

Supplement VI.C: XML

For Introduction to Java Programming

By Y. Daniel Liang

This supplement covers the following topics:

- Creating XML Documents
- Creating DTD
- XPath
- XSLT
- Oracle XSQL

12.1 Introduction

XML, eXtensible Markup Language, has become an Internet standard for data exchange on the Web. XML is similar to HTML in the sense that both are markup languages that annotate text. They both use the tags to describe documents. They are, however, very different in many ways. HTML is to describe the documents for display by Web browsers. It tells the Web browser how to render the documents visually. XML is for describing the contents of the data and their structural relationships. It is intended for use by programs to share and exchange data. XML is interoperable with HTML and can be transformed into HTML for display on Web browsers.

NOTE: Basic knowledge of HTML is required for this chapter. To learn HTML, please read Appendix VI.A, "HTML Tutorial."

Two main advantages of using XML are the following:

- XML is a metalanguage that enables it to describe itself and create its own tags. So XML is flexible and can create descriptive contents.
- XML is a text that enables it to be portable on all platforms. So XML data contents can be easily shared and exchanged on the Web.

In this chapter, you will learn how to describe data using XML, how to define XML documents using the document type definitions, how to transform XML into other documents using XSLT, and how to utilize XML to format and display query results and process database operations in Oracle.

12.2 Creating XML Documents

Let us begin with a simple example that demonstrates how to write an XML document, as shown in Listing 12.1.

Listing 12.1: Listing12_1.xml

*****PD: Please add line numbers in the following code*****

```
<?xml version = "1.0"?>
<!-- XML document for students -->
<students>
  <student num = "1">
    <ssn>444111110</ssn>
    <firstname>Jacob</firstname>
    <mi>R</mi>
    <lastname>Smith</lastname>
    <birthdate>4/9/1985</birthdate>
    <phone>9129219434</phone>
    <street>99 Kingston Street</street>
    <zipcode>31435</zipcode>
  </student>
  <student num = "2">
    <ssn>444111111</ssn>
    <firstname>John</firstname>
    <mi>K</mi>
    <lastname>Stevenson</lastname>
    <birthdate>4/9/1985</birthdate>
    <phone>9129219434</phone>
    <street>100 Main Street</street>
    <zipcode>31411</zipcode>
  </student>
</students>
```

As you see, the document is self-explanatory. It describes two students. The first line declares that it is an XML document. The second line is a comment. The students are contained in the tags `<students>` and `</students>`. The ssn, first name, mi, last name, birth date, street, and zip code of each student are described in the document.

The document can be stored as a text file with extension .xml by convention. The document itself contains data. You can use a program to process the data and display data in desired format. You will learn how to format the data for display in Section 12.4, "XML Stylesheet." Internet Explorer has a built-in tool to parse and display XML documents. If an XML document is syntactically correct, Internet Explorer displays it in the form of a hierarchical indexed list. The expand symbol in front of an item indicates that it contains subitems. You can see the subitems by clicking on the expand

symbol (+). Assume Listing 12.1 is stored in the file named listing12_1.xml. Figure 12.1 shows that student.xml is displayed in Internet Explorer.

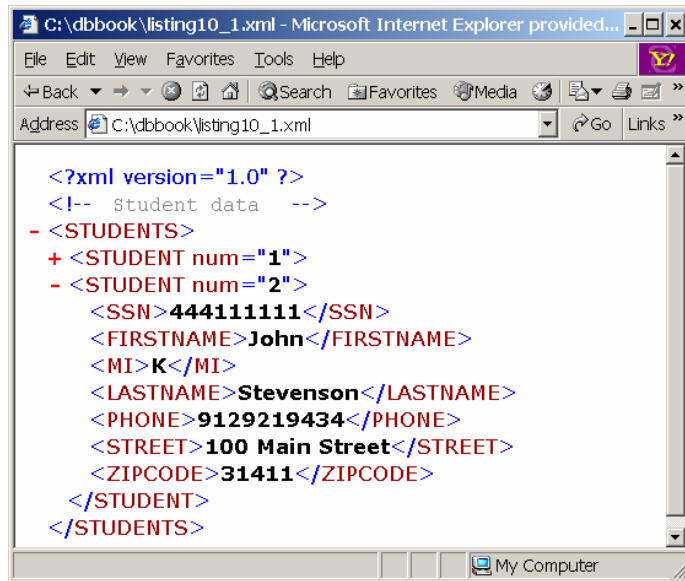


Figure 12.1

Internet Explorer can parse and display XML documents.

The following sections examine the syntax of XML using the preceding document in Listing 12.1 as example.

12.2.1 Declaration

Line 1 is an XML declaration to state that the document conforms to the XML version 1.0. The declaration is optional, but it is a good practice to use it. Otherwise, a document without the declaration may be assumed of a different version, which may lead to errors.

If an XML declaration is present, it must be the first item to appear in the document. This is because an XML processor looks for the first line to obtain information about the document so that it can be processed correctly.

12.2.2 Comments

Line 2 is a comment intended to make the XML source text easier to understand. XML comment always begins with `<!--` and end with `-->`. Comments cannot be inside another comment. For example, the following is wrong:

```
<!-- comment 1 <-- comment 2 --> -->
```

12.2.3 Tags

XML tags are used to describe the contents of the XML document. You can create your own tags. In the preceding example, students, student, ssn, firstname, mi, lastname, birthdate, street, and zipcode are the tags to describe students, student, ssn, fistName, mi, lastName, birthDate, and street. Obviously, you should choose descriptive names for the tags so that they can be easily understood.

Each tag in XML must be used in a pair of the starting tag and the closing tag. A starting tag begins with < followed by the tag name, and ends with >. A closing tag is the same as its starting tag except it begins with </. For example, <students> is a starting tag and </students> is a closing tag.

NOTE: The XML tag names are case-sensitive, whereas HTML tags are not. So, <students> is different from <students> in XML. Every starting tag in XML must have a matching closing tag, whereas some tags in HTML does not need closing tags.

12.2.4 Elements, the Root Element, and Leaf Elements

<side remark: root element>

<side remark: leaf element>

An XML document consists of elements described by tags. An element is enclosed between a starting tag and a closing tag. XML elements are organized in a tree-like hierarchy. Elements may contain subelements, but there is only one root element in an XML document. All the elements must be enclosed inside the root tag. For example, students is a root tag and all elements in the document are enclosed inside <students> and </students>. If an element does not contain other elements, it is called a *leaf element*.

12.2.5 Empty Elements

XML allows you to use empty elements as placeholders in the document. The syntax for an empty element is <tagName />. Listing 12.2 gives an example with empty elements.

Listing 12.2: Listing12_2.xml

*****PD: Please add line numbers in the following code*****

```
<?xml version = '1.0'?>
<!-- Use empty elements -->
<examples>
  <source />
  <example>
    An example here;
  </example>
</examples>
```

12.2.6 Attributes

Attributes provide additional information to describe elements. For example, the num attribute in Line 4 in Listing 12.1

```
<student num="1">
```

indicates that this is the first student in the student list.

12.2.7 Special Characters and Unicodes

The characters <, >, ", ', and & have special meanings in XML. For example, angle brackets are reserved for delimiting tags. To use these characters in the content of an element or attribute, you must use *entity references*, which begin with an ampersand (&) and end with a semicolon (;). The entity references for these characters are as follows:

- Less than sign (<) uses <.
- Greater than sign (>) uses >.
- Ampersand (&) uses &.
- Single quote or apostrophe (') uses '.
- Double quotation mark (") uses ".

You can also represent the Unicode characters in XML using the entity reference notation like &#abcd;, where abcd represents a four-hex digit Unicode character.

NOTE: The semicolon (;) at the end of the special characters and Unicode characters are required.

Listing 12.3 gives an example of using special characters and Unicodes in XML. It is displayed in Internet Explorer, as shown in Figure 12.2.

Listing 12.3: Listing12_3.xml

*****PD: Please add line numbers in the following code*****

```
<?xml version = '1.0'?>
<!-- Use Special Characters and Unicode Characters -->
<examples>
  <example num="1">
    &lt; &gt; &amp; &apos; &quot;
  </example>
  <example num="2">
    &#x00C4; &#x00D6; &#x00DC; &#x00DF;
    &#x00E4; &#x00F6; &#x00FC;
  </example>
  <example num="3">
    &#x03B1; &#x03B2; &#x03B3; &#x03B4;
    &#x03B5; &#x03B6; &#x03B7;
  </example>
</examples>
```

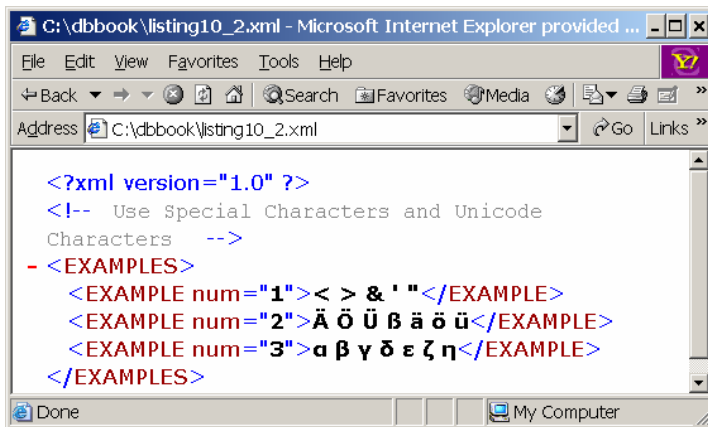


Figure 12.2

You can use special characters and Unicode in XML documents.

12.2.8 CDATA Sections

Suppose you need to write an XML document that contains a lot of special characters in the content. You would have to use a lot of entity references, which makes the document difficult to read. To avoid it, you can use the XML CDATA sections to contain the elements with the special characters without using the entity references. To use the CDATA section, enclose the element between <![CDATA[and]]>.

Listing 12.4 gives an example of using the CDATA sections.

The document is displayed in Internet Explorer, as shown in Figure 12.3.

Listing 12.4: Listing12_4.xml

*****PD: Please add line numbers in the following code*****

```
<?xml version = '1.0'?>
<!-- Use CDATA -->
<examples>
  <myxmldoc>
    <![CDATA[
      1 < 2 < 3 < 4 < 5 and 5 >= 5
    ]]>
  </myxmldoc>
</examples>
```

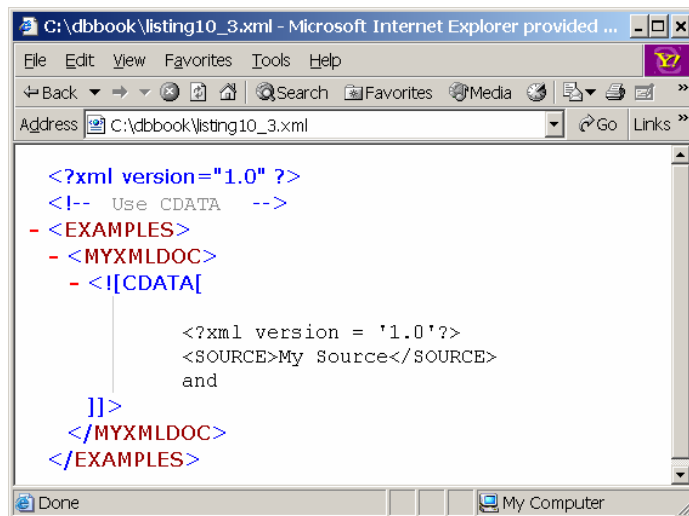


Figure 12.3

You can use the CDATA section to contain the elements with special characters.

12.2.9 Well-Formed XML Documents

An XML document is said to be *well-formed* if it is syntactically correct. The XML parser reads the XML document, checks its syntax, reports errors if any. The Internet Explorer has a built-in parser. When you display an XML document in Internet Explorer, the parser is invoked to check the document syntax. If the document is well-formed, it is displayed. Otherwise, Internet Explorer reports errors. For example, if you mistyped <students> to <student> in Listing 12.1, an error is displayed, as shown in Figure 12.4.

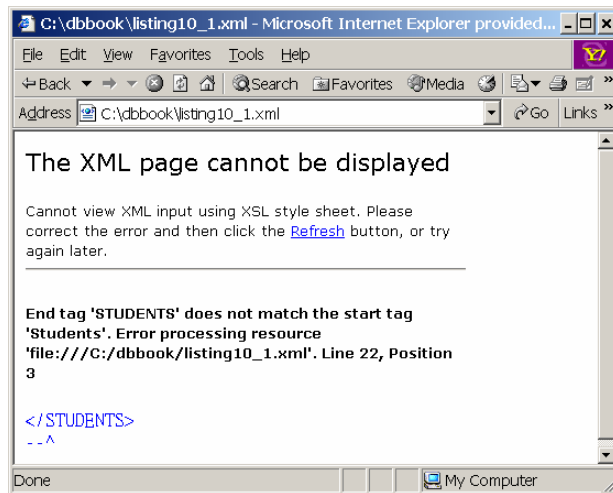


Figure 12.4

The XML parser checks if the document is well-formed.

12.3 Document Type Definition (Optional)

A document type definition (DTD) defines the types of elements and the order that can appear in the XML document. An XML document is not required to have a corresponding DTD, but a DTD can be used to ensure the documents of the same type conform to the same rules. This is particularly important when the XML documents are used in B2B (Business-to-business) and B2C (Business to customer) transactions. Listing 12.5 gives a simple example of DTD.

Listing 12.5: student.dtd

*****PD: Please add line numbers in the following code*****

```
<!DOCTYPE students [
  <!ELEMENT student (ssn, firstName, mi, lastname, birthdate,
    phone, street, zipcode)>
  <!ELEMENT ssn (#PCDATA)>
  <!ELEMENT firstName (#PCDATA)>
  <!ELEMENT mi (#PCDATA)>
  <!ELEMENT lastname (#PCDATA)>
  <!ELEMENT birthdate (#PCDATA)>
  <!ELEMENT phone (#PCDATA)>
  <!ELEMENT street (#PCDATA)>
  <!ELEMENT zipcode (#PCDATA)>
  <!ELEMENT students (student+)>
]>
```

Line 2 declares that the student element consists of subelements ssn, firstName, lastname, phone, street, and zipcode in this order. Line 4 declares that the ssn element is #PCDATA (parsable character data). Similarly, firstName,

lastname, phone, street, and zipcode are declared as #PCDATA in Lines 4-8. The last line declares that the students element consists one or more student elements.

The following sections introduce DTD in more details.

12.3.1 *Element Type Declarations*

You must declare every element you intend to use in a DTD. The general syntax to declare an element is:

```
<!ELEMENT elementName (elementDescription)>
```

This line declares the contents of the elementName in elementDescription.

12.3.1.1 *PCDATA*

You can describe the contents as parsable characters using PCDATA. For example, the following line declares that the ssn element consists of parsable characters. Parsable characters does not contain XML special characters like ≤, ≥, &, ", and '.

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

12.3.1.2 *The Order of Elements*

DTD allows you to use a comma (,) to specify the order of sub-elements for a parent element. For example, the following line declares that the student element consists of exactly ssn, firstName, mi, lastname, phone, street, and zipcode in this sequence.

```
<!ELEMENT student (ssn, firstName, mi, lastname, phone, street, zipcode)>
```

12.3.1.3 *The Selection of Elements*

DTD allows you to use a vertical bar (|) to specify the selection of sub-elements for a parent element. For example, the following line declares that the employee element is either faculty or staff, but not both.

```
<!ELEMENT employee (faculty | staff)>
```

12.3.1.4 *Repetitions of Elements*

DTD allows you to specify how many times an element may appear using the characters +, *, or ?. The *plus sign* (+)

indicates that the element can appear *one or more* times. The *asterisk sign* (*) indicates that the element can appear *zero or more* times. The *question mark* (?) indicates that the element can appear *zero or one* time.

For example, the following line declares that the department element contains one chair element, one or more student elements, zero or more faculty elements, zero or one staff element, and zero or one technician element.

```
<!ELEMENT department (chair, student+, faculty*, (staff, technician)?)>
```

NOTE: The elements can be grouped. In the preceding declaration, staff and technician are grouped.

12.3.1.5 EMPTY Elements

DTD allows you to specify an empty element using the keyword EMPTY. For example, the following line declares that the cup element is empty.

```
<!ELEMENT cup EMPTY>
```

12.3.1.6 ANY Elements

When developing a DTD in the early stage, sometime you don't have the exact definition in place; you can use the ANY keyword to specify that an empty element can contain any content. Later you should replace the ANY keyword with the exact definition. For example, the following line declares that the employee element can have any content.

```
<!ELEMENT employee ANY>
```

12.3.2 *Internal and External DTD*

How do you associate an XML document with a specified DTD? There are two ways: *internal DTD* and *external DTD*. An internal DTD is combined with an XML document in one file. An external DTD is placed in a separate file.

12.3.2.1 *Internal DTD*

An XML with an internal DTD has the following syntax:

```
<?xml version = '1.0'?>
<!DOCTYPE rootElement [
  <!-- Place DTD here -->
```

```

1>
<rootElement>
  <!-- Describe elements -->
</rootElement>

```

An example of an XML with an internal DTD is shown in Listing 12.6.

Listing 12.6: Listing12_6.xml

*****PD: Please add line numbers in the following code*****

```

1 <?xml version = '1.0'?>
2 <!DOCTYPE students [
3   <!ELEMENT student (ssn, firstName, mi, lastname, birthdate,
4     phone, street, zipcode)>
5   <!ELEMENT ssn (#PCDATA)>
6   <!ELEMENT firstName (#PCDATA)>
7   <!ELEMENT mi (#PCDATA)>
8   <!ELEMENT lastname (#PCDATA)>
9   <!ELEMENT birthdate (#PCDATA)>
10  <!ELEMENT phone (#PCDATA)>
11  <!ELEMENT street (#PCDATA)>
12  <!ELEMENT zipcode (#PCDATA)>
13  <!ELEMENT students (student+)>
14 ]>
15 <students>
16   <student>
17     <ssn>4441111110</ssn>
18     <firstname>Jacob</firstname>
19     <mi>R</mi>
20     <lastname>Smith</lastname>
21     <birthdate>4/9/1985</birthdate>
22     <phone>9129219434</phone>
23     <street>99 Kingston Street</street>
24     <zipcode>31435</zipcode>
25   </student>
26   <student>
27     <ssn>4441111111</ssn>
28     <firstname>John</firstname>
29     <mi>K</mi>
30     <lastname>Stevenson</lastname>
31     <birthdate>4/9/1985</birthdate>
32     <phone>9129219434</phone>
33     <street>100 Main Street</street>
34     <zipcode>31411</zipcode>
35   </student>
36 </students>

```

12.3.2.1 External DTD

An internal DTD is used only once in an XML document. An external DTD can be shared by many XML documents. Suppose you have stored Listing 12.5 as an external DTD in the file named `student.dtd`. Instead of including the entire DTD in the XML file, you can use the following syntax to reference the external DTD:

```
<!DOCTYPE rootElement SYSTEM "student.dtd"> or
```

```
<!DOCTYPE rootElement PUBLIC "student.dtd">
```

The keyword `SYSTEM` and `PUBLIC` indicate that an external DTD is used. `SYSTEM` is used to indicate a DTD is located on the local machine and restricted to a user or a group of users, and `PUBLIC` indicates that the external DTD can be accessed by the public. Often you use `PUBLIC` to access an external DTD from a Website as shown in the following example:

```
<!DOCTYPE students PUBLIC  
"http://www.cs.armstrong.edu/liang/intro6e/book/student.dtd">
```

12.3.3 Validating XML Documents

You can use a software, called *XML DTD validator*, to validate whether an XML document conforms to its DTD. Many validators are available free. Some XML parsers have the capabilities to validate DTD. Such XML parsers are known as *validating parsers*. The parser in Internet Explorer, however, is nonvalidating. You can use Microsoft FrontPage to edit and validate XML documents, as shown in Figure 12.5.

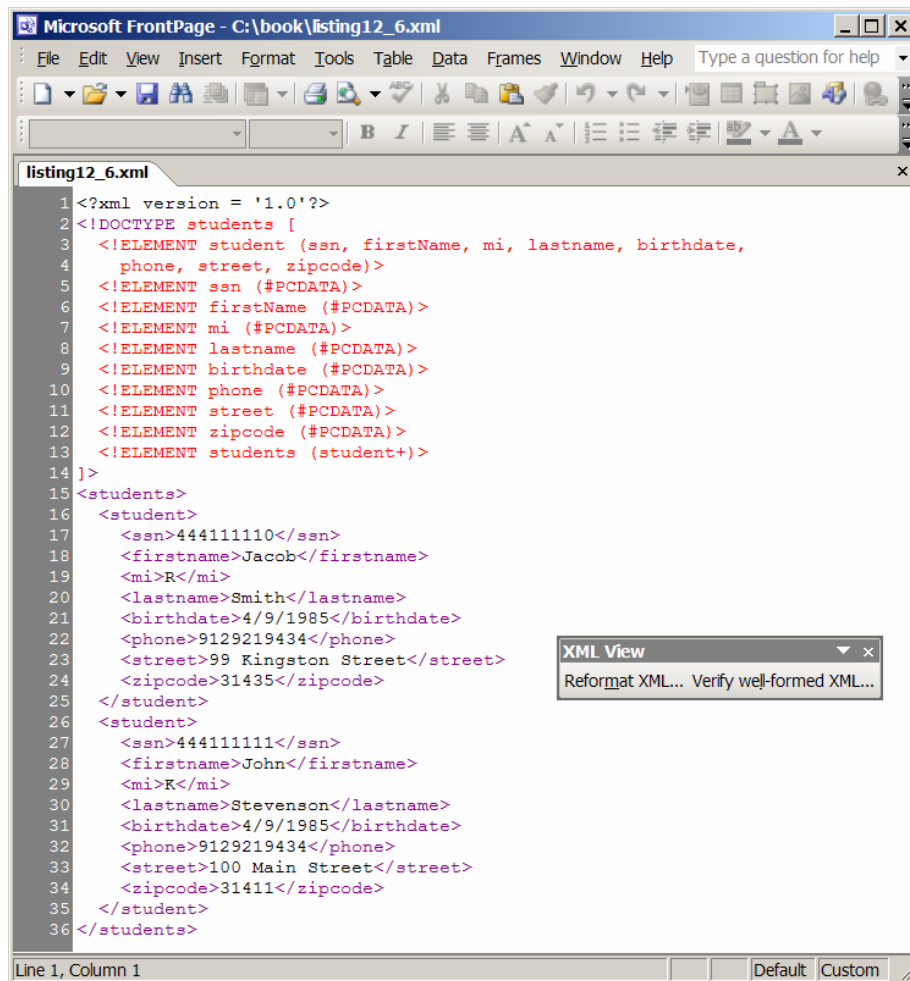


Figure 12.5

You can use Microsoft FrontPage to edit and validate XML documents.

NOTE: By definition, a valid XML document is also well-formed, but a well-formed document may not be valid.

12.3.4 Entities

You have learned to use the entity references for the five special characters (<, >, ", ', and &) in XML. You can define your own entity references using the <!ENTITY> tag.

12.3.4.1 Internal Entities

You can define entity references inside a DTD as shortcuts. For example, Listing 12.7 defines two shortcuts for

"Computer Science" and "Mathematics." CS is defined as an entity reference for Computer Science, and MATH is for Mathematics. The entity are referenced using the notation `&CS;` and `&MATH;` in the XML document. The parser replaces the entity reference with the actual value. Listing 12.7 is displayed as shown in Figure 12.6.

Listing 12.7: Listing12_7.xml

*****PD: Please add line numbers in the following code*****

```
<?xml version = '1.0'?>
<!DOCTYPE school [
  <!ELEMENT school (department+)>
  <!ELEMENT department (#PCDATA)>
  <!ENTITY CS "Computer Science">
  <!ENTITY MATH "Mathematics">
]>
<school>
  <department>
    &CS;
  </department>
  <department>
    &MATH;
  </department>
</school>
```

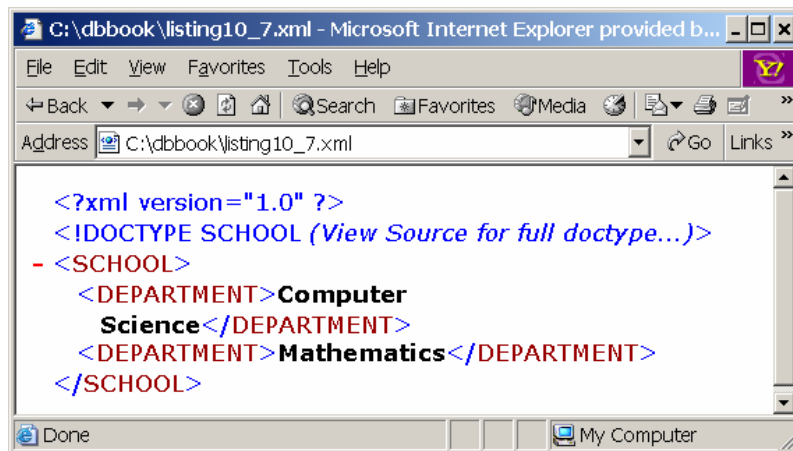


Figure 12.6

The entity references are replaced by the entity value in the browser.

NOTE: In Listing 12.7, entities are defined inside a DTD within an XML document. The entity can only be referenced in the XML document. You can store the DTD externally so that the entity definitions can be shared by other XML documents.

12.3.4.2 External Entities

You can declare an entity reference for a whole document stored in a separate file. For example, Listing 12.8 declares `myStudent` to reference the document `Student.xml`. `Student.xml` was given in Listing 12.1. The entity is referenced using the notation `&myStudents;` in the XML document. The parser replaces the entity reference with the external document. Listing 12.8 is displayed as shown in Figure 12.7.

Listing 12.8: Listing12_8.xml

*****PD: Please add line numbers in the following code*****

```
<?xml version = '1.0'?>
<!DOCTYPE school [
  <!ELEMENT school (department+)>
  <!ELEMENT department (#PCDATA)>
  <!ENTITY CS "Computer Science">
  <!ENTITY MATH "Mathematics">
  <!ENTITY myStudents SYSTEM "Student.xml">
]>
<school>
  &myStudents;
  <department>
    &CS;
  </department>
  <department>
    &MATH;
  </department>
</school>
```

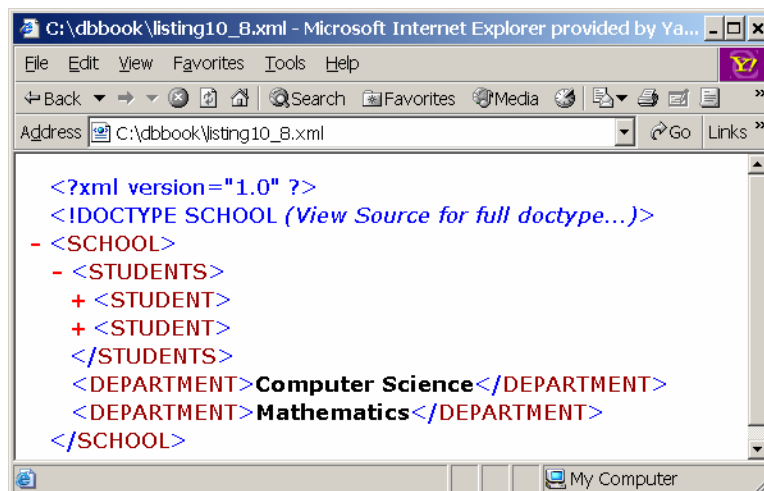


Figure 12.7

The entity references are replaced by the external document in the browser.

12.3.4.3 Unparsed Entities

Not all data exist in XML format. To incorporate non-XML data into your XML document, you can use unparsed entities. For example, the following declaration specifies that the external data is non-XML.

```
<!ENTITY nonXMLEntity SYSTEM "description.html" NDATA html>
```

The keyword NDATA indicates that the content of the external file is non-XML.

XML does not recognize any non-XML format. For the HTML files to be recognized as an external file, you have to declare it using the <!NOTATION> tag as follows:

```
<!NOTATION html SYSTEM "iexplorer">
```

This syntax declares that html files can be unparsed and the program that handles html files is IExplorer.

12.3.5 Attribute Declarations

You learned how to write DTD that declares elements. An XML document may also contain attributes. For example, num in the following line is an attribute.

```
<student num="1">
```

Attributes are subject to DTD rules. You must declare every attribute you intend to use in a DTD. The general syntax to declare an element is:

```
<!ATTLIST elementName attributeName dataType defaultValue>
```

The elementName is the name for which the attribute is to be used. The attributeName is the name of the declared attribute. The dataType defines the data type for the attribute.

12.3.5.1 Attribute Defaults

DTD allows you to specify default values for attributes using the keywords #REQUIRED, #IMPLIED, and #FIXED. #REQUIRED indicates that the attribute must appear in the element. The document is invalid if a required attribute is missing. #IMPLIED indicates that the attribute is optional in the element. #FIXED specifies that the attribute value is fixed in the element, which means any two values of the same

fixed attribute must be same in the element.

12.3.5.2 CDATA Attributes

CDATA declares that attribute can contain any character except the special characters <, >, ", ', and &.

12.3.5.3 NMTOKEN and NMTOKENS Attributes

NMTOKEN is the same as CDATA except that an NMTOKEN value must start with a letter or an underscore. NMTOKENS can be used to specify values separated by spaces.

12.3.5.4 ID, IDREF, and IDREFS Attributes

ID can be used to specify that the attribute value is unique for the element in the document. IDREF and IDREFS point to the element(s) with a given ID attribute value, which is already given in the document. IDREF and IDREFS provide a link between elements through XML tree structure. For example, Listing 12.9 shows an XML with the DTD that uses the attributes ID, IDREF and IDREFS.

Listing 12.9: Listing12_9.xml

*****PD: Please add line numbers in the following code*****

```
<?xml version = '1.0'?>
<!DOCTYPE teaching [
  <!ELEMENT teaching (faculty, course+)>
  <!ELEMENT faculty (#PCDATA)>
  <!ATTLIST faculty facultyID ID #REQUIRED>
  <!ATTLIST faculty X CDATA #IMPLIED>
  <!ELEMENT course (#PCDATA)>
  <!ATTLIST course taughtby CDATA #IMPLIED>
]>
<teaching>
  <faculty facultyID = "F1">
    Steve Jones
  </faculty>
  <course taughtby = "F1">
    Database Systems
  </course>
</teaching>
```

In Listing 12.9, facultyID is unique. If you add another faculty element with the same facultyID, the document is not valid.

NOTE: Since ID attribute value is unique for the elements, you cannot declare ID as #FIXED.

12.3.5.5 ENTITY and ENTITIES Attributes

ENTITY can be used to specify that the attribute value is an unparsed entity or entities. For example,

```
<!ENTITY myText "This is my text">
```

declares that myText is an entity reference for "This is my text." This entity might be used as follows:

```
<newText>&myText</newText>
```

The ENTITIES type can be used to indicate that an attribute may have multiple entities for its value. For example

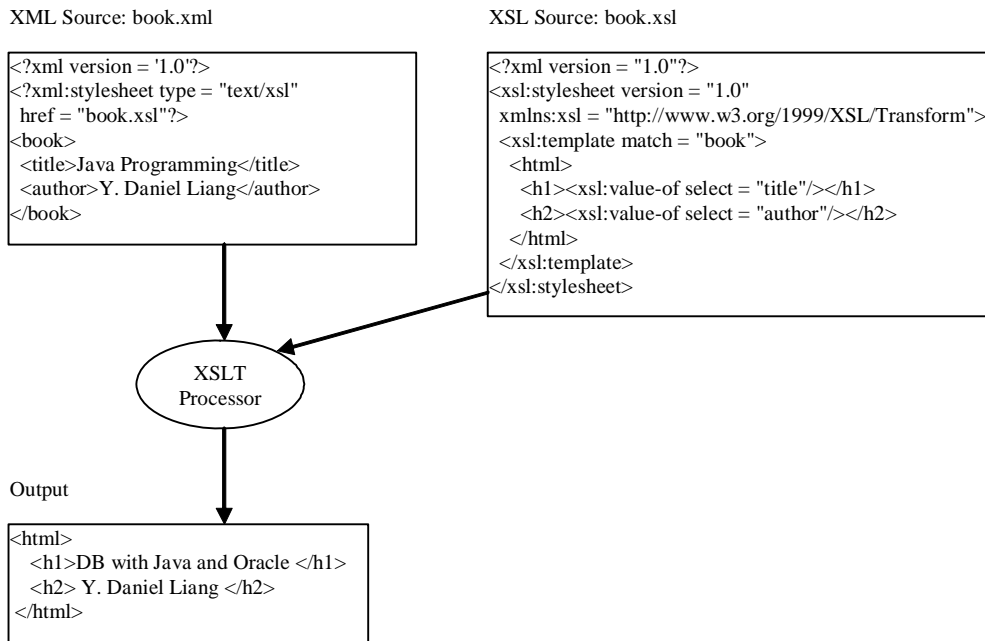
```
<!ENTITY department DOCUMENT ENTITIES>
```

specifies that DOCUMENT attribute contains multiple entities. An XML markup that conforms to this might look like this:

```
<department DOCUMENT = "Doc1 Doc2 Doc3">
```

12.4 XSLT

A key advantage of using XML is that its presentation can be separate from its content. An XML document can be formatted using a stylesheet. The stylesheet defines how an XML document is presented. The same XML document can be presented in many ways using different stylesheets. The program that translates an XML using a stylesheet is known as an *XSLT(eXtensible Stylesheet Translation) processor*, as shown in Figure 12.8.

**Figure 12.8**

The XSLT processor translates the original XML document into a new XML document.

To attach a stylesheet to an XML document, use the stylesheet declaration in the XML document:

```
<?xml:stylesheet type = "text/xsl" href = "book.xsl"?>
```

The type attribute specifies the stylesheet type. There are two types: XSL and CSS (Cascading Stylesheet). Cascading stylesheet originates from HTML. You can use cascading stylesheet in both HTML and XML. For information on CSS, please see Supplement IV.B. The href attribute points to the stylesheet file.

Let us look at the XSL source in Figure 12.8. An XSL stylesheet itself is an XML document. The name space **http://www.w3.org/1999/XSL/Transform** is required to identify the XSL elements in the stylesheet. All the XSL tags begin with prefix xsl. The root element for an XSL document is always xsl:stylesheet. The <xsl:template match = "book"> element contains the processing instructions for translating the XML source for the book element. The <xsl:value-of select = "title"/> element returns the value of the title element under the book element. Similarly, The <xsl:value-of select = "author"/> element returns the value of the author

element in the `book` element. So, it generates the output as shown in Figure 12.8. You will learn in details how to write the XSL stylesheet and how the translation is processed in following sections.

NOTE: Internet Explorer contains an XSLT processor, which translates the XML source to another XML document using the stylesheet. If the output is HTML, it renders it using the Internet Explorer HTML processor, as shown in Figure 12.9.

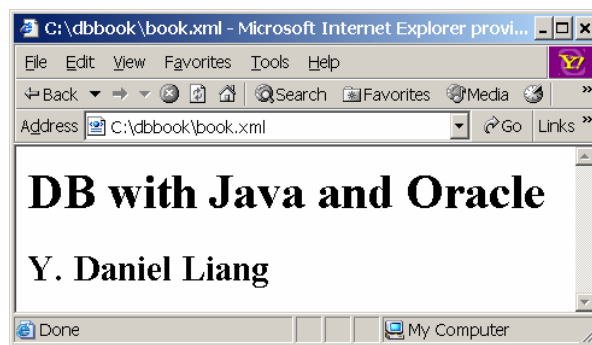


Figure 12.9

Internet Explorer translates the XML document and renders the HTML content in the browser.

12.4.1 The Stylesheet Structure

In general, a stylesheet has the following syntax:

```
<?xml version = "1.0"?>
<xsl:stylesheet version = "1.0"
  xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
  <xsl:template match = pattern1>
    [actions1]
  </xsl:template>
  <xsl:template match = pattern2>
    [actions2]
  </xsl:template>
  ...
</xsl:stylesheet>
```

A stylesheet consists of templates. Each template has its matching pattern specified using XPath expression. XPath is introduced in the next section. For the element(s) in the XML source document that match the pattern, actions are used to produce the output.

12.4.2 The XPath Tree

Before you learn how to write XSL stylesheets, you need to know the *XPath tree*, which conceptually models the underlying structure for the XML document. When the XML parser processes an XML document, it creates an XPath tree for the document. All the components in an XML are represented as nodes to mirror the structural relationships of the components in the XML document. Figure 12.10 shows the XPath tree of the XML document in Listing 12.1.

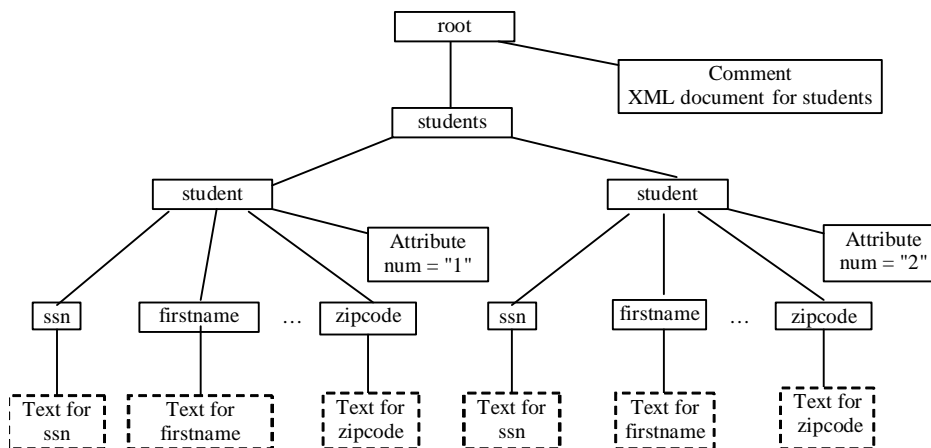


Figure 12.10

The XML parser parses an XML document into an XPath tree.

An XPath tree has seven types of nodes: *root*, *element*, *attribute*, *text*, *comment*, *processing instruction* and *name space*. There is only one root node that contains all the components in an XML document. Each element, attribute, comment, text, processing instruction, and name space in an XML document corresponds to a node in an XPath tree. XPath trees are usually generated by an XML parser if the XML document is well-formed.

NOTE: The order of the nodes in an XPath tree mirrors the order of their corresponding components in the XML document. For example, If element A is a subelement of B in the XML document, the node for A is a child node of the node for B. If element A appears before element B in the XML document, then the node for A is on the left side of the node for B in the XPath tree.

XSLT processor uses an XPath tree to traverse and locate

components in order to match the pattern for templates. Once a node in the tree is found to match the pattern, the corresponding element in the XML is processed using the actions specified for the pattern in the XML stylesheet.

A pattern is expressed using a *location path*, which is an expression that specifies how to locate a node or nodes. It has the following syntax:

axis::nodeset[predicate]

In most cases, axis and predicate are omitted.

12.4.2.1 Axis

The nodes in the XPath are traversed in some order. The current node being visited is called the "context node." An *axis* indicates which direction to traverse from the context node. Table 12.1 lists the axes.

Table 12.1: XPath Axes

Axis Name	Direction	Description
child	Forward	The context node's children backward
parent	Backward	The context node's parent backward
descendant	Forward	The context node's descendants
ancestor	Backward	The context node's ancestor
attribute	Forward	The context node's attribute
namespace	Forward	The context node's namespace
following	Forward	The nodes in the XML document following the context node, not including descendants
following-sibling	Forward	The sibling nodes following the context node
preceding	Backward	The nodes in the XML document preceding the context node, not including ancestors
preceding-sibling	Backward	The sibling nodes preceding the context node, not including ancestors
self	None	The context node itself
descendant-or-self	Forward	The context node's descendant and also itself
ancestor-or-self	Backward	The context node's ancestor and also itself

NOTE: An axis has a *principal node type* that corresponds to the nodes in the axis. All the axes have the *element principal node type*, except the attribute axes and namespace axes.

12.4.2.2 Node Set Test

An axis selects a set of nodes from the XPath tree. You can narrow the selection based on the types of the nodes. Table 12.2 lists node set tests.

Table 12.2: Node Set Test

Node Set Test	Description
*	Selects all nodes of the same principal node type.
node()	Selects all nodes, regardless of their type.
text()	Selects all text nodes.
comment()	Selects all comment nodes.
processing-instruction()	Selects all processing-instruction nodes.
nodename	Selects all nodes with the specified name.

The location path in the following template specifies a pattern child::*, which matches all the non-attribute and non-namespace child nodes of the current context, since the child axis is of the element principal node type.

```
<?xsl:template match = "child::*">
  [actions]
</xsl:template>
```

The child axis can be omitted. So child::* is equivalent to *.

The location path

```
child::comment()
```

selects all text child nodes.

<side remark: Node Set Operator>

XPath provides the | (the pipe character) operator to perform the union of two nodes sets, and the / operator to specify location steps. It also provides the shorthand operators //, @, ., and .. to simplify notations. The // operator is the same as /descendant-or-self::node()/, the @ operator is the same as attribut::, and the . operator is same as self::node(), and the .. operator is the same as parent::node(). Table 12.3 summarizes these operators.

Table 12.3: Node Set Operators

Node Set Operator	Description
	Performs union of two node sets.
/	Separates location paths.
//	Same as /descendant-or-self::node()/
@	Same as attribut::
.	Same as self::node()
..	Same as parent::node()

For example,

```
"student/firstName | student/lastname"
```

specifies the node set firstName and lastname of a student node.

"*/firstName" specifies all firstName at the child level of the current node.

"student/*" specifies any child element of a student node.

"students//*" specifies any descendants of students.

NOTE: Use the slash character alone means the root element in the tree.

12.4.2.3 Predicate

An axis selects a set of nodes from the XPath tree. The node set test narrows the selection. You can use the predicate to further narrow the selection. For example, there are two student nodes in the XPath tree in Figure 12.10. These two nodes would match the following template:

```
<xsl:template match = "child::student">
  [actions]
</xsl:template>
```

If you want only the first student node to match the template, you could apply a restriction on the attribute using a predicate as follows:

```
<xsl:template match =
  "child::student[attribute::num = '1']">
  [actions]
</xsl:template>
```

The predicate **attribute::num = '1'** specifies that attribute num of the student element is 1. As you see, the predicate provides an additional restriction on the node set. A predicate is a Boolean expression that results in true or false. You can use the relational operators in Table 12.4 in the predicates.

Table 12.4: Operators in Predicates

Operator	Description
----------	-------------

<u>=</u>	equal
<u>!=</u>	not equal
<u><</u>	less than
<u>></u>	greater than
<u><=</u>	less than or equal to
<u>>=</u>	greater than or equal to

You can also use node-set functions in Table 12.5 and string functions in Table 12.6 in the predicates. For example,

```
child::student[position() = 2]
```

specifies the second student element. This expression can be shorted as simply

```
child::student[2]
```

Table 12.5: Node-Set Functions

Node Set Function	Description
count(node-set)	Returns the number of nodes in the specified node-set.
position()	Returns the position number of the current node in the selected node set.
last()	Returns the position number of last node in the current node set.
id(string)	Returns the element node whose ID attribute matches the value specified by argument <i>string</i> .

Table 12.6: String Functions

String Function	Description
concat(str1, str2, ..., strn)	Concatenates various strings into one.
starts-with(str1, str2)	Returns true if <i>str1</i> starts with <i>str2</i> .
contains(str1, str2)	Returns true if <i>str1</i> contains <i>str2</i> .
string-length(str)	Returns the number of characters in the string.

12.4.3 How Does an XSLT Processor Work?

The XSLT processor processes the XML document by applying the template actions on the nodes in the XPath tree. The nodes of the element, text, comment, and processing types are traversed in depth-first order. For example, the nodes in Figure 12.11 are traversed as follows: A, B, D, I, J, K, E, F, L, C, G, H, M, N, O.

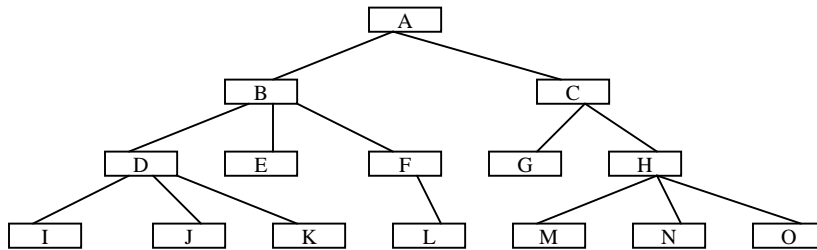


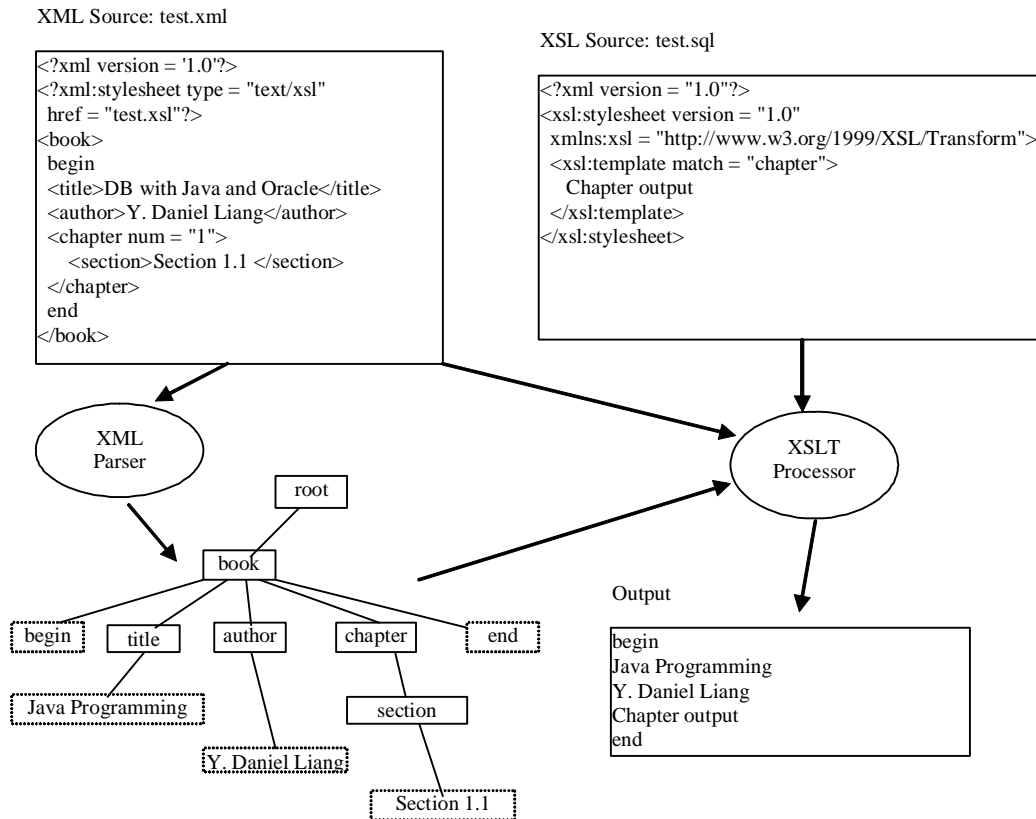
Figure 12.11

The nodes in the XPath tree are traversed in depth-first order for processing by the XSLT processor.

For each node traversed, the XSLT checks if the node matches a pattern in the stylesheet. If a matching pattern is found, the actions for the pattern are processed and the descendants of this node are skipped (not traversed). If a text node is traversed, the text is output to the result.

NOTE: Only the nodes of the element, text, comment, and processing types are processed. The attribute nodes and namespace nodes are ignored.

Let's demonstrate the process using Figure 12.12. The XML parser parses the XML document to generate the XPath tree. The XSLT processor traverses the root first. Since there is no match for the root, it traverses the text node, so the text "begin" is sent to the output. The node title is traversed, since there is no match for this node, its child text node is traversed, the text "Java Programming" is sent to the output. The author node is traversed. Since there is no match for this node, its child text node is traversed, the text "Y. Daniel Liang" is sent to the output. The chapter node is traversed. It matches the pattern in the template, the action for the pattern is processed and the text "Chapter output" is sent to the output. Finally, the text "end" is sent to the output.

**Figure 12.12**

The XSLT processor uses the XPath tree and templates to translate XML documents.

12.4.3.1 Resolving Conflict

What happens if a node matches two or more patterns in the stylesheet? In theory, it is not allowed. The XSLT processor should report an error in the stylesheet. However, Internet Explorer does not report any error. Internet Explorer resolves the conflict by applying the last template that matches the node in the stylesheet. For example, `actions2` is used to process the `ssn` node in the XML document in Listing 12.1 if the following stylesheet is used:

```
<xsl:template match = "student/ssn">
  [action1]
</xsl:template>

<xsl:template match = "student/ssn">
  [action2]
</xsl:template>
```

12.4.4 Actions

Once a node matches a pattern, the XSLT processor performs the actions in the template to process the node. This section introduces the syntax for actions.

12.4.4.1 The `<xsl:apply-templates>` Tag

The `<xsl:apply-templates select = "nodeset">` tag tells the XSLT processor to process the selected nodeset. `<xsl:apply-templates/>` is an empty tag, which processes the current node. For example, Figure 12.13 shows the effect of adding this tag in test.xml.

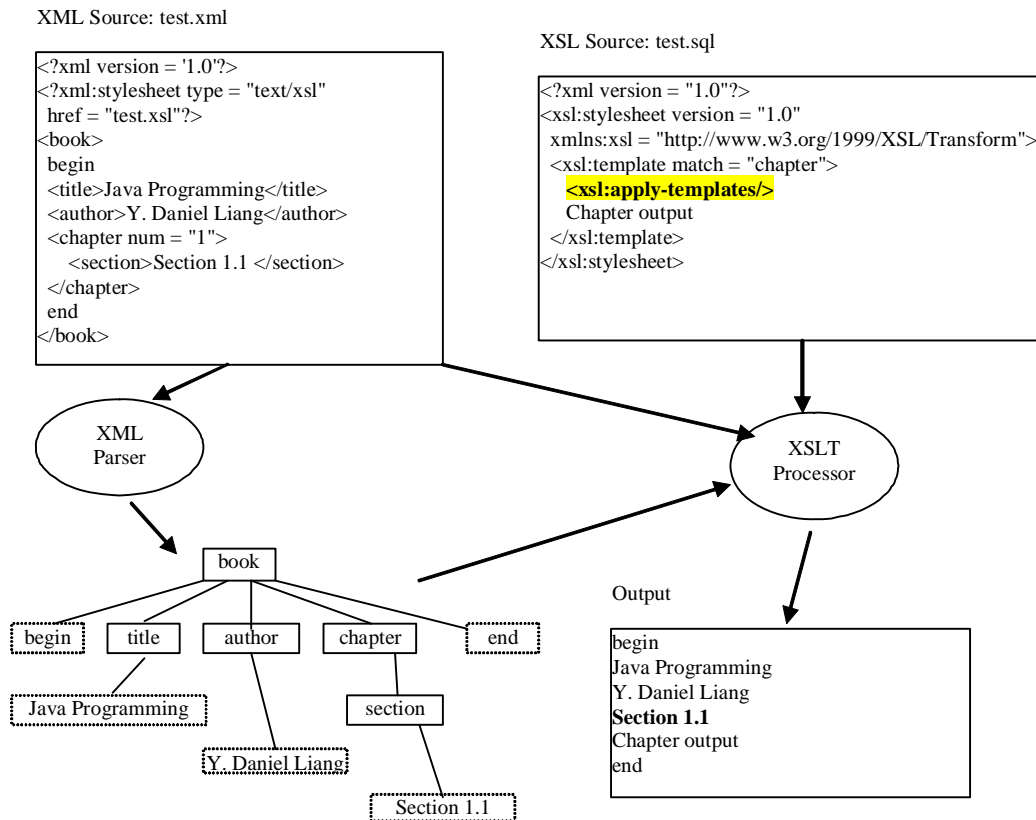


Figure 12.13

The `<xsl:apply-templates/>` tag is for processing the current node and its descendants.

12.4.4.2 The `<xsl:text>` Tag

The `<xsl:text>` tag is used to enter text string. This tag can be omitted. For example, the following two stylesheets in Figure 12.14 are equivalent.

```

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

```

```

<?xml version = "1.0"?>
<xsl:stylesheet version = "1.0"
xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
<xsl:template match = "chapter">
  <xsl:text>Chapter output</xsl:text>
</xsl:template>
</xsl:stylesheet>

```

Figure 12.14

The <xsl:text> tag can be omitted for entering text strings.

12.4.4.3 The <xsl:value-of select = exp/> Tag

The <xsl:value-of select = exp/> tag is for selecting a text and attribute value from the specified XPath expression in the XML source. For example,

```
<xsl:value-of select = "SECTION"/>
```

selects the text in the SECTION element.

```
<xsl:value-of select = "@NUM"/>
```

selects the value the NUM attribute.

12.4.4.4 The <xsl:if test = "condition"> Tag

The <xsl:if = condition> tag has the following syntax:

```

<xsl:if = condition>
  [actions]
</xsl:if>

```

It checks the condition. If true, perform the actions.

The condition is an XPath predicate. For example, the condition @NUM = 2 in following template checks is whether the NUM attribute for the student node is 2. If so, select the text values of the ssn, firstName, and lastname nodes.

```

<xsl:template match = "student">
  <xsl:if test = "@NUM = '2'">
    <xsl:value-of select = "ssn"/>
    <xsl:value-of select = "firstName"/>
    <xsl:value-of select = "lastname"/>
  </xsl:if>

```

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

12.4.4.5 The <xsl:choose> Tag

The <xsl:choose> tag has the following syntax:

```
<xsl:choose>
  <xsl:when test = "condition1">action1</xsl:when>
  <xsl:when test = "condition2">action2</xsl:when>
  ...
  <xsl:otherwise>otheraction</xsl:otherwise>
</xsl:choose>
```

It tests multiple conditions in the order specified in the <xsl:choose> element. If a condition is true, its corresponding action is performed and the rest of the conditions are skipped. The <xsl:choose> tag, which is optional, covers all the cases not specified in the <xsl:when> tags.

For example, the following <xsl:choose> tag tests whether the NUM attribute of the student node is 1 or 2. If it is 1, select the text value of the firstname node of the student node. If it is 2, select the text value of the lastname node of the student node. Otherwise, output the text No Match.

```
<xsl:template match = "student">
  <xsl:choose>
    <xsl:when test = "@num = '1'">
      <xsl:value-of select = "firstname"/>
    </xsl:when>
    <xsl:when test = "@NUM = '2'">
      <xsl:value-of select = "lastname"/>
    </xsl:when>
    <xsl:otherwise>
      No match
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

12.4.4.6 The <xsl:for-each select = "node-set"> Tag

The <xsl:for-each select = "node-set"> tag has the following syntax:

```
<xsl:for-each select = "node-set">
  [action]
</xsl:for-each>
```

It performs the action repeatedly for the matching node set.

For example, the following template matches the students node and looks for each student/firstName node to display its firstName text value. Here is an HTML tag for list.

```
<xsl:template match = "students">
  <xsl:for-each select = "student/firstName">
    <li>
      <xsl:value-of select = "self::firstName"/>
    </li>
  </xsl:for-each>
</xsl:template>
```

12.4.4.7 The <xsl:sort> Tag

The <xsl:sort> tag has the following syntax:

```
<xsl:for-each select = "node-set" order = "order"
  lang = "language" data-type = "data-type"
  case-order = "case-order"/>
```

It can be used with a <xsl:for-each select = "node-set"> loop or a <xsl:apply-templates> to sort the node set selected in the loop. The select attribute is required to specify on what key to sort. All other attributes are optional. The order attribute can be ascending or descending, which specifies whether the key strings should be sorted in ascending or descending order. By default, it is ascending. The lang attribute specifies the language of the sort keys. If no lang value is specified, the language should be determined from the system environment. The available values for lang are the language code, that is to say, one of the lowercase two-letter codes defined by ISO-639. See the Website www.computing.armstrong.edu/liang/iso639.html for a list of the language codes. The data-type attribute has two possible values (text and number), which specifies the data type whether is string or number. By default, it is a string. The case-order attribute has two possible values (upper-first and lower-first), which specifies whether upper-case letters are be sorted before lower-case letters. The default is language dependent. For the English language, the default is upper-first.

For example, the following template matches the students node and looks for each student node to output its firstName and lastname text values in descending order of the lastname values.

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

```

<xsl:for-each select = "student">
  <xsl:sort select = "lastname"
    order = "descending"/>
  <li>
    <xsl:value-of select = "firstname"/>
    <xsl:value-of select = "lastname"/>
  </li>
</xsl:for-each>
</xsl:template>

```

NOTE: You can use multiple `<xsl:sort>` tags to specify the primary sort key, the secondary sort key, and so on.

NOTE: For complete reference on XSLT, please visit <http://www.w3.org/TR/xslt>.

12.4.4.8 Presenting XML Using Stylesheets

An XML document can be presented in various ways, both in appearance and organization, simply by applying different stylesheets. This section demonstrates presenting the student XML document in Listing 12.1 using two different stylesheets.

12.4.4.8.1 Displaying the Student XML in a Table

The first stylesheet, shown in Listing 12.10, displays ssn, First Name, mi, and Last Name from the XML document in a table, as shown in Figure 12.15. The stylesheet has a single template to match the `students` element node in the XML document. The action of the template sorts all `student` elements in descending order of their `lastname` and generates an HTML table.

Listing 12.10: stylesheet1.xsl

*****PD: Please add line numbers in the following code*****

```

<?xml version = "1.0"?>
<xsl:stylesheet version = "1.0"
  xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
  <xsl:template match = "students">
    <html><body><center><table border = "1">
      <tr><th>ssn</th><th>First Name</th>
      <th>mi</th><th>Last Name</th></tr>
      <xsl:for-each select = "student">
        <xsl:sort select = "lastname" order = "descending"/>
        <tr>
          <td>
            <xsl:value-of select = "ssn"/>
          </td>

```



```

        <td>
        <xsl:value-of select = "firstName"/>
        </td>
        <td>
        <xsl:value-of select = "mi"/>
        </td>
        <td>
        <xsl:value-of select = "lastname"/>
        </td>
    </tr>
</xsl:for-each>
</table></center></body></html>
</xsl:template>
</xsl:stylesheet>

```

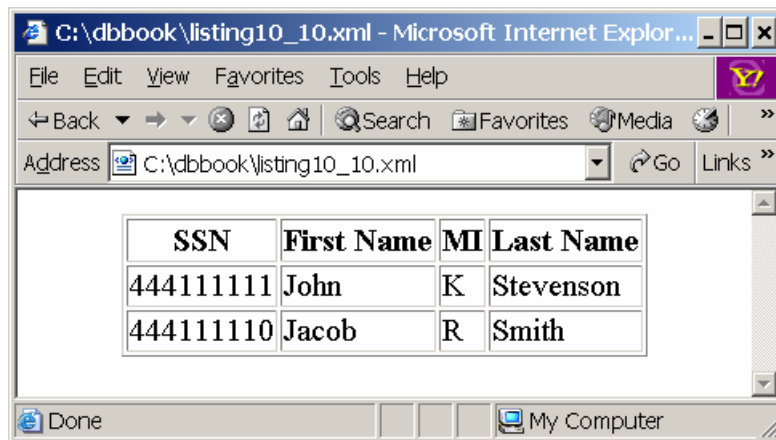


Figure 12.15

The contents in the XML document are displayed in a table.

12.4.4.8.1 Displaying the Student XML in a List

The second stylesheet, shown in Listing 12.11, displays `ssn`, `First Name`, `mi`, and `Last Name` from the XML document in a list, as shown in Figure 12.16. The stylesheet has a single template to match the `students` element node in the XML document. The action of the template sorts all `student` elements in ascending order of their `lastname` and generates an HTML list.

Listing 12.11: stylesheet2.xsl

*****PD: Please add line numbers in the following code*****

```

<?xml version = "1.0"?>
<xsl:stylesheet version = "1.0"
  xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
  <xsl:template match = "students">
    <html><body>
      <h2>Student List</h2>
      <xsl:for-each select = "student">
        <xsl:sort select = "lastname" order = "ascending"/>

```

```

<li>
  <xsl:value-of select = "ssn"/>
  <xsl:text> </xsl:text>
  <xsl:value-of select = "firstName"/>
  <xsl:text> </xsl:text>
  <xsl:value-of select = "mi"/>
  <xsl:text> </xsl:text>
  <xsl:value-of select = "lastName"/>
</li>
</xsl:for-each>
</body></html>
</xsl:template>
</xsl:stylesheet>

```

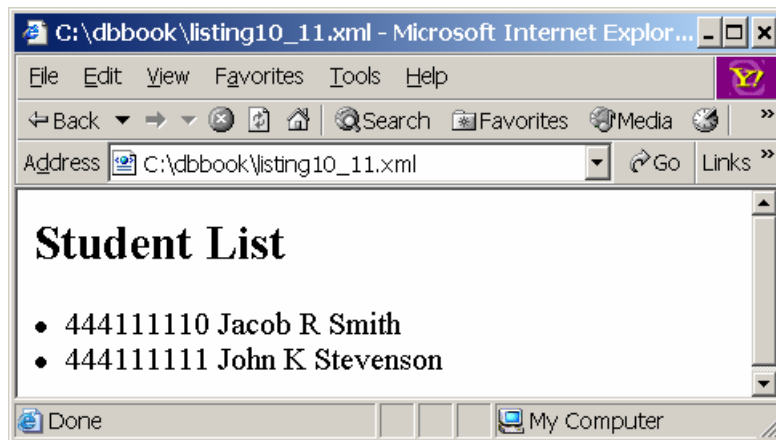


Figure 12.16

The contents in the XML document are displayed in a list.

NOTE: Often the stylesheet contains only one template as shown in this example. In this case, you can write a stylesheet without explicitly using the `<xsl:template match>` tag, as shown in Listing 12.12.

Listing 12.12: stylesheet3.xsl

*****PD: Please add line numbers in the following code*****

```

<?xml version = "1.0"?>
<html xsl:version = "1.0"
  xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
  <body>
    <h2>Student List</h2>
    <xsl:for-each select="students/student">
      <xsl:sort select = "lastName" order = "ascending"/>
      <li>
        <xsl:value-of select = "ssn"/>
        <xsl:text> </xsl:text>
        <xsl:value-of select = "firstName"/>
        <xsl:text> </xsl:text>
        <xsl:value-of select = "mi"/>
        <xsl:text> </xsl:text>
        <xsl:value-of select = "lastName"/>
      </li>
    </xsl:for-each>
  </body>
</html>

```

```

</li>
</xsl:for-each>
</body>
</html>

```

12.5 Developing Web Applications Using Oracle XSQL (Optional)

Oracle XSQL is a tool that integrates SQL with XML to generate dynamic XML contents. You can embed SQL statements inside XML tags to create an XML file called *XSQL page*. The XSQL page can be dynamically processed like a Java servlet to generate new XML contents that contain database data. For example, the XSQL page in Listing 12.13 in Figure 12.18 contains a SQL query to obtain all the records in the *Student* table. *Student.xml* can be processed by a Java servlet called *XSQL processor* to generate the XML file as shown in Listing 12.14 in Figure 12.17.

Listing 10.13: student.xml

```

<?xml version = "1.0"?>
<xsql:query connection="demo"
  xmlns:xsql="urn:oracle-xsql">
  select * from Student
</xsql:query>

```

Listing 10.14: The Generated XML

```

<?xml version = '1.0'?>
<ROWSET>
  <ROW num="1">
    <SSN>444111110</SSN>
    <FIRSTNAME>Jacob</FIRSTNAME>
    <MI>R</MI>
    <LASTNAME>Smith</LASTNAME>
    <BIRTHDATE>4/9/1985 0:0:0</BIRTHDATE>
    <STREET>99 Kingston Street</STREET>
    <ZIPCODE>31435</ZIPCODE>
  </ROW>
  ...
</ROWSET>

```

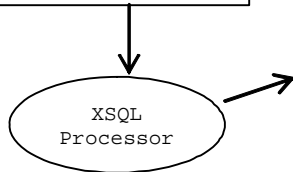
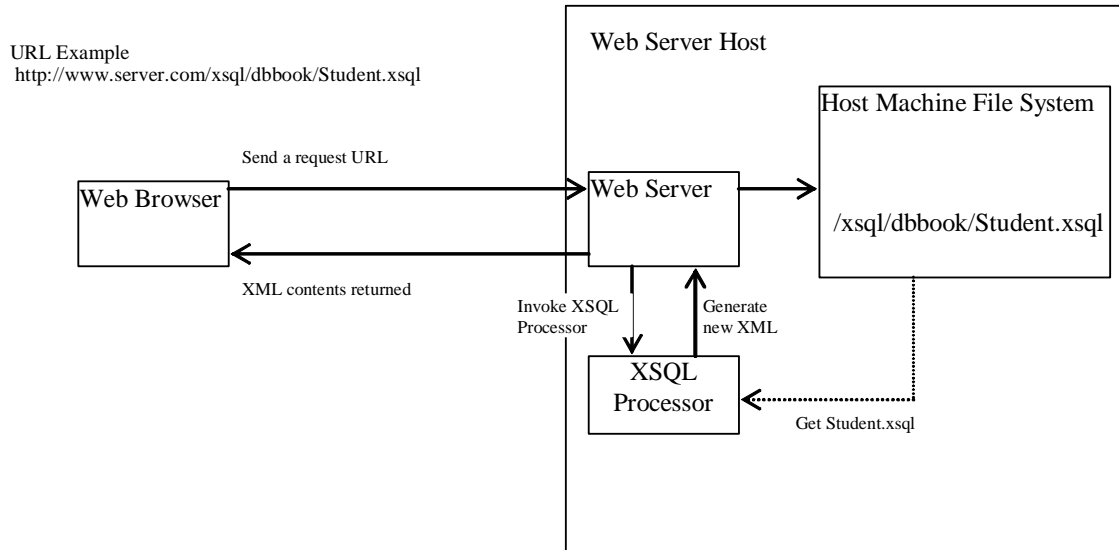


Figure 12.17

The XSQL processor produces an XML file that contains database data generated from SQL queries in the XSQL page.

With the XSQL processor properly installed and configured on your Web server, you can place *student.xml* to a directory under your Web server's virtual directory hierarchy and obtain the resulting XML file using an appropriate URL from a Web browser as shown in Figure 12.18.

**Figure 12.18**

A Web browser requests a dynamic XML page from a Web server.

By default, the XSQL processor is automatically installed in Oracle 9i Enterprise and configured to work with Apache Web server. Create a directory named dbbook under c:\oracle9i\jdk\demo\java\xsql and place student.xsql under c:\oracle9i\jdk\demo\java\xsql\dbbook. You can invoke student.xsql from the URL <http://localhost/xsql/dbbook/student.xsql>, as shown in Figure 12.19.

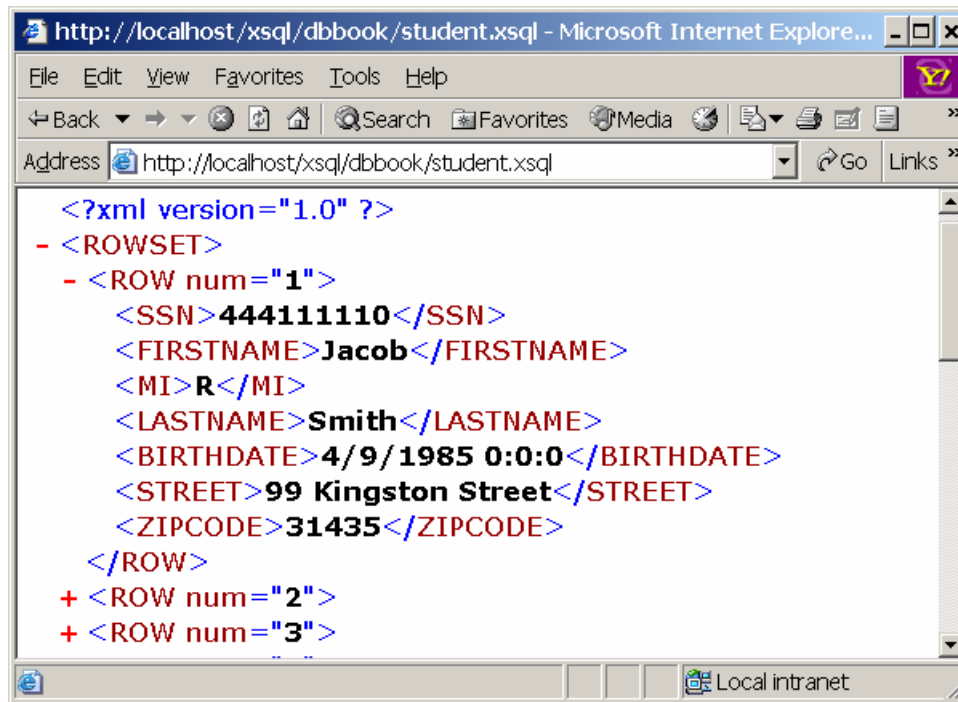


Figure 12.19

The *student.xsql* can be invoked like a Java servlet.

The results of the SQL query are in XML format. You can supply a stylesheet to display the XML data in HTML. Listing 12.15 adds a stylesheet in *student.xsql*. Listing 12.16 gives a stylesheet. Figure 12.20 shows the result of invoking the XSQL page with a stylesheet.

Listing 12.15: *studnet.xml*

*****PD: Please add line numbers in the following code*****

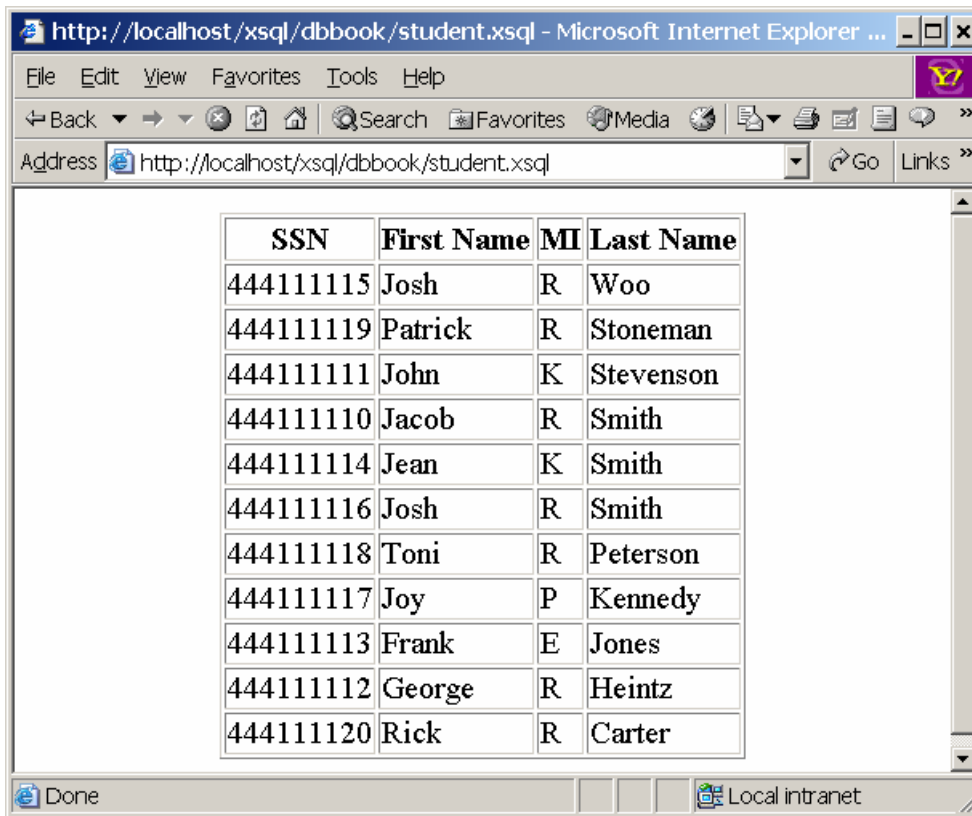
```
<?xml version = "1.0"?>
<?xml-stylesheet type="text/xsl" href="student.xsl"?>
<xsql:query connection="demo"
  xmlns:xsql="urn:oracle-xsql">
  select * from Student
</xsql:query>
```

Listing 12.16: *student.xsl*

*****PD: Please add line numbers in the following code*****

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

```
<xsl:stylesheet version = "1.0"
  xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
  <xsl:template match = "ROWSET">
    <html><body><center><table border = "1">
      <tr><th>ssn</th><th>First Name</th>
        <th>mi</th><th>Last Name</th></tr>
      <xsl:for-each select = "ROW">
        <xsl:sort select = "lastname" order = "descending"/>
        <tr>
          <td>
            <xsl:value-of select = "ssn"/>
          </td>
          <td>
            <xsl:value-of select = "firstName"/>
          </td>
          <td>
            <xsl:value-of select = "mi"/>
          </td>
          <td>
            <xsl:value-of select = "lastname"/>
          </td>
        </tr>
      </xsl:for-each>
    </table></center></body></html>
  </xsl:template>
</xsl:stylesheet>
```



SSN	First Name	MI	Last Name
444111115	Josh	R	Woo
444111119	Patrick	R	Stoneman
444111111	John	K	Stevenson
444111110	Jacob	R	Smith
444111114	Jean	K	Smith
444111116	Josh	R	Smith
444111118	Toni	R	Peterson
444111117	Joy	P	Kennedy
444111113	Frank	E	Jones
444111112	George	R	Heintz
444111120	Rick	R	Carter

Figure 12.20

You can use a stylesheet in XSQL pages.

You have used the `<xsql:query>` tag in `student.xsql`. All XSQL tags begin with the namespace prefix `xsql`. The `xsql` tags specify actions to be performed by XSQL processor. The following sections introduce several frequently used `xsql` tags.

12.5.1 The `<xsql:query>` Tag

The `<xsql:query>` tag specifies a SQL query statement as follows:

```
<xsql:query connection = "connname"
  xmlns:xsql="urn:oracle-xsql">
  SQL Query Statement
</xsql:query>
```

The `connection` attribute specifies a database connection name. The connection name must be defined in `c:\oracle9i\jdk\admin\XSQLConfig.xml`. The connection name `demo` is pre-defined in `XSQLConfig.xml` as follows:

```

<connection name="demo">
  <username>scott</username>
  <password>tiger</password>
  <dburl>jdbc:oracle:thin:@localhost:1521:ORCL</dburl>
  <driver>oracle.jdbc.driver.OracleDriver</driver>
</connection>

```

You can modify demo or create a new connection name to connect to a local or remote database. The database can be any relational database with appropriate JDBC drivers.

The xmlns attribute declares XSQL namespace identifier urn:oracle-xsql.

By default, the result data of the SQL query is saved in XML format that reflects the column structure of the query result with <ROWSET> as the tag for the root element. For example, if the query is

```

select firstName || ' ' || mi || ' ' || lastName as FullName
from Student
where phone like '707%';

```

the structure of the resulting XML will have the column name as the element tag as follows:

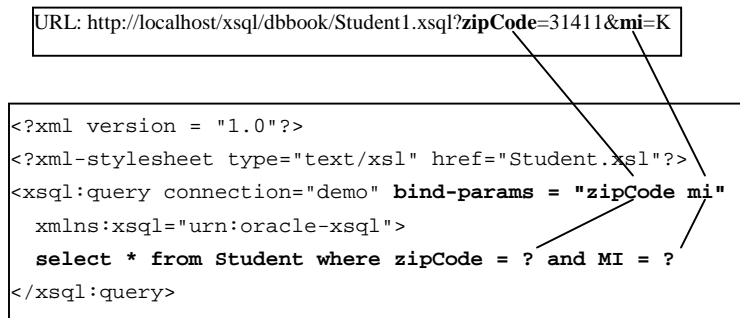
```

<?xml version = '1.0'?>
<ROWSET>
  <ROW num="1">
    <FullNAME>fullname1</FullNAME>
  </ROW>
  ...
</ROWSET>

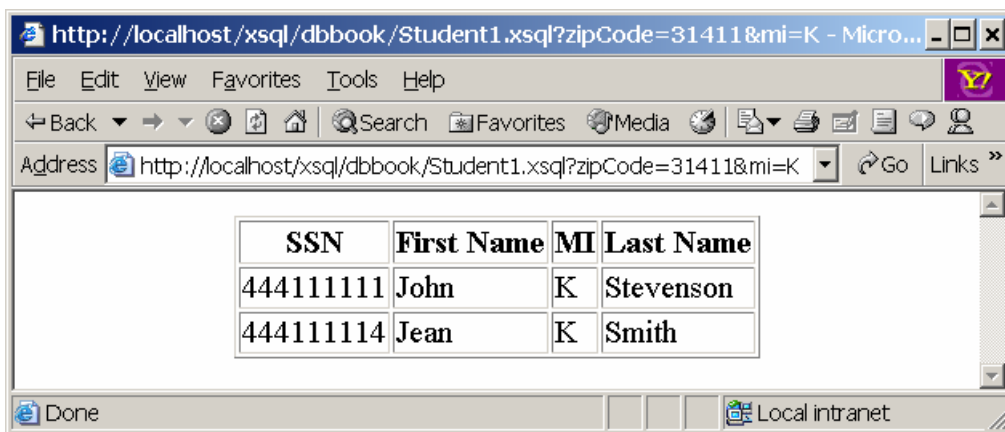
```

12.5.2 Using Parameters in XSQL

You can pass parameters in XSQL to obtain results dynamically based on the parameters. For example, the XSQL page in Figure 12.21 has a parameter on zipCode and a parameter on mi, which are passed to the SQL query to select the students with the specified zipCode and mi. You can invoke the query using the URL as shown in Figure 12.22.

**Figure 12.21**

You can pass parameters to the SQL statement in XSQL pages.

**Figure 12.22**

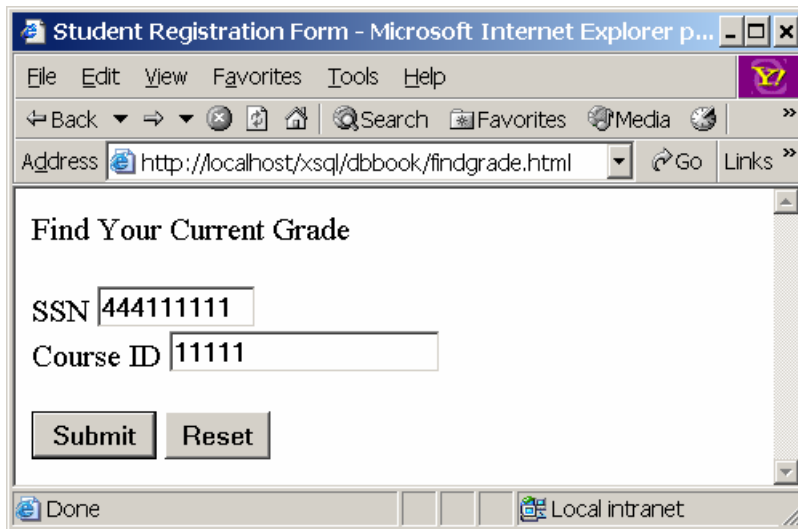
You can request an XSQL page with parameters from a Web browser.

The `bind-params` attribute lists all the parameters in order separated by space. The question marks are the placeholders for the parameters. The parameters from left to right in the list are bound to the question marks in order of their appearance in the SQL statement. i.e. The first question mark corresponds to the first parameter in the parameter list and the second question corresponds to the second parameter in the list. Figure 12.23 shows the result of invoking the parameterized XSQL page with `zipCode` 31411 and `mi` K.

Example 12.1

Retrieving and Presenting Data Using XSQL

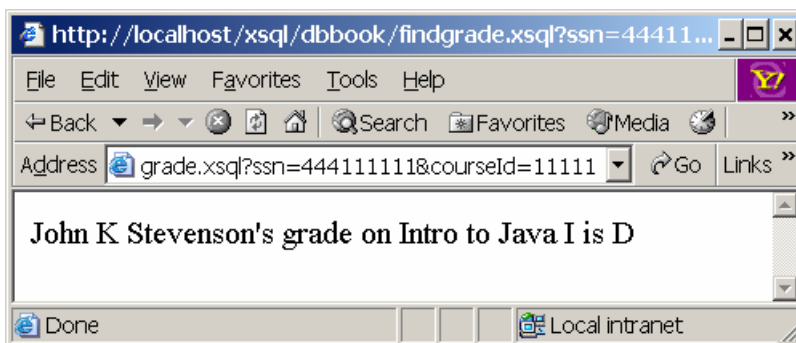
This example demonstrates developing database Web applications using XML and XSQL. The program starts with an HTML form that prompts the user to enter the ssn and Course ID, as shown in Figure 12.23. Pressing the Submit button invokes the XSQL page to retrieve the student first name, mi, last name, course title, and grade, as shown in Figure 12.24.



The screenshot shows a Microsoft Internet Explorer window titled "Student Registration Form - Microsoft Internet Explorer p...". The address bar displays "http://localhost/xsql/dbbook/findgrade.html". The main content area contains a form titled "Find Your Current Grade". The form has two input fields: "SSN" with the value "444111111" and "Course ID" with the value "11111". Below the input fields are two buttons: "Submit" and "Reset". The status bar at the bottom shows "Done" and "Local intranet".

Figure 12.23

The form prompts the user to enter ssn and Course ID for finding student's grade for the course.



The screenshot shows a Microsoft Internet Explorer window titled "http://localhost/xsql/dbbook/findgrade.xsql?ssn=44411...". The address bar displays "http://localhost/xsql/dbbook/findgrade.xsql?ssn=444111111&courseId=11111". The main content area displays the text "John K Stevenson's grade on Intro to Java I is D". The status bar at the bottom shows "Done" and "Local intranet".

Figure 12.24

The grade for the student on the course is displayed.

The example contains three files:

findgrade.html (Listing 12.17) displays an HTML form that enables the user to enter ssn and Course ID and submit the request to the server.

findgrade.xsql (Listing 12.18) retrieves the first name, mi, last name, course title, and grade for the student with the specified ssn and course ID.

findgrade.xsl (Listing 12.19) describes the stylesheet for the XML data generated from findgrade.xsql.

Listing 12.17: findgrade.html

*****PD: Please add line numbers in the following code*****

```
<html>
<head>
<title>Student Registration Form</title>
</head>
<body>
Find Your Current Grade

<form action="findgrade.xsql" method="get">
  ssn <input type="text" name="ssn" size="10"/><br/>
  Course ID <input type="text" name="courseId" size="20" />
  <p><input type="submit" value="Submit"/>
    <input type="reset" value="Reset"/></p>
</form>
</body>
```

Listing 12.18: findgrade.xsql

*****PD: Please add line numbers in the following code*****

```
<?xml version = "1.0"?>
<?xml-stylesheet type="text/xsl" href="findgrade.xsl"?>
<xsql:query connection="demo" bind-params = "ssn courseId"
  xmlns:xsql="urn:oracle-xsql">
  select firstName, mi, lastName, title, grade
  from Student, Enrollment, Course
  where Student.ssn = ? and Enrollment.courseId = ? and
    Enrollment.ssn = Student.ssn and
    Enrollment.courseId = Course.courseId
</xsql:query>
```

Listing 12.19: findgrade.xml

*****PD: Please add line numbers in the following code*****

```

<?xml version = "1.0"?>
<xsl:stylesheet version = "1.0"
  xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
  <xsl:template match = "ROWSET">
    <xsl:choose>
      <xsl:when test = 'ROW'>
        <html><body>
          <xsl:value-of select = "ROW/firstName"/>
          <xsl:text> </xsl:text>
          <xsl:value-of select = "ROW/mi"/>
          <xsl:text> </xsl:text>
          <xsl:value-of select = "ROW/lastname"/>
          <xsl:text>'s grade on </xsl:text>
          <xsl:value-of select = "ROW/TITLE"/>
          <xsl:text> is </xsl:text>
          <xsl:value-of select = "ROW/GRADE"/>
        </body></html>
      </xsl:when>
      <xsl:otherwise>
        <xsl:text>not found</xsl:text>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>

```

Example Review

The XSQL page is invoked from the HTML form (Line X in findgrade.html). The parameters ssn and courseId are passed to the XSQL page upon invoked.

The XSQL page uses the SQL query statement (Line X in findgrade.xsql) to retrieve first name, mi, last name, course title, and grade for the student with the passed ssn and course id. The result of the query is formatted in XML. The XSQL page specifies the XSL stylesheet findgrade.xsl, which is used to transform the XML data into HTML for presentation on the browser.

The XSL stylesheet uses the <xsl:choose> tag (Line X) to specify the output. If a grade is found, the student name, course title, and grade are displayed. Otherwise, the text no found is displayed.

12.5.3 The <xsql:dml> Tag

You can use the <xsql:dml> tag to contain SQL DML and DDL statements as follows:

```
<xsql:dml connection = connname  
  bind-params = "parameters"  
  commit = "booleanValue"  
  xmlns:xsql = "urn:oracle-xsql">  
  SQL DML and DDL statements  
</xsql:dml>
```

The attributes connection, bind-params, and xmlns:xsql are the same as in the <xsql:query> tag. The commit attribute specifies whether an SQL commit statement is called after a successful execution of the DML statement. The default value of commit is no.

Example 12.2

Using DML Statements in XSQL Pages

This example demonstrates using DML statements in XSQL. The program starts with an HTML form for receiving student information, as shown in Figure 12.25. Pressing the Submit button invokes the XSQL page to store the student information into the database. If successful, no errors are displayed as shown in Figure 12.26.

Simple Registration without Confirmation - Microsoft Internet Explorer provided by Yahoo!

File Edit View Favorites Tools Help

Back Forward Stop Home Search Favorites Media Print View Source Help

Address http://localhost/xsql/dbbook/registration.html Go Links

Please register to your instructor's student address book.

SSN * 555889999

Last Name * Doe First Name * John MI K

Phone 9129214545 Birth Date 09-OCT-69

Street 100 Main Street

Zip 31411

Submit Reset

* required fields

Done Local intranet

Figure 12.25

The form prompts the user to enter student information.

http://localhost/xsql/dbbook/registration.xsql - Microsoft Internet Explore...

File Edit View Favorites Tools Help

Back Forward Stop Home Search Favorites Media Print View Source Help

Address http://localhost/xsql/dbbook/registration.xsql Go Links

```
<?xml version="1.0" ?>
<xsql-status action="xsql:dml" rows="0" />
```

Done Local intranet

Figure 12.26

The student information is stored in the database with no errors.

The example contains two files:

registration.html (Listing 12.20) displays an HTML form that enables the user to enter student information and submit the information to the server.

registration.xsql (Listing 12.21) inserts the student information to the database.

Listing 12.20:

findgrade.html

*****PD: Please add line numbers in the following code*****

```

<html>
<head>
<title>Simple Registration without Confirmation</title>
</head>
<body>
Please register to your instructor's student address book.</font>

<form method="post" action="registration.xsql">
<p>ssn <font color="#FF0000">*</font>
    <input type="text" name="ssn">&nbsp;
<p>Last Name <font color="#FF0000">*</font>
    <input type="text" name="lastName">&nbsp;
First Name <font color="#FF0000">*</font>
    <input type="text" name="firstName">&nbsp;
mi <input type="text" name="mi" size="3"></p>
<p>Phone <input type="text" name="phone" size="20">&nbsp;
    Birth Date <input type="text" name="birthDate" size="20">&nbsp;
<p>Street <input type="text" name="street" size="50"></p>
<p>Zip <input type="text" name="zipCode" size="9"></p>
<p><input type="submit" name="Submit" value="Submit">
    <input type="reset" value="Reset"></p>
</form>
<p><font color="#FF0000">* required fields</font></p>
</body>
</html>

```

Listing 12.21:

registration.xsql

*****PD: Please add line numbers in the following code*****

```

<?xml version = "1.0"?>
<xsql:dml connection="demo" commit="yes" bind-params =
    "ssn firstName mi lastName phone birthDate street zipCode"
    xmlns:xsql="urn:oracle-xsql">
begin
    insert into Student values (?, ?, ?, ?, ?, ?, ?, ?);
    commit;
end;
</xsql:dml>

```

Example Review

The HTML form passes ssn, firstName, mi, lastname, phone, birthDate, street, and zipCode to the XSQL page, which uses the insert statement to store these values into the Student table. The commit attribute (Line 2 in registration.html) tells the XSQL processor to call a commit statement after the SQL insert statement is successfully executed. Due to a bug, the commit statement is not automatically invoked. To fix the bug, the commit statement is explicitly called in the XSQL page (Line X).

This example is similar to Example 8.3, "Registering a Student in Database." Example 8.3 is implemented using a Java servlet, but this example is implemented using XSQL. As you see, it is much simpler to use XSQL.

Chapter Summary

This chapter introduced XML, DTD, XPath, XSLT, and XSQL and using them to develop useful applications. You learned to use XML to structure data, use DTD to define the structure of XML documents, use XPath to traverse XML documents, use XSLT to transform XML documents, and use XSQL to access database and obtain results in XML.

Review Questions

- 12.1 What are the differences between XML and HTML? Is XML case-sensitive? Is HTML case-sensitive?
- 12.2 What is a well-formed HTML document?
- 12.3 How do you declare an XML document?
- 12.4 How do write comments in XML?
- 12.5 What is a well-formed HTML document?
- 12.6 What is an empty element? How do you write an empty element?
- 12.7 What is an XML attribute?
- 12.8 How do you denote special characters ≤, ≥, ", ', and & in XML?
- 12.9 What is a CDATA section in XML?
- 12.10 What is a well-formed XML document? What is a valid XML document?
- 12.11 What is a DTD (Document Type Definition) for?
- 12.12 What are an internal DTD and an external DTD?
- 12.13 What are the symbols +, *, and ? for in DTD?
- 12.14 Describe the attribute types: CDATA, NMTOKEN, NMTOKENS, ID, IDREF, IDREFS, ENTITY, and ENTITIES.
- 12.15 What is an XSL stylesheet?
- 12.16 What does an XSLT processor do?
- 12.17 What is an XPath tree? What are the axes? What are the node set operators? What are the predicates?
- 12.18 List some node set functions.
- 12.19 List some string functions.
- 12.20 How does an XSLT processor work?

12.21 What happens if a node matches two or more patterns in the stylesheet?

12.22 Describe the following XSLT action tags:

```
<xsl:apply-templates select = "nodeset">
<xsl:text>
<xsl:value-of select = exp/>
<xsl:if = condition>
<xsl:choose>
<xsl:for-each select = "node-set">
<xsl:sort>
```

12.23 What is XSQL?

12.24 What is the role for XSQL processor?

12.25 How do you specify the connection to a database from XSQL?

12.26 How do you pass parameters in XSQL?

12.27 Can you execute SQL DML and DDL statements in XSQL?

Programming Exercises

12.1 Write a stylesheet to transform student.xml in Listing 12.1 into a new XML document as follows:

```
<?xml version = "1.0"?>
<!-- XML document for students -->
<ROWS>
  <ROW NUM = "1">
    <ssn>444111110</ssn>
    <firstname>Jacob</firstname>
    <mi>R</mi>
    <lastname>Smith</lastname>
    <birthdate>4/9/1985</birthdate>
    <phone>9129219434</phone>
    <street>99 Kingston Street</street>
    <zipcode>31435</zipcode>
  </ROW>
  <ROW NUM = "2">
    <ssn>444111111</ssn>
    <firstname>John</firstname>
    <mi>K</mi>
    <lastname>Stevenson</lastname>
    <birthdate>4/9/1985</birthdate>
    <phone>9129219434</phone>
    <street>100 Main Street</street>
    <zipcode>31411</zipcode>
  </ROW>
</ROWS>
```

The <STUDNETS> and <student> tags in original document are transformed to <ROWS> and <ROW>.

- 12.2 Write an XSQL page to list all faculty in the Math department.
- 12.3 Write an XSLQ page to list all faculty in the specified department passed as a parameter.