions

Let study the XPath string functions. Here are the functions of XPath that are used for manipulating strings:

- concat()
- contains()
- normalize-space()
- starts-with()
- string()
- string-length()
- substring()
- substring-after()
- substring-before()
- translate()

The concat() Function

The `concat()` function concatenates the arguments passed to it. Here's the `XML` document, given in Listing 27.34 (you can find this file in the `Code/XML/Chapter 27/Xpath_String_Functions` folder on the CD):

Listing 27.34: `poem.xml`

```
<?xml version='1.0'?>
<Poem>
<Phrase>Twinkle Twinkle Little Star</Phrase>
<Phrase>How R Wonder what U R</Phrase>
<Phrase>Twinkle Twinkle Little Star is a famous poem</Phrase>
</Poem>
```

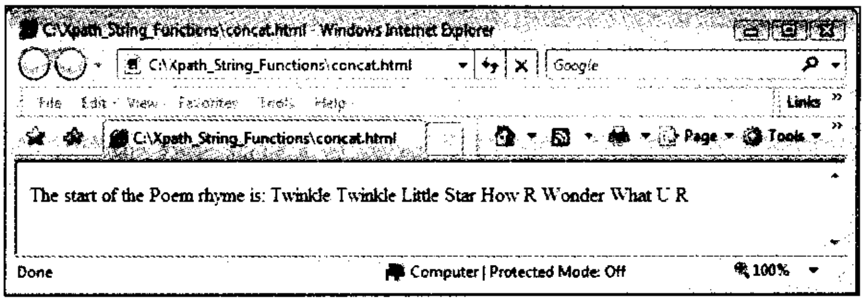
Here's the stylesheet, given in Listing 27.35, for applying it on the `XML` document of Listing 27.34 (you can find this file in the `Code/XML/Chapter 27/Xpath_String_Functions` folder on the CD):

Listing 27.35: `concat.xsl`

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
```

```
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<head>
<title></title>
</head>
<body>
<p>

The start of the Poem rhyme is:
<xsl:value-of select="concat(string(Poem/Phrase[position()=1]),'
', string(Poem/Phrase[position()=2]))"/></p>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

After processing with the SAXON processor, the browser will display the resultant document like the one shown in Figure 27.21:



The start of the Poem rhyme is: Twinkle Twinkle Little Star How R Wonder What U R

C:\Xpath_String_Functions\concat.html - Windows Internet Explorer
C:\Xpath_String_Functions\concat.html
Google
File Edit View Favorites Tools Help
Links
Page Tools
Done
Computer | Protected Mode: Off
100%

Figure 27.21: Using Xpath's concat() Function

The contains() Function

The `contains()` function takes two String arguments and checks whether the second string is present in the first string. If yes, then it returns Boolean value `true`, otherwise it returns `false`.

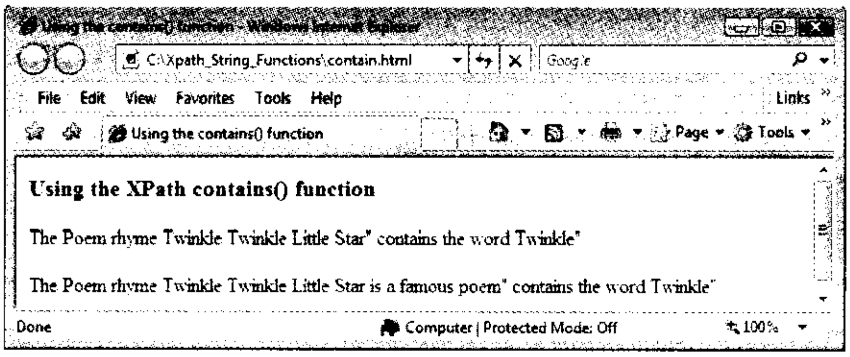
Here's the stylesheet, given in Listing 27.36, for applying it on the XML document of Listing 27.34 (you can find this file in the `Code/XML/Chapter 27/Xpath_String_Functions` folder on the CD):

Listing 27.36: `contain.xsl`

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<head>
<title>Using the contains() function</title>
</head>
<body>

<h1>Using the XPath contains() function</h1>
<xsl:apply-templates select="Poem/Phrase"/>
</body>
</html>

</xsl:template>
<xsl:template match="Phrase">
<xsl:if test="contains(.,'Twinkle')">
<p>The Poem rhyme <xsl:value-of select="."/>" contains the word Twinkle"</p>
```

```
</xsl:if>
</xsl:template>
</xsl:stylesheet>
```

After processing with the SAXON processor, the browser will display the resultant document like the one shown in Figure 27.22:


C:\Xpath_String_Functions\contain.html
Google
File Edit View Favorites Tools Help
Links
Using the contains() function
Page
Tools
Done
Computer | Protected Mode: Off
100%

Using the XPath contains() function

The Poem rhyme Twinkle Twinkle Little Star" contains the word Twinkle"

The Poem rhyme Twinkle Twinkle Little Star is a famous poem" contains the word Twinkle"

Figure 27.22: Using Xpath's contains() Function

The normalize-space() Function

The `normalize-space()` function is used to remove the leading and trailing whitespace from a string by replacing any internal sequences of whitespace characters by a single space character. The `normalize-space()` function takes zero or one argument. If no argument is passed then the string value of the context node is evaluated. Otherwise, the argument is converted to a string and then evaluated. Here's the `XML` document, given in Listing 27.37, that contains many whitespace and leading space characters (you can find this file in the `Code/XML/Chapter 27/Xpath_String_Functions` folder on the CD):

Listing 27.37: whitespace.xml

```
<?xml version='1.0'?>
<whitespace>
<Text> This has leading spaces and one big gap
caused by excess whitespace.</Text>
<Text>This doesn't have leading space but does
have sequences of space characters .</Text>
</whitespace>
```

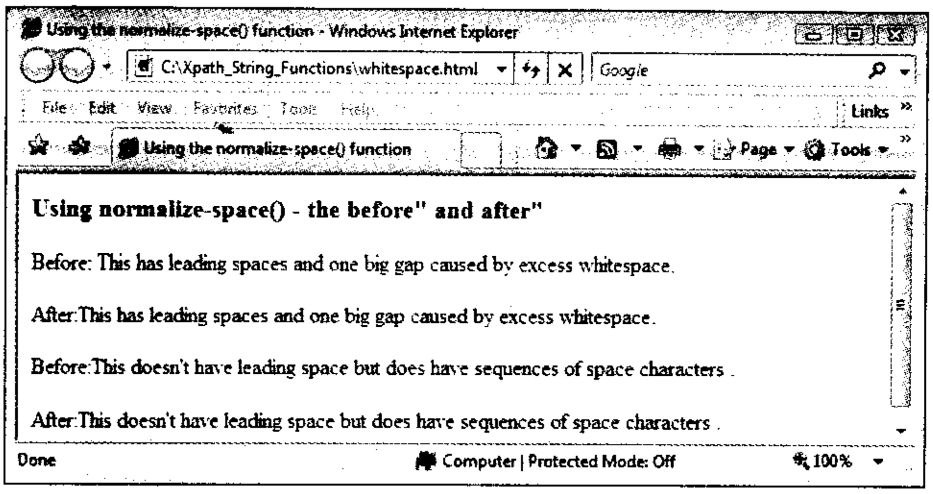
Here's the stylesheet, given in Listing 27.38, for applying it on the `XML` document of Listing 27.37 (you can find this file in the `Code/XML/Chapter 27/Xpath_String_Functions` folder on the CD):

Listing 27.38: whitespace.xsl

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<head>
<title>Using the normalize-space() function</title>
</head>
<body>
<h3>Using normalize-space() - the before" and after"</h3>
<xsl:apply-templates select="whitespace/Text"/>
</body>
</html>
</xsl:template>
<xsl:template match="Text">
<p>Before:<xsl:value-of select="."/></p>
<p> After:<xsl:value-of select="normalize-space(.)"/></p>
```

```
</xsl:template>
</xsl:stylesheet>
```

After processing with the SAXON processor, the browser will display the resultant document like the one shown in Figure 27.23:


Using the normalize-space() function - Windows Internet Explorer
C:\Xpath_String_Functions\whitespace.html
Google
File Edit View Favorites Tools Help
Links
Using the normalize-space() function
Page Tools
Done
Computer | Protected Mode: Off
100%

Using normalize-space() - the before" and after"

Before: This has leading spaces and one big gap caused by excess whitespace.

After:This has leading spaces and one big gap caused by excess whitespace.

Before:This doesn't have leading space but does have sequences of space characters .

After:This doesn't have leading space but does have sequences of space characters .

Figure 27.23: Using Xpath's normalize-sace () Function

Here's the code, given in Listing 27.39, for this document (you can find this file in the `Code/XML/Chapter 27/Xpath_String_Functions` folder on the CD):

Listing 27.39: whitespace.html

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Using the normalize-space() function</title>
</head>
<body>
  <h3>Using normalize-space() - the before" and after"</h3>
  <p>Before: This has leading spaces and one big
  gap caused by excess whitespace.
  </p>
  <p> After:This has leading spaces and one big gap caused by excess
  whitespace.</p>
  <p>Before:This doesn't have leading space but does
  have sequences of space characters .
  </p>
  <p> After:This doesn't have leading space but does have sequences of
  space characters .</p>
  </body>
</html>
```

The starts-with () Function

The `starts-with ()` function takes two string arguments. It checks whether the first string argument starts with the second string argument.

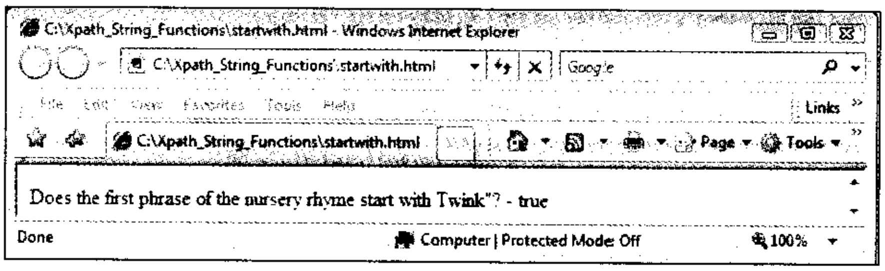
Here's the stylesheet, given in Listing 27.40, for applying it on the `XML` document of Listing 27.34 (you can find this file in the `Code/XML/Chapter 27/Xpath_String_Functions` folder on the CD):

Listing 27.40: startwith.xsl

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

```
<xsl:template match="/">
<html>
<head>
<title></title>
</head>
<body>
<p>
Does the first phrase of the nursery rhyme start with Twink"? -
<xsl:value-of select="starts-with(
string(Poem/Phrase[position()=1]),'Twink')"/></p>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

After processing with the SAXON processor, the browser will display the resultant document like the one shown in Figure 27.24:


C:\Xpath_String_Functions\startwith.html - Windows Internet Explorer
C:\Xpath_String_Functions\startwith.html
Google
Links
Page
Tools

Does the first phrase of the nursery rhyme start with Twink"? - true

Done
Computer | Protected Mode: Off
100%

Figure 27.24: Using Xpath's starts-with() Function

The string() Function

The `string()` function simply converts its argument to a string.

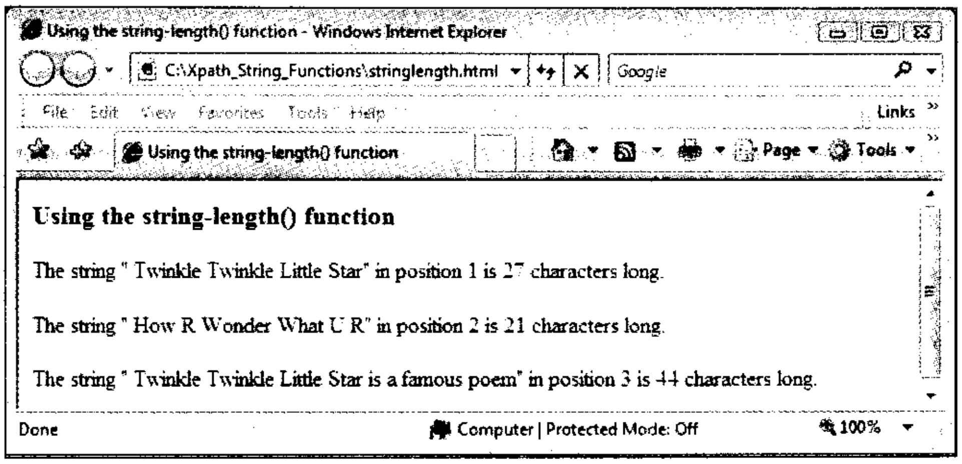
The string-length() Function

The `string-length()` function returns the length of the string passed to it as the argument. Here's the way to apply the stylesheet, given in Listing 27.41, on the `XML` document of Listing 27.34 (you can find this file in the `Code/XML/Chapter 27/Xpath_String_Functions` folder on the CD):

Listing 27.41: stringlength.xsl

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<head>
<title>Using the string-length() function</title>
</head>
<body>
<h3>Using the string-length() function</h3>
<xsl:apply-templates select="Poem/Phrase"/>
</body>
</html>
</xsl:template>
<xsl:template match="Phrase">
<p>The string " <xsl:value-of select="."/>" in position <xsl:value-of
select="position()"/> is
<xsl:value-of select="string-length(.)"/> characters long.</p>
</xsl:template>
</xsl:stylesheet>
```

After processing with the SAXON processor, the browser will display the resultant document like the one shown in Figure 27.25:


Using the string-length() function - Windows Internet Explorer
C:\Xpath_String_Functions\stringlength.html
Google
File Edit View Favorites Tools Help
Links
Using the string-length() function
Page Tools
Done
Computer | Protected Mode: Off
100%

Using the string-length() function

The string " Twinkle Twinkle Little Star" in position 1 is 27 characters long.

The string " How R Wonder What U R" in position 2 is 21 characters long.

The string " Twinkle Twinkle Little Star is a famous poem" in position 3 is 44 characters long.

Figure 27.25: Using Xpath's string-length() Function

The substring() Function

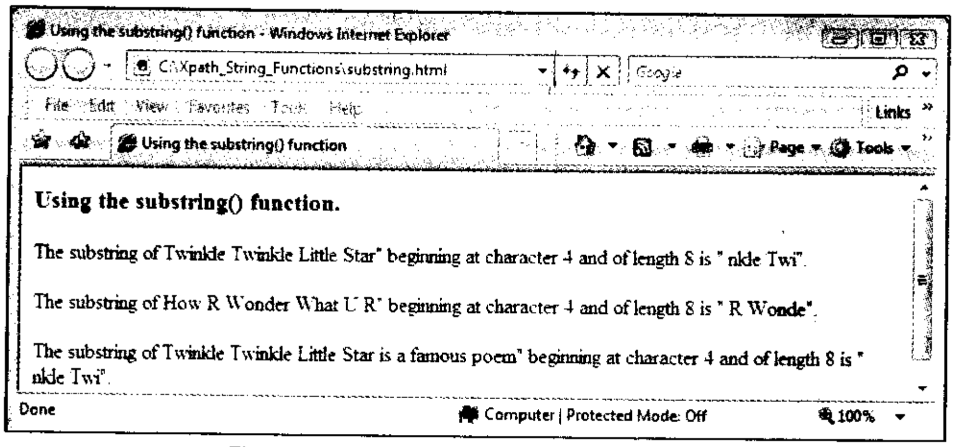
The `substring()` function has three arguments. The first argument specifies the string from which the substring is to be extracted. The second argument specifies the character from where the extraction starts, and the third argument specifies the length of the string to be extracted.

Here's the way to apply the stylesheet, given in Listing 27.42, on the `XML` document of Listing 27.34 (you can find this file in the `Code/XML/Chapter 27/Xpath_String_Functions` folder on the CD):

Listing 27.42: substring.xsl

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

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

<html>

<head>

<title>Using the substring() function</title>

</head>
<body>
<h3>Using the substring() function.</h3>
<xsl:apply-templates select="Poem/Phrase"/>
</body>
</html>
</xsl:template>
<xsl:template match="Phrase">
<p>The substring of <xsl:value-of select="."/>" beginning at character
4 and of length 8 is "
<xsl:value-of select="substring(., 4, 8)"/>".</p>
</xsl:template>
</xsl:stylesheet>
```

After processing with the SAXON processor, the browser will display the resultant document like the one shown in Figure 27.26:



Using the substring() function.

The substring of Twinkle Twinkle Little Star" beginning at character 4 and of length 8 is " nkle Twi".

The substring of How R Wonder What U R" beginning at character 4 and of length 8 is " R Wonde".

The substring of Twinkle Twinkle Little Star is a famous poem" beginning at character 4 and of length 8 is " nkle Twi".

Figure 27.26: Using Xpath's substring() Function

The substring-after() Function

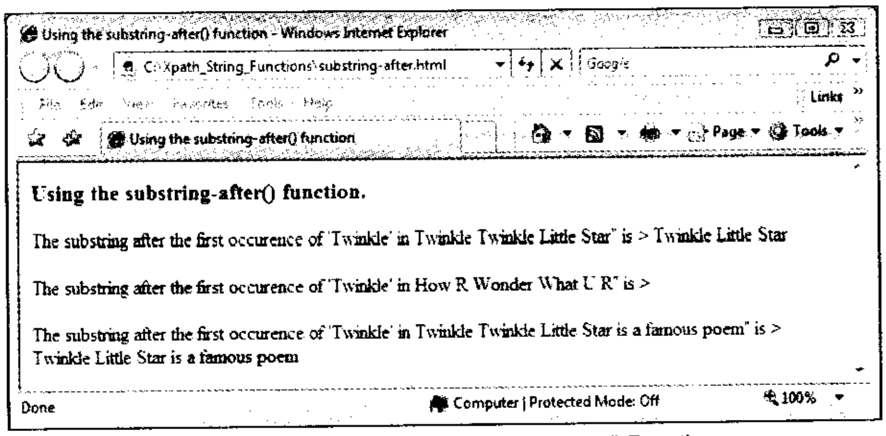
The `substring-after()` function returns the substring that occurs in its first argument after the first occurrence of the string in its second argument.

Here's the way to apply the stylesheet, given in Listing 27.43, on the `XML` document of Listing 27.34 (you can find this file in the `Code/XML/Chapter 27/Xpath_String_Functions` folder on the CD):

Listing 27.43: substring-after.xsl

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">

<html>

<head>

<title>Using the substring-after() function</title>

</head>

<body>
<h3>Using the substring() function.</h3>
<xsl:apply-templates select="Poem/Phrase"/>

</body>

</html>
</xsl:template>
<xsl:template match="Phrase">
<p>The substring after the first occurence of 'Twinkle' in <xsl:value-of
 select="."/>" is &gt;
<xsl:value-of select="substring-after(., 'Twinkle')"/></p>
</xsl:template>
</xsl:stylesheet>
```

After processing with the SAXON processor, the browser will display the resultant document like the one shown in Figure 27.27:



Using the substring-after() function - Windows Internet Explorer

Using the substring-after() function.

The substring after the first occurence of 'Twinkle' in Twinkle Twinkle Little Star" is > Twinkle Little Star

The substring after the first occurence of 'Twinkle' in How R Wonder What U R" is >

The substring after the first occurence of 'Twinkle' in Twinkle Twinkle Little Star is a famous poem" is > Twinkle Little Star is a famous poem

Figure 27.27: Using Xpath's substring-after() Function

The function `substring-before()` works similarly, but it returns the substring that occurs in its first argument before the first occurrence of the string in its second argument.

The translate() Function

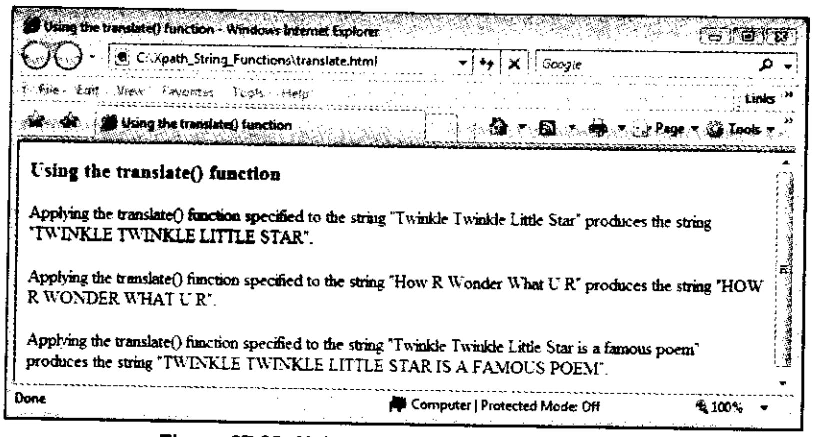
The `translate()` function translates the first string argument with the occurrence of characters specified in the second argument by replacing the characters specified in the third argument.

Here's the way to apply the stylesheet, given in Listing 27.44, on the `XML` document of Listing 27.34 (you can find this file in the `Code/XML/Chapter 27/Xpath_String_Functions` folder on the CD):

Listing 27.44: translate.xsl

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

<html>

<head>

<title>Using the translate() function</title>

</head>

<body>
<h3>Using the translate() function</h3>
<xsl:apply-templates select="Poem/Phrase"/>
</body>
</html>
</xsl:template>
<xsl:template match="Phrase">
<p>Applying the translate() function specified to the string
"<xsl:value-of select="."/>" produces the string
"<xsl:value-of select="translate(., 'abcdefghijklmnopqrstuvwxyz',
'ABCDEFGHIJKLMNOPQRSTUVWXYZ')"/>".</p>
</xsl:template>

</xsl:stylesheet>
```

After processing the SAXON processor, the browser will display the resultant document like the one shown in Figure 27.28:


Using the translate() function - Windows Internet Explorer
C:\Xpath_String_Functions\translate.html
Google
File Edit View Favorites Tools Help
Links
Using the translate() function
Page
Tools
Done
Computer | Protected Mode: Off
100%

Using the translate() function

Applying the translate() function specified to the string "Twinkle Twinkle Little Star" produces the string "TWINKLE TWINKLE LITTLE STAR".

Applying the translate() function specified to the string "How R Wonder What U R" produces the string "HOW R WONDER WHAT U R".

Applying the translate() function specified to the string "Twinkle Twinkle Little Star is a famous poem" produces the string "TWINKLE TWINKLE LITTLE STAR IS A FAMOUS POEM".

Figure 27.28: Using Xpath's translate () Function

Boolean Functions

Let's now understand the XPath Boolean functions. These functions are used for manipulating Boolean values `true` and `false`. Here are the following Boolean functions of XPath:

- boolean()
- true()
- false()
- lang()
- not()

The boolean() Function

The `boolean()` function takes one argument and converts it into boolean value. The argument may be a number, string, boolean, or node set. Every non-zero number is treated as true, and zero or NaN (not a number) is treated as false. If the argument is a string then every non-empty string is treated as true and an empty string is treated as false. If the argument is a node set then every non-empty node set is treated as true and empty node set is treated as false. If the argument is a boolean value then that value is unchanged.

The true() Function

The `true()` function does not take any arguments. The `true()` function returns a boolean value of `true`. It can be used in a situation where a boolean constant might, otherwise, be required.

The false() Function

The `false()` function does not take any arguments. The `false ()` function returns a boolean value of `false`. It can be used in a situation where a boolean constant might otherwise be required.

The not() Function

This function takes one argument. The `not ()` function negates the value of the condition passed in the argument. For example, if the condition passed in the argument is evaluated as `true` then the negate function will negate this value and return `false`.

The lang() Function

The `lang()` function takes one argument. The purpose of the `lang()` function is to test whether the language of the context node, as determined by the relevant `xml:lang` attribute, matches the language passed to the `lang()` function as its argument.

Now, let's discuss XML linking mechanism.

XML Linking Mechanism

The World Wide Web Consortium's (W3C) XML Linking Working group has designed hypertext links for XML. The developers of this group has explicitly defined the ways so that links itself are written in XML and makes a distinction for links between external link objects and internal link objects to locations within XML documents. Link is basically a relationship, which is asserted to exist between two or more data objects or portions of data objects. The main agenda of the W3C XML Linking Working Group is to design highly advanced, scalable, and maintainable hyperlinking and addressing functionality for XML. Now, XML allows the Web developers to use their own personal markup tags in a way that XML-ready browsers can interpret them. The XML Linking Language (XLink), which is XML markup language is used for creating hyperlinks in XML documents, and provides a mechanism for adding links between XML pages. On the other hand, the XML Pointer Language (XPointer), which is designed to address structural aspects of XML, allows hyperlinks to point to more specific parts (fragments) in the XML document. The XPointer uses XPath expressions to navigate in the XML document.

You all are familiar with HTML hyperlinks, which are generally used to get to and from XML pages. Though XLink linking mechanism is more complicated than a traditional HTML linking mechanism, but it provides a more classy way of linking the multilayered structure of XML documents.

Linking with XLink and XPointer

XLink and XPointer both are W3C recommendation. XLink 1.0 received the W3C recommendation status on 2001 June 27 and on March 28, 2006, XLink 1.1 entered W3C Candidate Recommendation status. The XML Pointer Language (XPointer) received the W3C recommendation status on 25. March 2003.

A set of attributes are defined in XLink that are added to elements of other XML namespaces. XLink provides two kinds of hyperlinking technique to use in XML documents such as:

- **Simple links**—Offers similar functionality to HTML links, which are in-band links. A simple link is similar HTML link and it simply creates a unidirectional hyperlink arc from one element to another through URI. You can see example of a simple link in the following code snippet:

```
<?xml version="1.0"?>
<document xmlns="http://example.org/xmlns/2002/document"
  xmlns:xlink="http://www.w3.org/1999/xlink">
 <heading id="Kogent">Kogent Document</heading>
 <para>The <anchor xlink:type="simple" xlink:href="# Kogent "> Kogent Document </anchor>
  header.</para>
</document>
```

- **Extended links**—Offers out-of-band hyperlinks that can link resources over which the link editor has no control in a linkbase document. You can connect multiple resources, either remote or local, by multiple arcs through extended links. Arcs are unidirectional and only define traversal in a single direction.

An extended link can attain particular traversal pathways among the resources, by grouping resources with labels and using one or more arcs. It is not mandatory to contain all extended links in the same document as the elements they link to. It helps in associating metadata or other supplemental information with resources without editing those resources. The browser support for XPointer is minimal. Some of the important software, which supports for XLink, as of 2006 June, are as follows:

- **Mozilla FireFox**—Mozilla Firefox 1.5.0 has very limited support for simple Xlinks. The CSS-formatted XML is only supported.
- **Internet Explorer**—Internet Explorer support very limited Xlinks, if msxml version 4.0 is used.
- **Netscape**—Netscape 7.2 has the same support for simple XLinks as Mozilla Firefox, but the xlink:show="new" attribute works correctly in Netscape 7.2.
- **Resource Directory Description Language (RDDL)**—RDDL, an extension to XHTML Basic uses XLink simple links.

In HTML, you can create a hyperlink, which either points to an HTML page or to a bookmark within an HTML page with the help of hash (#). Sometimes there are certain situations in which, we need to point to more specific content in a document. Let's say, you want to link to the second item in a particular list in a document. You can do this easily with the help of XPointer.

In the xlink:href attribute, you can add an XPointer part after the URL, to navigate to a specific position in the XML document, if the hyperlink points to an XML document. For example, XPointer is pointing to the sixth item in a list with a unique id of "computer", in the following code snippet:

```
href="http://www.kogent.com/cdlist.xml#id('computer').child(6,item)"
```

Working with XLink

Now, let's see how to work with XLink in a XML document. For that create a XML file named employee.xml, which gives information of the employees of a company along with their photos. Listing 27.45 shows code for the XML file (you can find the code of employee.xml file in the Code\XML\Chapter 27\XML folder on the CD):

Listing 27.45: employee.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<company xmlns:xlink="http://www.w3.org/1999/xlink">
<employee name="John">
  <description
  xlink:type="simple"
  xlink:href="http://kogent.com/images/John.gif"
  xlink:show="new">
  John has joined kogent five years back. Now, he is in the board of directors. He has
  done his MS from Oxford university......
  </description>
</employee>
<employee name ="Lisa">
  <description
  xlink:type="simple"
  xlink:href="http://kogent.com/images/Lisa.gif"
  xlink:show="new">
  Lisa has joined kogent ten years back. Now, she is the vice president sales of the
  company. She has done her studies from the university of Virginia.......
  </description>
</employee>
</company>
```

In the preceding listing:

- The XLink namespace (xmlns:xlink="http://www.w3.org/1999/xlink"), is declared at the top of the document, which shows that document has access to the XLink attributes and features.
- The xlink:type="simple" creates a simple "HTML-like" link.
- The xlink:href attribute specifies the URL to link to.
- The xlink:show attribute specifies where to open the link.
- The xlink:show="new" means that the link should open in a new window.

Working with XPointer

Now, let's see how to work with XPointer in combination with XLink to point to a specific part of another document. For that create a XML file named bird.xml. Listing 27.46 shows code for the XML file (you can find the code of bird.xml file in the Code\XML\Chapter 27\XML folder on the CD):

Listing 27.46: bird.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<birdbreeds>
<bird breed="Diamond Dove" id="Diamond Dove">
<picture url="http://animal.com/Dove.gif"/>
<description>Diamond Dove There are many varieties of the dove that vary in color. Some
  of the more popular varieties of dove are: Ring-Neck
Dove - A grey colored dove with a dark ring around its neck.....
```

```
    </description>
    <Feeding> A dove's metabolism is very active and can starve to death in as little as 24
      hours if it does not eat. Doves should eat a staple
diet of fresh fortified finch seed, parakeet seed or pellet daily. Doves only eat off the
      top of what is offered, so be sure to check the food
daily.....
    </Feeding>
    </bird>
    <bird breed="Cockatiels" id="Cockatiels">
    <picture url="http://animal.com/Cockatiels.gif"/>
    <description> Grey Cockatiel If properly cared for, a cockatiel can live up to thirty
      years. This is a smaller member of the parrot family.....
</description>
    <Feeding>Cockatiels should eat a staple diet of fresh fortified cockatiel seed or pellet
      daily. Cockatiels only eat off the top of what is
offered, so be sure to check the food daily. Besides a variety of seed mix or pellet,
      offer chopped dark green and yellow vegetables and a
variety of fresh fruits in addition to a protein source like mature legumes, hard cooked
      chopped egg, and grated cheese.....
    </Feeding>
    </bird>
    </birdbreeds>
```

In the preceeding listing, you can see that the XML document uses id attributes on each element, which we may want to link to.

Summary

In this chapter, you have learned about XML and its syntax. You also learned about DTD and DOM with XML. Along with XML parser, you have learned how to use java with XML and SAX. Chapter also focused on transforming an XML document using XSLT, XPath, and XML linking mechanisms with its implementation.

In the next chapter, you will learn the concept of AJAX.

Quick Revise

Q1. What is XML?

Ans. Extensible Markup Language (XML) is a markup language based on simple, platform-independent rules for processing and displaying textual information in a structured way. The platform-independent nature of XML makes an XML document an ideal format for exchanging structured textual information among different applications. XML provides customized tags to format and display textual information. XML documents represent data in a platform-neutral manner. For example, an XML document generated by an application running on Microsoft Windows can be easily consumed by an application running on Sun Solaris.

Q2. What syntax rules must be followed while creating an XML document?

Ans. The syntax used to create an XML document is called markup syntax. It is used to define the structure of data in the document. The following rules are associated with the markup syntax:

- XML documents must have a starting tag and closing tag
- XML tags are case-sensitive
- XML elements must be properly nested
- XML documents must have one and only one root element
- XML attributes values must be quoted
- XML preserves white spaces, although you can use white spaces in content including line breaks

Q3. What is XML declaration statement? And why it is used?

Ans. The XML declaration statement is used to indicate that the specified document is an XML document. Although it is not necessary for you to have an XML declaration in an XML document, but it is considered as a good practice to include it. If your document has the XML declaration statement, then it must be the first line in the document which defines the XML version and character encoding. An XML declaration looks like this:

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

Some important points must be noted when using the XML declaration statement, are given as follows:

- XML declaration starts with `<? xml`, and ends with `?>`
- XML declaration must include the `version` attribute as it is mandatory, but the encoding and standalone attributes are optional
- XML declaration must be at the beginning of the file
- XML declaration must maintain the order of version, encoding, and standalone attributes

Q4. **What is XML Parser?**

Ans. XML Parser is application, which is used to read, update, create, and manipulate the XML document. For manipulating the XML document, the XML parser loads the document into the computer's memory and then the data is manipulated using the DOM node-tree structure. The XML parser is a part of the software, which reads the XML files and tests whether the XML document is well-formed against the given DTD or the XML schema. Moreover, the XML Parser also makes the XML files available to the application with the use of the DOM. Some examples of XML parsers are: Microsoft's XML parser and Mozilla's XML parser.

Q5. **What is DTD?**

Ans. DTD is the acronym of Document Type Definition, which is used to define the XML document structure with a list of legal elements and attributes. It defines rules and attributes, which decide how to use tags in an XML document. A DTD can be declared either within an XML document, or can be used as an external reference.

Q6. **What is the syntax to provide a reference to the external DTD?**

Ans. A reference to the external DTD can be provided in the following ways:

- <!DOCTYPE collection SYSTEM "<dtd_name>.dtd">—This syntax is used when both the <dtd_name>.dtd file and the XML document are residing in same directory.
- <!DOCTYPE collection SYSTEM "<directory_name>://<dtd_name>.dtd">—This syntax is used when the <dtd_name>.dtd file and the parent XML document reside in separate directories.

Q7. **What is XHTML?**

Ans. XHTML refers to the Extensible HyperText Markup Language whose main focus of existence is to replace HTML. XHTML is the HTML defined in the form of the XML application. XHTML is very similar to HTML 4.01. From 26 January 2000, the W3C defined XHTML as the latest version of HTML. All browsers support XHTML. As we know HTML is used to display data and XML is used to describe data. However, XHTML combines the features of both markup languages, HTML and XML. XHTML documents must also follow some rules. These rules are given as follows:

- XHTML documents should have nested elements
- XHTML documents should have the end tag or closing tag for elements
- XHTML documents should have elements in lower-case
- XHTML documents should have one root element

Q8. **What is DOM?**

Ans. Document Object Model (DOM) is a platform and language independent convention for representing and interacting with objects in HTML, XHTML and XML documents. In DOM, objects are also called Elements, which are specified and addressed according to the syntax and rules of the programming language. These syntax and rules of the programming language are specified in the DOM Application Programming Interface (API).

DOM presents an XML document as the tree-structure having the root node as the parent element and the elements, attributes, and text defined as the child nodes. With the help of the DOM tree, the elements containing the text and the attributes can be manipulated and accessed. In a DOM tree, you can add new elements, modify the existing ones, or can remove the unwanted ones. Note that, in a DOM structure, all the elements, their text, and their attributes are called as nodes. The entire document is considered as the Document node. The XML tag or the XML element is recognized as the Element node. The text in XML elements is referred to as the Text node, the XML attributes are considered as the Attribute nodes and the comments are considered as the Comment node. In the DOM tree structure, the nodes have a hierarchical relationship with each other. The terms 'parent' and 'child' are used to describe the relationships between the nodes.

Q9. **What is SAX?**

Ans. SAX is the acronym of Simple API for XML, which provides a mechanism to read data from an XML document. It is the serial access parser, which is an alternative of DOM. The SAX parser is the Event Driven parser. The user defines the number of callback methods which would be called when an event occurs during parsing. The following is the list of SAX events:

- XML Text nodes
- XML Element nodes
- XML processing instructions
- XML Comments

These events are fired at the start and end of each XML node, instruction or comments whenever they are encountered. For example, at the start and end of a Text node, the XML Text nodes event will be fired; or XML Comments event will be fired at the start and end of comments. Note that SAX parsing is unidirectional, which means the previously parsed data cannot be read again, until the parsing operation is started again.

Q10. **What is XSLT?**

Ans. XSLT stands for Extensible Stylesheet Language Transformations, which is used for transforming the structure and content of XML document into the required output. XSLT transforms XML documents into other XML documents. XSLT processors parse the input XML document, as well as the XSLT stylesheet, and then process the instructions found in the XSLT stylesheet, using the elements from the input XML document. During the processing of the XSLT instructions, a structured XML output is created. XSLT instructions are in the form of XML elements, and use XML attributes to access and process the content of the elements in the XML input document. As XSLT converts the XML data into human readable format; therefore, it is used for displaying XML data in other formats, such as HTML and PDF.

Q11. **What is XPath?**

Ans. XPath is an expression language used for finding information in XML documents. XPath enables you to address specific parts of XML documents. XPath language is always used in conjunction with other XML technologies, for instance XSLT. It is used with XML applications, but have different syntax. XPath represents an XML document in the form of a tree, which is known as XPath Data Model. According to this model, an XPath document has seven nodes. These nodes are as follows:

- **Root node**—Represents the primary node of a tree in XPath Data Model. This node cannot occur anywhere else other than the root of the tree. All other nodes are always occurring in the Root node.
- **Element node**—Represents an element in the source XML document. An Element node may contain an ordered list of child Element nodes. The ordered list is useful when we want to access nodes according to their positions.
- **Attributes node**—Represents an attribute in the source XML file. The parent node of Attribute node is Element node
- **Namespaces node**—Represents a namespace that is in scope on the element in the source XML document represented by the parent Element node of the namespace node.
- **Processing-instruction node**—Represents a processing-instruction in the source XML document.
- **Comment node**—Represents a comment in the XML document.
- **Text node**—Represents the text content of an element.

Q12. What are node set functions of XPath?

Ans. XPath provides various functions that allow us to access a selection from the node set and return another node set. These functions are called as node set functions. Node set functions are generally used with those path expressions, which use predicates. Some examples of node set functions are: count(), last(), local-name(), name(), and namespace-uri().

Q13. What is lang() function in XPath?

Ans. The `lang()` function is used to test the language of the context node. It takes one argument of language, which is compared with the language that is determined by the relevant `xml:lang` attribute.

Q14. What is JAPX?

Ans. JAPX is acronym of Java API for XML Processing, which is a high-level API for writing vendor neutral applications that process XML. JAXP is used to process XML data by using applications built on the Java platform. JAXP provides an extra layer of adaptor around the vendor-specific parser and transformer implementations. With JAXP API, you can choose either Simple API for XML Parsing (SAX) parser or Document Object Model (DOM) parser to `parse` an XML document using a stream of events or using DOM object representation.

JAXP also supports the Extensible Stylesheet Language Transformations (XSLT) standard, which enables you to control the presentation of the data, and convert the data to other XML documents or other formats, for instance HTML. It also provides namespace support, which allows you to work with DTDs that might otherwise have naming conflicts.

Q15. What is XML Namespace?

Ans. An XML namespace enables you to differentiate elements and attributes of different XML document types from each other when combining them together into other documents, or even when processing multiple documents simultaneously. It is not necessary that every XML document have namespaces. Namespaces are optional components of basic XML documents. However, namespace declarations are recommended if your XML document is going to be shared with other XML documents that may share the same element names. Note that newer XML-based technologies, such as XML Schemas, SOAP, and WSDL, make heavy use of XML namespaces to identify data encoding types and important elements of their structure.

Q16. What is CDATA?

Ans. An XML parser parses the text data of an XML document. Parsed Character Data (PCDATA) is the term that is used for text data, which is normally parsed by an XML parser. However, Character Data (CDATA) is the term that is used for that text data, which is not parsed by the XML parser. For example, the characters, < and & are illegal in XML elements. This is because; the < character is interpreted by parser as the start of a new element; and the & character is interpreted as the start of a character entity. Therefore, a CDATA section is used in an XML document, which contains the data that could not be parsed by the parser. The syntax to use the CDATA is

```
<![CDATA[
...
...
...
]]>
```

Note that a CDATA section cannot contain the]]> string, and not nested CDATA sections.

PART 6
CREATING AJAX APPLICATIONS