Unit 3
Working with XML

Introduction to XML:

XML stands for e**X**tensible **M**arkup **L**anguage and is a text-based markup language derived from Standard Generalized Markup Language (SGML). The primary purpose of this standard is to provide way to store self describing data easily. Self-describing data are those that describe both their structure and their content. But, HTML documents describe how data should appear on the browsers screen and no information about the data. XML documents, on the other hand describe the meaning of data. The content and structure of XML documents are accessed by software module called XML processor.

XML Characteristics:

1. **XML is extensible :** XML essentially allows you to create your own language, or tags, that suits your application.
2. **XML separates data from presentation :** XML allows you to store content with regard to how it will be presented.
3. **XML is a public standard :** XML was developed by an organization called the World Wide Web Consortium (W3C) and available as an open standard.

XML Usage:

A short list of XML's usage says it all

- XML can work behind the scene to simplify the creation of HTML documents for large web sites.
- XML can be used to exchange of information between organizations and systems.
- XML can be used for offloading and reloading of databases.
- XML can be used to store and arrange data in a way that is customizable for your needs.
- XML can easily be mixed with stylesheets to create almost any output desired.

XML features:

- XML allows the user to define his own tags and his own document structure.
- XML document is pure information wrapped in XML tags.
- XML is a text based language, plain text files can be used to share data.
- XML provides a software and hardware independent way of sharing data.

XML document structure

An XML document consists of following parts: 1) Prolog 2) Body

1. Prolog:

This part of XML document may contain following parts: XML declaration, Optional processing instructions, Comments and Document Type Declaration

XML Declaration:

Every XML document should start with one-line XML declaration which describes document itself. The XML declaration is written as below:

Syn: <?xml version="1.0" encoding="UTF-8"?>

Where *version* is the XML version and *encoding* specify the character encoding used in the document. UTF-8 stands for Unicode Transformation Format is used for set of ASCII characters. It also have *standalone* attribute indicates whether the document can be processed as standalone document or is dependent on other document like Document Type Declaration(DTD).

Syn: <?xml version="1.0" encoding="UTF-8" standalone="yes|no"?>

Processing Instruction:

Processing Instructions starts with left angular bracket along with question mark(<?),ending with question mark followed by the right angular bracket(?>). These parameters instruct the application about how to interpret XML document. XML parser's do not take care of processing instructions and are not text portion of XML document.

Ex: <?xsl-stylesheet href="simple.xsl" type="text/xsl"?>

Comments:
Like HTML, comments may use anywhere in XML documents. An XML comments starts with <!—and ends with -->. Everything with in these will be ignored by the parsers and will not be parsed.

Syn: <!-- this is comments -->

Following points should be remembered while using comments: do not use double hyphens, never place inside entity declaration or within any tag, never place before XML declaration

Document Type Declaration(DTD):

XML allows to create new tags and have meaning if it has some logical structure created using set of related tags. <!DOCTYPE > is used to specify the logical structure of XML document by imposing constraints on what tags can be used and where. DTD may contain Name of root element, reference to external DTD, element and entity declarations.

2. Body:

This portion of XML document contains textual data marked up by tags. It must have one element called Document or Root element, which defines content in the XML document. Root element must be the top-level element in the document hierarchy and there can be one and only one root element.

Ex: <?xml version="1.0"?>

```
<book>
    <title>WT</title>
    <author>Uttam Roy</author>
    <price>500</price>
</book>
```

In this document, the name of root element id <book> which contains sub tags <title>, <author> and <price>. Each of these tags contains text "WT", "Uttam Roy" and "500" respectively.

XML Elements

An XML element consists of starting tag, an ending tag and its contents and attributes. The contents may be simple text or other element or both. XML tags are very much similar to that of HTML tags. A tag begins with less than(<) and ends with greater than(>) character. It takes the form <tag-name> and must have corresponding ending tag(</tag-name>). An element consists of opening tag, closing tag and contents. Few tag may not contain any content and hence know as Empty elements. According to the well-formedness constraint, every XML element must have closing tag. XML provides two ways for XML empty elements as follows:

*Syn:
</br> or
*

Following are the rules that need to be followed for XML elements:

- An element *name* can contain any alphanumeric characters. The only punctuation allowed in names are the hyphen (-), under-score (_) and period (.)
- Names are case sensitive. For example Address, address, ADDRESS are different names
- Element start and end tag should be identical
- An element which is a container can contain text or elements as seen in the above example

Attributes: Attributes are used to describe elements or to provide more information about elements. They appear in the starting tag of element. The syntax of specifying an attribute in element is:

Syn: <element-name attribute-name="value">...</elment-name>

Ex: <employee gener="male">ABCD</employee>

There is no strict rules that describes when to use elements and when to use attributes. However, it is recommended not to use attributes as far as possible due to following reasons:

- Too many attributes reduce readability of XML document
- Attributes cannot contain multiple values, but elements can
- Attributes are not easily extendable
- Attributes cannot represent logical structure, but elements together with their child elements can
- Attributes are difficult to access by parsers

- Attribute values are not easy to check against DTD

Well-formed XML:

An XML document is said to be well-formed if it contains text and tags that conform with the basic XML well-formedness constraints. XML can extend existing documents by creating new elements that fit their applications. The only thing is to remember the well-formedness constraints. The following rules must be followed by XML documents:

- An XML document must have one and only one root element
- All tags must be closed
- All tags must be properly nested
- XML tags are case-sensitive
- Attributes must always be quoted
- Certain characters are reserved for processing like pre-defined entities

Pre-defined Entities: W3C specification defined few entities each of which represents a special character that cannot be used in XML document directly. All XML processors must recognize those entities, whether they are declared or not.

Entity Name	Entity Number	Description	Character
&lt;	&#60;	Less than	<
&gt;	&#62;	Greater than	>
&amp;	&#38;	Amprersand	&
&quot;	&#34;	Quotation mark	“
&apos;	&#39;	Apostrophe	‘

Valid XML

Well-formed XML documents obey only basic well formedness constraints. So, valid XML documents are those that are well formed and comply with rules specified in DTC or Schema.

Name Space

XML was developed to be used by many applications. If many applications want to communicate using XML documents, problems may occur. In XML document, element and attribute names are selected by developers. In some cases two different documents may have same root element. For example, both client.xml and server.xml contains same root tag <config> as shown below.

Client.xml	**Server.xml**
<config> <version>1.0</version> </config>	<config> <version>1.0</version> </config>

XML namespace provides simple, straightforward way to distinguish between element names in XML document. Namespace suggests to use prefix with every element as follows:

Client.xml	**Server.xml**
<c:config> <c:version>1.0</c:version> </c:config>	<s:config> <s:version>1.0</s:version> </s:config>

Uniform Resource Identifier(URI) is used to guarantee the prefixes used by different developers. In general URL are used to choose unique name. But, URL must be prefixed for each tag instead of them we use prefix. Prefixes are just shorthand placeholders of URLs. Association of prefix and URL is done in the starting tag using reserved XML attribute *xmlns*.

Syn: xmlns:prefix=”URI”

Name Space Rules: The *xmlns* attribute identifies namespace and makes association between prefix and created namespace. Many prefixes may be associated with one namespace.

Default Namespace: Namespaces may not have their associated prefixes and are called default namespace. In such cases, a blank prefix is assumed for element and all of its descendants.

Document Type Declaration (DTD)

XML Schema languages:

Schema is an abstract representation of object characteristics and its relationship to other objects. An XML schema represents internal relationship between elements and attributes in XML document. It defines structure of XML documents by specifying list of valid elements and attributes. XML schema language is a formal language to express XML schemas. Most popular and primary schema languages are: DTD and W3C Schema.

1. Document Type Declaration(DTD):

It is one of the several XML schema languages and was introduced as part of XML 1.0. Even though DTD is not mandatory for an application to read and understand XML document, many developers recommend writing DTDs for XML applications. Using DTD we can specify various elements types, attributes and their relationship with in another. Basically DTD is used to specify set of rules for structuring data in any XML file.

1.1 Using DTD in XML document:

To validate XML document against DTD, we must tell validator where to find DTD so that it knows rules to be verified during validation. A Document Type Declaration is used to make such link and DOCTYPE keyword is used for this purpose. There are three ways to make this link: Internal DTD, External DTD and Combined internal and external.

1. Internal DTD:

When we embed DTD in XML document, DTD information is included within XML document itself. Specifically, DTD information is placed between square brackets in DOCTYPE declaration. The general syntax of internal DTD is

Syn: *<!DOCTYPE root-element [*
element-declarations
]>

Where *root-element* is the name of root element and *element-declarations* is where we declare the elements. Since every XML document must have one and only one root element, this is also structure definition of the entire document. Here, DOCTYPE must be in uppercase, document type declaration must appear before first element and name following word DOCTYPE i.e. root-element must match with name of root element.

Advantage of internal DTD is that we have to handle only single xml document instead of many which is useful for debugging and editing. It is a good idea to use with smaller sized documents. Problem of internal DTD is that it makes documents difficult to read for big sized document.

Ex: <?xml version="1.0" ?>

```
<!DOCTYPE bookstore [
        <!ELEMENT bookstore (book*)>
        <!ELEMENT book (title,author,price)>
        <!ELEMENT title (#PCDATA)>
        <!ELEMENT author (#PCDATA)>
        <!ELEMENT publisher (#PCDATA)>
        <!ELEMENT price (#PCDATA)>
]>
<bookstore>
    <book>
    <title>WT</title>
```

```
        <author>Uttam Roy</author>
        <publisher>Oxford</publisher>
        <price>500</price>
        </book>
        <book>
        <title>AJ</title>
        <author>Schildt</author>
        <publisher>TMH</publisher>
        <price>200</price>
        </book>
</bookstore>
```

2. External DTD:

Another way of connection DTD to XML document is to reference it with in XML document i.e. create separate document, put DTD information there and point to it from XML document. The general syntax for external DTD is.

Syn: *<!DOCTYPE root-element SYSTEM | PUBLIC "uri">*

Where *uri* is the Uniform Resource Identifier of the *.dtd* file. This declaration states that we are going to define structure of root-element of XML document and its definition can be found from *uri* specified like *book.dtd*. both *xml* and *dtd* files should be kept in same directory.

Ex:

book.xml

```
<?xml version="1.0" ?>
<!DOCTYPE book SYSTEM "book.dtd">
<bookstore>
<book>
        <title>WT</title>
        <author>Uttam Roy</author>
        <publisher>Oxford</publisher>
        <price>500</price>
</book>
<book>
        <title>AJ</title>
        <author>Schildt</author>
        <publisher>TMH</publisher>
        <price>200</price>
</book>
</bookstore>
```

book.dtd

```
<!ELEMENT bookstore (book*)>
<!ELEMENT book (title,author,price)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
<!ELEMENT price (#PCDATA)>
```

Location of DTD need not always be local file, it can be any valid URL. Following declaration for XHTML uses PUBLIC DTD:

Syn: *<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.0 Transitional//EN'>*

Disadvantage of using separate DTD is we have to deal with two documents.

3. Combining Internal and External DTD:

External DTD are useful for common rules for set of XML documents, whereas internal DTDs are beneficial for defining customized rules for specific document. XML allows to combine both internal and external DTD for complete collection of rules for given document. The general form of such DTD is:

Syn: *<!DOCTYPE root-element SYSTEM | PUBLIC "uri" [DTD declarations…] >*

Ex:

```
<?xml version="1.0" ?>
<!DOCTYPE book SYSTEM "book.dtd"
[       <!ELEMENT excl '&#21;'>
]>
<msg>Hello, World&excl; </msg>
```

1.2 DTD validation:

We'll use Java based DTD validator to validate the *bookstore.xml* against the *books.dtd*

DTDValidator.java

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
public class DTDValidator
{
  public static void main(String[] args) {
    try {
                DocumentBuilderFactory f = DocumentBuilderFactory.newInstance();
                f.setValidating(true); // Default is false
                Document d = f.newDocumentBuilder().parse(arg[0]);
    }
        catch (Exception e) {       System.out.println(e.toString());          }
 }
```

1.3 Element Type Declaration:

Elements are primary building blocks in XML document. Element type declaration set the rules for type and number of elements that may appear in XML document, what order they may appear in.

Syn: *<!ELEMENT element-name type>*

Or

<!ELEMENT element-name (content)>

Here, *element-name* is name of element to be defined. The *content* could include specific rule, data or another element(s). The keyword ELEMENT must be in upper case, element names are case sensitive, all elements used in XML must be declared and same name cannot be used in multiple declarations.

In DTD, elements are classified depending upon their content as follows:

- **Standalone/Empty elements**: these elements cannot have any content and may have attributes. They can be declared using type keyword EMPTY as follows:
  Syn: *<!ELEMENT element-name EMPTY>* *Ex:* *<!ELEMENT br EMPTY>*
- **Unrestricted elements**: element with content can be created using content type as ANY. Keyword ANY indicates that *element-name* can be anything including text and other elements in any order and any number of times.
  Syn: *<!ELEMENT element-name ANY>* *Ex:* *<!ELEMENT msg ANY>*
- **Simple elements**: simple element cannot contain other elements, but contains only text.
  Syn: *<!ELEMENT element-name (#PCDATA)>* *Ex:* *<!ELEMENT author (#PCDATA)>*
  This interprets that element *element-name* can have only text content. The type of text id PCDATA means Parsed Character DATA and the text will be parsed by parser and will examined for entities and markups and expanded as and when necessary. Sometimes we can use CDATA means Character DATA in place of PCDATA.
- **Compound elements**: compound elements ca contains other elements known as child elements.
  Syn: *<!ELEMENT element-name (child-elements-names)>*
  Ex: <!ELEMENT book (title, author, price)>

Occurrence Indicator: sometimes it is necessary to specify how many times element may occur in document which is done by Occurrence Indicator. When no occurrence indicator is specified, child element must occur exactly once in XML document

Operator	Syntax	Description
None	a	Exactly one occurrence of a
* (Astrisck)	a*	Zero or more occurrences of a i.e. any number of times
+ (Plus)	a+	One or more occurrences of a i.e. at least once

? (Question mark)	a?	Zero or one occurrences of a i.e. at most once

Declaring multiple children: elements with multiple children are declared with names of the child elements inside parenthesis. The child elements must also be declared.

Operator	Syntax	Description
, (Sequence)	a , b	a followed by b
| (Choice)	a | b	a or b
() (Singleton)	(expression)	Expression is treated as a unit

1.4 Attribute Declaration:

Attributes are used to associate name, value pairs with elements. They are useful when we want to provide some additional information about elements content. The declaration starts with ATTLIST followed by name of the element the attributes are associated with and declaration of individual declarations:

Syn: <!ATTLIST element-name attribute-name attribute-type default-vale>

Ex: <!ATTLIST employee geneder CDATA 'male'/>

Here, ATTLIST must be in upper case. The *default-value* can be any of the following:

- **Default**: in this case, attribute is optional and developer may or may not provide this attribute. When attribute is declared with default value, the value of attribute is whatever value appears as attributes content in instance document.
  Ex: <!ATTLIST line width CDATA '100'/>
- **#REQUIRED**: attribute must be specified with value every time enclosing element is listed
  Ex: <!ATTLIST line width CDATA #RQUIRED />
- **#FIXED**: attribute is optional and is used to ensure that the attributes are set to particular values.
  Ex: <!ATTLIST line width CDATA #FIXED '50'/>
- **#IMPLIED**: similar to that of default attribute except that no default value is provided by XML
  Ex: <!ATTLIST line width CDATA '#IMPLIED' />

Attribute types:

The *attribute-type* can be one among string or CDATA, tokenized and enumerated types.

- **String type**: may take any literal string as value and can be declared using keyword CDATA. An attribute of CDATA type can contain any character if it conforms to well formedness constraints. Some it can contains escape characters like <, > etc.
- **Tokenized type**: following tokenized types are available
  - **ID**: it is globally unique identifier of attribute, this means value of ID attribute must not appear more than once throughout the XML document and resembles primary key concept of data base.
    <!ATTLIST question no ID #REQUIRED>
  - **IDREF**: similar to that of foreign key concept in databases and is used to establish connections between elements. IDREF value of the attribute must refer to ID value declared
    <!ATTLIST answer qno IDREF #REQUIRED>
  - **IDREFS**: it allows a list of ID values separated by white spaces
    <!ATTLIST answer qno IDREFS #REQUIRED>
  - **NMTOKEN**: it restricts attributes value to one that is valid XML name means allows punctuation marks and white spaces.
    <!ATTLIST car serial NMTOKEN #REQUIRED>
  - **NMTOKENS**: can contains same characters and white spaces as NMTOKEN. White space includes one or more characters, carriage returns, line feeds, tabs
    <!ATTLIST car serial NMTOKENS #REQUIRED>
  - **ENTITY**: refers to external non parsed entities
    <!ATTLIST car serial ENTITY #REQUIRED>
  - **ENTITIES**: values of ENTITIES attribute may contain multiple entity names separated by one or more white spaces
    <!ATTLIST car serial ENTITIES #REQUIRED>

- **Enumerated type**: enumerated attribute values are used when we want attribute value to be one of fixed set of values. There are two kinds of enumerated types:
  - **Enumeration**: attributes are defined by a list of acceptable values from which document author must choose a value. The values are explicitly specified in declaration, separated by pipe(|)
    <!ATTLIST employee gender (male|female) #REQUIRED>
  - **Notation**: it allows to use value that has been declared a NOTATION in DTD. Notation is used to specify format of non-XML data and common used is to describe MIME types like image/gif, image/jpeg etc.
    <!NOTATION jpg SYSTEM 'image/gif'>
    <!ENTITY logo SYSTEM 'logo.jpg' NDATA jpg>
    <!ATTLIST photo format NOTATION (jpg) #IMPLIED>

1.5 Entity Declaration:

Entities are variables that represent other values. If a text contains entities, the value of entity is substituted by its actual value whenever the text is parsed. Entity must be defined in DTD declaration to use custom entites in XML document. Built-in entities and character entities do not require any declaration. There are two types of entity declarations: General entity and Paramter entity. Each type can be again Parsed or Unparsed.

- **General and Parameter entities**: General entities are used with in the document content. Parameter entities are parsed entites used with in DTD. These two types of entites use different forms of references and are recognized in different contexts. They occupy different namespaces
- **Parsed and Unparsed entities**: Parsed entity is an entity whose content is parsed and checked for well formedness constraint during parsing procedure. Unparsed entity is resource whose contents may or may not be text and if text may not be XML. It means there are no constrainst on conetents of unparsed entities. Each unparsed entity has associated notation, identified by name.

1.5.1 General Entity Declaration:

There are three kinds of general entitity declarations:

- **Internal parsed**: an internal entity declaration has following form
  Syn: <!ENTITY entity-name "entity-value">
  Ex: <!ENTITY UKR "Uttam Kumar Roy">
  The entity UKR can be referred in XML document as follows:
  <author>&UKR;</author>
  This will be interpreted as : <author>Uttam Kumar Roy</author>
- **External parsed**: external entities allow an XML document to refer to external resource.Parse external entites refer to data that an XML parser has to parse and used for long replacement text which is kept in another file. There are two type of external parsed entities: Public and Private. Public external enties are identified by PUBLIC keyword and intended for general use. Private external entites are identified by SYSTEM keyword and are intended for use by single author or group of authors.
  Syn: <!ENTITY entity-name SYSTRM | PUBLIC "URI">
  Ex: <!ENTITY author SYSTEM "author.xml">
- **External unparsed**: refer to data that an XML processor does not have to parse. For example, there are numerious ASCII text files, JPRG photographs etc. None of theseare well formed XML. Mechanism that XML suggests for embedding these files is enternal unparsed entity. They can be either private or public.
  Syn: <!ENTITY logo SYSTEM "logo.jpg" NDATA jpeg>

1.5.2 Parameter Entity Declaration:

DTD supports another kind of entity called parameter entity. It is used within DTD which allows to assign collection of elements, attributes and attribute values to name and refer them using name instead of explicitly listimg them every time they are used.

- **Internal parsed entity**: it has following form
  Syn: <!ENTITY % entity-name entity-definition>
  Ex: <!ENTITY % name "firstname, middlename, lastname">
  This parameter entity describes portion of content model that can be referenced with in elements ind DTD. They can be referenced using entity name between precent sign(%) and semicolor(;).
  Syn: %Entity-name; Ex: %name;
- **External parsed entity**: These are used to link external DTDs. It may be private or public and is identified by keywords SYSTEM and PUBLIC. Private entites are intended for use by single author where as public entities can be used by anyone.
  Syn: <!ENTITY % entity-name SYSTEM | PUBLIC "URI">

2. XML Schema:

XML Schema Definition commonly known as XSD, is a way to describe precisely the XML language. XSD check the validity of structure and vocabulary of an XML document against the grammatical rules of the appropriate XML language.An XML document can be defined as:

- **Well-formed**: If the XML document adheres to all the general XML rules such as tags must be properly nested, opening and closing tags must be balanced and empty tags must end with '/>', then it is called as *well-formed*.
- **Valid**: An XML document said to be valid when it is not only *well-formed*, but it also conforms to available XSD that specifies which tags it uses, what attributes those tags can contain and which tags can occur inside other tags, among other properties.

Limitaions of Document Type Declaration (DTD)

- There is no bult-in data type in DTDs
- No new data types can be created in DTDs
- The use of cardinatlity in DTDs is limited
- Namespaces are not supported
- DTDs provide very limited support for modularity and reuse
- We can not put any restrictions on text content
- Defaults for elements can not be specified
- We have very little control over mixed content
- DTDs are written in strange format and are difficult to validate

Strengths of XML Schema(XSD)

- XML schema provided much grater specification than DTDs
- They support large number of built-in data types
- They support namespaces
- They are extensible to future additions
- They support uniqueness and referencial integrity constraints in much better way
- It is easier to define data restrictions

2.1 XSD Structure:

An XML XSD is kept in a separate document and then the document having extension *.xsd* and is be linked to the XML document to use it. Schema is the root element of XSD and it is always required.

Syn: *<?xml version="1.0"?>*
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
...
</xs:schema>

Above fragement specifies that elements and datatypes used in the schema are defined in "http://www.w3.org/2001/XMLSchema" namespace and these elements/data types should be prefixed with *xs*. Similarly, XSD can be linked to xml file with following syntax:

Syn: <roo-tag xmlns:xsi=" http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="uri">
Above fragement specifies default namespace declaration i.e. "http://www.w3.org/2001/XMLSchema-instance". This namespace is used by schema validator check that all the elements are part of this namespace. It is optional. Use schemaLocation attribute to specify the location of the *xsd* file.

Ex: book.xml

```
<?xml version="1.0" ?>
<bookstore xsi:schemaLocation="book.xsd" xmlns:xsi=" http://www.w3.org/2001/XMLSchema-instance"
>
<book>
        <title>WT</title>
        <author>Uttam Roy</author>
        <publisher>Oxford</publisher>
        <price>500</price>
</book>
<book>
        <title>AJ</title>
        <author>Schildt</author>
        <publisher>TMH</publisher>
        <price>200</price>
</book>
</bookstore>
```

book.xsd

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="bookstore">
<xs:complexType>
        <xs:sequence>
                <xs:element name="book">
                <xs:complexType>
                        <xs:sequence>
                        <xs:element name="title" type="xs:string"/>
                        <xs:element name="author" type="xs:string"/>
                        <xs:element name="publisher" type="xs:string"/>
                        <xs:element name="price" type="xs:integer"/>
                        </xs:sequence>
                </xs:complexType>
                </xs:element>
        </xs:sequence>
</xs:complexType>
</xs:element>
```

2.2 XSD Validation:

We'll use Java based XSD validator to validate the *bookstore.xml* against the *books.xsd.*

XSDValidator.java

```
import java.io.*;
import javax.xml.*;
import javax.xml.transform.dom.*;
import   javax.xml.parsers.*;
import javax.xml.validation.*;
import org.w3c.dom.*;
```

```
public class XSDValidator {
  public static void main(String[] args) {
    try {
      SchemaFactory factory =
        SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
          Schema schema = factory.newSchema(new File(args[1]));
           Validator validator = schema.newValidator();
          DocumentBuilder parser=DocumentBuilderFactory.newInstance().newDocumentBuilder();
          Document document=parser.parse(new File(args[0]));
           validator.validate(new DOMSource(document));
    }
    catch (Exception e)
   {          System.out.println("Exception: "+e.getMessage());        }
}
```

2.3 Element declaration:

Elements are primary building blocks in XML document. Element type declaration can be done using <xs:element> tag with following syntax.

Syn: <xs:element name=”element-name” type=”element-type”>

Ex: <xs:element name=”title” type=”xs:string”>

Each element declaration within the XSD has mandatory attribute *name*. the value of this name attribute is the element name attribute is the element name that will appear in the XML document. Element definition may also have optional *type* attribute, which provides description of what can be containted within the element when it appears in XML document. Every XML document must have root element. This root element must be declared first in schema for conforming XML documents.

```
Ex: <!xml version=”1.0”?>
        <xsd:schema xmlns:xsd=”http://www.w3.org/2001/XMLSchema”>
              <xs:element name=”bookstore”>
              </xs:element>
        </xsd:schema>
```

2.3.1 Declarting simple elements:

Simple type elements can contain only text and/or data. They can not have child elements or attributes, and can not be empty. Simple elements are defined as follows:

Syn: <xs:element name=”element-name” type=”element-type”>

Ex: <xs:element name=”title” type=”xs:string”/>

The value of *type* attributes specifies an elements content type and can be any simple type. This attribute can be any complex type.

- **Default Value:** Simple Element can have default value that specifies the default content to be used when not content is specified. When an element is declared with default value, the value of the element is whatever value appears as elements content in instance document. Following example illustrates this:

  <xs:element name="gender" type="xs:boolean" default="true" />

- **Fixed Value:** Simple Element can also have optional fixed value. Fixed attribute is used to ensure that elements content is always set to particular value. Consider the following syntax:

  <xs:element name="branch" type="xs:string" fixed="IT" />

- **Occurance indicators:** an element have two optional attributes : minOccurs and maxOccurs. They are used to specify the number of times an element can occur in XML document.
  - **minOccurs**: this attribute specifies minimum number of times an element can occur. The following is example of usage of this attribute:

    <xs:element name=”option” type=”xs:string” minOccurs=”0”/>

- **maxOccurs**: this attribute specifies maximum number of times an element can occur. The declaration of element will be as follows:
  <xs:element name="option" type="xs:string" maxOccurs="10"/>

Schema	DTD	Meaning
minOccurs='0', maxOccurs='unbounded'	*	Zero or more
minOccurs='1', maxOccurs='unbounded'	+	One or more
Minoccurs='0'	?	Optional
None	None	Exactly one

2.3.2 Declarting complex elements:

Complex types can be named or can be anonymous. They are associated with complex elements in the same manner, typically using a type definition and an element declaration. By default, complex type elements have complex content i.e. they have child elements. Complex type elements can be limited to having simple content i.e. they contain only text. General form of element declaration is:

Syn: <xs:complexType name="complex-type-name"><xs:sequence>
</xs:sequence></xs:complexType>

Ex: <xs:complexType name="sName"><xs:sequence>
<xs:element name="first" type="xs:string"/>
<xs:element name="middle" type="xs:string"/>
<xs:element name="lase" type="xs:string"/>
</xs:sequence></xs:complexType>

2.3 Atribute declaration:

Attrbibutes are used to describe properties of an element. Attributes themselves are always declared as simple types as follows:

Syn: <xs:attribute name"attribute-name" type="attribute-type">
Ex: <xs:attribute name="id" type="xs:string'/>

Simple types can not have attributs. Element that have attributes are complex types. So, attributes declaration always occurs as part of complex type declaration, immediately after its content model.

```
Ex: <xs:complexType name="sName"><xs:sequence>
        <xs:element name="first" type="xs:string"/>
    <xs:element name="middle" type="xs:string"/>
    <xs:element name="lase" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:string"/>
</xs:complexType>
```

A part from this simple definition, there can be additional specifications for attributes:

- **Attribute element properties:**
  - **use**: possible values are optional, required and prohibited.
    <xs:attribute name="id" type="xs:string" use="required"/>
  - **default**: this specifies the value to be used if attribute is not specified
    <xs:attribute name="gender" type="xs:boolean" default="false"/>
  - **fixed**: it specifies that attribute, if it appears must always have fixed value specified. If the attribute does not appear, the schema processor will provide attribute with value specified here.
    <xs:attribute name="unit" type="xs:boolean" default="rpm"/>
- **Order Indicators**
  - **All**: Child elements can occur in any order.

```
<xs:all>
        <xs:element name="first" type="xs:string"/>
        <xs:element name="middle" type="xs:string"/>
        <xs:element name="last" type="xs:string"/>
```

```
</xs:all>
```

- **Choice**: Only one of the child element can occur.

```
<xs:choice>
        <xs:element name="first" type="xs:string"/>
        <xs:element name="middle" type="xs:string"/>
        <xs:element name="last" type="xs:string"/>
</xs:choice>
```

- **Sequence**: Child element can occur only in specified order.

```
<xs:sequence>
        <xs:element name="first" type="xs:string"/>
        <xs:element name="middle" type="xs:string"/>
        <xs:element name="last" type="xs:string"/>
</xs:sequence>
```

- **Occurence Indicators**
  - **maxOccurs** - Child element can occur only maxOccurs number of times.
  - **minOccurs** - Child element must occur minOccurs number of times.
- **Group Indicators**
  - **Group**: a set of related elements can be created using this indicator. the general form for creating an element group is as follows:

```
Syn:  <xs:group name="group-name">  ...  </ xs:group>
Ex:   <xs:group name="personInfo">
            <xs:element name="first" type="xs:string"/>
            <xs:element name="middle" type="xs:string"/>
            <xs:element name="last" type="xs:string"/>
      </ xs:group>
```

  - **attributeGroup**: XML Schema provides this element, which is used to group a set of attributes declarations so that they can be incorporated into complex types definitions with syntax:

```
Syn:  <xs:attributeGroup name="group-name">  ...  </ xs:attributeGroup>
Ex:   <xs: attributeGroup name="personInfo">
            <xs:element name="first" type="xs:string"/>
            <xs:element name="middle" type="xs:string"/>
            <xs:element name="last" type="xs:string"/>
      </ xs: attributeGroup>
```

2.4 Annotations declaration:

XML schema provides three annotation elements for documentation purposes in XML schema instance. They provide a way to write realistic ans structured comments for the benefit of applications. An annotation is represented by <annotation> element which typically appears at the beginning of most schemas. However, it can appear inside any complex element definition. It can contain only two elements <appinfo> and <documentation> any number of times. Following is an example:

```
<xs:annotation>
    <xs:documentation> <author>Uttam Roy</author></xs:documentation>
    <xs:appinfo><version>2.1</version></xs:appinfo>
</xs.annotation>
```

2.5 XML Scheme data types:

An element is limited by its type. Depending upon content model, elements are categorized as Simple or Complex type. A simple type can further be divided into three types: Atomic, List and Union. XML schema 1.0 specification provides about 46 built in data types. All built-in data types except anyType are considered as simple types. Some of the built-in data types are as follows:

2.5.1 XSD String Data Types:

String data types are used to represent characters in the XML documents.

- **<xs:string>:** The <xs:string> data type can take characters, line feeds, carriage returns, and tab characters. XML processor do not replace line feeds, carriage returns, and tab characters in the content with space and keep them intact. For example, multiple spaces or tabs are preserved during display.
  Syn: <xs:element name="elment-name" type="xs:string"/>
  Ex: < xs:element name="sname" type="xs:string"/>
- **<xs:token>:** The <xs:token> data type is derived from <string> data type and can take characters, line feeds, carriage returns, and tab characters. XML processor removes line feeds, carriage returns, and tab characters in the content and keep them intact. For example, multiple spaces or tabs are removed during display.
  Syn: <xs:element name="element-name" type="xs:token"/>

Following is the list of commonly used data types which are derived from <string> data type.

- **ID:** Represents the ID attribute in XML and is used in schema attributes.
- **IDREF:** Represents the IDREF attribute in XML and is used in schema attributes.
- **Language:**Represents a valid language id
- **Name:** Represents a valid XML name
- **NMTOKEN:** Represents a NMTOKEN attribute in XML and is used in schema attributes.
- **normalizedString:** Represents a string that does not contain line feeds, carriage returns, or tabs.
- **String:** Represents a string that can contain line feeds, carriage returns, or tabs.
- **Token:** Represents a string that does not contain line feeds, carriage returns, tabs, leading or trailing spaces, or multiple spaces

2.5.2 XSD Date & Time Data Types:

Date and Time data types are used to represent date and time in the XML documents.

- **<xs:date> data type:** The <xs:date> data type is used to represent date in YYYY-MM-DD format. Each component is required. **YYYY**- represents year, **MM**- represents month, **DD**- represents day
  Syn: <xs:element name="birthdate" type="xs:date"/>
  Ex: <birthdate>1980-03-23</birthdate>
- **<xs:time> data type**: The <xs:time> data type is used to represent time in hh:mm:ss format. Each component is required. **hh**- represents hours, **mm**- represents minutes, **ss**- represents seconds
  Syn: <xs:element name="startTime" type="xs:time"/>
  Ex: <startTime>10:20:15</startTime>
- **<xs:datetime> data type:** The <xs:datetime> data type is used to represent date and time in YYYY-MM-DDThh:mm:ss format. Each component is required. **YYYY**- represents year, **MM**- represents month, **DD**- represents day, **T**- represents start of time section, **hh**- represents hours, **mm**- represents minutes, **ss**- represents seconds
  Syn: <xs:element name="startTime" type="xs:datetime"/>
  Ex: <startTime>1980-03-23T10:20:15</startTime>
- **<xs:duration> data type:** The <xs:duration> data type is used to represent time interval in PnYnMnDTnHnMnS format. Each component is optional except P. **P**- represents year, **nY**- represents month, **nM**- represents day, **nD**- represents day, **T**- represents start of time section, **nH**- represents hours, **nM**- represents minutes, **nS**- represents seconds
  Syn: <xs:element name="period" type="xs:duration"/>
  Element usage in xml to represent period of 6 years, 3 months, 10 days and 15 hours.
  Ex: <period>P6Y3M10DT15H</period>

Following is the list of commonly used date data types .

- **Date:** Represents a date value
- **dateTime:** Represents a date and time value
- **duration:** Represents a time interval
- **gDay:** Represents a part of a date as the day (DD)

- **gMonth:** Represents a part of a date as the month (MM)
- **gMonthDay:** Represents a part of a date as the month and day (MM-DD)
- **gYear:** Represents a part of a date as the year (YYYY)
- **gYearMonth:** Represents a part of a date as the year and month (YYYY-MM)
- **time:** Represents a time value

2.5.3 XSD Numeric Data Types:

Numeric data types are used to represent numbers in the XML documents.

- **<xs:decimal> data type:** The <xs:decimal> data type is used to represent numeric values. It support decimal numbers upto 18 digits.
  Syn: *<xs:element name="score" type="xs:decimal"/>*
  Ex: *<score>9.12</score>*
- **<xs:integer> data type:** The <xs:integer> data type is used to represent integer values.
  Syn: *<xs:element name="score" type="xs:integer"/>*
  Ex: *<score>9</score>*

Following is the list of commonly used numeric data types .

- **Byte:** A signed 8 bit integer
- **Decimal:** A decimal value
- **Int:** A signed 32 bit integer
- **Integer:** An integer value
- **Long:** A signed 64 bit integer
- **negativeInteger:** An integer having only negative values (..,-2,-1)
- **nonNegativeInteger:** An integer having only non-negative values (0,1,2,..)
- **nonPositiveInteger:** An integer having only non-positive values (..,-2,-1,0)
- **positiveInteger:** An integer having only positive values (1,2,..)
- **short:** A signed 16 bit integer
- **unsignedLong:** An unsigned 64 bit integer
- **unsignedInt:** An unsigned 32 bit integer
- **unsignedShort:** An unsigned 16 bit integer
- **unsignedByte:** An unsigned 8 bit integer

2.5.4 XSD Miscalleneous Data Types

Other Important Miscellaneous data types used are boolean, binary and anyURI.

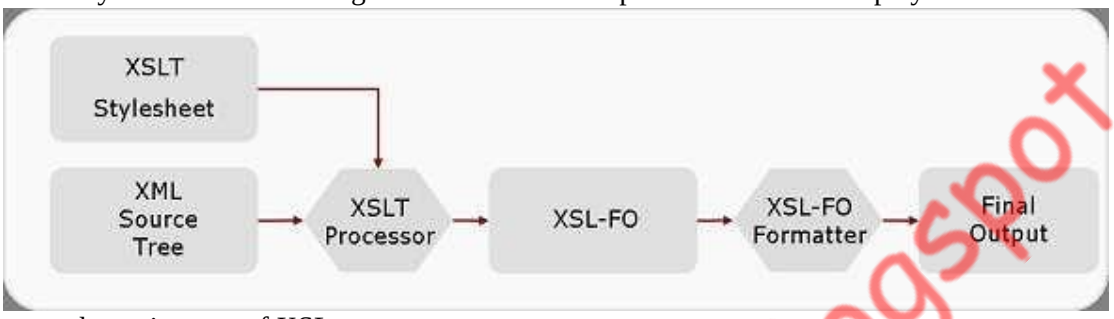
- **<xs:boolean> data type:** The <xs:boolean> data type is used to represent true, false, 1 (for true) or 0 (for false) value.
  Syn: *<xs:element name="pass" type="xs:boolean"/>*
  Ex: *<pass>false</pass>*
- **Binary data types:** The Binary data types are used to represent binary values. Two binary types are common in use. **base64Binary**- represents base64 encoded binary data, **hexBinary** - represents hexadecimal encoded binary data
  Syn: *<xs:element name="blob" type="xs:hexBinary"/>*
  Ex: *<blob>9FEEF</blob>*
- **<xs:anyURI> data type**: The <xs:anyURI> data type is used to represent URI.
  Syn: *<xs:attribute name="resource" type="xs:anyURI"/>*
  Ex: *<image resource="http://www.tutorialspoint.com/images/smiley.jpg" />*

eXtensible Stylesheet Language Transformation(XSLT):

XML documents contain self-describing and structured data. The set of tags and their structure varies widely in different applications. Web browsers can not display such non-HTML files as they have no prior knowledge about the meaning of set of tags used in different XML documents. Users may also want to

generate new XML documents from one or more existing XML documents for processing or sharing of data between different applications. One possible solution is to generate separate XML document such that the former contains only insensitive data. XSLT comes into play in this scenario.

XSLT, Extensible Stylesheet Language Transformations provides the ability to transform XML data from one format to another automatically. An XSLT stylesheet is used to define the transformation rules to be applied on the target XML document. XSLT stylesheet is written in XML format. XSLT Processor takes the XSLT stylesheet and apply the transformation rules on the target XML document and then it generates a formatted document in form of XML, HTML or text format. This formatted document then is utilized by XSLT formatter to generate the actual output which is to be displayed to the end user.



Following are the main parts of XSL.

- **XSLT** - used to transform XML document into various other types of document.
- **XPath** - used to navigate XML document.
- **XSL-FO** - used to format XML document.

In general following tasks can be performed using XSLT: Constant text generation, Reformatting of information, sensitive information suppression, adding new information, copying information and Sorting document with respect to a criteria.

Advantages

- Independent of programming. Transformations are written in a seperate xsl file which is again an XML document.
- Output can be altered by simply modifing the transformations in xsl file. No need to change in any code. So Web designers can edit stylesheet and can see the change in output quickly.

1. Stylesheet strcture:

XSLT files are themselves XML documents and hence must follow the well-formedness constraints. The W3C defined the exact syntax of an XSLT 2.0 document by XML schema. XSLT file starts with XML declaration. Every XSLT file must have either <stylesheet> or <transform> as root element. Following are simple structure of XSLT document:

```
<?xml version="1.0"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSLT/Transform">
...
</ xsl:stylesheet>
```

Or

```
<?xml version="1.0"?>
<xsl:transform version="2.0" xmlns:xsl="http://www.w3.org/1999/XSLT/Transform">
...
</ xsl: transform >
```

These elements must have the attribute *version* and namespace attribute *xmlns*. Version attribute indicate version of XSLT being used. Namespace attribute distinguishes XSLT elements from other elements. There are different ways to apply XSLT document to XML document. One way to add link to XML document which points to actual XSLT files and lets the browsers do transformation with following declaration:

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

<?xml-stylesheet type="text/xsl" href="URI">
<root> ... </root>

students.xml	*students.xsl*
<?xml version="1.0"?> *<?xml-stylesheet type="text/xsl" href="students.xsl">* *<class>* *...* *</class>*	*<?xml version="1.0"?>* *<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSLT/Transform">* *...* *</ xsl:stylesheet>*

2. XSLT Elements:

An XSLT file contains elements, which instruct processor how an XML document is to be transformed. It may contain elements that are not defined by XSLT. In such cases, XSLT processor does not process these non-XSLT elements and add them to the ouput in the same order they occurred in the source XSLT document. This means that the transformed XML document may use original mark-ups as well as new mark-ups.

Element	Description
stylesheet	Defines the root element of a style sheet
transform	Defines the root element of a style sheet
template	Rules to apply when a specified node is matched
apply-templates	Applies a template rule to the current element or to the current element's child nodes
call-template	Calls a named template
element	Creates an element node in the output document
variable	Declares a local or global variable
param	Declares a local or global parameter
value-of	Extracts the value of a selected node
attribute	Adds an attribute
attribute-set	Defines a named set of attributes
if	Contains a template that will be applied only if a specified condition is true
choose	Used in conjunction with <when> and <otherwise> to express multiple conditional tests
when	Specifies an action for the <choose> element
for-each	Loops through each node in a specified node set
import	Imports the contents of one style sheet into another. **Note:** An imported style sheet has lower precedence than the importing style sheet
include	Includes the contents of one style sheet into another. **Note:** An included style sheet has the same precedence as the including style sheet
sort	Sorts the output
processing-instruction	Writes a processing instruction to the output
comment	Creates a comment node in the result tree
copy	Creates a copy of the current node (without child nodes and attributes)
copy-of	Creates a copy of the current node (with child nodes and attributes)

3. XSLT templates:

An XSLT document is all about template rules. A template specifies rule and instruction, which is executed when rule matches. The rule is specified by XSLT <template> element. It has attribute *match*, which specifies pattern. The value of match attribute is subset of expression.

Syn: *<xsl:template match="expression">*
...
</xsl:template>

Ex: *<xsl:template match="/">*
<h1>Hello! World.</h1>
</xsl:template>

XSLT document contain single template rule. It has match attribute with expressin "/", which means the document root of any XML document. Ths instruction with in this template specifies the string *Hello! World* has to be added to the output and the resulting document obtained is as follows:

<html> <body><h1>Hello! World.</h1></body> </html>

Applying templates:

In general, if a node matches with template pattern, the templates action part is processed. It is also possible to instruct XSLT processor to process other template rules if any. This is done using <apply-templates> element with following syntax:

```
<xsl:template match="/">
  <xsl:apply-templates/>
</xsl:template>
```

This example states that whenever document root is encountered, XSLT processor has to process all templates that match with document roots children roots. The XSLT engine in turn, compares each child element of document root aginst templates in style sheet and if match is found, it processes the corresponding template.

Processing Sequence and default templates:

When XSLT processor is supplied XML document for transformation using XSLT document, it first creates document tree. Processing always starts from document root of this tree. So, XSLT processor looks for template for it. If no template is found for document root, XSLT processor provides default templates. This default template for document root looks like this:

```
<xsl:template match="/">
  <xsl:apply-templates/>
</xsl:template>
```

The behaviour of default template for any element node looks as follows:

```
<xsl:template match="*">
        <xsl:apply-templates/>
</xsl:template>
```

Default template for text nodes as follows:

```
<xsl:template match="text()">
        <xsl:apply-templates/>
</xsl:template>
```

Default templates and their behavior:

- **Root**: process template for its children
- **Element**: process templates for its children
- **Attribute**: output attribute name and value
- **Text**: output text value
- **Processing instruction**: do nothing
- **Comment**: do nothing

Named templates:

XSLT named templates resemble the functions in any procedural programming language. The <template> element has *name* attribute, which can be used to give name to template. Once template is created this way, it can be called by using <call-template> element and specifying its name.

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

```
<xsl:template name="header">
        <title>XSLT</title>
</xsl:template>
```

4. Selecting values:

The value of a node can also be added using <value-of> element. Value of node depends on type of the node. For example, the value of text node is the text itself, whereas the value of element node is concatenation of values of all text descendents. If multiple nodes are selected by select attribute, value is concatenation of values of those selected attributes. Consider simple XML document:

```
<book>
        <title>Web Technologies</title>
</book>
```

One can now extract the value of title element using <value-of> elemement as follows:

```
<xsl:template match="/">
  Title:    <xsl:value-of select="book/title"/>
</xsl:template>
```

This XSLT file, on applying previous XML document produces following result:

Title: Web Technologies

Values of different node types:

- **Text**: text of node
- **Element**: concatenation of values of all text descendants
- **Attribute**: attribute value without quotation marks
- **Namespace**: the URI of the namespace
- **Comment:** anything between <!--and -->
- **Processing instruction**: anything between <? and ?>

XSLT has another element <copy-of>, which returnas all selected elements including nested elements and text. Consider the following XSLT document.

```
<xsl:template match="/">
  <xsl:copy-of select="."/>
</xsl:template>
```

When we apply this XSLT document to any XML document, it produces the same XML document. This is because, when root element (/) is selected, <copy-of> copies root element together with all child elements recursively.

5. Varaibale and Parameters:

Named templates resembles the functions in any procedural programming language. Like function, named templates may accept argument. Formal parameters are declard with in template using <param> element as follows:

```
<xsl:template name="add">
        <xsl:param name="a"/>
        <xsl:param name="b"/>
        <xsl:value-of select="$a+$b"/>
</xsl:template>
```

This example defines named template *add*, which takes two parameters *a* and *b*. The purspose of this template is to add two arguments taken and produce the result to output. Arguments can then be passed to template using <with-param> element during template call.

```
<xsl:call-template name="add">
        <xsl:with-param name="a" select="2"/>
        <xsl:with-param name="b" select="4"/>
</xsl:call-template>
```

This code calls template add with parameters 2 and 4. If this XSLT applied to XML document the output will be 6. The scope of formal is with in the template only. XSLT allows to declare anduse variable. Consider the following code:

```
<xsl:template>
        <xsl:variable name="a">4</xsl:variable>
        <xsl:variable name="b"> 6</xsl:variable>
        <xsl:value-of select="$a+$b"/>
</xsl:template>
```

6. Conditional Processing:

There are two types of branching constructs in XSLT: <if> and <choose>

6.1 Using if:

XSLT <if> element has attribute *test*, which takes Boolean expression. If the effective Boolean value of this expression is evaluated to true, the action under <if> construct is followed. The general syntax of <if> construct is as follows:

```
<xsl:if test="condition">
  ...
</xsl:if>
```

The following extracts information about only that book having title as "Web Technologies":

```
<xsl:template match="//book">
  <xsl:if test="@title='Web Technologies'">
        Author: <xsl:value-of select="@author"/>
        Price:  <xsl:value-of select="@price"/>
  </xsl:if>
</xsl:template>
```

6.2 Using choose:

XSLT <choose> element allows us to select particular condition among set of conditions specified by <when> element. The general format of <choose> construct is:

```
<xsl:choose>
  <xsl:when  text="expression1">..</xsl:when>
  <xsl:when  text="expression2">..</xsl:when>
  ...
  <xsl:when  text="expressionN">..</xsl:when>
  <xsl:otherwise>...</xsl:otherwise>
</xsl:choose>
```

Consider the following XML file *result.xml*, containing marks of different students:

```
<result>
        <student><rollno>01</rollno><marks>80</marks></student>
        <student><rollno>02</rollno><marks>70</marks></student>
        <student><rollno>03</rollno><marks>60</marks></student>
        <student><rollno>04</rollno><marks>55</marks></student>
        <student><rollno>05</rollno><marks>77</marks></student>
</result>
```

The following XSLT document displays results of the students:

```
<xsl:choose>
        <xsl:when test="marks &gt; 80 and marks &lt;= 100">A Grade</xsl:when>
        <xsl:when test="marks &gt;  70 and marks  &lt;=  80">B Grade</xsl:when>
        <xsl:when test="marks &gt; 60 and marks  &lt;=  70">C  Grade</xsl:when>
        <xsl:when test="marks &lt;=60">D Grade</xsl:when>
```

```
</xsl:choose>
```

6. Repetition:

XSLT allows <for-each> construct, which can be used to process set of instructions repetedly for different items in sequence. The attribute *select* evaluates sequence of nodes. For each of telements in this sequence, instruction under <for-each> element are processed. Consider the following XML file *result.xml*, containing marks of different students:

```
<result>
        <student><rollno>01</rollno><marks>80</marks></student>
        <student><rollno>02</rollno><marks>70</marks></student>
        <student><rollno>03</rollno><marks>60</marks></student>
        <student><rollno>04</rollno><marks>55</marks></student>
        <student><rollno>05</rollno><marks>77</marks></student>
</result>
```

The following XSLT document displays results of the students:

```
<xsl:for-each select="result">
        Roll No: <xsl:value-of select ="rollno"/><br>
        Marks: <xsl:value-of select ="marks"/><br>
</xsl:for-each>
```

7. Creating nodes and Sequences:

XSLT allows to directly create custom nodes such as element node, text nodes etc. or sequences of nodes and atomic values that appear in output.

7.1 Creating element nodes:

An element node is created using <element> tag. The content of created element is whatever is generated between the starting and closing of <element> tag. If an element has attributes, they are declared using <attribute> tag described in the next section.

```
<xsl:element name="msg">
  Hello world!
</xsl:element>
```

7.2 Create attribute node:

An attributes of an element is created using enclosed <attribute> tag. The mandatory attribute *name* specifies name of the generated attributes. The value is indicated by contnent of <attribute> element.

```
<xsl:element name="msg">
        <xsl:attribute name="lang">en</xsl:attribute>
        Hello world!
</xsl:element>
```

This code segment creates the element *msg* with attribute *lang* as follows:

```
<msg lang="en">Hellow World!</msg>
```

7.3 Create text nodes:

Generally, XSLT processor outputs text that appears in the stylesheet. However, extra white spaces are not provided in such case. Secondaly, special characters such as < and & are represented in text by escape character sequence < and & respectively. For this reason, it provides <text> element to add literal text to result with following syntax:

```
<xsl:text> Hello World &amp;</xsl:text>
```

7.4 Creating document node:

XSLT allows to create new document node using <document> element. For example, following code create temporary document node, which is stored in varaibel named "tempTree".

```
<xsl:variable name="tempTree" as ="document-node()">
  <xsl:document> <xsl:apply-templates/> </xsl:document>
</xsl:variable>
```

7.5 Creating processing instructions:

Processing instruction is added in the result using <processing-instruction> element. The most popular use of this element is to insert the <stylesheet> element in output HTML/XML document with syntax.

```
<xsl:processing-instruction name="xml-stylesheet">
  <xsl:text> href="sort.xsl" type="text/xsl"</xsl:text>
</xsl:processing-instruction>
```

7.5 Creating comments:

Comment is added using <comment> element as follows:

```
<xsl:comment>This is XSLT document</xsl:document>
```

8. Grouping nodes:

XSLT allows us to group related items based on common values. Consider the following XML document.

```
<result>
        <student><rollno>01</rollno><marks>80</marks><dept>IT</dept></student>
        <student><rollno>02</rollno><marks>70</marks><dept>IT</dept></student>
        <student><rollno>03</rollno><marks>60</marks><dept>CSE</dept></student>
        <student><rollno>04</rollno><marks>55</marks><dept>IT</dept></student>
        <student><rollno>05</rollno><marks>77</marks><dept>CSE</dept></student>
</result>
```

The following XSLT document displays results of the students as groups by *dept*:

```
<xsl:template match="/result">
        <xsl:for-each-group select="student" group-by="@dept">
        <xsl:value-of select="current-grouping-key()" />
        <xsl:for-each select="current-group()">
                <xsl:value-of select="@rollno"/>
        </xsl:for-each>
</xsl:template>
```

This enumerates group items based either on common value of grouping key or pattern specified by group-by attribute. The current-group() function returns the current group item in the iteration and current-grouping-key() returns commn key of current group.

9. Sorting nodes:

We can sort group of similar elements using <sort> element. The attributes of the <sort> element describe how to perform sorting. For example, sorting can be doen alphabetically or numerically or in increasing or decreating order. The attribute select is used to specify sorting key. The order attribute specifies order and can have values accending or decending. The type of data to be sorted can be specified using attribute data-type. Following example sorts list of student respect to their marks.

```
<table><xsl:for-each select="/result">
  <xsl:sort select="marks" data-type="number"/>
        <tr><td><xsl:value-of select="rollno"/></td>
        <td><xsl:value-of select="marks"/></td>
        <td><xsl:value-of select="dept"/></td></tr>
</xsl:for-each></table>
```

10. Functions:

XSLT also allows custom functions to be defined in stylesheet. A function is defined using <function> element. It has attribute name, which specifies the name of the function. Once function is defined, it can be called from any expressin. The function name must have prefix. This is required to avoid conflict with any function from default namespace. A prefix can not be bound to reserved namespace.

```
<xsl:function name="f:fact">
        <xsl:param name="n">
```

```
        <xsl:value-of select="if ($n le 1) then 1 else $n*f:fact($n-1)"/>
    </xsl:function>
    <xsl:template match="/">
        <xsl:value-of select="f:fact(3)"/>
    < xsl:template>
```

11. Copying nodes:

The <copy> element copies the current node to the output. If the node is an element node, its namespace nodes are copied automatically, but attributes and children of element nodes are not copied automatically. Consider the simple XML document:

```
<result>
        <student><rollno>01</rollno><marks>80</marks><dept>IT</dept></student>
        <student><rollno>02</rollno><marks>70</marks><dept>IT</dept></student>
        <student><rollno>03</rollno><marks>60</marks><dept>CSE</dept></student>
        <student><rollno>04</rollno><marks>55</marks><dept>IT</dept></student>
        <student><rollno>05</rollno><marks>77</marks><dept>CSE</dept></student>
</result>
```

Now consider following XSLT document:

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

12. Numbering:

The <number> element allows to insert and format number into the result tree.

```
<xsl:template match="/result">
        <xsl:for-each-group select="student" group-by="@dept">
        <xsl:number value="position()"/>
        <xsl:value-of select="current-grouping-key()" />
        <xsl:for-each select="current-group()">
                <xsl:number value="position()"/>
                <xsl:value-of select="@rollno"/>
        </xsl:for-each>
</xsl:template>
```

Document Object Model (DOM):

The Document Object Model(DOM) is an application programming interface(API) for HTML and XML documents. It defines the logical structure of documents and the way a document is accesses and manipulated. DOM is a set of platform independent and language neutral application programming interface(API) which describes how to access and manipulate the information stored in XML or in HTML documents. Main objectives of DOM are Accessing the elements of document, deleting the elements of documents and changing the elements of document.

DOM models document as hierarchical structure consisting of different kinds of nodes. Each of these nodes represents specific portion of the document. Some kind of nodes may have children of different types. Some nodes cannot have anything below it in the hierarchical structure and are leaf nodes. With the Document Object Model, programmers can build documents, navigate their structure, and add, modify, or delete elements and content. Anything found in an HTML or XML document can be accessed, changed, deleted, or added using the document object model. The DOM is separated into 3 different parts/levels:

1. Core DOM: standard model for any structured document.
2. HTML DOM:standard model for HTML documents.
3. XML DOM: standard model for XML documents.

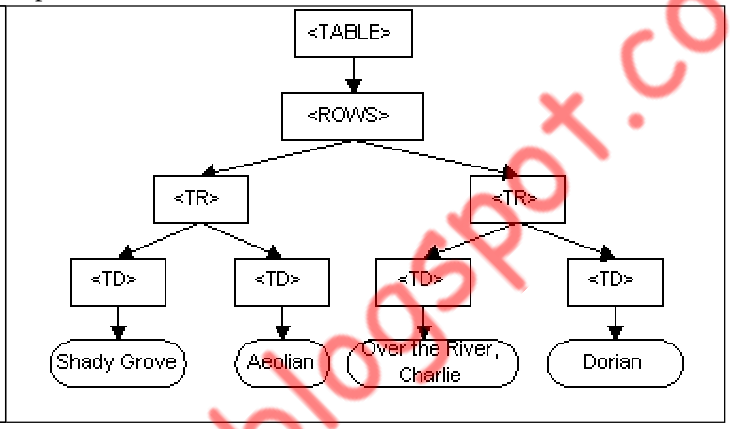
1. Core DOM:

This portion defines the basic set of interfaces and objtects for any structured document.

2. HTML DOM:

The HTML Document Object Model (DOM) is a programming API for HTML documents. It defines the logical structure of documents and the way a document is accessed and manipulated. With the Document Object Model, programmers can create and build documents, navigate their structure, and add, modify, or delete elements and content. Anything found in an HTML document can be accessed, changed, deleted, or added using the Document Object Model, with a few exceptions - in particular, the DOM interfaces for the internal subset and external subset have not yet been specified.

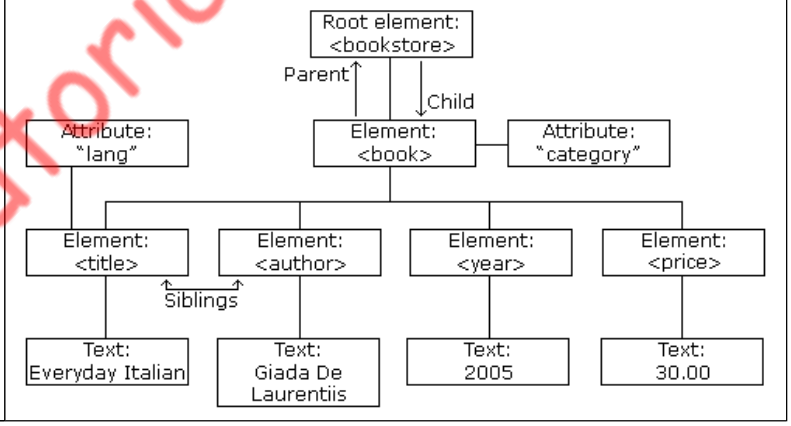
```
<TABLE>
 <ROWS>
 <TR>
    <TD>Shady Grove</TD>
    <TD>Aeolian</TD>
 </TR>
 <TR>
    <TD>Over the River, Charlie</TD>
    <TD>Dorian</TD>
 </TR>
 </ROWS>
 </TABLE>
```



3. XML DOM:

According to the DOM, everything in an XML document is a Node. The DOM says: The entire document is a document node, Every XML element is an element node, The text in the XML elements are text nodes, Every attribute is an attribute node, Comments are comment nodes.

```
<bookstore>
<book category=”cooking”>
<title lang=”en”>Everday Italian</title>
<author>Giada De Laurentiis</author
<year>2005</year>
<price>30.00</price>
</book>
</bookstore>
```



The root node in the XML above is named <bookstore>. All other nodes in the document are contained within <bookstore>. The root node <bookstore> holds four <book> nodes. The first <book> node holds four nodes: <title>, <author>, <year>, and <price>, which contains one text node each, "Everyday Italian", "Giada De Laurentiis", "2005", and "30.00". The XML DOM views an XML document as a tree-structure. The tree structure is called a node-tree. All nodes can be accessed through the tree. Their contents can be modified or deleted, and new elements can be created.

The node tree shows the set of nodes, and the connections between them. The tree starts at the root node and branches out to the text nodes at the lowest level of the tree. The nodes in the node tree have a hierarchical relationship to each other. The terms parent, child, and sibling are used to describe the relationships. Parent nodes have children. Children on the same level are called siblings (brothers or sisters). In a node tree, the top node is called the root, Every node except the root has exactly one parent node, A

node can have any number of children, A leaf is a node with no children, Siblings are nodes with the same parent.

Using XML processors:

Parsing XML refers to going through XML document to access data or to modify data in one or other way. XML Parser provides way how to access or modify data present in an XML document. Java provides multiple options to parse XML document. Following are various types of parsers which are commonly used to parse XML documents.

- **Dom Parser**: Parses the document by loading the complete contents of the document and creating its complete hiearchical tree in memory.
- **SAX Parser**: Parses the document on event based triggers. Does not load the complete document into the memory.

Difference between DOM and SAX:

DOM	SAX
DOM is a tree based parsing method	SAX is an event based parsing method
We can insert or delete a node	We can insert or delete a node
Traverse in any direction	Top to bottom traversing
Stores the entire XML document in to memory before processing	Parses node by node
Occupies more memory	Doesn't store the XML in memory
DOM preserves comments	SAX doesn't preserve comments.
import javax.xml.parsers.*; import org.w3c.dom.*;	import javax.xml.sax.*; import org.xml.sax.helpers.*;

1. Java DOM Parser:

The Document Object Model is an official recommendation of the World Wide Web Consortium (W3C). It defines an interface that enables programs to access and update the style, structure,and contents of XML documents. XML parsers that support the DOM implement that interface. In order to use this, we need to know a lot about the structure of a document, need to move parts of the document around and need to use the information in the document more than once. When we parse an XML document with a DOM parser, we get back a tree structure that contains all of the elements of your document. The DOM provides a variety of functions you can use to examine the contents and structure of the document.

DOM interfaces:

The DOM defines several Java interfaces. Here are the most common interfaces:

- **Node**: The base datatype of the DOM.
- **Element**: The vast majority of the objects you'll deal with are Elements.
- **Attr:** Represents an attribute of an element.
- **Text:** The actual content of an Element or Attr.
- **Document:** Represents entire XML document, a Document object is often referred to as a DOM tree.

Common DOM methods:

When you are working with the DOM, there are several methods you'll use often:

- **Document.getDocumentElement()** - Returns the root element of the document.
- **Node.getFirstChild()** - Returns the first child of a given Node.
- **Node.getLastChild()** - Returns the last child of a given Node.
- **Node.getNextSibling()** - These methods return the next sibling of a given Node.
- **Node.getPreviousSibling()** - These methods return the previous sibling of a given Node.
- **Node.getAttribute(attrName)** - For a given Node, returns the attribute with the requested name.

Steps to Use DOM parser:

Following are the steps used while parsing a document using DOM Parser.

1. Import XML-related packages

import org.w3c.dom.;*
import javax.xml.parsers.;*
import java.io.;*

2. Create a DocumentBuilder

DocumentBuilderFactory factory =DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();

3. Create a Document from a file or stream

StringBuilder strb = new StringBuilder();
strb.append("<?xml version="1.0"?> <bookstore> </bookstore>");
ByteArrayInputStream input = new ByteArrayInputStream(strb.toString().getBytes("UTF-8"));
Document doc = builder.parse(input);

4. Extract the root element

Element root = document.getDocumentElement();

5. Examine attributes

getAttribute("attributeName");
getAttributes();

6. Examine sub-elements

getElementsByTagName("subelementName");
getChildNodes();

Example for using of DOM parser:

class.xml

```
<?xml version="1.0"?>
<class>
        <student rollno="393">
                <firstname>dinkar</firstname>
                <lastname>kad</lastname>
                <nickname>dinkar</nickname>
                <marks>85</marks>
        </student>
        <student rollno="493">
                <firstname>Vaneet</firstname>
                <lastname>Gupta</lastname>
                <nickname>vinni</nickname>
                <marks>95</marks>
         </student>
</class>
```

DomParserDemo.java

```
import java.io.File;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import org.w3c.dom.Element;
public class DomParserDemo
{
  public static void main(String[] args){
 try {
      File inputFile = new File("input.txt");
```

```
        DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
        DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
        Document doc = dBuilder.parse(inputFile);
        doc.getDocumentElement().normalize();
        System.out.println("Root element :" + doc.getDocumentElement().getNodeName());
        NodeList nList = doc.getElementsByTagName("student");
        System.out.println(" ---------------------------- ");
        for (int temp = 0; temp < nList.getLength(); temp++)
            {           Node nNode = nList.item(temp);
                        System.out.println("\nCurrent Element  :"  + nNode.getNodeName());
                        if (nNode.getNodeType() == Node.ELEMENT_NODE)
                       {           Element eElement = (Element) nNode;
                                   System.out.println("Student roll no : " + eElement.getAttribute("rollno"));
System.out.println("First                                   Name                                    :
"+eElement.getElementsByTagName("firstname").item(0).getTextContent());
System.out.println("Last                    Name                    :                               "                                       +
eElement.getElementsByTagName("lastname").item(0).getTextContent());
System.out.println("Nick                                    Name                                    :
"+eElement.getElementsByTagName("nickname").item(0).getTextContent());
System.out.println("Marks : " +eElement.getElementsByTagName("marks").item(0).getTextContent());
            }
        }
    } catch (Exception e) {       e.printStackTrace();      }
  }
}
```

2. Java SAX Parser:

SAX (the Simple API for XML) is an event-based parser for xml documents.Unlike a DOM parser, a SAX parser creates no parse tree. SAX is a streaming interface for XML, which means that applications using SAX receive event notifications about the XML document being processed an element, and attribute, at a time in sequential order starting at the top of the document, and ending with the closing of the ROOT element. Reads an XML document from top to bottom, recognizing the tokens that make up a well-formed XML document. Tokens are processed in the same order that they appear in the document. Reports the application program the nature of tokens that the parser has encountered as they occur. The application program provides an "event" handler that must be registered with the parser. As the tokens are identified, callback methods in the handler are invoked with the relevant information

ContentHandler Interface

This interface specifies the callback methods that the SAX parser uses to notify an application program of the components of the XML document that it has seen.

- **void startDocument()** - Called at the beginning of a document.
- **void endDocument()** - Called at the end of a document.
- **void startElement(String uri, String localName, String qName, Attributes atts)** - Called at the beginning of an element.
- **void endElement(String uri, String localName,String qName)** - Called at the end of an element.
- **void characters(char[] ch, int start, int length)** - Called when character data is encountered.
- **void ignorableWhitespace(char[] ch, int start, int length)** - Called when a DTD is present and ignorable whitespace is encountered.
- **void processingInstruction(String target, String data)** - Called when a processing instruction is recognized.

- **void setDocumentLocator(Locator locator))** - Provides a Locator that can be used to identify positions in the document.
- **void skippedEntity(String name)** - Called when an unresolved entity is encountered.
- **void startPrefixMapping(String prefix, String uri)** - Called when a new namespace mapping is defined.
- **void endPrefixMapping(String prefix)** - Called when a namespace definition ends its scope.

Attributes Interface

This interface specifies methods for processing the attributes connected to an element.

- **int getLength()** - Returns number of attributes.
- **String getQName(int index)**
- **String getValue(int index)**
- **String getValue(String qname)**

Example for using of SAX parser:

class.xml

```
<?xml version="1.0"?>
<class>
        <student rollno="393">
                <firstname>dinkar</firstname>
                <lastname>kad</lastname>
                <nickname>dinkar</nickname>
                <marks>85</marks>
        </student>
        <student rollno="493">
                <firstname>Vaneet</firstname>
                <lastname>Gupta</lastname>
                <nickname>vinni</nickname>
                <marks>95</marks>
         </student>
</class>
```

SAXParserDemo.java

```
import java.io.File;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class SAXParserDemo {
   public static void main(String[] args){
      try {
            File inputFile = new File("input.txt");
            SAXParserFactory factory = SAXParserFactory.newInstance();
            SAXParser saxParser = factory.newSAXParser();
            UserHandler userhandler = new UserHandler();
            saxParser.parse(inputFile, userhandler);
      } catch (Exception e) {        e.printStackTrace();      }
   }
}
class UserHandler extends DefaultHandler {
   boolean bFirstName = false;
```

```
boolean bLastName = false;
boolean bNickName = false;
boolean bMarks = false;
public void startElement(String uri, String localName, String qName, Attributes attributes)
    throws SAXException {
        if (qName.equalsIgnoreCase("student"))
        {           String rollNo = attributes.getValue("rollno");
                    System.out.println("Roll No : " + rollNo);
        }
        else if (qName.equalsIgnoreCase("firstname"))
        {       bFirstName = true;      }
        else if (qName.equalsIgnoreCase("lastname"))
        {       bLastName = true;       }
        else if (qName.equalsIgnoreCase("nickname"))
        {       bNickName = true;       }
         else if (qName.equalsIgnoreCase("marks"))
        {       bMarks = true;          }
}
public void endElement(String uri,  String localName, String qName) throws SAXException {
  if (qName.equalsIgnoreCase("student"))
        {       System.out.println("End  Element :" + qName);       }
}
public void characters(char ch[],   int start, int length) throws SAXException {
  if (bFirstName) {
    System.out.println("First Name: " + new String(ch, start, length));
    bFirstName = false;
  } else if (bLastName) {
    System.out.println("Last Name: " + new String(ch, start, length));
    bLastName = false;
  } else if (bNickName) {
    System.out.println("Nick Name: " + new String(ch, start, length));
    bNickName = false;
  } else if (bMarks) {
    System.out.println("Marks: "    + new String(ch, start, length));
    bMarks = false;
  }
}
}
```

AJAX (Asynchronous JavaScript and XML)

AJAX is an acronym for **Asynchronous JavaScript and XML**. It is a group of inter-related technologies like javascript, dom, xml, html, css etc. AJAX allows you to send and receive data asynchronously without reloading the entire web page. So it is fast.

AJAX allows you to send only important information to the server not the entire page. So only valuable data from the client side is routed to the server side. It makes your application interactive and faster.

Where it is used?

There are too many web applications running on the web that are using AJAX Technology. Some are:

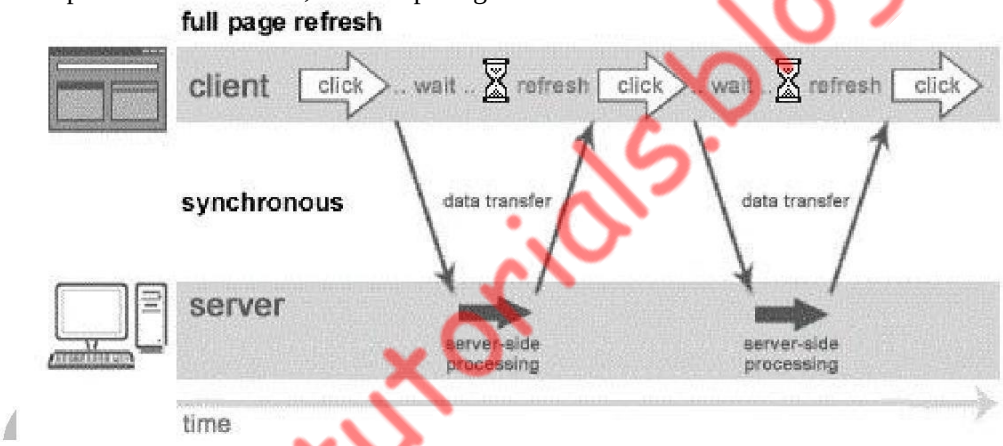
1. Gmail
2. Facebook
3. Twitter
4. Google maps
5. YouTube etc.,

Synchronous Vs. Asynchronous Application

Before understanding AJAX, let's understand classic web application model and AJAX Web application model.

- **Synchronous (Classic Web-Application Model)**

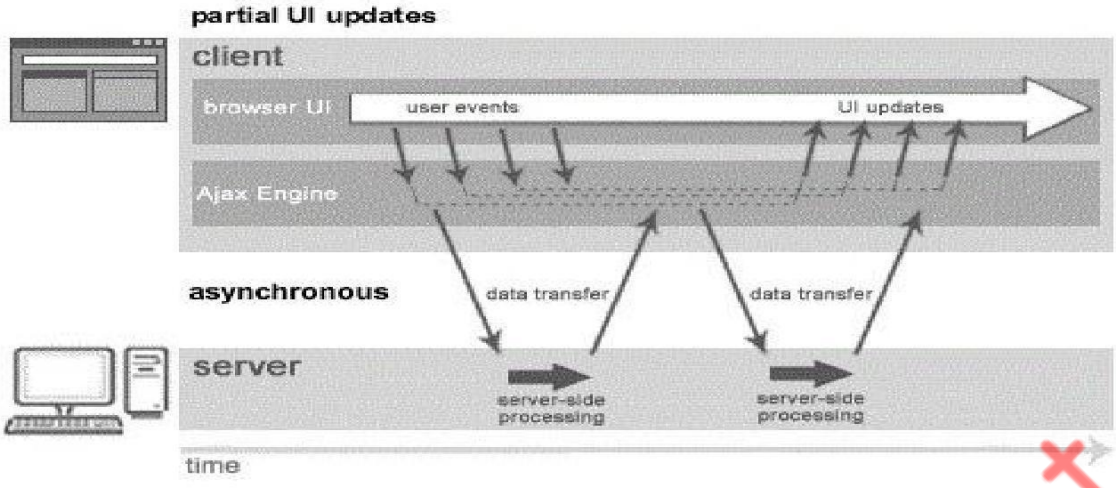
A synchronous request blocks the client until operation completes i.e. browser is not unresponsive. In such case, JavaScript Engine of the browser is blocked.



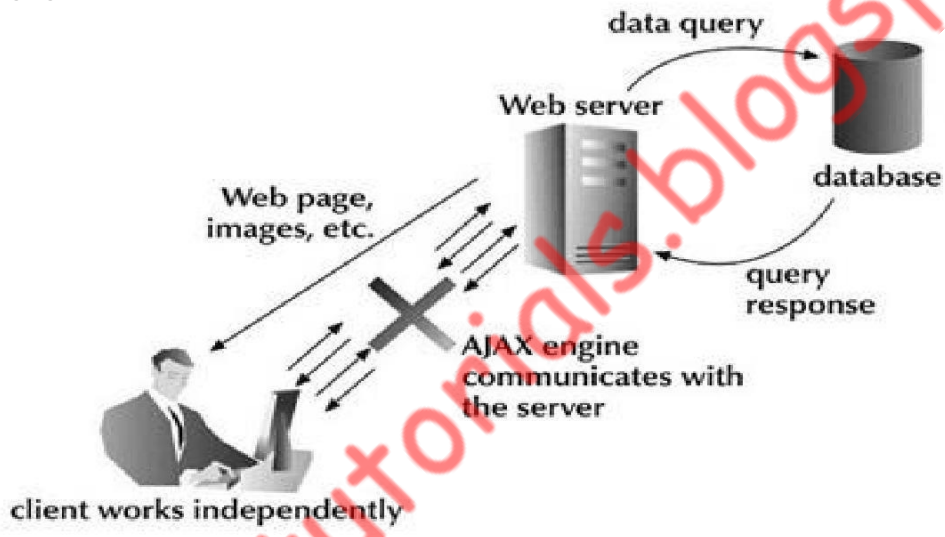
As you can see in the above image, full page is refreshed at request time and user is blocked until request completes. Let's understand it another way.

- **Asynchronous (AJAX Web-Application Model)**

An asynchronous request doesn't block the client i.e. browser is responsive. At that time, user can perform other operations also. In such case, JavaScript Engine of the browser is not blocked.



As you can see in the above image, full page is not refreshed at request time and user gets response from the AJAX Engine. Let's try to understand asynchronous communication by the image given below.



AJAX Technologies

AJAX is not a Technology but group of inter-related technologies. AJAX Technologies includes:

- HTML/XHTML and CSS
- DOM
- XML or JSON(JavaScript Object Notation)
- XMLHttpRequest
- JavaScript

- **HTML/XHTML and CSS**
  These technologies are used for displaying content and style. It is mainly used for presentation.

- **DOM**
  It is used for dynamic display and interaction with data.

- **XML or JSON**
  For carrying data to and from server. JSON is like XML but short and faster than XML.

- XMLHttpRequest
  For asynchronous communication between client and server.

- JavaScript
  It is used to bring above technologies together. Independently, it is used mainly for client-side validation.

Understanding XMLHttpRequest

An object of XMLHttpRequest is used for asynchronous communication between client and server. It performs following operations:

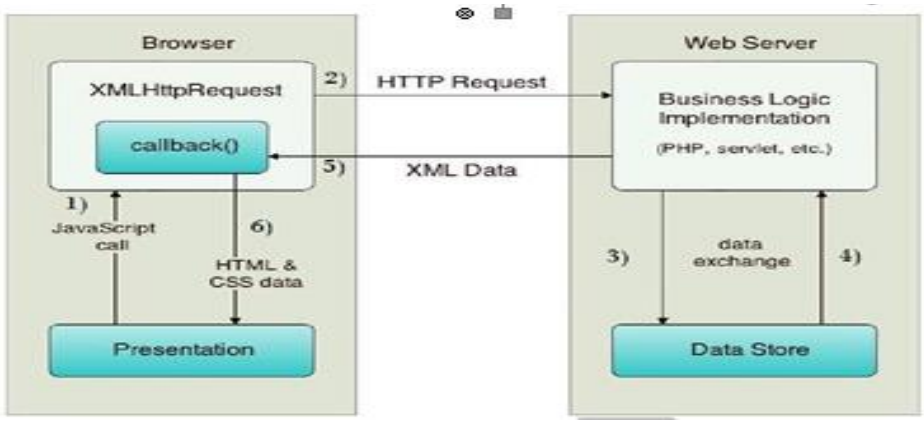
1. Sends data from the client in the background
2. Receives the data from the server
3. Updates the webpage without reloading it.

- **Properties of XMLHttpRequest object:**

Property	Description
onReadyStateChange	It is called whenever readystate attribute changes. It must not be used with synchronous requests.
readyState	Represents the state of the request. It ranges from 0 to 4. **0** UNOPENED open() is not called. **1** OPENED open is called but send() is not called. **2** HEADERS_RECEIVED send() is called, and headers and status are available. **3** LOADING Downloading data; responseText holds the data. **4** DONE The operation is completed fully.
reponseText	Returns response as TEXT.
responseXML	Returns response as XML

Method	Description
void open(method, URL)	Opens the request specifying get or post method and url.
void open(method, URL, async)	Same as above but specifies asynchronous or not.
void open(method, URL, async, username, password)	Same as above but specifies username and password.
void send()	Sends GET request.
void send(string)	Sends POST request.
setRequestHeader(header,value)	It adds request headers.

1. User sends a request from the UI and a javascript call goes to XMLHttpRequest object.
2. HTTP Request is sent to the server by XMLHttpRequest object.
3. Server interacts with the database using JSP, PHP, Servlet, ASP.net etc.
4. Data is retrieved.
5. Server sends XML data or JSON data to the XMLHttpRequest callback function.
6. HTML and CSS data is displayed on the browser.



Integrating PHP and AJAX:-

The following example will demonstrate how a web page can communicate with a web server while a user type characters in an input field:

Example

Start typing a name in the input field below:

First name:

Suggestions:

Example Explained

In the example above, when a user types a character in the input field, a function called "showHint()" is executed.

The function is triggered by the onkeyup event.

Here is the HTML code:

Example

```
<html>
<head>
<script>
function showHint(str) {
   if (str.length == 0) {
      document.getElementById("txtHint").innerHTML = "";
      return;
   } else {
```

```
        var xmlhttp = new XMLHttpRequest();
        xmlhttp.onreadystatechange = function() {
            if (this.readyState == 4 && this.status == 200) {
                document.getElementById("txtHint").innerHTML = this.responseText;
            }
        };
        xmlhttp.open("GET", "gethint.php?q=" + str, true);
        xmlhttp.send();
    }
}
</script>
</head>
<body>

<p><b>Start typing a name in the input field below:</b></p>
<form>
First name: <input type="text" onkeyup="showHint(this.value)">
</form>
<p>Suggestions: <span id="txtHint"></span></p>
</body>
</html>
```

Code explanation:

First, check if the input field is empty (str.length == 0). If it is, clear the content of the txtHint placeholder and exit the function.

However, if the input field is not empty, do the following:

- Create an XMLHttpRequest object
- Create the function to be executed when the server response is ready
- Send the request off to a PHP file (gethint.php) on the server
- Notice that q parameter is added to the url (gethint.php?q="+str)
- And the str variable holds the content of the input field

The PHP File - "gethint.php"

The PHP file checks an array of names, and returns the corresponding name(s) to the browser:

```
<?php
// Array with names
$a[] = "Anna";
$a[] = "Brittany";
$a[] = "Cinderella";
$a[] = "Diana";
$a[] = "Eva";
$a[] = "Fiona";
$a[] = "Gunda";
$a[] = "Hege";
$a[] = "Inga";
$a[] = "Johanna";
```

```
$a[] = "Kitty";
$a[] = "Linda";
$a[] = "Nina";
$a[] = "Ophelia";
$a[] = "Petunia";
$a[] = "Amanda";
$a[] = "Raquel";
$a[] = "Cindy";
$a[] = "Doris";
$a[] = "Eve";
$a[] = "Evita";
$a[] = "Sunniva";
$a[] = "Tove";
$a[] = "Unni";
$a[] = "Violet";
$a[] = "Liza";
$a[] = "Elizabeth";
$a[] = "Ellen";
$a[] = "Wenche";
$a[] = "Vicky";

// get the q parameter from URL
$q = $_REQUEST["q"];

$hint = "";

// lookup all hints from array if $q is different from ""
if ($q !== "") {
    $q = strtolower($q);
    $len=strlen($q);
    foreach($a as $name) {
        if (stristr($q, substr($name, 0, $len))) {
            if ($hint === "") {
                $hint = $name;
            } else {
                $hint .= ", $name";
            }
        }
    }
}

// Output "no suggestion" if no hint was found or output correct values
echo $hint === "" ? "no suggestion" : $hint;
?>
```

Introduction to Web Services

Technology keep on changing, users were forces to learn new application on continuous basis. With internet, focus is shifting to-wards services based software. Users may access these services using wide range of devices such as PDAs, mobile phones, desktop computers etc. Service oriented software development is possible using man known techniques such as COM, CORBA, RMI, JINI, RPC etc. some of them are capable of delivering services over web & some or not. Most of these technologies uses particular protocols for communication & with no standardization. **Web service** is the concept of creating services that can be accessed over web. Most of these

What are Web Services?

A web services may be defines as: An application component accessible via standard web protocols. It is like unit of application logic. It provides services & data to remote clients & other applications. Remote clients & application access web services with internet protocols. They use XML for data transport & SOAP for using services. Accessing service is independent of implementation. With component development model, web service must have following characteristics:

- Registration with lookup service
- Public interface for client to invoke service
- It should use standard web protocols for communication
- It should be accessible over web
- It should support loose coupling between uncoupled distributed systems

Web services receive information from clients as messages, containing instructions about what client wants, similar to method calls with parameters. These message delivered by web services are encoded using XML.XML enabled web services are interoperable with other web services.

Web Service Technologies:

Wide variety of technologies supports web services. Following technologies are available for creation of web services. These are vendor neutral technologies. They are:

- Simple Object Access Protocol(SOAP)
- Web Services Description Language(WSDL)
- UDDI(Universal Description Discovery and Integration)

Simple Object Access Protocol (SOAP):

SOAP is a light weight & simple XML based protocol. It enables exchange of structured & typed information on web by describing messaging format for machine to machine communication. It also enables creation of web services based on open infrastructure. SOAP consists of three parts:

- **SOAP Envelope**: defines what is in message, who is the recipient, whether message is optional or mandatory

- **SOAP Encoding Rules**: defines set of rules for exchanging instances of application defined data types
- **SOAP RPC Representation**: defines convention for representing remote procedure calls & response

SOAP can be used in combination with variety of existing internet protocols & formats including HTTP, SMTP etc. Typical SOAP message is shown below:

```
<IVORY:Envelope xmlns:IVORY="http://schemas.xmlsoap.org/soap/envelope"
                IVORY:encodingStyle="http://schemas.xmlsoap.org/soap/encoding">
<IVORY:Body>
    <m:GetLastTradePrice xmlns:m="Some-URI">
    <symbol>DIS</symbol>
    </m:GetLastTradePrice>
</IVORY:Body>
</IVORY:Envelope>
```

The consumer of web service creates SOAP message as above, embeds it in HTTP POST request & sends it to web service for processing:

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml;
charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"
    ....
SOAP Message
....
```

The message now contains requested stock price. A typical returned SOAP message may look like following:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope"
SOAP-ENV:encodingStyle=" http://schemas.xmlsoap.org/soap/encoding" />
    <SOAP-ENV:Body>
        <m:GetLastTradePrice xmlns:m="Some-URI">
            <Price>34.5</Price>
        </m:GetLastTradePrice>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Interoperability:

The major goal in design of SOAP was to allow for easy creation of interoperable distributed web services. Few details of SOAP specifications are open for interpretation; implementation may differ across different vendors. SOAP message though it is conformant XML message, may not strictly follow SOAP specification.

Implementations:

SOAP technology was developed by DevelopMentor, IBM, Lotus, Microsoft etc. More than 50 vendors have currently implemented SOAP. Most popular implementations are by Apache which is open source java based implementation & by Microsoft in .NET platform. SOAP specification has been submitted to W3C, which is now working on new specifications called XMLP (XML Protocol)

SOAP Messages with Attachments (SwA)

SOAP can send message with an attachment containing of another document or image etc. On Internet, GIF, JPEG data formats are treated as standards for image transmission. Second iteration of SOAP specification allowed for attachments to be combined with SOAP message by using multipart

MIME structure. This multi part structure is called as **SOAP Message Package**. This new specification was developed by HP & Microsoft. Sample SOAP message attachment is shown here:

```
MIME-Version: 1.0
Content-Type: Multipart/Related; boundary=MIME_boundary;
type=text/xml; start=”<myimagedoc.xml@mystie.com>”
Content-Description: This is the optional message description.
--MIME_boundary
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <myimagedoc.xml@mysite.com>
<?xml version="1.0"?>
<SOAP-ENV: Envelope xmlns:SOAP-ENV=”http://schemas.xmlsoap.org/soap/envelope”
<SOAP-ENV:Body>
        …
        <theSignedForm href=”cid:myimage.tiff@mysite.com” />
        …
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
        --MIME_boundary
        Content-Type: image/tiff
        Content-Transfer-Encoding: binary
        Content-ID: <myimagedoc.xml@mysite.com>
        …binary TIFF image…
        --MIME_boundary--
```

Web Services Description Language (WSDL)

WSDL is an XML format for describing web service interface. WSDL file defines set of operations permitted on the server & format that client must follow while requesting service. WSDL file acts like contract between client & service for effective communication between two parties. Client has to request service by sending well formed & conformant SOAP request.

If we are creating web service that offered latest stock quotes, we need to create WSDL file on server that describes service. Client obtains copy of this file, understand contract, create SOAP request based on contract & dispatch request to server using HTTP post. Server validates the request, if found valid executes request. The result which is latest stock price for requested symbol is then returned to client as SOAP response.

WSDL Document:

WSDL document is an XML document that contains of set of definitions. First we declare name spaces required by schema definition:

```
<schema xmlns=”http://www.w3.org/2000/10/XMLSchema”
xmlns:wsdl=”http://schemas.xmlsoap.org/wsdl/”
targetNameSpace=http://schemas.xmlsoap.org/wsdl/ elementFormDefault=”qualified”>
```

The root element is definitions as shown below:

```
<wsdl:definitions name=”nmtoken”? targetNameSpace=”uri”?>
        <import namespace=”uri” location=”uri”/>
<wsdl:documentation ….. />?
        …
</wsdl:definitions>
```

The *name* attribute is optional & can serve as light weight form of documentation. The *nmtoken* represents name token that are qualified strings similar to CDATA, but character usage is limited to letters, digits, underscores, colons, periods & dashes. A *targetNamespace* may be specified by providing uri. The *import* tag may be used to associate namespace with document locations. Following code segment shows how declared namespace is associated with document location specified in *import* statement:

```
<definitions                                    name=”StockQuote”
        targetNameSpace=”http://example.com/stockquote/definitions”
```

```
        xmlns:tns=”http://example.com/stockquote/definitions”
        xmlns:xsdl=”http://example.com/stockquote/schemas”
        xmlns:soap=”http://schemas.xmlsoap.org/wsdl/soap/”
        xmlns=”http://schemas.xmlsoap.org/wsdl/”>
<import namespace=”http://example.com/stockquote/schemas”
        Location=”http://example.com/stockquote/stockquote.xsd”/>
```

Finally, optional *wsdl:documentation* element is used for declaring human readable documentation. The element may contain any arbitrary text. There are six major elements in document structure that describes service. These are as follows:

- **Types Element:** it provides definitions for data types used to describe how messages will exchange data. Syntax for types element is as follows:

```
<wsdl:types> ?
        <wsdl:documentation
        .../> <xsd:schema .../>
        <-- extensibility element -->
</wsdl:types>
```

The *wsdl:documentation* tag is optional as in case of *definitions*. The *xsd* type system may be used to define types in message. WSDL allows type systems to be added via extensibility element.

- **Message Element:** It represents abstract definition of data begin transmitted. Syntax for message element:

```
<wsdl:message name=”nktoken”> *
        <wsdl;documentation .../>
            <part name=”nmtoken” element=”qname”? type=”qname”? /> *
</wsdl:message>
```

The *message name* attribute is used for defining unique name for message with in document scope. The *wsdl:documentation* is optional & may be used for declaring human readable documentation. The message consists of one or more logical parts. The *part* describes logical abstract content of message. Each part consists of name & optional element & type attributes.\

- **Port Type Element:** It defines set of abstract operations. An operation consists of both input & output messages. The *operation* tag defines name of operation, *input* defines input for operation & *output* defines output format for result. The *fault* element is used for describing contents of SOAP fault details element. It specifies abstract message format for error messages that may be output as result of operation:

```
<wsdl:portType name=”nmtoken”>*
        <wsdl:documentation ..../>?
    <wsdl:operation name=”nmtoken”>*
        <wsdl:documentation ..../>?
            <wsdl:input name=”nmtoken”? message=”qname”>?
        <wsdl:documentation ..../>?
        </wsdl:input>
            <wsdl:output name=”nmtoken”? message=”qname”>?
        <wsdl:documentation ..../>?
        </wsdl:output>
            <wsdl:fault name=”nmtoken”? message=”qname”>?
            <wsdl:documentation ..../>?
        </wsdl:fault>
    </wsdl:operation>
</wsdl:portType>
```

- **Binding Element:** It defines protocol to be used & specifies data format for operations & messages defined by particular *portType*. The full syntax for binding is given below:

```
<wsdl:binding name="nmtoken" type="qname"> *
        <wsdl:documentation ..../>?
        <--Extensibility element -->*
    <wsdl:operation name="nmtoken">*
        <wsdl:documentation ..../>?
        <--Extensibility element -->*
    <wsdl:input> ?
        <wsdl:documentation ..../>?
        <--Extensibility element -->*
    </wsdl:input>
    <wsdl:output> ?
        <wsdl:documentation ..../>?
        <--Extensibility element -->*
    </wsdl:output>
    <wsdl:fault name="nmtoken"> *
        <wsdl:documentation ..../>?
        <--Extensibility element -->*
    </wsdl:fault>
    </wsdl:operation>
</wsdl:binding>
```

The operation in WSDL file can be document oriented or remote procedure call (RPC) oriented. The style attribute of <*soap:binding*> element defines type of operation. If operation is document oriented, input & output messages will consist of XML documents. If operation is RPC oriented, input message contains operations input parameters & output message contains result of operation.

❖ **Port Element:** It defines individual end point by specifying single address for binding:

```
<wsdl:port name="nmtoken" binding="qname"> *
    <--Extensibility element (1) -->
</wsdl:port>
```

The *name* attribute defines unique name for port with current WSDL document. The *binding* attribute refers to binding & extensibility element is used to specify address information for port.

❖ **Service Element:** it aggregates set of related ports. Each port specifies address for binding:

```
<wsdl:service name="nmtoken"> *
        <wsdl:documentation ..../>?
    <wsdl:port name="nktoken" binding="qname"> *
        <wsdl:documentation .../> ?
        <--Extensibility element -->
    </wsdl:port>
        <--Extensibility element -->
</wsdl:service>
```

Universal Description, Discovery & Integration (UDDI)

We need to publish web services so that customers & business partners can use the services. It requires common registry to register web service for clients to find it. For this several vendors including IBM, HP, Oracle, Sun Microsystem etc. formed an industry consortium known as UDDI. Today more than 250 companies have joined UDDI project. The main task of this project is to develop specifications for web based business registry. The registry should be able to describe web service & allow others to discover registered web services.

UDDI allows any organization to publish information about its web services. The framework defines standard for businesses to share information, describe their services & their business & to decide what information is made public & what information is kept private. The interface is based on XML & SOAP, uses HTTP to interact with registry.

Registry itself holds information about business such as company name, contact etc. it holds both descriptive & technical information about web service. It provides search facilities that allow to search specific industry segment or geographic location.

Implementation:

This is global, public registry called UDDI business registry. It is possible for individuals to set up private UDDI registries. The implementations for creating private registries are available from IBM, Idoox etc. Microsoft has developed UDDI SDK that allows visual basic programmer to write program code to interact with UDDI registry. The use of SDK greatly simplifies interaction with registry & shields programmer from local level details of XML & SOAP.

Electronic Business XML (ebXML):

ebXML is set of specifications that allows businesses to collaborate. It enables global electronic market place where business can meet & tranasact with help of XML based messages. Business may be geographically located anywhere in world & could be of any size to participate in global marketplace. The framework defines specifications for sharing of web based business services. It includes specifications for message service, collaborative partner agreements, core components, business process methodology, registry & repository.

It defines registry & repository where business can register themselves by providing their contact information, address & so on. Such information is called Core Component. After business has registered with ebXML registry, other partners can look up registry to locate that business. Once business partner is located, the core components of located business are downloaded. Once buyer is satisfied with fact that seller service can meet its requirements, it negotiates contract with seller. Such collaborative partner agreements are defined in ebXML. Once both parties agree on contract terms, sign agreements & collaborative business transaction by exchanging their private documents. ebXML provides marketplace & defines several XML based documents for business to join & transact in such marketplace.