UNIT-2 XML

XML - XML stands for **Extensible Mark-up Language**, developed by W3C in 1996. It is a text-based mark-up language derived from Standard Generalized Mark-up Language (SGML). XML 1.0 was officially adopted as a W3C recommendation in 1998. XML was designed to carry data, not to display data. XML is designed to be self-descriptive. XML is a subset of SGML that can define your own tags. A Meta Language and tags describe the content. XML Supports CSS, XSL, DOM. XML does not qualify to be a programming language as it does not performs any computation or algorithms. It is usually stored in a simple text file and is processed by special software that is capable of interpreting XML.

The Difference between XML and HTML

1. HTML is about displaying information, where asXML is about carrying information. In other words, XML was created to structure, store, and transport information. HTML was designed to display the data.
2. Using XML, we can create own tags where as in HTML it is not possible instead it offers several built in tags.
3. XML is platform independent neutral and language independent.
4. XML tags and attribute names are case-sensitive where as in HTML it is not.
5. XML attribute values must be single or double quoted where as in HTML it is not compulsory.
6. XML elements must be properly nested.
7. All XML elements must have a closing tag.

Well Formed XML Documents

A "Well Formed" XML document must have the following correct XML syntax:

- XML documents must have a root element
- XML elements must have a closing tag(start tag must have matching end tag).
- XML tags are case sensitive
- XML elements must be properly nested Ex:<one><two>Hello</two></one>
- XML attribute values must be quoted

XML with correct syntax is "Well Formed" XML. XML validated against a DTD is "Valid" XML.

What is Markup?

XML is a markup language that defines set of rules for encoding documents in a format that is both human-readable and machine-readable.

Example for XML Document

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?> <!—xml declaration-->
<note>
<to>MRCET</to>
<from>MRGI</from>
<heading>KALPANA</heading>
<body>Hello, world! </body>
</note>
```

- Xml document begins with XML declaration statement: <? xml version="1.0" encoding="ISO-8859-1"?> .
- The next line describes the **root element** of the document: **<note>**.

- This element is "the parent" of all other elements.
- The next 4 lines describe 4**child elements** of the root: to, from, heading, and body. And finally the last line defines the end of the root element : **< /note>.**
- The XML declaration has no closing tag i.e. </?xml>
- The **default standalone value** is set to **no**. Setting it to **yes** tells the processor there are no external declarations (DTD) required for parsing the document. The file name extension used for xml program is.xml.

Valid XML document
If an XML document is well- formed and has an associated Document Type Declaration (DTD), then it is said to be a valid XML document. We will study more about DTD in the chapter XML - DTDs.

XML DTD

Document Type Definition purpose is to define the structure of an XML document. It defines the structure with a list of defined elements in the xml document. Using DTD we can specify the various elements types, attributes and their relationship with one another. Basically DTD is used to specify the set of rules for structuring data in any XML file.

Why use a DTD?
XML provides an application independent way of sharing data. With a DTD, independent groups of people can agree to use a common DTD for interchanging data. Your application can use a standard DTD to verify that data that you receive from the outside world is valid. You can also use a DTD to verify your own data.

DTD - XML building blocks

Various building blocks of XML are-

1. Elements: The basic entity is **element**. The elements are used for defining the tags. The elements typically consist of opening and closing tag. Mostly only one element is used to define a single tag.

Syntax1: <!ELEMENT element- name (element-content)>

Syntax 2: <!ELEMENT element- name (#CDATA)>

#CDATA means the element contains character data that is not supposed to be parsed by a parser. or

Syntax 3: <!ELEMENT element- name (#PCDATA)>

#PCDATA means that the element contains data that IS going to be parsed by a parser. or

Syntax 4: <!ELEMENT element- name (ANY)>

The keyword ANY declares an element with any content.

Example:

<!ELEMENT note (#PCDATA)>

Elements with children (sequences)

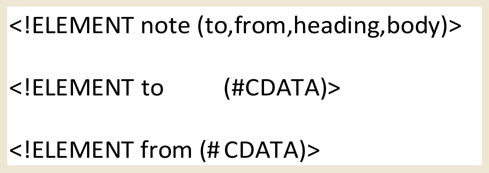
Elements with one or more children are defined with the name of the children elements inside the parentheses:

<!ELEMENT parent-name (child-element- name)> **EX:**`<!ELEMENT student (id)>`

`<!ELEMENT id (#PCDATA)>` or

<!ELEMENT element-name (child-element-name, child-element-name,. ...)>

Example: <!ELEMENT note (to,from,heading,body)>



```
<!ELEMENT note (to,from,heading,body)>

<!ELEMENT to        (#CDATA)>

<!ELEMENT from (# CDATA)>
```

2. Tags

Tags are used to markup elements. A starting tag like <element_name> mark up the beginning of an element, and an ending tag like </element_name> mark up the end of an element.

Examples:

A body element: <body>body text in between</body>.

A message element: <message>some message in between</message>

3. Attribute: The attributes are generally used to specify the values of the element. These are specified within the double quotes. Ex: <flag type=‖true‖>

4. Entities

Entities as variables used to define common text. Entity references are references to entities. Most of you will known the HTML entity reference: "&nbsp;" that is used to insert an extra space in an HTML document. Entities are expanded when a document is parsed by an XML parser.

The following entities are predefined in XML:

&lt; (<), &gt;(>), &amp;(&), &quot;(") and &apos;(').

5. CDATA: It stands for character data. CDATA is text that will **NOT be parsed by a parser**. Tags inside the text will NOT be treated as markup and entities will not be expanded.

6. PCDATA: It stands for Parsed Character Data(i.e., text). Any parsed character data should not contain the markup characters. The markup characters are < or > or &. If we want to use these characters then make use of &lt; , &gt; or &amp;. Think of character data as the text found between the start tag and the end tag of an XML element. PCDATA is text that will be **parsed by a parser**. Tags inside the text will be treated as markup and entities will be expanded.

```
<!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

Where PCDATA refers parsed character data. In the above xml document the elements to, from, heading, body carries some text, so that, these elements are declared to carry text in DTD file.

This definition file is stored with **.dtd** extension.

DTD identifier is an identifier for the document type definition, which may be the path to a file on the system or URL to a file on the internet. If the DTD is pointing to external path, it is called External Subset.

The square brackets [ ] enclose an optional list of entity declarations called Internal Subset.

Types of DTD:

1. Internal DTD
2. External DTD

1. Internal DTD

A DTD is referred to as an internal DTD if elements are declared within the XML files. To refer it as internal DTD, standalone attribute in XML declaration must be set to yes. This means, the declaration works independent of external source.

Syntax:

The syntax of internal DTD is as shown:

```
<!DOCTYPE root-element [element-declarations]>
```

Where root-element is the name of root element and element-declarations is where you declare the elements.

Example:

Following is a simple example of internal DTD:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE address [
  <!ELEMENT address (name,company,phone)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT company (#PCDATA)>
  <!ELEMENT phone (#PCDATA)>
]>
<address>
  <name>Kalpana</name>
  <company>MRCET</company>
  <phone>(040) 123-4567</phone>
</address>
```

Let us go through the above code:

Start Declaration- Begin the XML declaration with following statement <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>

DTD- Immediately after the XML header, the document type declaration follows, commonly referred to as the DOCTYPE:

<!DOCTYPE address [

The DOCTYPE declaration has an exclamation mark (!) at the start of the element name. The DOCTYPE informs the parser that a DTD is associated with this XML document.

DTD Body- The DOCTYPE declaration is followed by body of the DTD, where you declare elements, attributes, entities, and notations:

```
<!ELEMENT address (name,company,phone)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT company (#PCDATA)>
<!ELEMENT phone_no (#PCDATA)>
```

Several elements are declared here that make up the vocabulary of the <name> document. <!ELEMENT name (#PCDATA)> defines the element name to be of type "#PCDATA". Here #PCDATA means parse-able text data. End Declaration - Finally, the declaration section of the DTD is closed using a closing bracket and a closing angle bracket (]>). This effectively ends the definition, and thereafter, the XML document follows immediately.

Rules

✓ The document type declaration must appear at the start of the document (preceded only by the XML header) — it is not permitted anywhere else within the document.

✓ Similar to the DOCTYPE declaration, the element declarations must start with an exclamation mark.

✓ The Name in the document type declaration must match the element type of the root element.

External DTD

In external DTD elements are declared outside the XML file. They are accessed by specifying the system attributes which may be either the legal .dtd file or a va lid URL. To refer it as external DTD, standalone attribute in the XML declaration must be set as no. This means, declaration includes information from the externalsource.

Syntax Following is the syntax for external DTD:

```
<!DOCTYPE root-element SYSTEM "file-name">
```

where file- name is the file with **.dtd** extension.

Example The following example shows external DTD usage:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE address SYSTEM "address.dtd">
<address>
  <name>Kalpana</name>
  <company>MRCET</company>
  <phone>(040) 123-4567</phone>
</address>
```

The content of the DTD file **address.dtd** are as shown:

```
<!ELEMENT address (name,company,phone)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT company (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
```

Types

You can refer to an external DTD by using either system identifiers or public identifiers.

SYSTEM IDENTIFIERS

A system identifier enables you to specify the location of an external file containing DTD declarations. Syntax is as follows:

```
<!DOCTYPE name SYSTEM "address.dtd" [...]>
```

As you can see, it contains keyword SYSTEM and a URI reference pointing to the location of the document.

PUBLIC IDENTIFIERS

Public identifiers provide a mechanism to locate DTD resources and are written as below:

```
<!DOCTYPE name PUBLIC "-//Beginning XML//DTD Address Example//EN">
```

As you can see, it begins with keyword PUBLIC, followed by a specialized identifier. Public identifiers are used to identify an entry in a catalog. Public identifiers can follow any format; however, a commonly used format is called Formal Public Identifiers, or FPIs.

XML Schemas

- XML Schema is commonly known as XML Schema Definition (XSD). It is used to describe and validate the structure and the content of XML data. XML schema defines the elements, attributes and data types. Schema element supports Namespaces. It is similar to a database schema that describ es the data in a database. XSD extension is**".xsd"**.
- This can be used as an alternative to XML DTD. The XML schema became the W#C recommendation in 2001.
- XML schema defines elements, attributes, element having child elements, order of child elements. It also defines fixed and default values of elements and attributes.
- XML schema also allows the developer to us **data types**.

Syntax : You need to declare a schema in your XML document as follows:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

Example : contact.xsd

The following example shows how to use schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="contact">
<xs:complexType>
<xs:sequence>
        <xs:element name="name" type="xs:string" />
        <xs:element name="company" type="xs:string" />
        <xs:element name="phone" type="xs:int" />
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

The basic idea behind XML Schemas is that they describe the legitimate format that an XML document can take.

XML Document: myschema.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<contact xmlns:xsi=http://www.w3.org/2001/XMLSchema- instance

xsi:noNamespaceSchemaLocation="contact.xsd">
<name>KALPANA</name>
<company>04024056789</company>
<phone>9876543210</phone>
</contact>
```

Limitations of DTD:

- There is no built- in data type in DTDs.
- No new data type can be created in DTDs.
- The use of cardinality (no. of occurrences) in DTDs is limited.
- Namespaces are not supported.
- DTDs provide very limited support for modularity and reuse.
- We cannot put any restrictions on text content.
- Defaults for elements cannot be specified.
- DTDs are written in a non-XML format and are difficult to validate.

Strengths of Schema:

- XML schemas provide much greater specificity than DTDs.
- They supports large number of built- in-data types.
- They are namespace-aware.
- They are extensible to future additions.
- They support the uniqueness.
- It is easier to define data facets (restrictions on data).

SCHEMA STRUCTURE

The Schema Element

<xs: schema xmlns: xs="http://www.w3.org/2001/XMLSchema">

Element definitions

As we saw in the chapter XML - Elements, elements are the building blocks of XML document. An element can be defined within an XSD as follows:
<xs:element name="x" type="y"/>

Data types:

These can be used to specify the type of data stored in an Element.

- String (xs:string)
- Date (xs:date or xs:time)
- Numeric (xs:integer or xs:decimal)
- Boolean (xs:boolean)

EX: Sample.xsd

```
<?xml version=‖1.0‖ encoading=‖UTF-8‖?>
<xs:schema xmlns:xs=http://www.w3.org/XMLSchema>

  <xs:element name="sname‖ type=‖xs:string"/>

/* <xs:element name="dob” type=”xs:date"/>

    <xs:element name="dobtime” type=”xs:time"/>

    <xs:element name="marks” type=”xs:integer"/>

    <xs:element name="avg” type=”xs:decimal"/>

    <xs:element name="flag” type=”xs:boolean"/> */

</xs:schema>
```

Sample.xml:

```
<?xml version=‖1.0‖ encoading=‖UTF-8‖?>

<sname xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="sample.xsd">
```

Kalpana *\/*yyyy-mm-dd 23:14:34 600 92.5 true/false */*

</sname>

Definition Types

You can define XML schema elements in following ways:

Simple Type - Simple type element is used only in the context of the text. Some of predefined simple types are: xs:integer, xs:boolean, xs:string, xs:date. For example:
<xs:element name="phone_number" type="xs:int" />
<phone>9876543210</phone>

Default and Fixed Values for Simple Elements

In the following example the default value is "red":

```
<xs:element name="color" type="xs:string" default="red"/>
```

In the following example the fixed value is "red":

```
<xs:element name="color" type="xs:string" fixed="red"/>
```

Complex Type - A complex type is a container for other element definitions. This allows you to specify which child elements an element can contain and to provide some structure within your XML documents. For example:

```
<xs:element name="Address">
<xs:complexType>
<xs:sequence>
<xs:element name="name" type="xs:string" />
<xs:element name="company" type="xs:string" />
 <xs:element name="phone" type="xs:int" />
</xs:sequence>
</xs:complexType>
</xs:element>
```

In the above example, Address element consists of child elements. This is a container for other <xs:element> definitions, that allows to build a simple hierarchy of elements in the XML document.

Global Types - With global type, you can define a single type in your document, which can be used by **all other references**. For example, suppose you want to generalize the person and company for different addresses of the company. In such case, you can define a general type as below:

```
<xs:element name="AddressType">

<xs:complexType>
<xs:sequence>
<xs:element name="name" type="xs:string" />
<xs:element name="company" type="xs:string" />
</xs:sequence>
</xs:complexType>
</xs:element>
```

Now let us use this type in our example as below:

```
<xs:element name="Address1">
<xs:complexType>
<xs:sequence>
<xs:element name="address" type="AddressType " />

   <xs:element name="phone1" type="xs:int" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="Address2">
<xs:complexType>
<xs:sequence>
<xs:element name="address" type="AddressType " />

   <xs:element name="phone2" type="xs:int" />
    </xs:sequence>     </xs:complexType> </xs:element>
```

Instead of having to define the name and the company twice (once for Address1 and once for Address2), we now have a single definition. This makes maintenance simpler, i.e., if you decide to add "Postcode" elements to the address, you need to add them at just one place.

Attributes

Simple elements cannot have attributes. If an element has attributes, it is considered to be of a complex type. But the attribute itself is always declared as a simple type. Attributes in XSD provide extra information within an element. Attributes have name and type property as shown below:

```
        <xs:attribute name="x" type="y"/>
```

Ex: <lastname lang="EN ">Smith</lastname>

```
    <xs:attribute name="lang" type="xs:string"/>
```

Default and Fixed Values for Attributes

```
<xs:attribute name="lang" type="xs:string" default="EN"/>
<xs:attribute name="lang" type="xs:string" fixed="EN"/>
```

Optional and Required Attributes

Attributes are optional by default. To specify that the attribute is required, use the "use" attribute:

```
<xs:attribute name="lang" type="xs:string" use="required"/>
```

Restrictions on Content

When an XML element or attribute has a data type defined, it puts restrictions on the element's or attribute's content. If an XML element is of type "xs:date" and contains a string like "Hello World", the element will not validate.

Restrictions on Values:

The value of **age cannot be lower than 0 or greater than 120:**

```
<xs:element name="age">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="120"/>
    </xs:restriction>
  </xs:simpleType> </xs:element>
```

Restrictions on a Set of Values

The example below defines an element called "car" with a restriction. The only acceptable values are: Audi, Golf, BMW:

```
<xs:element name="car">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="Audi"/>
      <xs:enumeration value="Golf"/>
      <xs:enumeration value="BMW"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Restrictions on Length

To limit the length of a value in an element, we would use the length, maxLength, and minLength constraints. **The value must be exactly eight characters:**

```
<xs:element name="password">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:length value="8"/>    [<xs:minLength value="5"/>     <xs:maxLength value="8"/>]
    </xs:restriction> </xs:simpleType> </xs:element>
```

XSD Indicators

We can control HOW elements are to be used in documents with indicators.

Indicators: There are seven indicators

Order indicators:

- All
- Choice
- Sequence

Occurrence indicators:

- maxOccurs
- minOccurs

Group indicators:

- Group name
- attributeGroup name

❼ Order Indicators

Order indicators are used to define the order of the elements.

All Indicator

The <all> indicator specifies that the child elements can appear in any order, and that each child element must occur only once:

```
<xs:element name="person">
  <xs:complexType>
    <xs:all>
     <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:all>
  </xs:complexType>
</xs:element>
```

Note: When using the <all> indicator you can set the <minOccurs> indicator to 0 or 1 and the <maxOccurs> indicator can only be set to 1 (the <minOccurs> and <maxOccurs> are described later).

Choice Indicator

The <choice> indicator specifies that either one child element or another can occur:

```
<xs:element name="person">
  <xs:complexType>
    <xs:choice>
<xs:element name="employee" type="employee"/>
      <xs:element name="member" type="member"/>
     </xs:choice>  </xs:complexType>   </xs:element>
```

Sequence Indicator

The <sequence> indicator specifies that the child elements must appear in a specific order:

```
<xs:element name="person">
   <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:sequence> </xs:complexType> </xs:element>
```

❼ Occurrence Indicators

Occurrence indicators are used to define how often an element can occur.

Note: For all "Order" and "Group" indicators (any, all, choice, sequence, group name, and group reference) **the default value for maxOccurs and minOccurs is 1**.

maxOccurs Indicator

The <maxOccurs> indicator specifies the maximum number of times an element can occur:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
 <xs:element name="full_name" type="xs:string"/>
  <xs:element name="child_name" type="xs:string" maxOccurs="10"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

minOccurs Indicator

The <minOccurs> indicator specifies the minimum number of times an element can occur:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
 <xs:element name="full_name" type="xs:string"/>
  <xs:element name="child_name" type="xs:string" maxOccurs="10" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Tip: To allow an element to appear an unlimited number of times, use the **maxOccurs="unbounded"** statement:

EX: An XML file called **"Myfamily.xml":**

```
<?xml version="1.0" encoding="UTF-8"?>
<persons xmlns:xsi="http://www.w3.org/2001/XMLSchema- instance"
xsi:noNamespaceSchemaLocation= "family.xsd">
<person>
  <full_name>KALPANA</full_name>
  <child_name>mrcet</child_name>
</person>
<person>
  <full_name>Tove Refsnes</full_name>
  <child_name>Hege</child_name>
  <child_name>Stale</child_name>
  <child_name>Jim</child_name>
  <child_name>Borge</child_name>
</person>
<person>
  <full_name>Stale Refsnes</full_name>
</person>
 </persons>
```

The XML file above contains a root element named "persons". Inside this root element we have defined three "person" elements. Each "person" element must contain a "full_name" element and it can contain up to five "child_name" elements.
Here is the schema file **"family.xsd":**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema
elementFormDefault="qualified">
<xs:element name="persons">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="person" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="full_name" type="xs:string"/>
            <xs:element name="child_name" type="xs:string" minOccurs="0" maxOccurs="5"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

❼ **Group Indicators:** Group indicators are used to define related sets of elements.

Element Groups
Element groups are defined with the group declaration, like this:

```
<xs:group name="groupname">
...
</xs:group>
```

You must define an all, choice, or sequence element inside the group declaration. The following example defines a group named "persongroup", that defines a group of elements that must occur in an exact sequence:

```
<xs:group name="persongroup">

  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
    <xs:element name="birthday" type="xs:date"/>
  </xs:sequence>
</xs:group>
```

After you have defined a group, **you can reference it in another definition**, like this:

```
<xs:element name="person" type="personinfo"/>
<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:group ref="persongroup"/>

    <xs:element name="country" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Attribute Groups

Attribute groups are defined with the attributeGroup declaration, like this:

```
<xs:attributeGroup name="groupname">
...
</xs:attributeGroup>
```

The following example defines an attribute group named "personattrgroup":

```
<xs:attributeGroup name="personattrgroup">
  <xs:attribute name="firstname" type="xs:string"/>
  <xs:attribute name="lastname" type="xs:string"/>
  <xs:attribute name="birthday" type="xs:date"/>
</xs:attributeGroup>
```

After you have defined an attribute group, you can reference it in another definition, like this:

```
<xs:element name="person">
  <xs:complexType>
    <xs:attributeGroup ref="personattrgroup"/> </xs:complexType> </xs:element>
```

Example Program: "shiporder.xml"

```
<?xml version="1.0" encoding="UTF-8"?>
<shiporder                              orderid="889923"
xmlns:xsi=http://www.w3.org/2001/XMLSchema- instance
xsi:noNamespaceSchemaLocation="shiporder.xsd">
  <orderperson>John Smith</orderperson>
<shipto>
    <name>Ola Nordmann</name>
    <address>Langgt 23</address>
    <city>4000 Stavanger</city>
    <country>Norway</country>
  </shipto>
<item>
    <title>Empire Burlesque</title>
    <note>Special Edition</note>
    <quantity>1</quantity>
    <price>10.90</price>
  </item>
<item>
    <title>Hide your heart</title>     <quantity>1</quantity>
    <price>9.90</price> </item>
</shiporder>
```

Create an XML Schema "shiporder.xsd":

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="shiporder">

  <xs:complexType>
    <xs:sequence>
      <xs:element name="orderperson" type="xs:string"/>

      <xs:element name="s hipto">

        <xs:complexType>
          <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="address" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
            <xs:element name="country" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="item" maxOccurs="unbounded">

        <xs:complexType>
          <xs:sequence>
            <xs:element name="title" type="xs:string"/>
            <xs:element name="note" type="xs:string" minOccurs="0"/>

            <xs:element name="quantity" type="xs:positiveInteger"/>
            <xs:element name="price" type="xs:decimal"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="orderid" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
</xs:schema>
```

XML DTD vs XML Schema

The schema has more advantages over DTD. A DTD can have two types of data in it, namely the CDATA and the PCDATA. The CDATA is not parsed by the parser whereas the PCDATA is parsed. In a schema you can have primitive data types and custom data types like you have used in programming.

Schema vs. DTD

- XML Schemas are extensible to future additions
- XML Schemas are richerand more powerful than DTDs
- XMLSchemas are written in XML
- XMLSchemas support data types
- XMLSchemas support namespaces

XML Parsers

An XML parser converts an XML document into an XML DOM object - which can then be manipulated with a JavaScript.

Two types of XML parsers:

- **Validating Parser**
  - It requires document type declaration
  - It generates error if document does not
    - Conform with DTD and
    - Meet XML validity constraints
- **Non-validating** Parser
  - It checks well- formedness for xml document
  - It can ignore external DTD

What is XML Parser?

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.

Types of parsers:

- **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.
- **JDOM Parser -** Parses the document in similar fashion to DOM parser but in more easier way.
- **StAX Parser -** Parses the document in similar fashion to SAX parser but in more efficient way.
- **XPath Parser** - Parses the XML based on expression and is used extensively in conjuction with XSLT.
- **DOM4J Parser** - A java library to parse XML, XPath and XSLT using Java Collections Framework , provides support for DOM, SAX and JAXP.

DOM-Document Object Model

The Document Object Model protocol converts an XML document into a collection of objects in your program. XML documents have a hierarchy of informational units called nodes; this hierarchy allows a developer to navigate through the tree looking for specific information. Because it is based on a hierarchy of information, the DOM is said to be tree based. DOM is a way of describing those nodes and the relationships between them.

You can then manipulate the object model in any way that makes sense. This mechanism is also known as the "random access" protocol, because you can visit any part of the data at any time. You can then modify the data, remove it, or insert new data.

The XML DOM, on the other hand, also provides an API that allows a developer to add, edit, move, or remove nodes in the tree at any point in order to create an application. A DOM parser creates a tree structure in memory from the input document and then waits for requests from client. A DOM parser always serves the client application with the **entire document no matter how much is actually needed** by the client. With DOM parser, method calls in client application have to be explicit and forms a kind of chained method calls.

Document Object Model is for defining the standard for accessing and manipulating XML documents. **XML DOM** is used for

- **Loading the xml document**
- **Accessing the xml document**
- **Deleting the elements of xml document**
- **Changing the elements of xml document**

According to the DOM, everything in an XML document is a node. It considers

- 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

The W3C DOM specification is divided into three major parts:

DOM Core- This portion defines the basic set of interfaces and objects for any structured documents.
XML DOM- This part specifies the standard set of objects and interfaces for XML documents only.
HTML DOM- This part specifies the objects and interfaces for HTML documents only.

DOM Levels

- Level 1 Core:W3C Recommendation, October 1998

✓ It has feature for primitive navigation and manipulation of XML trees
✓ other Level 1 features are: All HTML features

- Level 2 Core:W3C Recommendation, November 2000

✓ It adds Namespace support and minor new features
✓ other Level 2 features are: Events, Views, Style, Traversaland Range

- Level 3 Core: W3C Working Draft, April 2002

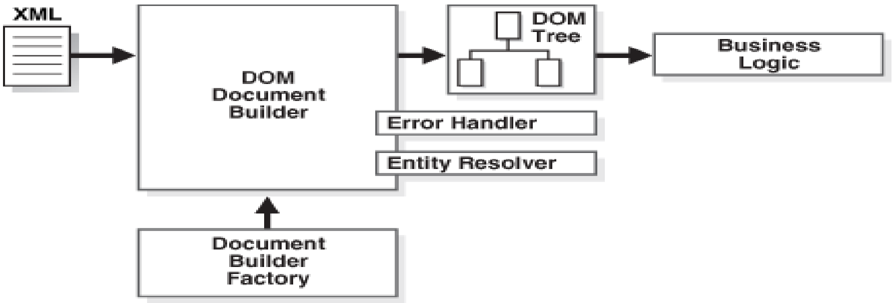
✓ It supports: Schemas, XPath, XSL, XSLT

We can access and parse the XML document in two ways:

➢ **Parsingusing DOM (tree based)**
➢ **Parsing using SAX (Event based)**

Parsing the XML doc. using DOM methods and properties are called as **tree based approach** whereas using SAX (Simple Api for Xml) methods and properties are called as **event based approach.**

Steps to Using DOM Parser
Let‘s note down some broad steps involved in using a DOM parser for parsing any XML file in java.



DOM based XML Parsing:(tree based)

JAXP is a tool, stands for Java Api for Xml Processing, used for accessing and manipulating xml document in a tree based manner.

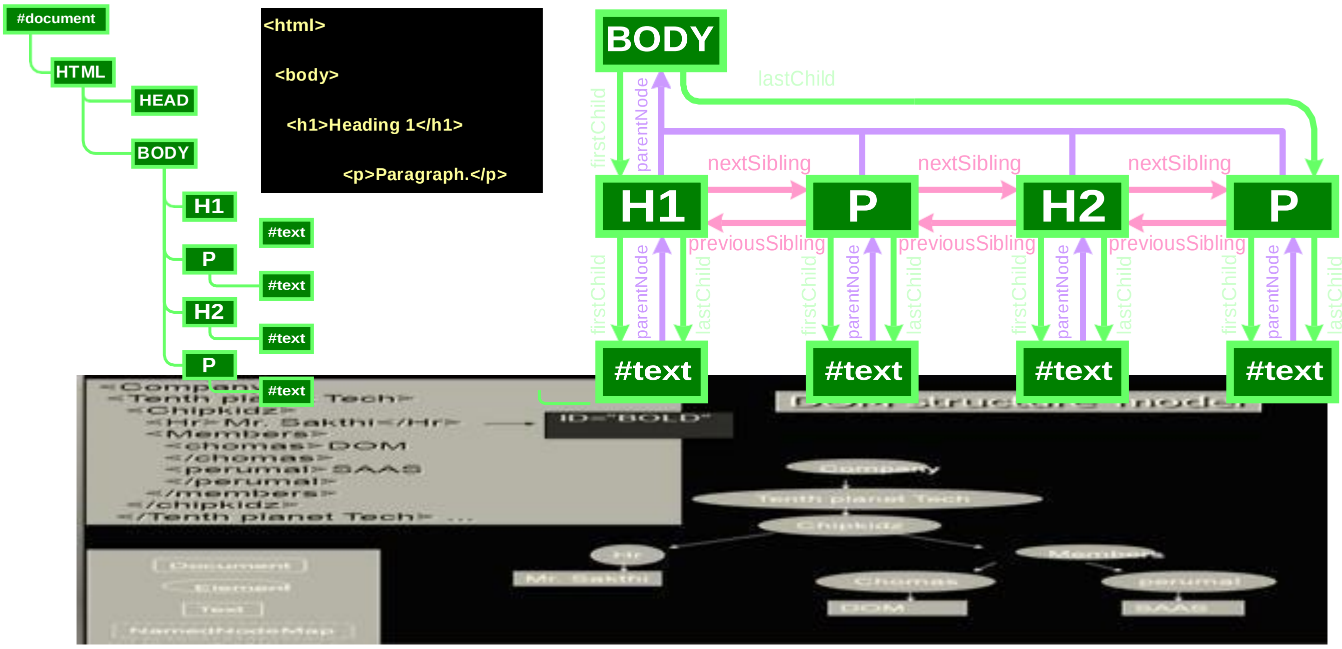
The following DOM javaClasses are necessary to process the XML document:

- DocumentBuilderFactory class creates the instance of DocumentBuilder.
- DocumentBuilder produces a Document (a DOM) that conforms to the DOM specification.

The following methods and properties are necessary to process the XML document:

Property	**Meaning**
nodeName	Finding the name of the node
nodeValue	Obtaining value of the node
parentNode	To get parnet node
childNodes	Obtain child nodes
Attributes	For getting the attributes values

Method	**Meaning**
getElementByTagName(name)	To access the element by specifying its name
appendChild(node)	To insert a child node
removeChild(node)	To remove existing child node



```
<html>
  <body>
    <h1>Heading 1</h1>
          <p>Paragraph.</p>
```

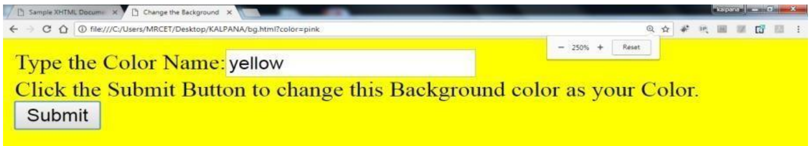
DOM Document Object

✓ **There are 12 types of nodes in a DOM *Document* object**

1. Document node	*7. EntityReference node*
2. Element node	*8. Entity node*
3. Text node	*9. Comment node*
4. Attribute node	*10. DocumentType node*
5. Processing instruction node	*11. DocumentFragment node*
6. CDATA Section node	*12. Notation node*

Examples for Document method

```
<html>
  <head>
  <title>Change the Background</title>
  </head>
 <body>
  <script language = "JavaScript">
      function background()
 {        var       color       =       document.bg.color.value;
       document.body.style.backgroundColor=color;          }
  </script>
  <form name="bg">
  Type the Color Name:<input type="text" name="color" size="20">
<br>
  Click the Submit Button to change this Background color as your Color.
<br>
  <input type="button" value="Submit" onClick='background()'>
  </form>
 </body>
  </html>
```



file:///C:/Users/MRCET/Desktop/KALPANA/bg.html?color=pink

Type the Color Name: yellow

Click the Submit Button to change this Background color as your Color.

Submit

DOM NODE Methods

Method Name	Description
appendChild	Appends a child node.
cloneNode	Duplicates the node.
getAttributes	Returns the node‘s attributes.
getChildNodes	Returns the node‘s child nodes.
getNodeName	Returns the node‘s name.
getNodeType	Returns the node‘s type (e.g., element, attribute, text, etc.).
getNodeValue	Returns the node‘s value.
getParentNode	Returns the node‘s parent.
hasChildNodes	Returns **true** if the node has child nodes.
removeChild	Removes a child node from the node.
replaceChild	Replaces a child node with another node.
setNodeValue	Sets the node‘s value.
insertBefore	Appends a child node in front of a child node.

DOM Advantages & Disadvantages

ADVANTAGES

- Robust API for the DOM tree
- Relatively simple to modify the data structure and extract data
- It is good when random access to widely separated parts of a document isrequired
- It supports both read and write operations
-

Disadvantages

- Stores the entire document in memory, It is memory inefficient
- As Dom was written for any language, method naming conventions don‘t follow standard java programming conventions

DOM or SAX

DOM

- Suitable for small documents
- Easily modify document
- Memory intensive;Load the complete XML document

SAX

- Suitable for large documents; saves significant amounts of memory
- Only traverse document once, start to end
- Event driven
- Limited standard functions.
-

Loading an XML file:one.html

```
<html> <body>
<script type=‖text/javascript‖> try
{
xmlDocument=new ActiveXObject(–Microsoft.XMLDOM‖);
}
catch(e)
{
try                                                              {
xmlDocument=document.implementation.createDocument("","",null);
}
catch(e){alert(e.message)}
}
try
{
xmlDocument.async=false;
xmlDocument.load(–faculty.xml‖);
document.write(–XML document student is loaded‖);
}
catch(e){alert(e.message)}
</script>
</body> </html>
```

faculty.xml:

```
<?xml version=‖1.0‖?>
< faculty >
  <eno>30</eno>
<personal_inf>
  <name>Kalpana</name>
  <address>Hyd</address>
  <phone>9959967192</phone>
</personal_inf>
<dept>CSE</dept>
<col>MRCET</col>
<group>MRGI</group>
</faculty>
```

OUTPUT: XML document student is loaded

ActiveXObject: It creates empty xml document object.

Use separate function for Loading an XML document: two.html

```
<html> <head>
<script         type=‖text/javascript‖>
Function My_function(doc_file)
{
try
{
xmlDocument=new ActiveXObject(–Microsoft.XMLDOM‖);
}
catch(e)
{
try
{
xmlDocument=document.implementation.createDocument("","",null);
}
catch(e){alert(e.message)}
}
try
{
xmlDocument.async=false;
xmlDocument.load(–faculty.xml‖);
return(xmlDocument);
}
catch(e){alert(e.message)}
return(null);
}
</script>
</head>
<body>
<script              type=‖text/javascript‖>
xmlDoc=‖My_function(–faculty.xml‖);
document.write(–XML document student is loaded‖);
</script>
</body> </html>
```

OUTPUT: XML document student is loaded

Use of properties and methods: three.html

```
<html> <head>
<script type=‖text/javascript‖ src=‖my_function_file.js‖></script>
</head> <body>
<script type=‖text/javascript‖>
xmlDocument=My_function("faculty.xml");
document.write(–XML document faculty is loaded and content of this file is:‖);
document.write(–<br>‖);
document.write(–ENO:‖+
xmlDocument.getElementsByTagName(–eno‖)[0].childNodes[0].nodeValue);
document.write(–<br>‖);
document.write(–Name:‖+
xmlDocument.getElementsByTagName(–name‖)[0].childNodes[0].nodeValue);
```

```
document.write(–<br>‖);
document.write(–ADDRESS:‖+
xmlDocument.getElementsByTagName(–address‖)[0].childNodes[0].nodeValue);
document.write(–<br>‖);
document.write(–PHONE:‖+
xmlDocument.getElementsByTagName(–phone‖)[0].childNodes[0].nodeValue);
document.write(–<br>‖);
document.write(–DEPARTMENT:‖+
xmlDocument.getElementsByTagName(–dept‖)[0].childNodes[0].nodeValue);
document.write(–<br>‖);
document.write(–COLLEGE:‖+
xmlDocument.getElementsByTagName(–col‖)[0].childNodes[0].nodeValue);
document.write(–<br>‖);
document.write(–GROUP:‖+
xmlDocument.getElementsByTagName(–group‖)[0].childNodes[0].nodeValue);
</script>
</body>
</html>
```

OUTPUT:

XML document faculty is loaded and content of this file is

```
ENO: 30
NAME: Kalpana
ADDRESS: Hyd
PHONE: 9959967192
DEPARTMENT: CSE
COLLEGE: MRCET
GROUP: MRGI
```

<u>We can access any XML element using the index value: four.html</u>

```
<html> <head>
<script type=‖text/javascript‖ src=‖my_function_file.js‖></script>
</head> <body>
<script                                type=‖text/javascript‖>
xmlDoc=My_function("faculty1.xml");
value=xmlDoc.        getElementsByTagName(-name‖);
document.write(-value[0].childNodes[0].nodeValue‖);

</script></body></html>
```

OUTPUT: Kalpana

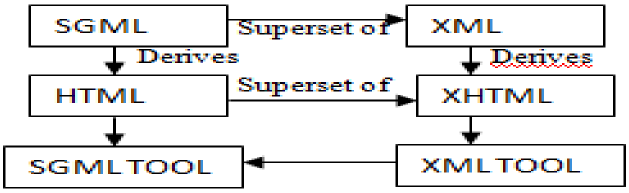
XHTML: eXtensible Hypertext Markup Language

Hypertext is simply a piece of text that works as a link. **Markup language** is a language of writing layout information within documents. The XHTML recommended by **W3C**. Basically an XHTML document is a plain text file and it is very much similar to HTML. It contains rich text, means text with tags. The extension to this program should b either **html or htm**. These programs can be opened in some web browsers and the corresponding web page can beviewed.

HTML Vs XHTML

HTML	XHTML
1. **The HTML tags are case insensitive.** **EX: <BoDy>-------- </body>**	**1.** The XHTML tags are casesensitive. EX: <body> ------- </body>
2. **We can omit the closing tags sometimes.**	**2.** For every tag there must be a closing tag. EX: <h1>---------</h1> or <h1------------ />
3. **The attribute values not always necessary to quote.**	**3.** The attribute values are must be quoted.
4. **In HTML there are some implicit attribute values.**	**4.** In XHTML the attribute values must be specified explicitly.
5. **In HTML even if we do not follow the nesting rules strictly it does not cause much difference.**	**5.** In XHTML the nesting rules must be strictly followed. These nesting rules are- - A form element cannot contain another form element. -an anchor element does not contain another form element -List element cannot be nested in the list element -If there are two nested elements then the inner element must be enclosed first before closing the outer element -Text element cannot be directly nested in form element

The relationship between SGML, XML, HTML and XHTML is as given below



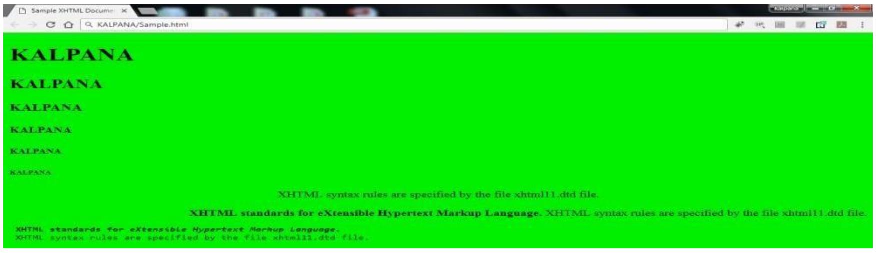
Standard structure: DOCTYPE, html, head and body

The doctype is specified by the DTD. The XHTML syntax rules are specified by the file xhtml11.dtd file.There are 3 types of DTDs.

1. **XHTML 1.0 Strict:** clean markup code
2. **XHTML 1.0 Transitional:** Use some html features in the existing XHTML document.
3. **XHTML 1.0 Frameset:** Use of Frames in an XHTML document.

EX:

```
<!DOCTYPE html PUBLIC"-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml11.dtd">

<html xmlns="http://www.w3c.org/1999/xhtml">

<head>
<title>Sample XHTML Document</title>
</head>
<body bgcolor=‖#FF0000‖>
<basefont face=‖arial‖ size=‖14‖ color=‖white‖>
```



KALPANA
KALPANA
KALPANA
KALPANA
KALPANA
KALPANA

XHTML syntax rules are specified by the file xhtml11.dtd file.

XHTML standards for eXtensible Hypertext Markup Language. XHTML syntax rules are specified by the file xhtml11.dtd file.

```
XHTML standards for eXtensible Hypertext Markup Language.
XHTML syntax rules are specified by the file xhtml11.dtd file.
```

```
<h1>MIST</h1>
<h2> MIST </h2>
<h3> MIST </h3>
<h4> KALPANA </h4>
<h5> KALPANA </h5>
<h6>KALPANA</h6>
<p><center> XHTML syntax rules are specified by the file xhtml11.dtd file. </center></p>
<div align="right"> <b>XHTML standards for eXtensible Hypertext Markup Language.</b>
 XHTML syntax rules are specified by the file xhtml11.dtd file.</div>
<pre> <b>XHTML standards for <i>eXtensible Hypertext Markup Language.</i></b>
 XHTML syntax rules are specified by the file xhtml11.dtd file.</pre>
</basefont>
</body>
</html>
```

DOM in JAVA

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 the 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 Using DOM

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

- Import XML-related packages.
- Create a DocumentBuilder
- Create a Document from a file or stream
- Extract the root element
- Examine attributes
- Examine sub-elements

DOM

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
public class parsing_DOMDemo
{
public static void main(String[] args)
{
try
{
System.out.println(–enter the name of XML document‖);
BufferedReader input=new Bufferedreader(new InputStreamReader(System.in));
String file_name=input.readLine();
File fp=new File(file_name);
if(fp.exists())
{
try
{
DocumentBuilderFactory Factory_obj= DocumentBuilderFactory.newInstance();
DocumentBuilder builder=Factory_obj.newDocumentBuilder();
InputSource ip_src=new InputSource(file_name);
Document doc=builder.parse(ip_src);
System.out.println(–file_name+‖is well- formed.‖);
}
catch (Exception e)
{
System.out.println(file_name+‖is not well- formed.‖);
System.exit(1);
} }
else
 {
System.out.println(–file not found:‖+file_name);
} }
catch(IOException ex)
{
ex.printStackTrace();
}
} }
```

SAX:

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 ROOTelement.

- 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

When to use?

You should use a SAX parser when:

- You can process the XML document in a linear fashion from the top down
- The document is not deeply nested
- You are processing a very large XML document whose DOM tree would consume too much memory.Typical DOM implementations use ten bytes of memory to represent one byte of XML
- The problem to be solved involves only part of the XML document
- Data is available as soon as it is seen by the parser, so SAX works well for an XML document that arrives over a stream

Disadvantages ofSAX

- We have no random access to an XML document since it is processed in a forward-only manner
- If you need to keep track of data the parser has seen or change the order of items, you must write the code and store the data on your own
- The data is broken into pieces and clients never have all the information as a whole unless they create their own data structure

The kinds of events are:

- The start of the document is encountered
- The end of the document is encountered
- The start tag of an element is encountered
- The end tag of an element is encountered
- Character data is encountered
- A processing instruction is encountered

Scanning the XML file from start to end, each event invokes a corresponding callback method that the programmer writes.

SAX packages

javax.xml.parsers: Describing the main classes needed for parsing

org.xml.sax: Describing few interfaces for parsing

SAX classes

- **SAXParser** Defines the API that wraps an XMLReader implementation class
- **SAXParserFactory** Defines a factory API that enables applications to configure and obtain a SAX based parser to parse XML documents
- **ContentHandler** Receive notification of the logical content of a document.

- **DTDHandler** Receive notification of basic DTD-related events.
- **EntityResolver** Basic interface for resolving entities.
- **ErrorHandler** Basic interface for SAX error handlers.
- **DefaultHandler** Default base class for SAX event handlers.

SAX parser methods

StartDocument() and endDocument() – methods called at the start and end of an XML document.

StartElement() and endElement() – methods called at the start and end of a document element.

Characters() – method called with the text contents in between the start and end tags of an XML document element.

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, etc.

SAX simple API for XML

```
import java.io.*;
import org.xml.sax;
import org.xml.sax.helpers;
public class parsing_SAXDemo
{
public static void main(String[] args) throws IOException
{
```

```
try {
System.out.println(–enter the name of XML document‖);
BufferedReader input=new Bufferedreader(new InputStreamReader(System.in));
String file_name=input.readLine();
File fp=new File(file_name);
if(fp.exists())
{
try
{
XMLReader reader=XMLReaderFactory.createXMLReader();
reader.parse(file_name);
System.out.println(–file_name+‖is well- formed.‖);
}
catch (Exception e)
{
System.out.println(file_name+‖is not well- formed.‖);
System.exit(1);
}
}
else
{
System.out.println(–file not found:‖+file_name);
}
}
```

Differences between DOM and SAX

DOM	SAX
Stores the entire XML document into memory before processing	Parses node by node
Occupies more memory	Doesn‘t store the XML in memory
We can insert or delete nodes	We can‘t insert or delete a node
DOM is a tree model parser	SAX is an event based parser
Document Object Model (DOM) API	SAX is a Simple API for XML
Preserves comments	Doesn‘t preserve comments
DOM is slower than SAX, heavy weight.	SAX generally runs a little faster than DOM, light weight.
Traverse in any direction.	Top to bottom traversing is done in this approach
Random Access	Serial Access

Packages required to import
import java x.xml.parsers.*;
import java x.xml.parsers.Docu mentBuilder;
import java x.xml.parsers.Docu mentBuilderFactory;
	Packages required to import
	import java.xml.parsers.*;
	import org.xml.sax.*;
	import org.xml.sax.helpers;

UNIT-2 HTML

- **HTML** - a *markup* language
    - To describe the general form and layout of documents
        - HTML is **not** a programming language - it cannot be used describe **computations**.
    - An HTML document is a mix of **content** and **controls**
        - Controls are **tags** and their **attributes**
            - Tags often delimit content and specify something about how the content should be arranged in the document
              For example, <p>Write a paragraph here </p> is an *element*.
            - Attributes provide additional information about the content of atag
              For example, <img src = "redhead.jpg"/> <font color ="Red" />
- Plug ins
    - Integrated into tools like word processors, effectively converting them to WYSIWYG HTML editors
- Filters
    - Convert documents in other formats to HTM

Advantages of both filters and plug- ins:

- Existing documents produced with other tools can be converted to HTML documents
- Use a tool you already know to produce HTML

- Disadvantages of both filters and plug-ins:
    - HTML output of both is not perfect - must be fine tuned
    - HTML may be non-standard
    - You have two versions of the document, which are difficult to synchronize
- XML
    - A meta-markup language (a language for defining markup language)
    - Used to create a new markup language for a particular purpose or area
    - Because the tags are designed for a specific area, they can be meaningful
- JavaScript
    - A client-side HTML-embedded scripting language
    - Provides a way to access elements of HTML documents and dynamicallychange them
- Flash
    - A system for building and displaying text, graphics, sound, interactivity, and animation (movies)
    - Two parts:
        1. Authoring environment
        2. Player

Supports both motion and shape animation
PHP
A server-side scripting language
Great for form processing and database access through the Web

No new technologies or languages

Much faster for Web applications that have extensive user/server interactions

Uses asynchronous requests to the server

Requests and receives small parts of documents, resulting in much faster responses

Java Web Software

Servlets – server-side Java classes

JavaServer Pages (JSP) – a Java-based approach to server-side scripting

JavaServer Faces – adds an event-driven interface model on JSP

ASP.NET

Does what JSP and JSF do, but in the .NET environment

Allows.N ET languages to be used as server-side scripting language

Ruby

A pure object-oriented interpreted scripting language

Every data value is an object, and all operations are via method calls

Both classes and objects are dynamic

Rails

A development framework for Web-based applications

Particularly useful for Web applications that access databases

Written in Ruby and uses Ruby as its primary user language

HTML Common tags:-

HTML is the building block for web pages. HTML is a format that tells a computer how to display a web page. The documents themselves are plain text files with special "tags" or codes that a web browser uses to interpret and display information on your computer screen.

- HTML stands for Hyper Text Markup Language
- An HTML file is a text file containing small markup tags
- The markup tags tell the Web browser how to display the page
- An HTML file must have an htm or html file extension.

HTML Tags:- HTML tags are used to mark-up HTML elements .HTML tags are surrounded by the two characters < and >. The surrounding characters are called angle brackets. HTML tags normally come in pairs like **and** The first tag in a pair is the start tag, the second tag is the end tag . The text between the start and end tags is the element content . HTML tags are not case sensitive, <B>**means the same as<b>.**

The most important tags in HTML are tags that define headings, paragraphs and line breaks.

Tag	**Description**
<!DOCTYPE...>	This tag defines the document type and HTML version.
<html>	This tag encloses the complete HTML document and mainly comprises of document header which is represented by <head>...</head> and document body which is represented by <body>...</body> tags.
<head>	This tag represents the document's header which can keep other HTML tags like <title>, <link> etc.
<title>	The <title> tag is used inside the <head> tag to mention the document title.

<body>	This tag represents the document's body which keeps other HTML tags like <h1>, <div>, <p> etc.
<p>	This tag represents a paragraph.
<h1> to <h6>	Defines header 1 to header 6
<br>	Inserts a single line break
<hr>	Defines a horizontal rule
<!-->	Defines a comment

Headings:-

Headings are defined with the <h1> to <h6> tags. <h1> defines the largest heading while <h6> defines the smallest.

```
<h1>This is a heading</h1>
<h2>This is a heading</h2>
<h3>This is a heading</h3>
<h4>This is a heading</h4>
<h5>This is a heading</h5>
<h6>This is a heading</h6>
```

Paragraphs:-

Paragraphs are defined with the <p> tag. Think of a paragraph as a block of text. You can use the align attribute with a paragraph tag as well.

```
<p align="left">This is a paragraph</p>
<p align="center">this is another paragraph</p>
```

Note: You must indicate paragraphs with <p> elements. A browser ignores any indentations or blank lines in the source text. Without <p> elements, the document becomes

Line Breaks:-

The <br> tag is used when you want to start a new line, but don't want to start a new paragraph. The <br> tag forces a line break wherever you place it. It is similar to single spacing in a document.

This Code	output
<p>This <br> is a para<br> graph with line breaks</p>	This<br>is a para<br>graph with line breaks

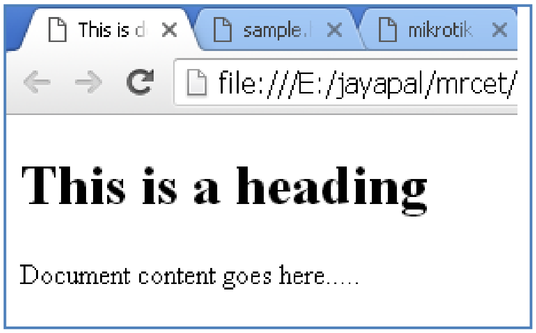
Horizontal Rule The element is used for horizontal rules that act as dividers between sections like this:

The horizontal rule does not have a closing tag. It takes attributes such as align and width

Code	Output
<hr width="50%" align="center">	

Sample html program

```
<!DOCTYPE html>
<html>
        <head>
                <title>This is document title
                </title>

  </head>
  <body>
<h1>This is a heading</h1>
<p>Document content goes here ....</p>
</body>

</html>
```



This is a heading

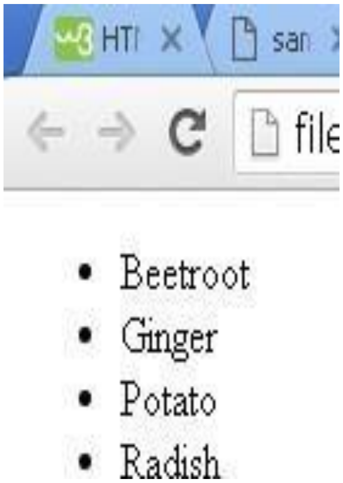
Document content goes here.....

Lists:-HTML offers web authors three ways for specifying lists of information. All lists must contain one or more list elements. Lists are of three types

1) Un ordered list
2)Ordered List
3)Definition list

HTML Unordered Lists: An unordered list is a collection of related items that have no special order or sequence. This list is created by using HTML <ul> tag. Each item in the list is marked with a bullet.

```
<!DOCTYPE html>
  <html>
            <head>
                      <title>HTML Unordered List</title>
            </head>
            <body>
                       <ul>     <li>Beetroot</li>
                                 <li>Ginger</li> <li>Potato</li>
                                 <li>Radish</li>

                      </ul>

            </body>

   </html>
```

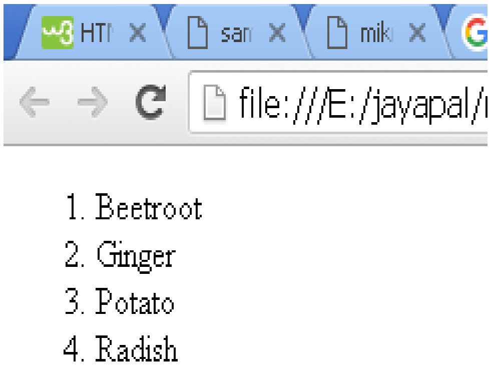


- Beetroot
- Ginger
- Potato
- Radish

Example

HTML Ordered Lists:- ite ms are numbered list instead of bulleted, This list is created by using **<ol>** tag.

```
<!DOCTYPE html>

<html>

        <head>

                <title>HTML Ordered List</title>

        </head>

        <body>

                <ol>

                        <li>Beetroot</li>
```

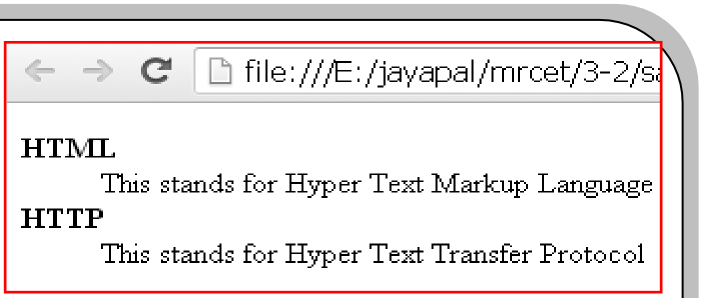


1. Beetroot
2. Ginger
3. Potato
4. Radish

HTML Definition Lists:- HTML and XHTML supports a list style which is called definition lists where entries are listed like in a dictionary or encyclopedia. The definition list is the ideal way to present a glossary, list of terms, or other name/value list. Definition List makes use of following three tags.

1). <dl> - Defines the start of the list
2). <dt> - A term
3). <dd> - Term definition
4). </dl> - Defines the end of the list

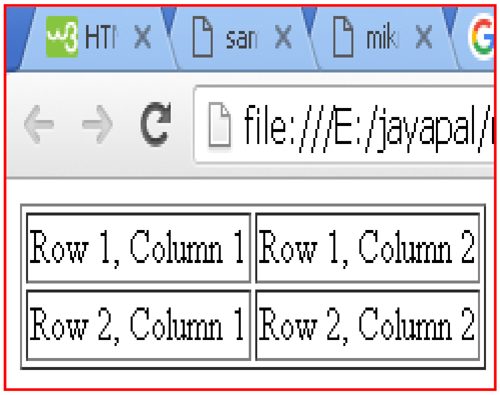
```
<!DOCTYPE html>

<html>

   <head>

        <title>HTML Definition List</title>

  </head>

  <body>

   <dl>
```



HTML
    This stands for Hyper Text Markup Language
HTTP
    This stands for Hyper Text Transfer Protocol

HTML tables:

The HTML tables allow web authors to arrange data like text, images, links, other tables, etc. into rows and columns of cells. The HTML tables are created using the <**table**> tag in which the **<tr>** tag is used to create table rows and **<td>** tag is used to create data cells. Exampl</head>



Row 1, Column 1	Row 1, Column 2
Row 2, Column 1	Row 2, Column 2

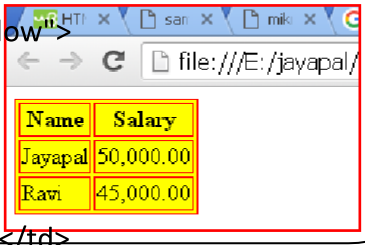
```
<!DOCTYPE html>

 <html>

 <head>

 <title>HTML Tables</title>

 <body>

        <table border="1">

                <tr>

                        <td>Row 1, Column 1</td> <td>Row 1, Column 2</td>

                </tr>
```

Table Heading: Table heading can be defined using **<th>** tag. This tag will be put to replace

<td> tag, which is used to represent actual data cell. Normally you will put your top row as table heading as shown below, otherwise you can use <th> element in any row.

Tables Backgrounds: set table background using one of the following two ways:
1)bgcolor attribute - You can set background color for whole table or just for one cell.
2)background attribute - You can set background image for whole table or just for one cell. You can also set border color also using bordercolor attribute.

```
<!DOCTYPE html>

 <html>

 <head>

 <title>HTML Tables</title> </head>

 <body>

          <table border="1"bordercolor="red" bgcolor="yellow">

                   <tr> <th>Name</th>

                    <th>Salary</th> </tr>

                            <td>Jayapal     </td> <td>50,000.00</td>
```



Name	Salary
Jayapal	50,000.00
Ravi	45,000.00

Images are very important to beautify as well as to depict many complex concepts in simple way on your web page.

Insert Image:

insert any image in the web page by using <**img**> tag.
<img align="left|right|middle|top|bottom">

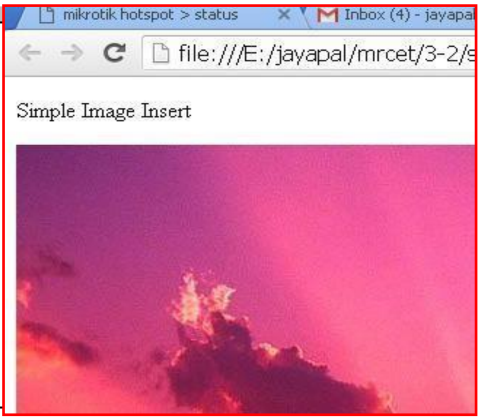
```
<img src="Image URL" ... attributes-
```

Attribute Values

Value	Description
left	Align the image to the left
right	Align the image to the right
middle	Align the image in the middle
top	Align the image at the top
bottom	Align the image at the bottom

Example

```
<!DOCTYPE html>
 <html>
        <head>
                <title>Using Image in Webpage</title>
        </head>
        <body> <p>Simple Image Insert</p>
```

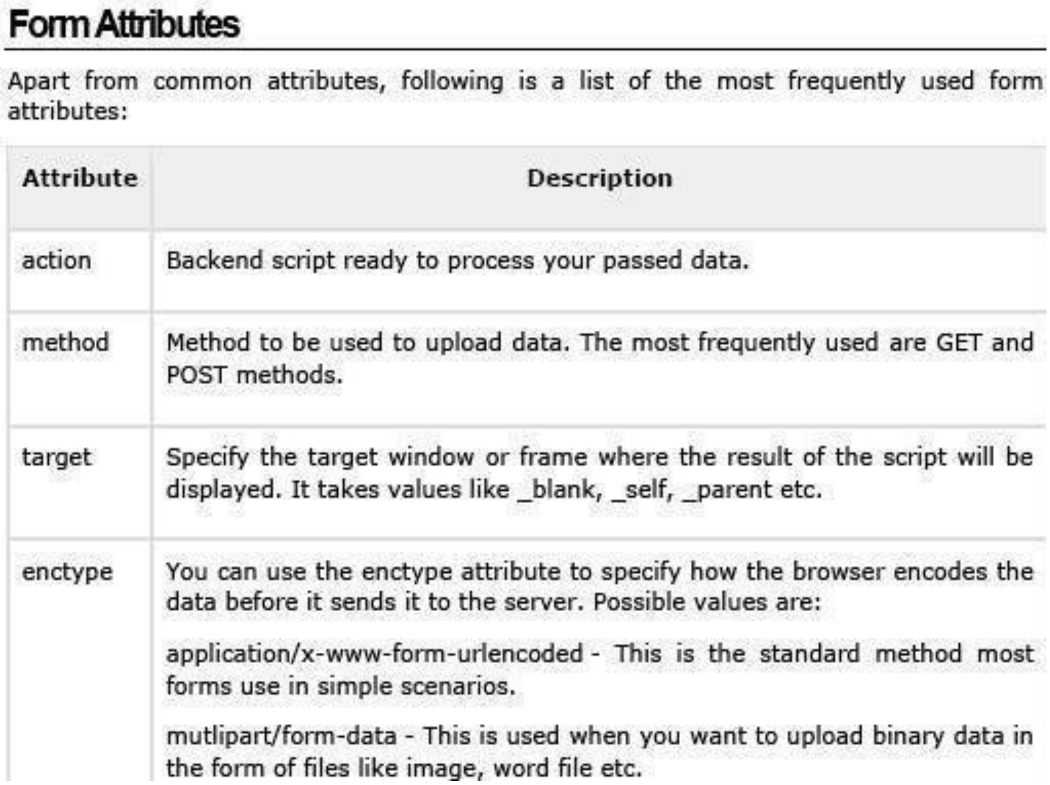


Simple Image Insert

HTML FORMS:

HTML Forms are required to collect some data from the site visitor. For example, during user registration you would like to collect information such as name, email address, credit card, etc. A form will take input from the site visitor and then will post it to a back-end application such as CGI, ASP Script or PHP script etc. The back-end application will perform required processing on the passed data based on defined business logic inside the application. There are various form elements available like text fields, text area fields, drop-down menus, radio buttons, checkboxes, etc.

```
<form action="Script URL" method="GET|POST"> form elements like input, text area etc. </form>
```



Form Attributes

Apart from common attributes, following is a list of the most frequently used form attributes:

Attribute	Description
action	Backend script ready to process your passed data.
method	Method to be used to upload data. The most frequently used are GET and POST methods.
target	Specify the target window or frame where the result of the script will be displayed. It takes values like _blank, _self, _parent etc.
enctype	You can use the enctype attribute to specify how the browser encodes the data before it sends it to the server. Possible values are: application/x-www-form-urlencoded - This is the standard method most forms use in simple scenarios. mutlipart/form-data - This is used when you want to upload binary data in the form of files like image, word file etc.

HTML Form Controls :

There are different types of form controls that you can use to collect data using HTML

- Text Input Controls
- Checkboxes Controls
- Radio Box Controls
- Select Box Controls
- File Select boxes
- Hidden Controls
- Clickable Buttons
- Submit and Reset Button

Text Input Controls:-

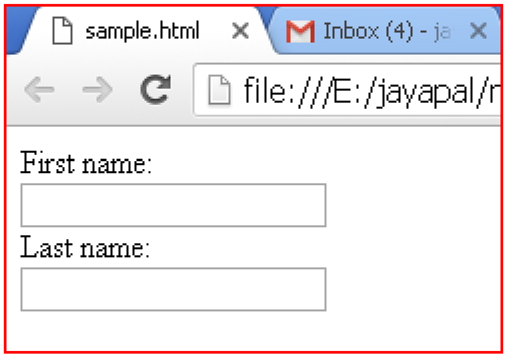
There are three types of text input used on forms:

1) **Single-line text input controls -** This control is used for items that require only one line of user input, such as search boxes or names. They are created using HTML **<input>** tag.

<input type="text"> defines a one-line input field for **text input**:

Example:

```
<form>
 First name:<br>
 <input type="text"
 name="firstname"><br> Last name:<br>
 <input type="text" name="lastname">
</form>
```



First name:

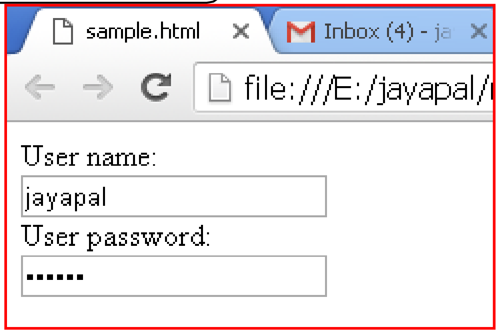
Last name:

2) **Password input controls -** This is also a single- line text input but it masks the character as soon as a user enters it. They are also created using HTML <input> tag.

Input Type Password

<input type="password"> defines a **password field**:

```
<form>

 User name:<br>

 <input type="text" name="username"><br>
 User password:<br>
```



User name:

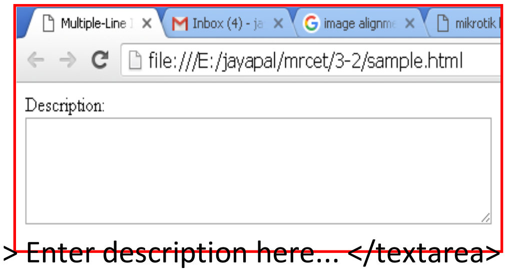
jayapal

User password:

••••••

3) **Multi-line text input controls -** This is used when the user is required to give details that may be longer than a single sentence. Multi- line input controls arecreated using HTML <textarea> tag.

```
<!DOCTYPE html>
<html>
  <head>
              <title>Multiple-Line Input Control</title>
  </head>
  <body>
   <form> Description: <br />
       <textarea rows="5" cols="50" name="description"> Enter description here... </textarea>

   </form>

  </body>

</html>
```



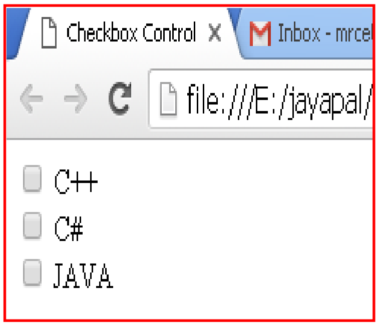
Description:

Checkboxes Controls:-

Checkboxes are used when more than one option is required to be selected. They are also created using HTML <input> tag but type attribute is set to checkbox.

Here is an example HTML code for a form with two checkboxes:

```
<!DOCTYPE html>
<html> <head> <title>Checkbox Control</title> </head>
 <body>
        <form>
               <input type="checkbox" name="C++" value="on"> C++
               <br>
               <input type="checkbox" name="C#" value="on"> C#

               <br>

               <input type="checkbox" name="JAVA" value="on"> JAVA
```

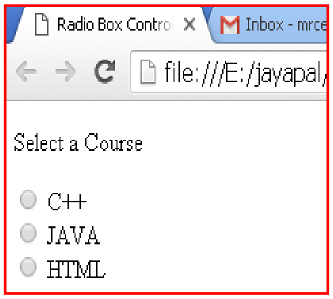


☐ C++
☐ C#
☐ JAVA

Radio Button Control:-

Radio buttons are used when out of many options, just one option is required to be selected. They are also created using HTML <input> tag but type attribute is set to radio.

```
 <!DOCTYPE html>
<html> <head> <tit le>Radio Box Control</title> </head>
         <body> <p>Select a Course</p>
                <form>
                       <input type="radio" name="subject" value="C++"> C++
                       <br>
                       <input type="radio" name="subject" value="JAVA"> JA VA<br>

                        <input type="radio" name="subject" value="HTML"> HTML

                </form>    </body>  </html>
```

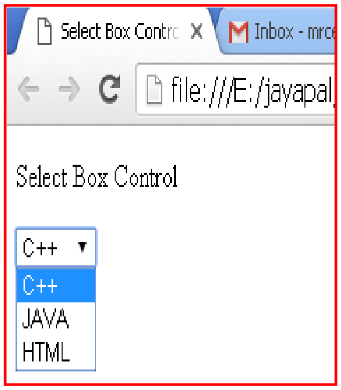


Select a Course

○ C++
○ JAVA
○ HTML

Select Box Controls :- A select box, also called drop down box which provi des option to list down various options in the form of drop down list, from where a user can select one or more options.

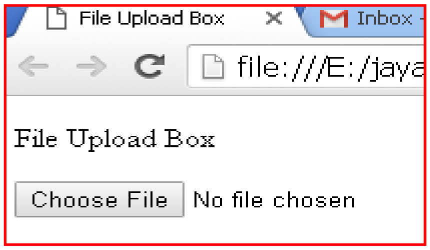
```
<!DOCTYPE html>

<html>

 <head>

         <title>Select Box Control</title>

  </head>

  <body>

         <form>

                 <select name="dropdown">

                         <option value="C++" selected>C++</option>

                         <option value="JAVA">JA VA</option>

                         <option value="HTML">HTML</option>

                 </select>

         </form>

  </body>

 </html>
```



File Select boxes:- If you want to allow a user to upload a file to your web site, you will need to use a file upload box, also known as a file select box. This is also created usingthe

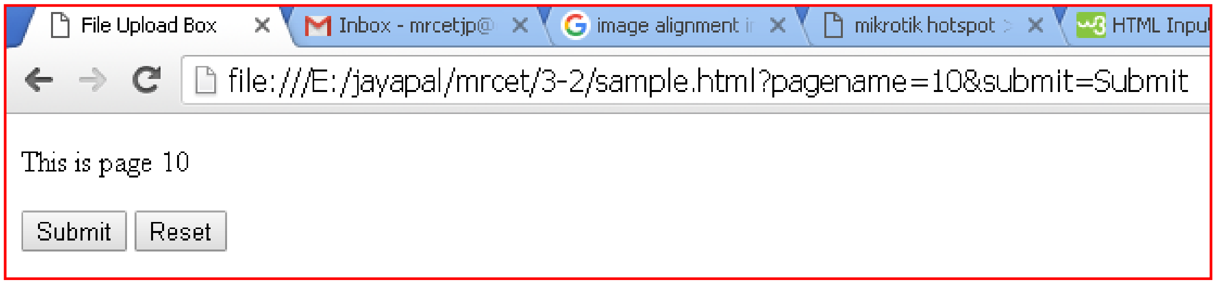
<input > element but type attribute is set to **file.**

```
<!DOCTYPE html>

<html>

        <head>

                <title>File Upload Box</title>

        </head>

        <body>
```



Hidden Controls:- Hidden form controls are used to hide data inside the page which later on can be pushed to the server. This control hides inside the code and does not appear on the actual page. For example, following hidden form is being used to keep current page number. When a user will click next page then the value of hidden control will be sent to the web server and there it will decide which page will be displayed next based on the passed current page.

```
<html>   <head>              <title>File Upload Box</title>          </head

                  <form>

                             <p>This is page 10</p>

                             <input type="hidden" name="pagename" value="10" />
```

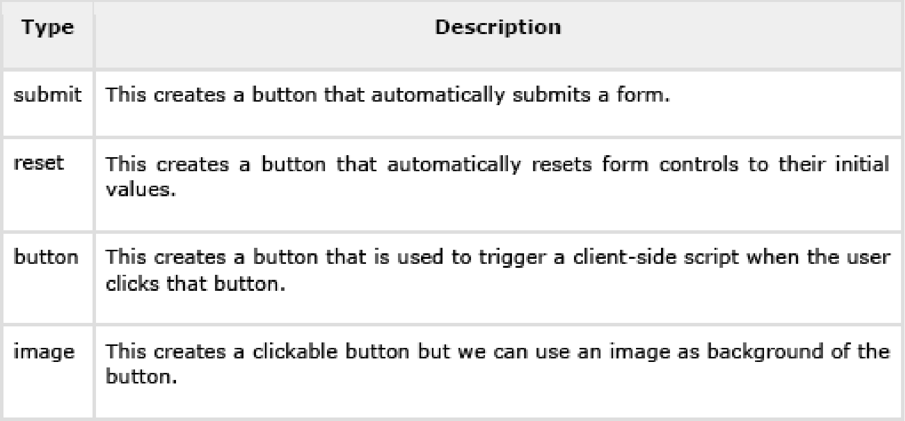


file:///E:/jayapal/mrcet/3-2/sample.html?pagename=10&submit=Submit

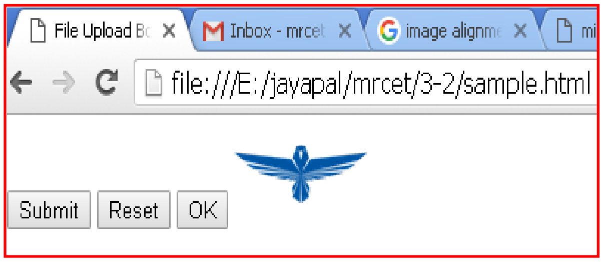
This is page 10

Button Controls:-

There are various ways in HTML to create clickable buttons. You can also create a clickable button using <input> tag by setting its type attribute to **button**. The type attribute can take the following values:



Type	Description
submit	This creates a button that automatically submits a form.
reset	This creates a button that automatically resets form controls to their initial values.
button	This creates a button that is used to trigger a client-side script when the user clicks that button.
image	This creates a clickable button but we can use an image as background of the button.



file:///E:/jayapal/mrcet/3-2/sample.html

<!DOCTYPE html>

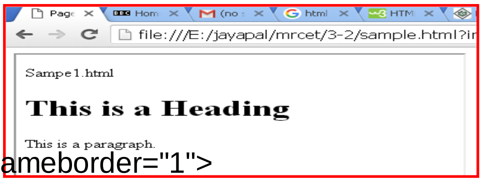
HTML frames: These are used to divide your browser window into multiple sections where each section can load a separate HTML document. A collection of frames in the browser window is known as a frameset. The window is divided into frames in a similar way the tables are organized: into rows and columns.

To use frames on a page we use <frameset> tag instead of <body> tag. The <frameset> tag defines, how to divide the window into frames. The **rows** attribute of <frameset> tag defines horizontal frames and **cols** attribute defines vertical frames. Each frame is indicated by <frame> tag and it defines which HTML document shall open into the frame.

Note: HTML **<frame>** Tag. Not Supported in HTML5.

```
<frameset cols="25%,50%,25%">
          <framesrc="frame_a.htm">
          <frame src="frame_b.htm">
          <frame   src="frame_c.htm">
</frameset>
```

```
<!DOCTYPE html>
<html>
        <head>
                <title>Page Title</title>
        </head>
        <body>
        <iframe src="sample1.html" height="400" width="400"frameborder="1">
```



file:///E:/jayapal/mrcet/3-2/sample.html?ir

Sampe1.html

This is a Heading

This is a paragraph.

CSS stands for Cascading Style Sheets

CSS describes **how HTML elements are to be displayed on screen, paper, or in other media**. CSS **saves a lot of work**. It can control the layout of multiple web pages all at once.

CSS can be added to HTML elements in 3 ways:

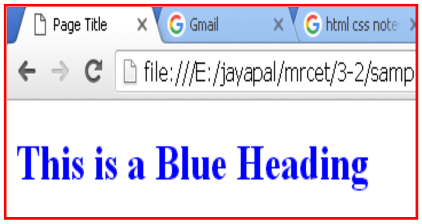
- **Inline** - by using the style attribute in HTML elements
- **Internal** - by using a <style> element in the <head> section
- **External** - by using an external CSS file

Inline CSS

An inline CSS is used to apply a unique style to a single HTML element.
An inline CSS uses the style attribute of an HTML element.
This example sets the text color of the < h1> element to blue:

```
<h1 style="color:blue;">This is a Blue Heading</h1>
```

```
<html>      <head>       <title>Page Title</title> </head>
         .   .
                <h1style="color:blue;">ThisisaBlueHeading</h1>
         </body>
```



file:///E:/jayapal/mrcet/3-2/samp

This is a Blue Heading

Internal CSS: An internal CSS is used to define a style for a single HTML page. An internal CSS is defined in the <head> section of an HTML page, within a <style> element:

```
<html>
          <head>
          <style>
          body {background-color: powderblue;}
          h1 {color: blue;}
          p    {color: red;}
          </style>
          </head>
```

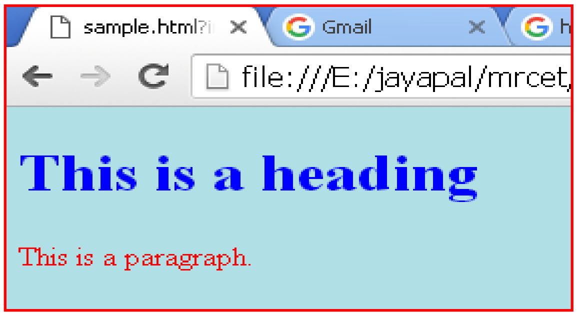
External CSS:-

An external style sheet is used to define the style for many HTML pages. **With an external style sheet, you can change the look of an entire web site, by changing one file!** To use an external style sheet, add a link to it in the <head> section of the HTML page:

```
<html>
          <head>
            <link rel="stylesheet" href="styles.css">
          </head>
          <body>
                    <h1>This is a heading</h1>
```

An external style sheet can be written in any text editor. The file must not contain any HTML code, and must be saved with **a .css extension**.
Here is how the "styles.css" looks:

```
body    {    background-color: powderblue;     }
h1      {   color: blue;    }
p        {   color: red;    }
```

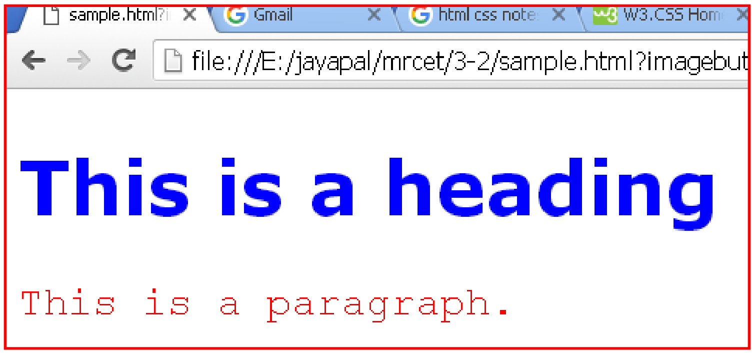


This is a heading

This is a paragraph.

CSS Fonts: The CSS **color** property defines the text color to be used. The CSS **font-family** property defines the font to be used. The CSS **font-size** property defines the text size to be used.

```
<html>

<head>

<style>
h1 {

    color: blue;

    font-family:
    verdana; font-size:
    300%;

}

p {

    color: red;

    font-family: courier;
    font-size: 160%;

}
```



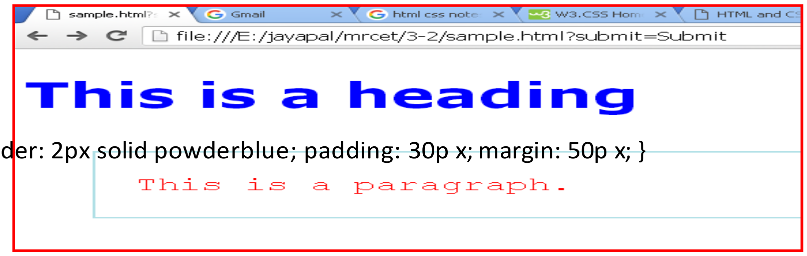
This is a heading

This is a paragraph.

CSS Border: The CSS **border** property defines a border around an HTML element.

CSS Padding: The CSS **padding** property defines a padding (space) between the text and the border.

```
<html> <head>

<style>
h1 {

   color: blue;

   font-family: verdana;
   font-size: 300%; }

p {

   color: red;    font-size: 160%; border: 2px solid powderblue; padding: 30p x; margin: 50p x; }

</style>

</head>
```



This is a heading

This is a paragraph.

CSS Margin: The CSS **margin** property defines a margin (space) outside