Unit II:
XML document rule, XML structuring, XML presentation technologies, XML Transformation, XSLT, XQUERY, XLINK, XPATH

1. XML document Syntax rule:

- FIRST item must be  XML declaration statement  

- All XML Elements Must Have a Closing Tag:
  - In HTML, few of them not having closing tag.
  - Example: `<p>This is a paragraph.</p>`
  - But XML all elements should have closing tag.

- XML tags are case sensitive:
  - The tag `<Book>` is different from `<book>`

- XML elements must be properly nested.
  - In html, one can have `<b><i>This text is bold and italic</i></b>`
  - In XML, all elements must be properly nested within each other:
    - `<b><i>This text is bold and italic</i></b>`

- XML Documents Must Have a Root Element
  - XML documents must contain one element that is the parent of all other elements. This element is called the root element.

```xml
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

- XML, the attribute values must always be quoted. `<book ISBN="21-458-65-0">`

- The syntax for writing comments in XML is similar to that of HTML.
  - `<!-- This is a comment -->`

- White-space is Preserved in XML
  - HTML truncates multiple white-space characters to one single white-space:
    - HTML: `Hello Tove`
    - Output: `Hello Tove`
  - With XML, the white-space in a document is not truncated.

- XML Stores New Line as LF

- Entity References
  - Some characters have a special meaning in XML. If you place a character like "<" inside an XML element, it will generate an error because the parser interprets it as the start of a new element.
  - This will generate an XML error: `<message>`if salary < 1000 then`</message>`
To avoid this error, replace the "<" character with an entity reference: 
<message>if salary &lt; 1000 then</message>

There are 5 predefined entity references in XML / Special Markup characters:

<table>
<thead>
<tr>
<th>&lt;</th>
<th>less than</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&amp;</td>
<td>ampersand</td>
</tr>
<tr>
<td>'</td>
<td>apostrophe</td>
</tr>
<tr>
<td>&quot;</td>
<td>quotation mark</td>
</tr>
</tbody>
</table>

2. XML Language fundamentals:

(i) XML Tag

An XML element, in turn, consists of a start tag and an end tag, except in the case of elements that are defined to be empty, which consist only of one tag. A start tag (also called an opening tag) starts with < and ends with >. End tags (also called closing tags) begin with </ and end with >. The XML specification is very specific about tag names; you can start a tag name with a letter, an underscore, or a colon. The next characters may be letters, digits, underscores, hyphens, periods, and colons (but no whitespace).

<table>
<thead>
<tr>
<th>Legal Tags</th>
<th>Illegal Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;DOCUMENT&gt;</td>
<td>&lt;2003DOCUMENT&gt;</td>
</tr>
<tr>
<td>&lt;document&gt;</td>
<td>&lt;.document&gt;</td>
</tr>
<tr>
<td>&lt;_Record&gt;</td>
<td>&lt;Record Number&gt;</td>
</tr>
<tr>
<td>&lt;customer&gt;</td>
<td>&lt;customer*name&gt;</td>
</tr>
<tr>
<td>&lt;PRODUCT&gt;</td>
<td>&lt;PRODUCT(ID)&gt;</td>
</tr>
</tbody>
</table>

Exercise 3:

```xml
<?xml version = "1.0" encoding="utf-8"?>
<DOCUMENT>
  <GREETING>
    Hello From XML
  </GREETING>
  <MESSAGE>
    Welcome to the wild and woolly world of XML.
  </MESSAGE>
</DOCUMENT>
```
Exercise 4:

<?xml version = "1.0" encoding="utf-8"?>
<name>
  <first>John</first>
  <middle>Fitzgerald Johansen</middle>
  <last>Doe</last>
</name>

<table>
<thead>
<tr>
<th>Types of Tags</th>
<th>Purpose</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty element</td>
<td>Represent information at a specific point in the document.</td>
<td>&lt;xref linkend=&quot;abc&quot;/&gt;</td>
</tr>
<tr>
<td>container element</td>
<td>Group together elements and character data.</td>
<td>&lt;p&gt;This is a paragraph.&lt;/p&gt;</td>
</tr>
<tr>
<td>Declaration</td>
<td>Add a new parameter, entity, or grammar definition to the parsing environment.</td>
<td>&lt;!ENTITY author &quot;Erik Ray&quot;&gt;</td>
</tr>
<tr>
<td>processing instruction</td>
<td>Feed a special instruction to a particular type of software.</td>
<td>&lt;?print-formatter force-linebreak?&gt;</td>
</tr>
<tr>
<td>Comment</td>
<td>Insert an annotation that will be ignored by the XML processor.</td>
<td>&lt;!— here's where I left off —&gt;</td>
</tr>
<tr>
<td>CDATA section</td>
<td>Create a section of character data that should not be parsed, preserving any special characters inside it.</td>
<td>&lt;![CDATA[Ampersands galore! &amp;&amp;&amp;&amp;&amp;&amp;]]&gt;</td>
</tr>
<tr>
<td>entity reference</td>
<td>Command the parser to insert some text stored elsewhere.</td>
<td>&amp;company-name;</td>
</tr>
</tbody>
</table>

(ii) XML Element

The fundamental unit of XML content is the element, which is an author-specified chunk of information. An element consists of an element name and element content. Consider the example an annotated version of our business card document to see examples of these content types.

```xml
<BusinessCard>
  <Name>
    <GivenName>Kevin</GivenName>
    <MiddleName>Stewart</MiddleName>
    <FamilyName>Dick</FamilyName>
  </Name>
  <Title>Software Technology Analyst</Title>
  <Author/>  
</BusinessCard>
```
In the above example, "BusinessCard" is the top-level element. In XML, there can be only one element at the top level. This element is called the document element or sometimes root element. Think of this element as the trunk of the tree from which all other elements branch. The following figure shows the corresponding tree for the above example with each node representing an element and identified with the element name. Conceptually the element content resides within the node.

- Empty elements:

Empty elements have only one tag, not a start and end tag and close an empty element with />

Exercise 5: Example of Empty element : <GREETING /> is empty element

```xml
<?xml version = "1.0" standalone="yes"?>
<DOCUMENT>
  <GREETING TEXT = "Hello From XML"/>
</DOCUMENT>
```

- Root element

Each well-formed XML document must contain one element that contains all the other elements. The containing element is called the root element. An XML document must have a single root tag, such that all other tags are contained within that root tag. All subsequent elements must be contained within the root tag, each nested within its parent tag.
Exercise 6: Example of Root element `<PLANT>` is root element

```xml
<?xml version="1.0"?>
<PLANT>
  <COMMON>Columbine</COMMON>
  <BOTANICAL>Aquilegia canadensis</BOTANICAL>
</PLANT>
```

- **Child elements**

The root node has only children. All other nodes have one parent node, as well as zero or more child nodes. Nodes can have elements that are related on the same hierarchical level.

Exercise 7: Example of Child element

- `<bookstore>` is a root element
- `<book>` is a child of root element as well as parent of `<title>`, `<author>`, `<year>`, `<price>`
- `<title>`, `<author>`, `<year>`, `<price>` are child elements

```xml
<?xml version="1.0"?>
<bookstore>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="CHILDREN">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
</bookstore>
```

- **Tree structure of XML document**

The elements in an XML document form a document tree. The tree starts at the root and branches to the lowest level of the tree. The example of tree structure for the Exercise 7 using one of the XML editor ‘firstobject’ is shown as follows:
XML elements are extensible

Exercise 8: For example Exercise 6 can be further added with few more elements

```xml
<?xml version="1.0"?>
<PLANT>
  <COMMON>Columbine</COMMON>
  <BOTANICAL>Aquilegia canadensis</BOTANICAL>
  <LIFE>12</LIFE>
  <COUNTRY>AFRICA</COUNTRY>
</PLANT>
```

(iii) XML attribute

In addition to tags and elements, XML documents can also include attributes. Attributes are simple name/value pairs associated with an element. Attributes must have values—even if that value is just an empty string (like "")

Exercise 9: They are attached to the start-tag, as shown below, but not to the end-tag:

```xml
<?xml version="1.0"?>
<name nickname="Shiny John">
  <first>John</first>
  <middle>Fitzgerald Johansen</middle>
  <last>Doe</last>
</name>
```
From a purely programming perspective, it could be stated that attributes should not be used because of the following reasons:

- Elements help to define tree structure and attributes do not.
- Attributes are not allowed to have multiple values whereas elements can.
- Programming is more complex using attributes.
- Attributes are more difficult to alter in XML documents / not easily expandable at a later stage.

Attributes are difficult to read and maintain. Use elements for data. Use attributes for information that is not relevant to the data.

Exercise 10: Convert attribute to element

<table>
<thead>
<tr>
<th>Exercise 10 a: Attribute</th>
<th>Exercise 10 b: Element</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;?xml version=&quot;1.0&quot;?&gt;</code>&lt;person sex=&quot;female&quot;&gt;</td>
<td><code>&lt;?xml version=&quot;1.0&quot;?&gt;</code>&lt;person&gt;</td>
</tr>
<tr>
<td>&lt;firstname&gt;Anna&lt;/firstname&gt;</td>
<td>&lt;sex&gt;female&lt;/sex&gt;</td>
</tr>
<tr>
<td>&lt;lastname&gt;Smith&lt;/lastname&gt;</td>
<td>&lt;firstname&gt;Anna&lt;/firstname&gt;</td>
</tr>
<tr>
<td>&lt;/person&gt;</td>
<td>&lt;lastname&gt;Smith&lt;/lastname&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/person&gt;</td>
</tr>
</tbody>
</table>

XML Attributes for Metadata: Attributes Can Provide Meta Data that May Not be Relevant to Most Applications Dealing with Our XML.

Exercise 11: Attribute is used as meta data to identify the element: Sometimes ID references are assigned to elements. These IDs can be used to identify XML elements in much the same way as the id attribute in HTML. This example demonstrates this:

```
<?xml version="1.0"?>
<messages>
  <note id="501">
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don't forget to XML Project!</body>
  </note>
  <note id="502">
    <to>Jani</to>
    <from>Tove</from>
    <heading>Re: Reminder</heading>
    <body>I will not</body>
  </note>
</messages>
```
(iv) Prolog

The prolog of an XML document, when present, precedes the document element. The prolog may, but need not, contain the following:
- An XML declaration
- Miscellaneous content—processing instructions or comments
- A Document Type Declaration, also called a DOCTYPE declaration

(v) XML declaration

The typical XML declaration looks like this

```xml
<?xml version='1.0' encoding='UTF-16' standalone='yes'?>
```
- The XML declaration starts with the characters `<?xml`, and ends with the characters `?>`.
- If you include it, you must include the version, but the encoding and standalone attributes are optional.
- The version, encoding, and standalone attributes must be in that order.
- The XML declaration must be right at the beginning of the file. That is, the first character in the file should be that `<?`; no line breaks or spaces. Some parsers are more forgiving about this than others.
- A character encoding is the method used to represent the numbers in a character code digitally (in other words how many bytes should be used for each number, etc.). Some of the encoding are utf-8, utf-16, ascii etc.
- If the standalone attribute is included in the XML declaration, it must be set to either yes or no. yes specifies that this document exists entirely on its own, without depending on any other files, no indicates that the document may depend on an external DTD.

Exercise 12: Example to include all the above declaration

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

Exercise 13: Using special mark up elements in element content

```xml
<?xml version= "1.0" ?>
<example>
  &lt;book&gt;
  &lt;title&gt;Sams Teach Yourself XML in 10 Minutes&lt;/title&gt;
  &lt;author&gt;Andrew Watt&lt;/author&gt;
  &lt;/book&gt;
</example>
```

Exercise 14: Using special mark up elements in element content

```xml
<?xml version="1.0" encoding="UTF-8"?>
<document>
  <heading>
    Hello From XML
  </heading>
  <message>
    This text is inside a &lt;message&gt; element.
  </message>
</document>
```

(vii) Use comment to exclude the document

Exercise 15:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<document >
  <!--
    <heading>
      Hello From XML
    </heading>
  -->
  <message>
    This is an XML document!
  </message>
</document>
```
(viii) XML tools

- XML editors: They are used to create XML document. Some of them are firstobject, Adobe frameworker, XML pro, Altova XML spy, stylus studio, xml writer, enotepad, xml notepad.
- XML browsers: IE 6, Netscape Navigator 6, Jumbo
- XML parsers: MSXML, SAX, expat, expat perl module, TC1Expat, LT XML, XML for Java, XML test pad, XP, SXP, Python and XML processing preliminary XML parser
- XML validators: W3C XML validator, Tidy, XML.com lark parser, LTP, STG, VS.net

Example using STG:

(ix) XML editors help to type and identifying the errors while parsing the XML document

Exercise 16: Open the Exercise 11. Do few mistakes and view the error in the document

(x) The appearance of XML document in MSWord

(xi) The appearance of XML document in MSExcel
3. XML Structuring

An XML document actually can do more than just hold data. Therefore it is required to specify the structure of that data as well. This structuring is very important when dealing with complex data. For example, you could store a long account statement in HTML, but after the first ten pages or so, that data would be prone to errors. But in XML, you can actually build in the syntax rules that specify the structure of the document so that the document can be checked to make sure it's set up correctly.

This emphasis on the correctness of your data's structure is strong in XML, and it makes it easy to detect problems. In HTML, a Web author could (and frequently did) write sloppy HTML, knowing that the Web browser would take care of any syntax problems. In fact, some people estimate that 50% or more of the code in modern browsers is there to take care of sloppy HTML in Web pages. But things are different in XML. The software that reads your XML—called an XML processor—is supposed to check your document; if there's a problem, the processor is supposed to quit. So how does an XML processor check your document? There are two main checks that XML processors make: checking that your document is well-formed and checking that it's valid.

3.1 Well-formed XML documents

Why to make XML documents well-formed?

- W3C doesn't consider an XML document to be XML unless it's well-formed.
- XML processors won't read XML documents unless those documents are well-formed.

What makes an XML documents well-formed?

A textual object is a well-formed XML document if:

(i) Taken as a whole, it matches the production labeled document. The document must follow the production, must have three parts:
   - a prolog (XML declaration, PI, DTD)
   - a root element (which can contain other elements)
   - a miscellaneous part (unlike the preceding two parts, this part is optional)

(ii) It meets all the well-formedness constraints given in this specification (that is, the XML 1.0 specification, http://www.w3.org/TR/REC-xml). Every XML document, must be well-formed. This means it must adhere to a number of rules, including the following:
   - Begin the Document with an XML Declaration
     Example: <?xml version = "1.0" encoding="UTF-8" standalone="yes"?>
   - Use Only Legal Character References. Note the characters that are legal in XML 1.0 differ somewhat from what's legal in XML 1.1.
Include at least one element (root element) and
There must be exactly one root element.
Using the Root Element to Contain All Other Elements
Structure Elements Correctly.

- Every start-tag must have a matching end-tag.
- Elements may nest, but may not overlap.

Exercise 16: Simple example of well-formed documents with all the above points

```xml
<?xml version= "1.0"?><employee>
    <name> Kelly </name>
    <hiredate>October 15, 2005</hiredate>
    <project>
        <product>Laptop</product>
        <id>222</id>
        <price>$989.00</price>
    </project>
</employee>
```

Attribute values must be quoted.

Example: `<cd serial="51234b" />

Use single quotes when attribute value contains quoted text
`<message text='I said, "No, no, no!"' />

When an attribute value contains both single and double quotation marks, you can escape " as &quot; and ' as &apos;
`<tree type="Maple" height="50&apos;6&quot;" />

Avoid Overuse of < and &

XML processors assume that < starts a tag and & starts an entity reference, so you should avoid using those characters for anything else.

Make Attribute Names Unique. An element may not have two attributes with the same name. (XML is case sensitive)
`<message Text="Hi there!" text="Hello!" />

Avoid Entity References and < in Attribute Values. No unescaped < or & signs may occur in the character data of an element or attribute
`project note="This is a <project> element.">
should be written as this, where you're escaping both < and >:
`<project note="This is a &lt;project&gt; element.">
Comments and processing instructions may not appear inside tags.

Comments begin with <!-- and end with the first occurrence of -->. For example:
<!-- I need to verify and update these links when I get a chance. -->

Comments may appear anywhere in the character data of a document. They may also appear before or after the root element. However, comments may not appear inside a tag or inside another comment.

XML provides the processing instruction as an alternative means of passing information to particular applications that may read the document. A processing instruction begins with <? and ends with ?>. Like comments, processing instructions may appear anywhere in an XML document outside of a tag, including before or after the root element. The most common processing instruction, xml-stylesheet, is used to attach stylesheets to documents.

Example 1: Exercise 17: XML file along with cascading style sheet : Example for PI

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/css" href="o.css"?>
<document>
  <heading> Hello From XML </heading>
  <message> This is an XML document! </message>
</document>
```

Where O.css file contains

```css
heading {display: block; font-size: 24pt; color: #ff0000; text-align: center}
message {display: block; font-size: 18pt; color: #0000ff; text-align: center}
```

Example 2: Processing instruction may also include program coding as shown below

```php
<?php
mysql_connect("database.unc.edu", "clerk", "password");
$result = mysql("HR", "SELECT LastName, FirstName FROM Employees ORDER BY LastName, FirstName");
$i = 0;
while ($i < mysql_numrows ($result)) {
  $fields = mysql_fetch_row($result);
  echo "<person>$fields[1] $fields[0] </person>\n";
  $i++;
}
?>
```

(iii) Each of the parsed entities, which is referenced directly or indirectly within the document, is well-formed.
3.2 Valid XML documents

Well-formed XML data is guaranteed to use proper XML syntax, and a properly nested (hierarchical) tree structure. This may be sufficient for relatively static internal applications, particularly if the XML data is computer-generated and/or computer consumed. In this case, it's the responsibility of the applications using the data to perform any structural or content verification, error handling, and interpretation of the data. The XML structural information, and the logic to do this, is usually hard-coded separately within the sending and receiving applications, from a common specification. Therefore, any change to the XML data structure must be made in three places: the specification, and the sending and receiving applications.

This can be a high-performance approach to handling XML data in certain limited circumstances. For example, an internal corporate application might use well-formed XML as a data transfer mechanism between two different Relational Database Management Systems. The sending RDBMS would be assumed to generate good data. At the other end, the receiving RDBMS is likely to already have an input data verification feature, so any data validation could occur after the XML-to-DBMS translation step. There would be no need to re-validate the data while it was within the XML domain - in this limited case, well-formed XML would be sufficient for the data transfer. However, when there's no formal description of XML data, it can be difficult to describe or modify the structure of that data, since its structure and content constraints are buried within the application code. Without the use of a formal description of our XML data, we're only using a fraction of the power of XML.

Any XML data object is considered a valid XML document if it is well-formed, meets certain further validity constraints, and matches a grammar describing the document's content (DTD / Schema). For many applications, simple well-formed XML data is not enough – we must ensure that the data is also valid, using either XML 1.0 DTDs, or an extension such as XML Schema.

In addition to ensuring that XML data is well formed, many, if not most, XML applications will also need to ensure that the data is valid XML. To do this, we need to:

- Describe and validate the data structure, preferably in a rigorous and formal manner
- Communicate this data structure to others - both applications and people
- Constrain element content
- Constrain attribute types and values, and perhaps provide default values

These functions could be handled by specific code within a pair of cooperating applications and their accompanying documentation. However, in cases where the XML data is more widely shared, say between multiple applications or users, maintaining these functions in each application becomes an exponential nightmare. This is a problem common to most XML applications, so ideally we'd like to take a more standardized approach. Separating the XML data description from individual applications allows all cooperating applications to share a single description of the data, known as the XML vocabulary. A group of XML documents that share a common XML vocabulary is known as a document type, and each individual document that conforms to a document type is a document instance.
### 3.2.1 XML DTD

A DTD is a set of declarations which can be incorporated within XML data, or exist as a separate document. The DTD defines the rules that describe the structure and permissible content of the XML data. Only one DTD may be associated with a given XML document or data object.

The most significant aspect of DTD validation is the definition of the structure of the hierarchical tree of elements. A validating parser and a DTD can ensure that all necessary elements and attributes are present in a document, and that there are no unauthorized elements or attributes. This ensures that the data has a valid structure before it is handed over to the application. A DTD can be used in conjunction with a validating parser to validate existing XML data or enforce validity during the creation of XML documents by a human author, by:

- Checking that required elements are present
- Prompting the author for their inclusion when using a DTD-aware XML editor
- Check that no disallowed elements are included, and prevent the author from using them
- Enforcing element content and tree structure (using <!ELEMENT> declarations)
- Enforcing element attributes and their permissible values (with <!ATTLIST> declarations)

The DTD lists all the elements, attributes, and entities the document uses and the contexts in which it uses them. There are many things the DTD does not say. In particular, it does not say the following:

1. What the root element of the document is
2. How many of instances of each kind of element appear in the document
3. What the character data inside the elements looks like
4. The semantic meaning of an element; for instance, whether it contains a date or a person's name

**DTD**

- Identify elements and attributes
- Identify method for storing consistent data
- Define meaningful structure of content of xml doc
- DTD Creating is similar to table in database
- Specify elements that can be present in xml document (columns)
- XML documents that confirm to DTD are considered as valid documents.
- The two types of DTD are: internal and external
  - Internal DTD is part of XML file and cannot use across multiple document
  - External DTD is a separate file and reference is included in XML document
Exercise 18: Internal DTD implementation

```xml
<?xml version = "1.0" standalone="yes"?>
<!DOCTYPE document [
<!ELEMENT document (employee)*>
<!ELEMENT employee (name, hiredate, projects)>
<!ELEMENT name (lastname, firstname)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT hiredate (#PCDATA)>
<!ELEMENT projects (project)>
<!ELEMENT product (id,price)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT price (#PCDATA)>
] >
<document>
  <employee>
    <name>
      <lastname>Kelly</lastname>
      <firstname>Grace</firstname>
    </name>
    <hiredate>October 15, 2005</hiredate>
    <projects>
      <project>
        <product>Printer</product>
        <id>111</id>
        <price>$111.00</price>
      </project>
      <project>
        <product>Laptop</product>
        <id>222</id>
        <price>$989.00</price>
      </project>
    </projects>
  </employee>
  <employee>
    <name>
      <lastname>Grant</lastname>
      <firstname>Cary</firstname>
    </name>
    <hiredate>October 20, 2005</hiredate>
    <projects>
      <project>
        <product>Desktop</product>
        <id>333</id>
        <price>$2995.00</price>
      </project>
      <project>
        <product>Scanner</product>
        <id>444</id>
        <price>$200.00</price>
      </project>
    </projects>
  </employee>
</document>
```
Exercise 19: External DTD implementation

<table>
<thead>
<tr>
<th>XML file:</th>
<th>DTD file</th>
</tr>
</thead>
</table>
| `<?xml version = "1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE document SYSTEM "Ex19.dtd">
<document>
  <employee>
    <name>
      <lastname>Kelly</lastname>
      <firstname>Grace</firstname>
    </name>
    <hiredate>October 15, 2005</hiredate>
    <projects>
      <project>
        <product>Printer</product>
        <id>111</id>
        <price>$111.00</price>
      </project>
      <project>
        <product>Laptop</product>
        <id>222</id>
        <price>$989.00</price>
      </project>
    </projects>
  </employee>
</document>` | `<!ELEMENT document (employee)*>
<!ELEMENT employee (name, hiredate, projects)>  
<!ELEMENT name (lastname, firstname)>  
<!ELEMENT lastname (#PCDATA)>  
<!ELEMENT firstname (#PCDATA)>  
<!ELEMENT hiredate (#PCDATA)>  
<!ELEMENT projects (project)*>  
<!ELEMENT project (product, id, price)>  
<!ELEMENT product (#PCDATA)>  
<!ELEMENT id (#PCDATA)>  
<!ELEMENT price (#PCDATA)>`

A parser is the most basic yet most important XML tool. Every XML application is based on a parser. A parser is a software component that sits between the application and the XML files. Its goal is to shield the developer from the intricacies of the XML syntax.
How to parse and validate the XML along with DTD information?

It requires XML validating parsers. There are two types of parsers:

- Non-validating – the parser merely ensures that a data object is well-formed XML
- Validating – the parser uses a DTD (or other type of schema) to ensure the validity of a well-formed data object's form and content

Some parsers work as both types, with configuration switches that determine whether or not the document will be validated.

We make use of MSXML parser, which uses XMLDOM object. To do validation using MSXML parser, the following javascript coding is used.

```html
<html>
<body>
<h3>
This demonstrates a parser error:
</h3>

<script type="text/javascript">
var xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
xmlDoc.async="false";
xmlDoc.validateOnParse="true";
xmlDoc.load("ex18.xml");

document.write("<br />Error Code: ");
document.write(xmlDoc.parseError.errorCode);
document.write("<br />Error Reason: ");
document.write(xmlDoc.parseError.reason);
document.write("<br />Error Line: ");
document.write(xmlDoc.parseError.line);
</script>

</body>
</html>
```

When the XML is valid, the output looks like this:

```
This demonstrates a parser error:

Error Code: 0
Error Reason:
Error Line: 0
```
When XML file is having any invalid statement the output looks like:

![Parser Error](image)

This demonstrates a parser error:

Error Code: -1072898028
Error Reason: Element content is invalid according to the DTD/Schema.
Error Line: 14

More about DTD:

The elements can be structured using DTD and checked against DTD as shown above. In a DTD, elements are declared using the following syntax:

```xml
<!ELEMENT elementname (content-type or content-model)>
```

Using the container elements, one can precisely specify which other elements are allowed inside an element, how often they may appear, and in what order. Example for element container model for the given XML code snippet is as follows:

<table>
<thead>
<tr>
<th>XML code snippet</th>
<th>DTD element container model</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;book&gt;</code></td>
<td><code>&lt;!ELEMENT book (title, author)&gt;</code></td>
</tr>
<tr>
<td><code>&lt;title&gt; XML in nutshell &lt;/title&gt;</code></td>
<td><code>&lt;!--Element content --&gt;</code></td>
</tr>
<tr>
<td><code>&lt;author&gt; elliotte &lt;/author&gt;</code></td>
<td><code>&lt;!ELEMENT title (#PCDATA)&gt;</code></td>
</tr>
<tr>
<td><code>&lt;/book&gt;</code></td>
<td><code>&lt;!ELEMENT author (#PCDATA)&gt;</code></td>
</tr>
</tbody>
</table>

Note: The title, author elements have the content type as PCDATA. PCDATA stands for parsable character data and is used to represent character content. To prevent from keyword with a normal element, the keyword is prefixed by hash.

The following table lists the symbols used while specifying the element content in a DTD:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>,</td>
<td>“and” in specified order</td>
<td>title, author</td>
<td>title and author in that order</td>
</tr>
<tr>
<td></td>
<td>“or”</td>
<td>title</td>
<td>author</td>
</tr>
<tr>
<td>?</td>
<td>“optional” can occur only once</td>
<td>author?</td>
<td>author need not be present, but if it is present, it can occur only once</td>
</tr>
<tr>
<td>*</td>
<td>Zero or multiple occurrence</td>
<td>(title</td>
<td>author)*</td>
</tr>
<tr>
<td>+</td>
<td>At least once occurrence</td>
<td>author+</td>
<td>There can be multiple author elements.</td>
</tr>
</tbody>
</table>
The following table lists the different value types that can be specified for an attribute in a DTD:

<table>
<thead>
<tr>
<th>Value type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCDATA</td>
<td>Used to represent plain text values</td>
</tr>
<tr>
<td>ID</td>
<td>Used to assign a unique value to each element in the document. The ID must begin with an alphabetic character.</td>
</tr>
<tr>
<td>(enumerated)</td>
<td>Used to assign a specific range of values. These values are specified within parenthesis.</td>
</tr>
</tbody>
</table>

The following describes the attribute is mandatory or optional:

<table>
<thead>
<tr>
<th>Attribute type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQUIRED</td>
<td>Value of attribute must be specified each time the element is used in XML</td>
</tr>
<tr>
<td>FIXED</td>
<td>Value of the attribute cannot be changed in the XML document</td>
</tr>
<tr>
<td>IMPLIED</td>
<td>Attribute is optional. Take text strings as their values</td>
</tr>
</tbody>
</table>

Few examples associated with the above tables:

```xml
<!ATTLIST product pid ID #REQUIRED>
```

An attribute called pid is declared for product element. The value type of attribute is set to ID, which indicates it should be unique for each appearance of product element. Also, the #REQUIRED indicates it is mandatory.

```xml
<!ATTLIST product category (TOY|BOOK) “TOY”>
```

The category attribute is declared for the product element. The value type for the attribute is enumerated list, in which default value is toy.

Exercise 20: DTD with CDATA

```xml
<?xml version="1.0"?>
<!DOCTYPE person [ 
 <!ELEMENT person (name+, profession*)> 
 <!ELEMENT name EMPTY> 
 <!ATTLIST name first CDATA #REQUIRED 
 last CDATA #REQUIRED> 
 <!-- The first and last attributes are required to be present 
 but they may be empty. For example, 
 <name first="Cher" last=""> -->
 <!ELEMENT profession EMPTY> 
 <!ATTLIST profession value CDATA #REQUIRED> 
 ]>
```
A CDATA attribute value can contain any string of text acceptable in a well-formed XML attribute value. This is the most general attribute type.

More on CDATA for processing:

When an XML processor parses an XML document, it interprets the markup in that document and replaces entity references. When parsed, those characters will be interpreted as part of the markup unless you convert them to &lt; and &amp;, which is called escaping them. To avoid that, you can specify that you don't want the XML processor to parse part of your text data by placing it in a CDATA section. CDATA stands for character data, as opposed to parsed character data, which is PCDATA.

Exercise 21: CDATA usage

```xml
<?xml version = "1.0" standalone="yes"?>
<document>
  <text>
    Here's how the element starts:
    <![CDATA[
      <employee status="retired">
        <name>
          <lastname>Kelly</lastname>
          <firstname>Grace</firstname>
        </name>
        <hiredate>October 15, 2005</hiredate>
        <projects>
          <project>
            <product>Printer</product>
            <id>111</id>
            <price>$111.00</price>
          </project>
        </projects>
      </employee>
    ]]>}
  </text>
</document>
```

### 3.2.2. XML Schema

An XML schema is used to define the structure of an XML document. A schema defines the list of elements and attributes that can be used in an XML document. In addition to the list of elements, an XML schema also specifies the order in which these elements appear in the XML document and their data types. To define the schema, Microsoft has developed XML schema...
definition languages (XSD). XML schemas have now become a W3C recommendation for creating valid XML documents.

An XML Schema:

- defines elements that can appear in a document
- defines attributes that can appear in a document
- defines which elements are child elements
- defines the order of child elements
- defines the number of child elements
- defines whether an element is empty or can include text
- defines data types for elements and attributes
- defines default and fixed values for elements and attributes

Advantages of XML over DTDs

An XML schema created using XSD is very similar to a DTD, which is also used for defining the structure of an XML document. However, an XML schema created using XSD has many advantages over DTD. Some of them are:

- XSD provides more control over the type of data that can be assigned to elements and attributes as compared to DTD
- DTD does not enable to define own customized data types.
- XSD allows to specify restrictions on data. For example, one can ensure the content of an element is a positive integer value.
- The syntax for defining DTD is different from the syntax of XML document. However, the XSD is the same as the syntax of XML.
- XSD is also supported by variety of parsers.

Figure 2.6: Both DTDs and XML Schemas may be used to define the structure of an XML document.
Exercise 22: XML schema

Create a schema file and XML file

<table>
<thead>
<tr>
<th>Schema file: ex22.xsd</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;?xml version=&quot;1.0&quot;?&gt;</code></td>
</tr>
<tr>
<td><code>&lt;xs:schema xmlns:xs=&quot;http://www.w3.org/2001/XMLSchema&quot;&gt;</code></td>
</tr>
<tr>
<td><code>&lt;xs:element name=&quot;fullName&quot; type=&quot;xs:string&quot;/&gt;</code></td>
</tr>
<tr>
<td><code>&lt;/xs:schema&gt;</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>XML file: ex22.xml</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;?xml version=&quot;1.0&quot;?&gt;</code></td>
</tr>
<tr>
<td><code>&lt;fullName xmlns:xsi=&quot;http://www.w3.org/2001/XMLSchema-instance&quot;</code></td>
</tr>
<tr>
<td><code>xsi:noNamespaceSchemaLocation=&quot;ex22.xsd&quot;&gt;SRM University</code></td>
</tr>
<tr>
<td><code>&lt;/fullName&gt;</code></td>
</tr>
</tbody>
</table>


Various XML schema validate tools: VS.Net, Topologi schematron validator, XML schema quality checker, Xerces, XSD schema validator, XSV, Xerces J, IE

To validate XML schema in IE, follow the steps given below:

1. Download the file iexmltls.exe in your system
2. Run the file and store it in the folder. This downloads the following file in the specified folder:
   - msxmlval.htm
   - msxmlval.inf
   - msxmlvw.htm
   - msxmlvw.inf
3. Navigate to the directory containing the above files.
4. Right-click on each of the .inf files and select the menu option **Install**.

The entries should now be added to the appropriate drop-down menu. By installing these files, entries will be added to the drop-down menu when you right-click on the browser window. These entries will provide the following options:

- Validate XML
- View XSL Output
The above Validation procedure is applied to Exercise 22:

Exercise 23: Create a XML schema for the existing XML file in .Net environment

Note: while choosing validate XML from .net, any error in xml file is blurred with yellow
Exercise 24: HTML file which is used to validate xml file:

```
<HTML>  <HEAD>  <TITLE>   Validating With XML Schemas       </TITLE>
   <SCRIPT LANGUAGE="JavaScript">
      document.write("<H1>Validating With XML Schemas</H1>");
      var parser = new ActiveXObject("Microsoft.DOMDOM");
      parser.validateOnParse = true;
      if (parser.load("ex22.xml")) {
         document.write("The document is valid!");
      } else {  if (parser.parseError.errorCode != 0) {
         document.write(parser.parseError.reason);
      }
   </SCRIPT>
   </HEAD>  <BODY></BODY> </HTML>
```

The output will be:

```
Validating With XML Schemas

The document is valid!
```

Exercise 25: Validate the XML document with the given XSD:

XML file:
```
<?xml version="1.0"?>
<document xmlns="http://xmlpowercorp"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlpowercorp ex25.xsd">
   <text>
      Welcome to XML Schemas!
   </text>
</document>
```

XSD file:
```
<?xml version="1.0"?>
<xsd:schema targetNamespace="http://xmlpowercorp"
   xmlns="http://xmlpowercorp"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   attributeFormDefault="qualified" elementFormDefault="qualified">
   <xsd:element name="document">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="text" type="xsd:string" minOccurs="1" />
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
</xsd:schema>
```
More about XML schema:

XML provides a list of predefined data types. These data types can be classified as:

<table>
<thead>
<tr>
<th>Data type classification</th>
<th>Data types in each classification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive</td>
<td>string</td>
<td>Collection of characters</td>
</tr>
<tr>
<td></td>
<td>decimal</td>
<td>Decimal numbers</td>
</tr>
<tr>
<td></td>
<td>float</td>
<td>32 bit single precision</td>
</tr>
<tr>
<td></td>
<td>boolean</td>
<td>true or false values</td>
</tr>
<tr>
<td></td>
<td>timeDuration</td>
<td>Duration of time</td>
</tr>
<tr>
<td></td>
<td>recurringDuration</td>
<td>Time duration after specific interval</td>
</tr>
<tr>
<td>Derived</td>
<td>integer</td>
<td>From decimal</td>
</tr>
<tr>
<td></td>
<td>long</td>
<td>From integer</td>
</tr>
<tr>
<td></td>
<td>nonNegativeInteger</td>
<td>From integer</td>
</tr>
<tr>
<td></td>
<td>positiveInteger</td>
<td>From nonNegativeInteger</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td>From long</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>recurringDuration</td>
</tr>
<tr>
<td></td>
<td>date</td>
<td>timeDuration</td>
</tr>
<tr>
<td>Atomic</td>
<td>string</td>
<td>A primitive or derived which cannot be further broken</td>
</tr>
<tr>
<td>List</td>
<td>pointlist</td>
<td>Set of values created for particular type</td>
</tr>
<tr>
<td>Union</td>
<td></td>
<td>From atomic and list datatypes</td>
</tr>
</tbody>
</table>

XSD allows definition of custom data types: simple and complex data type.

Element declarations: XML documents are composed primarily of nested elements, and the xs:element element is one of the most often-used declarations in a typical schema.

```xml
<xs:element name="fullName" type="xs:string">
    <xsd:element name="productprice" type="xsd:positiveInteger" />
</xs:element>```

This declaration uses two attributes to describe the element that can appear in the instance document: name and type. The name attribute is self-explanatory, but the type attribute requires some additional explanation. The other few optional attributes to describe xs:element are minOccurs, maxOccurs.

Exercise 26: Develop XML schema file for the following XML file:

```xml
<?xml version= "1.0" ?>
<productdata>
    <product>
        <productname> iPod </productname>  <desc> this is music instrument </desc>
        <price> 3000 </price>  <quantity> 50 </quantity>
    </product>
</productdata>```
Step 1: Define simple data type for child elements

```xml
<xsd:element name="productname" type="xsd:string"/>
<xsd:element name="desc" type="xsd:string"/>
<xsd:element name="price" type="xsd:positiveInteger"/>
<xsd:element name="quantity" type="xsd:nonNegativeInteger"/>
```

Step 2: Define complex data type for the parent element product

```xml
<xsd:element name="product"/>
<xsd:complexType>
    <xsd:sequence>
        <xsd:element name="productname" type="xsd:string"/>
        <xsd:element name="desc" type="xsd:string"/>
        <xsd:element name="price" type="xsd:positiveInteger"/>
        <xsd:element name="quantity" type="xsd:nonNegativeInteger"/>
    </xsd:sequence>
</xsd:complexType>
```

Step 3: Define complex data type for root element with the integration Schema elements

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="productdata" type="pdata"/>
    <xsd:complexType name="pdata">
        <xsd:sequence>
            <xsd:element name="product"/>
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="productname" type="xsd:string"/>
                    <xsd:element name="desc" type="xsd:string"/>
                    <xsd:element name="price" type="xsd:positiveInteger"/>
                    <xsd:element name="quantity" type="xsd:nonNegativeInteger"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```
Step 4: Include xmlschema file for processing in XML file:

```xml
<?xml version="1.0"?>
<productdata xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="ex26.xsd">
    <product>
        <productname>iPOD</productname>
        <desc>this is music instrument</desc>
        <price>3000</price>
        <quantity>50</quantity>
    </product>
</productdata>
```

The output in IE through Microsoft XML validation:

```
<?xml version="1.0"?>
<productdata xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="ex26.xsd">
    <product>
        <productname>iPOD</productname>
        <desc>this is music instrument</desc>
        <price>3000</price>
        <quantity>50</quantity>
    </product>
</productdata>
```

Validation Successful.
File://C:\Documents and Settings\Administrator\Desktop\micodesample\ex26.xml
Additional Information: Students are informed to explore the various schema options given in this example on their own:

XML file:
```xml
<?xml version="1.0"?>
<transaction borrowDate="2003-10-15">
  <Lender phone="607.555.2222">
    <name>Doug Glass</name>
    <street>416 Disk Drive</street>
    <city>Medfield</city>
    <state>MA</state>
  </Lender>
  <Borrower phone="310.555.1111">
    <name>Britta Regensburg</name>
    <street>219 Union Drive</street>
    <city>Medfield</city>
    <state>CA</state>
  </Borrower>
  <note>Lender wants these back in two weeks!</note>
  <books>
    <book bookID="123-4567-890">
      <bookTitle>Earthquakes for Breakfast</bookTitle>
      <pubDate>2003-10-20</pubDate>
      <replacementValue>15.95</replacementValue>
      <maxDaysOut>14</maxDaysOut>
    </book>
    <book bookID="123-4567-891">
      <bookTitle>Avalanches for Lunch</bookTitle>
      <pubDate>2003-10-21</pubDate>
      <replacementValue>19.99</replacementValue>
      <maxDaysOut>14</maxDaysOut>
    </book>
    <book bookID="123-4567-892">
      <bookTitle>Meteor Showers for Dinner</bookTitle>
      <pubDate>2003-10-22</pubDate>
      <replacementValue>11.95</replacementValue>
      <maxDaysOut>14</maxDaysOut>
    </book>
    <book bookID="123-4567-893">
      <bookTitle>Snacking on Volcanoes</bookTitle>
      <pubDate>2003-10-23</pubDate>
      <replacementValue>17.99</replacementValue>
      <maxDaysOut>14</maxDaysOut>
    </book>
  </books>
</transaction>
```
XSD file:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation>Book borrowing transaction schema.</xsd:documentation>
  </xsd:annotation>
  <xsd:element name="transaction" type="transactionType"/>
  <xsd:complexType name="transactionType">
    <xsd:sequence>
      <xsd:element name="Lender" type="address"/>
      <xsd:element name="Borrower" type="address"/>
      <xsd:element ref="note" minOccurs="0"/>
      <xsd:element name="books" type="books"/>
    </xsd:sequence>
    <xsd:attribute name="borrowDate" type="xsd:date"/>
  </xsd:complexType>
  <xsd:element name="note" type="xsd:string"/>
  <xsd:complexType name="address">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="street" type="xsd:string"/>
      <xsd:element name="city" type="xsd:string"/>
      <xsd:element name="state" type="xsd:NM_TOKEN"/>
    </xsd:sequence>
    <xsd:attribute name="phone" type="xsd:string" use="optional"/>
  </xsd:complexType>
  <xsd:complexType name="books">
    <xsd:sequence>
      <xsd:element name="book" minOccurs="0" maxOccurs="10">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="bookTitle" type="xsd:string"/>
            <xsd:element name="pubDate" type="xsd:date" minOccurs='0'/>
            <xsd:element name="replacementValue" type="xsd:decimal"/>
            <xsd:element name="maxDaysOut">
              <xsd:simpleType>
                <xsd:restriction base="xsd:integer">
                  <xsd:maxExclusive value="14"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
        <xsd:attribute name="bookID" type="catalogID"/>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="catalogID">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d{3}-\d{4}-\d{3}"/>
    </xsd:restriction>
  </xsd:complexType>
</xsd:schema>
```

Self-explore: Describe namespace usage, group elements, multiple documents in Schema
4. XML Presentation Technologies

Data is stored in XML document by using elements and attributes. XML focus on data storage not presentation of data. Rendering refers to the act of processing XML documents so that it can be displayed on a variety of targets, such as web browsers, email pagers, and cell phones. XML presentation technologies provide a modular way to deliver and display content to a variety of devices. Here we examine some technologies for display, including CSS, XSL, Xforms, and VoiceXML.

4.1 CSS

Cascading style sheets is an XML-supporting technology for adding style display properties such as fonts, colors, or spacing to Web documents. CSS origins may be traced to the SGML world, which used a style sheet technology called DSSSL to control the display of SGML documents.

Figure 2.10 shows, a style sheet tells a browser or other display engine how to display content. Each rule is made up of a selector—typically an element name such as an HTML heading (H1) or paragraph (P), or a user-defined XML element (Book)—and the style to be applied to the selector. The CSS specification defines numerous properties (color, font style, point size, and so on) that may be defined for an element. Each property takes a value which describes how the selector should be presented.

Exercise 27: Implement CSS presentation:

```javascript
/* Defaults for the entire document */
recipe {font-family: "New York", "Times New Roman", serif;
    font-size: 12pt }

/* Make the dish look like a headline */
dish {
    display: block;
    font-family: Helvetica, Arial, sans-serif;
    font-size: 20pt;
    font-weight: bold;
    text-align: center
}
```
/* A bulleted list */
ingredient {display: list-item; list-style-position: inside}

/* Format these two items as paragraphs */
directions, story {
  display: block;
  margin-top: 12pt;
  margin-left: 4pt
}

XML file:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<?xml-stylesheet type="text/css" href="recipe.css"?>
<recipe source="Marjorie Anderson">
  <dish>Southern Corn Bread</dish>
  <ingredients>
    <ingredient>
      <quantity>1 cup</quantity>
      <component>flour</component>
    </ingredient>
    <ingredient>
      <quantity>4 tablespoons</quantity>
      <component>Royal Baking Powder</component>
    </ingredient>
    <ingredient>
      <quantity>1/2 teaspoon</quantity>
      <component>salt</component>
    </ingredient>
    <ingredient>
      <quantity>1 cup</quantity>
      <component>corn meal</component>
    </ingredient>
    <ingredient>
      <quantity>1 1/2 cups</quantity>
      <component>whole milk</component>
    </ingredient>
    <ingredient>
      <quantity>4 tablespoons</quantity>
      <component>melted butter</component>
    </ingredient>
  </ingredients>
  <directions>
    <step>Sift flour, baking powder, sugar & salt together.</step>
    <step>Add 1 cup corn meal.</step>
    <step>Beat egg in cup and add beaten egg and 1 1/2 cups whole milk to make a batter. Stir well.</step>
    <step>Add melted shortening and beat until light and thoroughly mixed.</step>
  </directions>
</recipe>
```
Pour into greased shallow pan or greased muffin rings.

Bake in hot oven at 425 F for 25 minutes.

Cut into squares if cooked in shallow pan.

This food is well prepared by III MCA students died, Many persons used to like this in Chennai, Tamil Nadu.

The syntax for coding a CSS is as follows:

```css
elementname { property1:value; property2:value; }
```

Example: `quantity { color: RED; }
```

When the properties to be set for more than one element at the same time, it can be combined as a single group. For example, if quantity and component both to be displayed in red color,

```css
quantity, component { color: RED; }
```
4.2 XSL Formatting Objects (XSL-FO)

XSL 1.0 is a W3C Recommendation that provides users with the ability to describe how XML data and documents are to be formatted. XSL does this by defining "formatting objects," such as footnotes, headers, or columns.

An XSL style sheet is basically a series of pattern-action rules and looks like an XML document with a mixture of two kinds of elements: those defined by XSL and those defined by the object language. The patterns are similar to CSS's selectors, but the action part may create an arbitrary number of "objects." The action part of the rule is called the "template" in XSL, and a template and a pattern together are referred to as a "template rule."

The result of applying all matching patterns to a document recursively is a tree of objects, which is then interpreted top-down according to the definition of each object. For example, if they are HTML objects, an HTML document will be generated; if they are XML objects, XML will be the result.

Comparison of XSL and CSS

<table>
<thead>
<tr>
<th></th>
<th>CSS</th>
<th>XSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can be used with HTML?</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Can be used with XML?</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Transformation language?</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Syntax</td>
<td>CSS</td>
<td>XML</td>
</tr>
</tbody>
</table>

Figure 2.11 illustrates some of the different options for using CSS and XSL to create displays based on HTML or XML. The general principle is that if the document is to be simply rendered and not transformed in any way through the addition or deletion of items, then CSS is the more straightforward approach.
Exercise 28: Implement XSL formatting

XML file:
```xml
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="ex28.xsl"?>
<states>
  <state>
    <name>California</name>
    <population units="people">33871648</population><!--2000 census-->
    <capital>Sacramento</capital>
    <bird>Quail</bird>
    <flower>Golden Poppy</flower>
    <area units="square miles">155959</area>
  </state>
  <state>
    <name>New York</name>
    <population units="people">18976457</population><!--2000 census-->
    <capital>Albany</capital>
    <bird>Bluebird</bird>
    <flower>Rose</flower>
    <area units="square miles">47214</area>
  </state>
</states>
```

XSL file:
```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="states">
    <HTML>
      <BODY>
        <xsl:apply-templates/>
      </BODY>
    </HTML>
  </xsl:template>
  <xsl:template match="state">
    <P>
      <b><i><xsl:value-of select="name"/></i></b>
      <xsl:value-of select="capital"/>
      <xsl:value-of select="bird"/>
      <xsl:value-of select="flower"/>
      <xsl:value-of select="area" units="square miles"/>
    </P>
  </xsl:template>
</xsl:stylesheet>
```

The output looks like:
4.3 XFORMS

XForms is an XML approach that overcomes the limitations of HTML forms. XForms is a GUI toolkit for creating user interfaces and delivering the results in XML. Figure 2.12 illustrates how a single device-independent XML form definition, called the XForms Model, has the capability to work with a variety of standard or proprietary user interfaces. For example, the Voice Browser Working Group is looking at developing voice-based user interface components for XForms.

Self-Explore: Code for XFORMS

```xml
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:ev="http://www.w3.org/2001/xml-events"
     xmlns:xforms="http://www.w3.org/2002/xforms/cr">
  <head>
    <xforms:model>
      <xforms:submission localfile="data.xml"/>
      <xforms:instance>
        <data xmlns="">
          <input>Hello!</input>
          <select>1</select>
          <selectboolean>true</selectboolean>
          <message>Hello!</message>
        </data>
      </xforms:instance>
    </xforms:model>
  </head>

  <body>
    <h1>Using XForms</h1>
    <p>Input Control</p>
    <xforms:input ref="/data/input"></xforms:input>
    <p>Select Control</p>
    <xforms:select appearance="full" ref="/data/select">
      <xforms:item>
        <xforms:value>1</xforms:value>
        <xforms:label>Item 1</xforms:label>
      </xforms:item>
      <xforms:item>
        <xforms:value>2</xforms:value>
      </xforms:item>
    </xforms:select>
  </body>
</html>
```
4.4 XHTML

The capability of XHTML to be more flexible than HTML is attributable to the use of XHTML modules for creating XHTML-conforming markup languages. New XHTML-compliant languages must use the basic XHTML framework as well as other XHTML modules. As illustrated in Figure 2.13, modules plug together within the XHTML framework to define a markup language that is task or client specific. Documents developed based on the new markup language will be usable on any XHTML-conforming clients.
4.5 VoiceXML

VoiceXML is an emerging standard for speech-enabled applications. Its XML syntax defines elements to control a sequence of interaction dialogs between a user and an implementation platform. The elements defined as part of VoiceXML control dialogs and rules for presenting information to and extracting information from an end-user using speech. Figure 2.14 illustrates, VoiceXML documents are stored on Web servers. Translation from text to voice is carried out either on a specialized server that delivers voice directly to a phone or by the device itself using speech processing technology.

![Figure 2.14. VoiceXML documents are used to drive voice interactions over conventional or wireless phones.](image)

5. XML Infoset, Canonical XML

5.1 XML Infoset

XML documents excel at storing data, and this has led developers to wonder if XML will ultimately be able to solve an old problem: being able to directly compare and classify the data in multiple documents. For example, consider the World Wide Web as it stands today: There can be thousands of documents on a particular topic, but how can you possibly compare them? For example, a search for the term XML turns up millions of matches, but it would be extraordinarily difficult to write a program that would compare the data in those documents because all that data isn't stored in any remotely compatible format.

The idea behind XML information sets, also called infosets, is to set up an abstract way of looking at an XML document so that it can be compared to others. To have an infoset, XML documents may not use colons in tag and attribute names unless they are used to support
namespaces. Documents do not need to be valid to have an infoset, but they need to be well formed.

An XML document's information set consists of two or more information items (the information set for any well-formed XML document contains at least the document information item and one element information item). An information item is an abstract representation of some part of an XML document, and each information item has a set of properties, some of which are considered core and some of which are considered peripheral.

An XML information set can contain 15 different types of information items:

- A document information item (core)
- Element information items (core)
- Attribute information items (core)
- Processing instruction information items (core)
- Reference to skipped entity information items (core)
- Character information items (core)
- Comment information items (peripheral)
- A document type definition information item (peripheral)
- Entity information items (core for unparsed entities, peripheral for others)
- Notation information items (core)
- Entity start marker information items (peripheral)
- Entity end marker information items (peripheral)
- CDATA start marker information items (peripheral)
- CDATA end marker information items (peripheral)
- Namespace declaration information items (core)

### 5.2 Canonical XML:

Although infosets are a good idea, they are only abstract formulations of the information in an XML document. So without reducing an XML document to its infoset, how can you actually approach the goal of being able to actually compare XML documents byte by byte?. It turns out that there is a way: You can use canonical XML. Canonical XML is a companion standard to XML. The canonical XML syntax is very strict; for example, canonical XML uses UTF-8 character encoding only, carriage-return linefeed pairs are replaced with linefeeds, tabs in CDATA sections are replaced by spaces, all entity references must be expanded, and much more, as specified in [www.w3.org/TR/xml-c14n](http://www.w3.org/TR/xml-c14n). Because canonical XML is intended to be byte-by-byte correct, the upshot is that if you need a document in canonical form, you should use software to convert your XML documents to that form.

One such package that will convert valid XML documents to canonical form comes with the XML for Java software that you can get from IBM's AlphaWorks (the Web site is http://www.alphaworks.ibm.com/tech/xml4j).
6. XML Transformation

The transformation language lets you transform the structure of documents into different forms (such as PDF, WML, HTML, or another schema type), while the formatting language actually formats and styles documents in various ways. These two parts of XSL can function quite independently, and you can think of XSL as two languages, not one. In practice, you often transform a document before formatting it.

XSL is made up of the following parts:

- XSL Transformation (XSLT): It is an XML based language that allows to transform an XML document into another XML document.
- XMLPath (XPath): It is a language that is used to access different parts of an XML document, such as elements and attributes.
- XSL formatting objects (XSL-FO), which specify how the data is to be displayed.

One can transform documents in three ways:

- On the server— A server program, such as a Java servlet or a JavaServer Page (JSP), can use a stylesheet to transform a document automatically and serve it to the client. One such example is the XML Enabler, which is a servlet you'll find at the XML For Java Web site, www.alphaworks.ibm.com/tech/xml4j.
- On the client— A client program, such as a browser, can perform the transformation, reading in the stylesheet that you specify with the `<?xml-stylesheet?>` processing instruction. Internet Explorer can handle transformations this way, to some extent.
- With a separate program —Several standalone programs, usually based on Java, will perform XSLT transformations.

6.1 XSLT

To perform an XSL transformation, a program referred to as an XSLT processor reads both an XML document and an XSLT document that defines how to transform the XML (see Figure 2.15). An XSLT processor has the capability to read the XML source document, and rearrange and reassemble it in a variety of ways, even adding new text and tags.
Comparison between XSLT and CSS:

<table>
<thead>
<tr>
<th>CSS</th>
<th>XSLT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple to use and suitable for simple documents</td>
<td>Complex to use. It is a superset of CSS functionality.</td>
</tr>
<tr>
<td>Cannot reorder, add, delete, or perform operations on elements</td>
<td>Can render, add, delete, elements since it is aware of the structure of XML document</td>
</tr>
<tr>
<td>Does not offer access to non-elements, such as attributes and their values and processing instructions.</td>
<td>Able to access and manipulate the comments, processing instructions, and attribute values and names with an XML document</td>
</tr>
<tr>
<td>Uses less memory since it cannot reorder a document and therefore, does not need to build a tree representation of the document.</td>
<td>Uses more memory and processor power since reordering, adding, deleting and manipulating elements require a tree representation of the document in the memory.</td>
</tr>
<tr>
<td>Uses a different syntax than XML</td>
<td>Written using XML and therefore has the same syntax as that of XML</td>
</tr>
</tbody>
</table>

Working of XSLT

- Xslt style sheet
- Xml doc
The XSLT processor comes packaged along with MSXML parser. Since XSLT is an application of XML, the MSXML parser is also used to parse an XSLT document. The MSXML parser parses the XSLT stylesheet and creates a tree structure based on the elements and attributes used in an XSLT document. This tree is referred to as the XSLT tree.

The XSLT processor component of the MSXML parser takes the transformation information contained in the XSLT stylesheet, applies it to the data retrieved from the source document, and builds a resultant tree structure referred to as the result tree. This tree is then rendered to various targets, such as web browsers, pagers, and cell phones.

XSLT provides a number of elements for selecting and formatting data.

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

The stylesheet element is the root element of all XSLT stylesheets. The xsl prefix contains a reference to the namespace-URI for XSLT.

value-of element
<xsl:value-of select="productname" />

The value-of-element is an empty element. It is used to represent the name of the element or attribute whose value is to be displayed. To display the value of attribute @ symbol should be used:
<xsl:value-of select="@category" />

for-each element: This element is used to instruct the XSLT processor to process the information for each instance of the specified pattern. Parent/Child pattern:

<xsl:for-each select="productdata/product">
  <font color="blue"> <xsl:value-of select="productname" /> </font>
</xsl:for-each>

For ancestor-child pattern : ancestor//child

sort element: XSLT provides the sort element for sorting data based on values assigned to elements and attributes.
<xsl:sort select="PRICE" data-type="number" order="ascending" />

text element
The text element allows to generate constant text in the output. This element can be used to display labels.
<xsl:text> Product Name : </xsl:text>
XSLT template rules:

A template rule describes how an XML element and its contents are converted into a format that can be displayed in the browser. A template rule consists of two parts:

- A pattern that identifies an XML element in an XML document
- An action or processing code that details the transformation and rendering of the resulting element.

The template element: It is used to define a template for desired output. The syntax is as follows:

```xml
<xsl:template match="pattern">  
    [action to be taken]  
</xsl:template>
```

<table>
<thead>
<tr>
<th>Pattern example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;xsl:template match=&quot;/&quot;/&gt;</code></td>
<td>Start performing the action from the root element</td>
</tr>
<tr>
<td><code>&lt;xsl:template match=&quot;*&quot;/&gt;</code></td>
<td>Matches any element in an XML document</td>
</tr>
<tr>
<td><code>&lt;xsl:template match=&quot;pdt&quot;/&gt;</code></td>
<td>When the named element is encountered, perform the action</td>
</tr>
<tr>
<td>`&lt;xsl:template match=&quot;fn</td>
<td>ln&quot;/&gt;`</td>
</tr>
<tr>
<td><code>&lt;xsl:template match=&quot;pdt[@pid]&quot;/&gt;</code></td>
<td>Perform the action on the pid attribute of pdt element</td>
</tr>
<tr>
<td><code>&lt;xsl:template match=&quot;pdt[@pid='1']&quot;/&gt;</code></td>
<td>Match all the pdt elements that have a pid attribute value of ‘1’</td>
</tr>
<tr>
<td><code>&lt;xsl:template match=&quot;pdt/pname&quot;/&gt;</code></td>
<td>Perform the action on pname element having the pdt element as parent</td>
</tr>
<tr>
<td><code>&lt;xsl:template match=&quot;pdtdata//pname&quot;/&gt;</code></td>
<td>Perform the action on pname element having the pdtdata element as ancestor</td>
</tr>
</tbody>
</table>

The apply-templates elements

This element is used to instruct the XSLT processor to find an appropriate template and perform the specified tasks on each selected element. The syntax for using this element is as follows:

```xml
<xsl:apply-templates [select="pattern"/>]
```

The select attribute is optional and is used to specify the context in which the template should be executed. The default value for this attribute is “node()”, which means that the template should be executed for the children of the current node. The apply-templates element directs the XSLT processor to find an appropriate template to apply.
Exercise 29: XSLT transformation with various options

XML file:
```xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="ex29a.xsl"?>
<states>
  <state>
    <name>California</name>
    <population units="people">33871648</population><!-- 2000 census -->
    <capital>Sacramento</capital>
    <bird>Quail</bird>
    <flower>Golden Poppy</flower>
    <area units="square miles">155959</area>
  </state>
  <state>
    <name>New York</name>
    <population units="people">18976457</population><!-- 2000 census -->
    <capital>Albany</capital>
    <bird>Bluebird</bird>
    <flower>Rose</flower>
    <area units="square miles">47214</area>
  </state>
</states>
```

Ex29a.XSL file:
```xml
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>
</xsl:stylesheet>
```

Ex29b.XSL file:
```xml
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:apply-templates select="states/state"/>
  </xsl:template>
  <xsl:template match=""states/state"">
    <font color="green">
      <LI> <xsl:apply-templates /> </LI>
    </font>
  </xsl:template>
</xsl:stylesheet>
```
Exercise 30: Implement for-each and sort element option:

```xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="ex30.xsl"?>

<messages>
  <note id="501">
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don't forget to watch FIFA!</body>
  </note>
  <note id="502">
    <to>Jani</to>
    <from>Tove</from>
    <heading>Re: Reminder</heading>
    <body>I will not</body>
  </note>
</messages>

Ex30.xsl:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:for-each select="messages/note">
      <xsl:sort select="@id" order="ascending"/>
      <LI>
        <xsl:text>Note id: </xsl:text>
        <font color="red"><xsl:value-of select="@id" /></font>
        <xsl:text>To:   </xsl:text>
        <xsl:apply-templates select="to"/>
        <xsl:text> From:   </xsl:text>
        <xsl:apply-templates select="from"/>
        <xsl:text> Subject Heading   </xsl:text>
        <xsl:apply-templates select="heading"/>
        <xsl:text> Body of the text   </xsl:text>
        <xsl:apply-templates select="body"/>
      </LI>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
Exercise 31: Make use of if statement and choose-when

```
<?xml version= "1.0" ?>
<?xml-stylesheet type="text/xsl" href="ex31a.xsl"?>

<projects>
    <project>
        <product>Printer</product>
        <id>111</id>
        <price>111</price>
    </project>
    <project>
        <product>Laptop</product>
        <id>222</id>
        <price>989</price>
    </project>
    <project>
        <product>Keyboard</product>
        <id>555</id>
        <price>129</price>
    </project>
    <project>
        <product>Mouse</product>
        <id>666</id>
        <price>25</price>
    </project>
</projects>
```

Ex31a.xsl : using if statement

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <xsl:for-each select="projects/project">
            <xsl:if test="price[. &gt; 500]">
                <xsl:value-of select="product"/>
            </xsl:if>
        </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

The output looks like:

```
Laptop
```
Ex31b.xsl : using when choose statement

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:for-each select="projects/project">
      <xsl:choose>
        <xsl:when test="price[. &lt; 500]">
          <font color="red"> <xsl:value-of select="product" /> </font>
        </xsl:when>
        <xsl:otherwise>
          <font color="green">  <xsl:value-of select="product" /> </font>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

Note: The XSLT if element provides a simple if-then construct. The choose element is used to make a choice when there are two or more possible course of action. It provides a means for testing multiple conditions.

Various comparison and Boolean operators:

```plaintext
=  ! =  &lt;  &gt;  &lt;=  &gt;=  and  or  not
```
Additional information to explore students by their own:

On Java Server side (JSP) transformation is having the following Code snippet.

```jsp
<%@ page import="javax.xml.transform.*,
javax.xml.transform.stream.*, java.io.*" %>
<%
try
{
TransformerFactory transformerfactory = TransformerFactory.newInstance();

Transformer transformer = transformerfactory.newTransformer(new StreamSource(new
File (application.getRealPath("/") + "ex13.xsl")));
transformer.transform(new StreamSource(new File(application.getRealPath("/") +
"ex13.xml")), new StreamResult(new File(application.getRealPath("/") +
temp.html")));
}
catch(Exception e) {}

FileReader filereader = new FileReader(application.getRealPath("/") +"temp.html");
BufferedReader bufferedreader = new BufferedReader(filereader);

String instring;
while((instring = bufferedreader.readLine()) != null) {
<%= instring %>

}%
<%
filereader.close();
%
```

Sandalone program to transform XML documents:

```java
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import java.io.*;

public class ch13_05
{
    public static void main(String args[])
    {
        try
        {
            TransformerFactory transformerfactory = TransformerFactory.newInstance();
            Transformer transformer = transformerfactory.newTransformer (new
            StreamSource(new File(args[1])));
            
        }
    }
}
```
6.2 XPATH

The XML 1.0 syntax provides a straightforward, standard way to exchange information between computer programs. The XML Path Language, XPath, plays an important part in such exchange of information between computers or computer applications. XPath is used to navigate XML tree structures. XPath gets its name from its use of a path notation to navigate through the hierarchical tree structure of an XML document. Because all XML documents can be represented as a tree of nodes, XPath allows for the selection of a node or group of nodes through the use of a compact, non-XML syntax. It is an important XML technology due to its role in providing a common syntax and semantics for functionality in both XSLT and XPointer.

Figure 2.16 shows that XPath operates on the hierarchical tree structure of an XML document rather than its tag-based syntax. It is capable of distinguishing between different types of nodes, including element nodes, attribute nodes, and text node. XPath is designed to enable addressing of, or navigation to, chosen parts of an XML document. In support of that aim, XPath provides a number of functions for the manipulation of strings, numbers, Booleans, and node sets.

Exercise 32: Implement XPATH

```xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="ex31a.xsl"?>
<purchase>
  <product id="p001" name="computer" price="45000">
    <order id="o01"> <quantity> 30 </quantity> </order>
    <order id="o02"> <quantity> 40 </quantity> </order>
  </product>

  <product id="p002" name="printer" price="10000">
    <order id="o03"> <quantity> 24 </quantity> </order>
    <order id="o04"> <quantity> 50 </quantity> </order>
  </product>
</purchase>
```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <p align="center"> <font color="blue"> Computer shoppee </font> </p>
    <xsl:for-each select="purchase/product">
      <b> product id: </b> <xsl:value-of select="@id"/> <br/>
      Name : <xsl:value-of select="@name"/> <br/>
      Price per unit: <xsl:value-of select="@price"/> <br/>
    </xsl:for-each>

    <xsl:for-each select="order">
      <b> order id: </b> <xsl:value-of select="@id"/> <br/>
      quantity <xsl:value-of select="@quantity"/> <br/>
      Order value <xsl:value-of select="(./@price)*quantity"/> <br/>
    </xsl:for-each>

    <hr/>

    <b> Product sales value: </b> <xsl:value-of select="(./@price)*sum(./order/quantity)"/>
  </xsl:template>
</xsl:stylesheet>
Exercise 33: Data in tabular form

```xml
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet href="Ex33.xsl" type="text/xsl" ?>
<SalesReport>
  <Company>XMML.com</Company>
  <Period>2001-2002</Period>
  <Sales Region="EU">50,000</Sales>
  <Sales Region="NA">150,000</Sales>
  <Sales Region="AU">10,000</Sales>
</SalesReport>
```

Ex33.xsl

```xml
<?xml version='1.0'?>
xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
xsl:output method="html" omit-xml-declaration='yes' />
xsl:template match="/"
  <html>
    <head>
    </head>
    <body>
      <table width="50%">
        <tr><th>Region</th><th>Sales ($ 000's)</th></tr>
        <xsl:for-each select="/SalesReport/Sales">
          <tr><td align="center"><xsl:value-of select="@Region"/></td>
            <td align="center"><xsl:value-of select="."/></td>
          </tr>
        </xsl:for-each>
      </table>
    </body>
  </html>
</xsl:stylesheet>
```

The output looks like:

**XMML.com, Sales Report: 2001-2002**

<table>
<thead>
<tr>
<th>Region</th>
<th>Sales ($ 000's)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EU</td>
<td>50,000</td>
</tr>
<tr>
<td>NA</td>
<td>150,000</td>
</tr>
<tr>
<td>AU</td>
<td>10,000</td>
</tr>
</tbody>
</table>
6.3 XLINK

XLink will enable bidirectional Web linking. The notion of resources is universal to the World Wide Web. According to the Internet Engineering Task Force, a "resource" is any addressable unit of information or service. Examples include files, images, documents, programs, and query results. These resources are addressed using a URI reference. What XLink brings to the table is the ability to address a portion of a resource. For example, if the entire resource is an XML document, a useful portion of that resource might be a single element within the document.

Exercise 34: To implement XLINK:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/css" href="ex34.css"?>
<insurance>
  <title>Supporting XLinks</title>
  Looking for health insurance
  <link xmlns:xlink = "http://www.w3.org/1999/xlink"
    xlink:type = "simple"
    xlink:show = "new"
    xlink:href="http://www.w3c.org"
    onclick="location.href='http://www.w3c.org/default.htm'">
    health insurance
  </link>
</insurance>
```

6.4 XPOINTER

When you use XLinks, you can link to a particular document, but many times, you want to be more precise than that. XPointers let us point to specific locations inside a document, and they are coming into more common use.

The XML Pointer language, XPointer, is intended as a provider of fragment identifiers for XML documents. Expressed more formally, XPointer is the fragment identifier language for resources whose type is one of text/xml, application/xml, text/xmlexternal-parsed-entity, or application/xml-external-parsed-entity.

XPointer is based on XPath. Conformant XPointer processing depends on XPath processing. XPointer provides a means of examining the hierarchical representation of an XML document and the choice of internal parts of that representation by means of selecting, for example, element types, attribute values, and character content.
Consider the code snippet:

```xml
<list ID="MyList">
    <item>First</item>
    <item>Second</item>
    <item>Third</item>
</list>
```

If we want to refer to what XPointer terms a singleton location-set, consisting of the second of the items within the `<list>` element, we would use the following XPointer:

```
xpointer(id('MyList')/item[1]/range-to(following-sibling::item[2]))
```

6.6 XQUERY

XQuery is a W3C initiative to define a standard set of constructs for querying and searching XML documents. The XML Query Working Group draws its membership from both the document and the database communities, trying to hammer out an agreement on XML-based query syntax that meets the needs of both cultures.

XQuery is a standardized language that can be used to query XML documents much as SQL is used to query relational database tables. Essentially, XQuery consists of a data model and a set of query operators that operate on that data model.

Consider the example:

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<bookstore>
    <book category="COOKING">
        <title lang="en">Everyday Italian</title>
        <author>Giada De Laurentiis</author>
        <year>2005</year>
        <price>30.00</price>
    </book>
    <book category="CHILDREN">
        <title lang="en">Harry Potter</title>
        <author>J K. Rowling</author>
        <year>2006</year>
        <price>29.99</price>
    </book>
</bookstore>
```

The following XQUERY predicate is used to select all the book elements under the bookstore element that have a price element with a value that is less than 30:

```
The doc function in XPATH:

doc("books.xml")/bookstore/book[price<30]
```

Equivalent XQUERY statement for the above XPATH statement is

```xquery
for $i in doc("books.xml")/bookstore/book
where $i/price<30
order by $i/title
return $i/title
```