Unit – III :: JavaScript

3.1 INTRODUCTION TO JAVASCRIPT

- JavaScript is a very powerful **client-side scripting language**.
- JavaScript is used mainly for enhancing the interaction of a user with the webpage.
- JavaScript is also being used widely in game development and Mobile application development.
- You need a text editor to write JavaScript code and a browser to display your web page.

The <script> Tag

In HTML, JavaScript code is inserted between `<script>` and `</script>` tags.

JavaScript Output Display Possibilities

JavaScript can "display" data in different ways:

- Writing into an HTML element, using `innerHTML`.
- Writing into the HTML output using `document.write()`.
- Writing into an alert box, using `window.alert()`.
- Writing into the browser console, using `console.log()`.

Simple JavaScript Program

```
<html>
<head>
    <title>My First JavaScript code!!!</title>
    <script>
            window.alert("Hello World!");
    </script>
</head>
<body>
</body>
</html>
```

JavaScript statements are composed of Values, Operators, Expressions, Keywords, and Comments. JavaScript is case-sensitive, means that variables, language keywords, function names, and other identifiers must always be typed with a consistent capitalization of letters

JavaScript Comments

- Single line comments start with `//`.
- Multi-line comments start with `/*` and end with `*/`.

3.1.1 JavaScript Identifiers

All JavaScript **variables** must be **identified** with **unique names**.

The general rules for constructing names for variables (unique identifiers) are:

- Names can contain letters, digits, underscores, and dollar signs.
- Names must begin with a letter

- Names can also begin with $ and _ (but we will not use it in this tutorial)
- Names are case sensitive (y and Y are different variables)
- Reserved words (like JavaScript keywords) cannot be used as names.

3.1.2 JavaScript Operators

Different types of operators are:

- Arithmetic Operators
- Assignment Operators
- String Operators
- Comparison Operators
- Logical Operators
- Type Operators
- Bitwise Operator

3.1.3 JavaScript Functions

- A JavaScript function is a block of code designed to perform a particular task.

Example program

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Functions</h2>
<p id="demo"></p>
<script>
function myFunction(p1, p2) {
  return p1 * p2;
}
document.getElementById("demo").innerHTML = myFunction(4, 3);
</script>
</body>
</html>
```

3.1.4 JavaScript Data Types

There are six basic data types in JavaScript which can be divided into **three** main categories:

- **primitive (or *primary*)** -> String, Number, and Boolean
- ***composite* (or *reference*)** -> Object, Array, and Function (which are all types of objects)
- ***special* data types**-> Undefined and Null

3.1.5 JavaScript Errors

- The try statement lets you test a block of code for errors.
- The catch statement lets you handle the error.
- The throw statement lets you create custom errors.
- The finally statement lets you execute code, after try and catch, regardless of the result.

3.1.6 JavaScript Events

An event is something that happens when user interact with the web page.
Some of the java script events are:

- Mouse Events
- Keyboard Events
- Form Events
- The Submit Event (onsubmit)

3.2 Objects in Javascript

Objects, in JavaScript, is it's most important data-type and forms the building blocks for modern JavaScript. These objects are quite different from JavaScript's primitive data-types(Number, String, Boolean, null, undefined and symbol) in the sense that while these primitive data-types all store a single value each (depending on their types).

- Objects are more complex and each object may contain any combination of these primitive data-types as well as reference data-types.
- An object, is a reference data type. Variables that are assigned a reference value are given a reference or a pointer to that value. That reference or pointer points to the location in memory where the object is stored. The variables don't actually store the value.
- Loosely speaking, **objects in JavaScript may be defined as an unordered collection of related data, of primitive or reference types, in the form of "key: value" pairs.** These keys can be variables or functions and are called properties and methods, respectively, in the context of an object.

An object can be created with figure brackets {…} with an optional list of properties. A property is a "key: value" pair, where a key is a string (also called a "property name"), and value can be anything.
To understand this rather abstract definition, let us look at an example of a JavaScript Object :

```
let school = {
   name : "Vivekananda School",
   location : "Delhi",
   established : "1971"
}
```

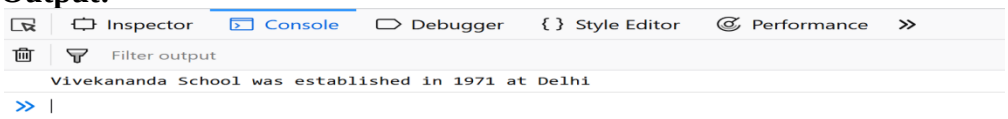
In the above example **"name", "location", "established"** are all **"keys"** and **"Vivekananda School", "Delhi" and 1971** are values of these keys respectively.

Each of these keys is referred to as **properties** of the object. An object in JavaScript may also have a function as a member, in which case it will be known as a **method** of that object.
Let us see such an example :

```
// javascript code demonstrating a simple object
let school = {
   name: 'Vivekananda School',
   location : 'Delhi',
   established : '1971',
   displayInfo : function(){
      console.log(`${school.name} was established
```

```
        in ${school.established} at ${school.location}`);
    }
}
school.displayInfo();
```

Output:



In the above example, **"displayinfo"** is a method of the school object that is being used to work with the object's data, stored in its properties.

Properties of JavaScript Object

The property names can be strings or numbers. In case the property names are numbers, they must be accessed using the "bracket notation" like this :

```
let school = {
    name: 'Vivekananda School',
    location : 'Delhi',
    established : '1971',
    20 : 1000,
    displayInfo : function(){
        console.log(`The value of the key 20 is ${school['20']}`);
    }
}
school.displayInfo();
```

Output:

But more on the bracket notation later.

Property names can also be strings with more than one space separated words. In which case, these property names must be enclosed in quotes :

```
let school = {
    "school name" : "Vivekananda School",
}
```

Like property names which are numbers, they must also be accessed using the bracket notation. Like if we want to access the 'Vivekananda' from 'Vivekananda School' we can do something like this:

```
// bracket notation
let school = {
    name: 'Vivekananda School',
    displayInfo : function(){
        console.log(`${school.name.split(' ')[0]}`);
    }
}
school.displayInfo(); // Vivekananda
```

Output:

In the above code, we made use of bracket notation and also split method provided by javascript which you will learn about in strings article.

2.2.1 Inherited Properties

Inherited properties of an object are those properties which have been inherited from the object's prototype, as opposed to being defined for the object itself, which is known as the object's Own property. To verify if a property is an objects Own property, we can use the **hasOwnProperty** method.

PropertyAttributes

Data properties in JavaScript have four attributes.

- **value:** The property's value.
- **writable:** When true, the property's value can be changed
- **enumerable:** When true, the property can be iterated over by "for-in" enumeration. Otherwise, the property is said to be non-enumerable.
- **configurable:**If false, attempts to delete the property, change the property to be an access-or property, or change its attributes (other than [[Value]], or changing [[Writable]] to false) will fail.

```
// hasOwnProperty code in js
const object1 = new Object();
object1.property1 = 42;

console.log(object1.hasOwnProperty('property1')); // true
```

Output:

3.2.2 Creating Objects

There are several ways or syntax's to create objects. One of which, known as the Object literal syntax, we have already used. Besides the object literal syntax, objects in JavaScript may also be created using the constructors, Object Constructor or the prototype pattern.

1. **Using the Object literal syntax :** Object literal syntax uses the {…} notation to initialize an object an its methods/properties directly.
   Let us look at an example of creating objects using this method :

```
var obj = {
    member1 : value1,
    member2 : value2,
};
```

   These members can be anything – strings, numbers, functions, arrays or even other objects. An object like this is referred to as an object literal. This is different from other methods of object creation which involve using constructors and classes or prototypes, which have been discussed below.

3.3 Dynamic HTML with Java Script

DHTML stands for Dynamic HTML. Dynamic means that the content of the web page can be customized or changed according to user inputs i.e. a page that is interactive with the user. In earlier times, HTML was used to create a static page. It only defined the structure of the content that was displayed on the page. With the help of CSS, we can beautify the HTML page by changing various properties like text size, background color etc. The HTML and CSS could manage to navigate between static pages but couldn't do anything else. If 1000 users view a page that had their information for eg. Admit card then there was a problem because 1000 static pages for this application build to work. As the number of users increase, the problem also increases and at some point it becomes impossible to handle this problem.

To overcome this problem, DHTML came into existence. DHTML included JavaScript along with HTML and CSS to make the page dynamic. This combo made the web pages dynamic and eliminated this problem of creating static page for each user. To integrate JavaScript into HTML, a Document Object Model(DOM) is made for the HTML document. In DOM, the document is represented as nodes and objects which are accessed by different languages like JavaScript to manipulate the document.
HTML document include JavaScript:: The JavaScript document is included in our html page using the html tag. <src> tag is used to specify the source of external JavaScript file.

Example 1: Example to understand how to use JavaScript in DHTML.

```
<html>

<head>

<title>DOM programming</title>

</head>

<body>

<h1>GeeksforGeeks</h1>

<p id = "geeks">Hello Geeks!</p>

<script style = "text/javascript">

document.getElementById("geeks").innerHTML =

        "A computer science portal for geeks";

</script>

</body>

</html>
```

Example 2: Creating an alert on click of a button.

```
<html>

<head>

<title>Create an alert</title>

</head>

<body>

<h1 id = "para1">GeeksforGeeks</h1>

<input type = "Submit" onclick = "Click()"/>

<script style = "text/javascript">

        function Click() {

document.getElementById("para1").style.color = "#009900";

window.alert("Color changed to green");

        }

</script>

</body>

</html>
```

Part-B(XML)

3.9 Document Type Definition Introduction

The XML Document Type Declaration, commonly known as DTD, is a way to describe XML language precisely. DTDs check vocabulary and validity of the structure of XML documents against grammatical rules of appropriate XML language.

An XML DTD can be either specified inside the document, or it can be kept in a separate document and then liked separately.

Syntax

Basic syntax of a DTD is as follows −

```
<! DOCTYPE element DTD identifier
[
declaration1
declaration2
]>
```

In the above syntax,

- The DTD starts with <!DOCTYPE delimiter.
- An element tells the parser to parse the document from the specified root element.
- 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*.

3.9.1 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 an external source.

Syntax:

Following is the syntax of internal DTD −

```
<!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>Tanmay Patil</name>
  <company>TutorialsPoint</company>
  <phone>(011) 123-4567</phone>
</address>
```

3.9.2 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 valid 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 external source.

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>Tanmay Patil</name>
   <company>TutorialsPoint</company>
   <phone>(011) 123-4567</phone>
</address>
```

The content of the DTD file address.dtd is 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 is written as follows −

```
<!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.

3.10 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 describes the data in a database.

Syntax

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

Example

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.

Elements

As we saw in the XML - Elements chapter, elements are the building blocks of XML document. An element can be defined within an XSD as follows −

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

Definition Types

You can define XML schema elements in the following ways −

3.10.1 Simple Type

Simple type element is used only in the context of the text. Some of the predefined simple types are: xs:integer, xs:boolean, xs:string, xs:date. For example −

```
<xs:element name = "phone_number" type = "xs:int" />
```

3.10.2 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 the 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 follows −

```
<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 follows −

```
<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

Attributes in XSD provide extra information within an element. Attributes have *name* and *type* property as shown below −

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

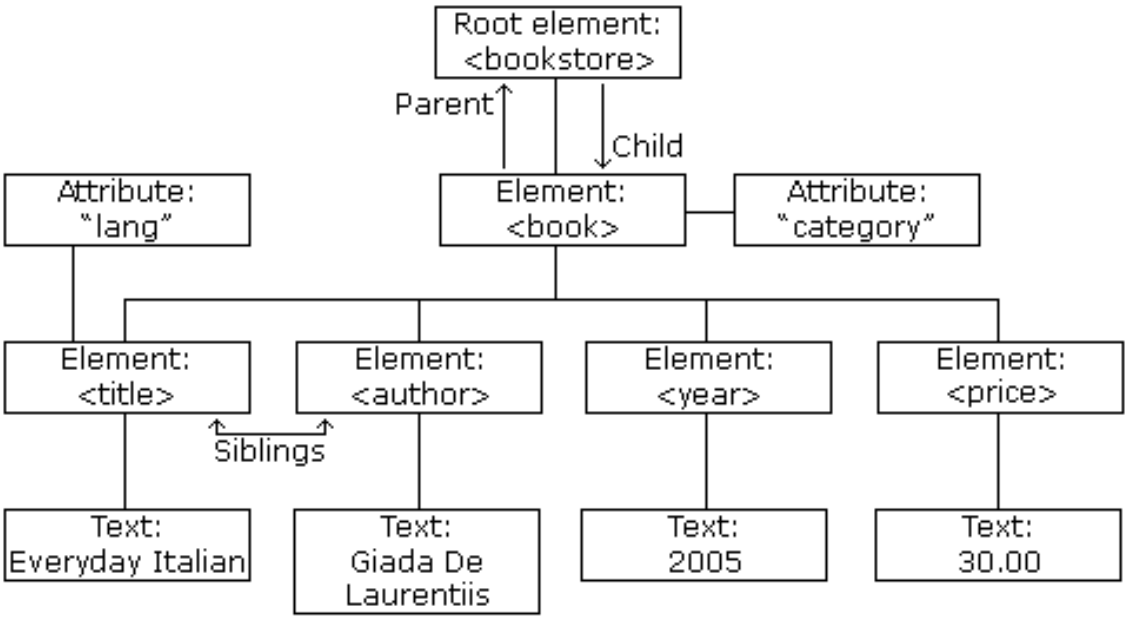
3.11 DOM Introduction

The **Document Object Model** (**DOM**) is a cross-platform and language-independent interface that treats an XML or HTML **document** as a tree structure wherein each node is an **object** representing a part of the **document**.

- DOM stands for Document Object Model
- DOM represents a document with a logical tree.
- The DOM defines a standard for accessing documents:
- Latest version: DOM4; November 19, 2015; 4 years ago
- First published: October 1, 1998; 21 years ago
- When a web page is loaded, the browser creates a Document Object Model of the page.

The **DOM** standard is separated into 3 different parts:

- **Core DOM -** standard model for all document types
- **XML DOM** - standard model for XML documents
- **HTML DOM** - standard model for HTML documents
  MLXML
  ML DOMXML



XML DOM

- The HTML DOM defines a standard way for accessing and manipulating HTML documents. It presents an HTML document as a tree-structure.

- The XML DOM defines a standard way for accessing and manipulating XML documents. It presents an XML document as a tree-structure.

The HTML DOM
All HTML elements can be accessed through the HTML DOM.
This example changes the value of an HTML element with id="demo":
EXAMPLE:

```
<h1 id="demo">This is a Heading</h1>
<button type="button"
onclick="document.getElementById('demo').innerHTML = 'Hello World!'">Click Me!
</button>
```

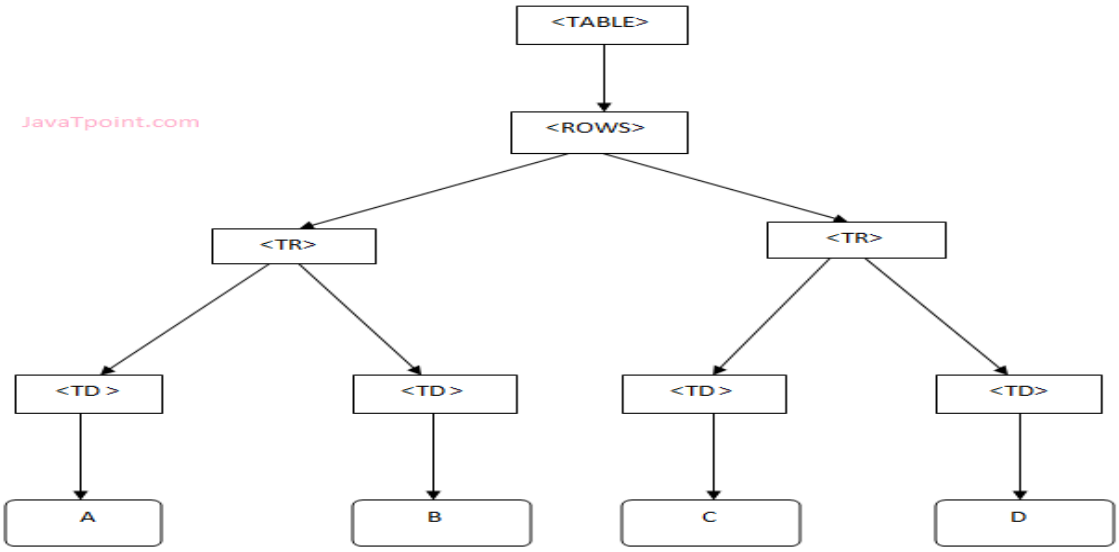
Tags Explanation

- The <!DOCTYPE html> declaration defines this document to be HTML5
- The <html> element is the root element of an HTML page
- The <head> element contains meta information about the document
- The <title> element specifies a title for the document
- The <body> element contains the visible page content
- The <h1> element defines a large heading
- The <button> element defines a clickable **button**.

What does XML DOM
The XML DOM makes a tree-structure view for an XML document.
We can access all elements through the DOM tree.
We can modify or delete their content and also create new elements. The elements, their content (text and attributes) are all known as nodes.
For example, consider this table, taken from an HTML document:

```
<TABLE>
<ROWS>
<TR>
<TD>A</TD>
<TD>B</TD>
</TR>
<TR>
<TD>C</TD>
<TD>D</TD>
</TR>
</ROWS>
</TABLE>
```

The Document Object Model represents this table like this:



XML DOM Example : Load XML File

Let's take an example to show how an XML document ("note.xml") is parsed into an XML DOM object.

This example parses an XML document (note.xml) into an XML DOM object and extracts information from it with JavaScript.

Let's see the XML file that contains message.

note.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
  <to>csec2020@svecw.edu.in</to>
  <from>ramaraocse@svecw.edu.in</from>
  <body>Hello XML DOM</body>
</note>
```

Let's see the HTML file that extracts the data of XML document using DOM.

xmldom.html

```
<!DOCTYPE html>
<html>
<body>
<h1>Important Note</h1>
<div>
<b>To:</b> <span id="to"></span><br>
<b>From:</b> <span id="from"></span><br>
<b>Message:</b> <span id="message"></span>
</div>
<script>
if (window.XMLHttpRequest)
```

```
 {// code for IE7+, Firefox, Chrome, Opera, Safari
 xmlhttp=new XMLHttpRequest();
 }
else
 {// code for IE6, IE5
 xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
 }
xmlhttp.open("GET","note.xml",false);
xmlhttp.send();
xmlDoc=xmlhttp.responseXML;
document.getElementById("to").innerHTML=
xmlDoc.getElementsByTagName("to")[0].childNodes[0].nodeValue;
document.getElementById("from").innerHTML=
xmlDoc.getElementsByTagName("from")[0].childNodes[0].nodeValue;
document.getElementById("message").innerHTML=
xmlDoc.getElementsByTagName("body")[0].childNodes[0].nodeValue;
</script>
</body>
</html>
```

Output:

Important Note
To: csec2020@svecw.edu.in
From: ramaraocse@svecw.edu.in
Message: Hello XML DOM

3.12 XSLT introduction

- XSL (eXtensible Stylesheet Language) is a styling language for XML.
- XSLT stands for XSL Transformations.
- XSLT is the most important part of XSL
- XSLT transforms an XML document into another XML document
- XSLT uses XPath to navigate in XML document.

Example:

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

```
    <h2>My CD Collection</h2>
    <table border="1">
      <tr bgcolor="#9acd32">
        <th>Title</th>
        <th>Artist</th>
      </tr>
      <xsl:for-each select="catalog/cd">
        <tr>
          <td><xsl:value-of select="title"/></td>
          <td><xsl:value-of select="artist"/></td>
        </tr>
      </xsl:for-each>
    </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

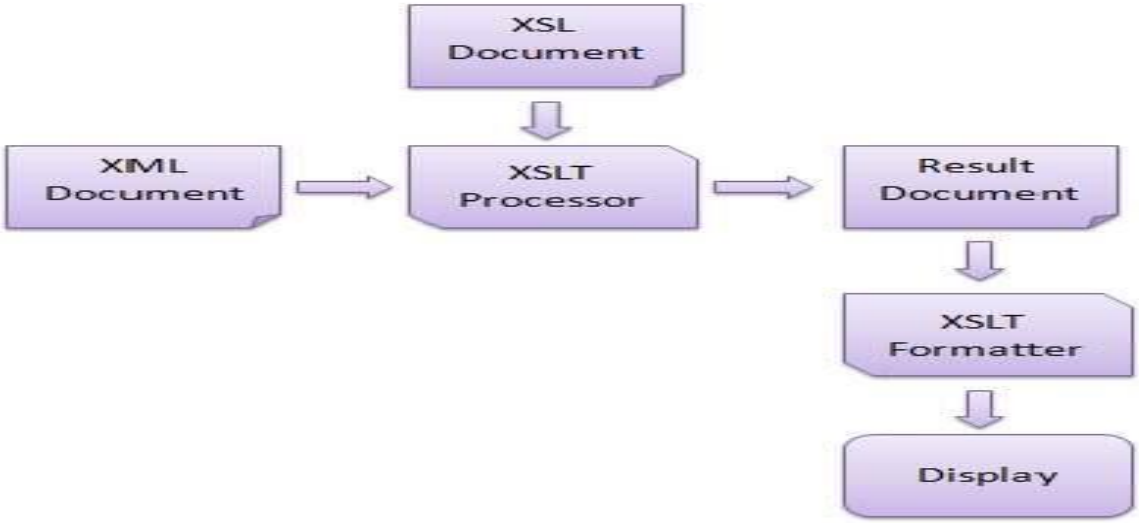
XSL - More Than a Style Sheet Language

XSL Consists of four:

XSLT - a language for transforming XML documents
XPath - a language for navigating in XML documents
XSL-FO - a language for formatting XML documents (discontinued in 2013)
XQuery - a language for querying XML documents

How XSLT Works

- 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 applies the transformation rules on the target XML document and then it generates a formatted document in the form of XML, HTML, or text format.
- This formatted document is then utilized by XSLT formatter to generate the actual output which is to be displayed to the end-user.



Advantages :

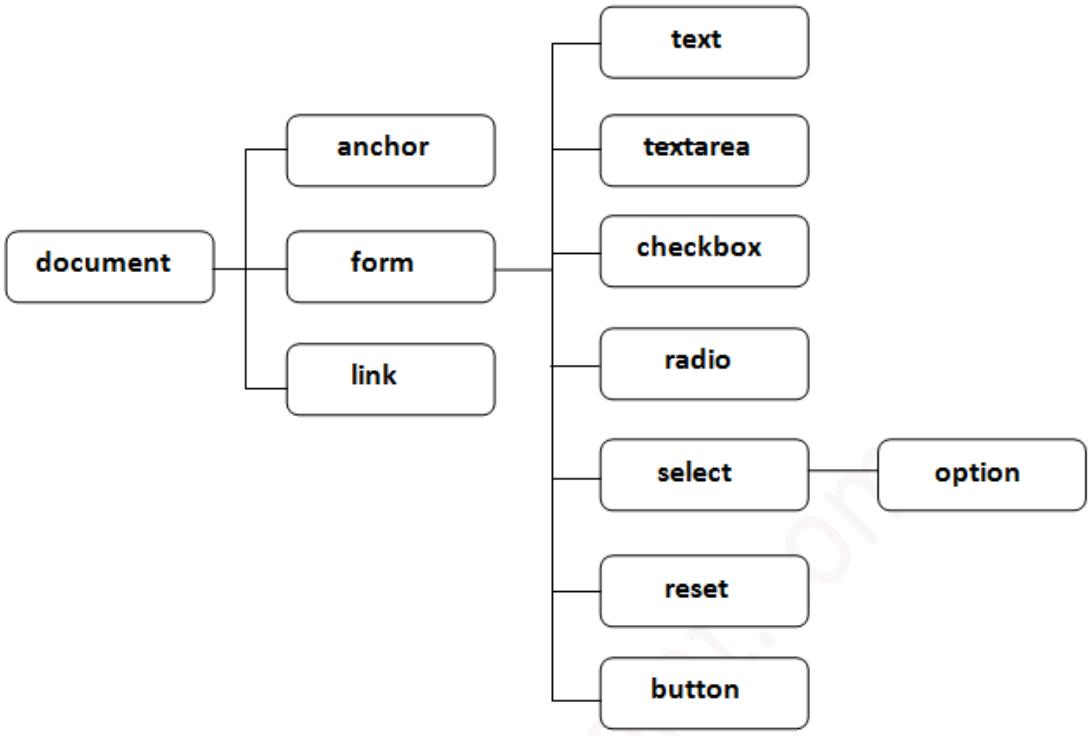
●Independent of programming. Transformations are written in a separate xsl file which is again an XML document.

●Output can be altered by simply modifying the transformations in xsl file. No need to change any code. So Web designers can edit the stylesheet and can see the change in the output quickly.

3.13 Document Object Model

The document object represents the whole html document.
When html document is loaded in the browser, it becomes a document object. It is the root element that represents the html document. It has properties and methods. By the help of document object, we can add dynamic content to our web page.
As mentioned earlier, it is the object of window. So window document Is same as document

Properties of document object :
Let's see the properties of document object that can be accessed and modified by the document object.



Methods of document object :
We can access and change the contents of document by its methods.

The important methods of document object are as follows:

Method	Description

write("string")	writes the given string on the doucment.
writeln("string")	writes the given string on the doucment with newline character at the end.
getElementById()	returns the element having the given id value.
getElementsByName()	returns all the elements having the given name value.
getElementsByTagName()	returns all the elements having the given tag name.
getElementsByClassName()	returns all the elements having the given class name.

Accessing field value by document object :
In this example, we are going to get the value of input text by user. Here, we are using document.form1.name.value to get the value of name field.

Here, document is the root element that represents the html document.
form1 is the name of the form.
name is the attribute name of the input text.
value is the property, that returns the value of the input text.

Let's see the simple example of document object that prints name with welcome message.

```
<script type="text/javascript">
function printvalue(){
var name=document.form1.name.value;
alert("Welcome: "+name);
}
</script>
<form name="form1">
Enter Name:<input type="text" name="name"/>
<input type="button" onclick="printvalue()" value="print name"/>
</form>
```

3.14 SAX (Simple API for XML) :
SAX (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.

When to Use?

You should use a SAX parser when −

- You can process the XML document in a linear fashion from top to 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 a 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 of SAX :

- 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 that the parser has seen or change the order of items, you must write the code and store the data on your own.

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)