

UNIT IV

Java Script and HTML Documents, The Document Object Model; Element access in JavaScript; Events and event handling; Handling events from the Body elements, Button elements, Text box and Password elements; The DOM 2 event model; The navigator object; DOM tree traversal and modification.

Introduction to dynamic documents; positioning elements; Moving elements; Element visibility; Changing colors and fonts; Dynamic content; Stacking elements; Locating the mouse cursor; Reacting to a mouse click; Slow movement of elements; Dragging and dropping elements.

XML: Introduction to XML, Anatomy of an XML, document, Creating XML Documents, Creating XML DTDs, XML Schemas, XSL, XML processors, Web services.

The Document Object Model

- DOM (Document Object Model) is an API (Application Programming Interface) that defines an interface between HTML documents and application program.
- It is an abstract model that applies to a variety of programming language.
- Essentially, the various structures in an HTML document that are marked up with tags are considered objects, complete with properties and methods.
- The attributes of an HTML tag corresponds to a property for the corresponding object.

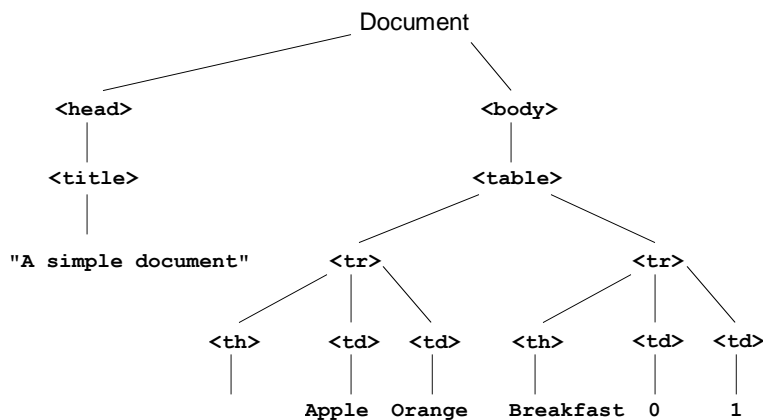
A table in XHTML

```
<!DOCTYPE html>
<!-- table2.html
    A simple table to demonstrate DOM trees
-->
<html lang = "en">
  <head> <title> A simple document </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <table>
      <tr>
        <th> </th>
        <td> Apple </td>
        <td> Orange </td>
      </tr>

      <tr>
        <th> Breakfast </th>
        <td> 1 </td>
        <td> 0 </td>
      </tr>

    </table>
  </body>
</html>
```

DOM Structure of the Document



Element Access in JavaScript

- Manipulating elements in an HTML document requires that you have the address of the corresponding object.
- This can be handled in several different ways, depending on which version of DOM to which you seek to conform.
- The address of a JavaScript object associated with an HTML element is called its DOM address.

Sample XHTML document

```
<html lang = "en">
<head> <title> Access to form elements </title>
<meta charset = "utf-8" />
</head>
```

```
<body>
<form name = "myForm" action = "">
<input type = "button" name = "turnItOn">
```

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

Getting the DOM Address

- We can use the forms and elements arrays:

```
var dom = document.forms[0].elements[0];
```

If another button were added before turnItOn, the address could no longer be accessed this way.

- We can use the name given to the item (this requires that every enclosing item (up to but not including body) must have a name:

```
var dom = document.myForm.turnItOn;
```

The problem is that XHTML 1.1 does not allow forms to have a name. This causes a validation problem.

getElementById

- A better way involves using getElementById:

```
var dom = document.getElementById("turnItOn");
```

- Since ids are useful for DOM addressing and form processing frequently requires name, it is not unusual for form elements to have both set to the same value.

- Since buttons in a group of checkboxes often share a name and a group of radio buttons will share a name, this won't work in finding their DOM addresses.

Events and event handling

Basic Concepts of Event Handling

- An event is a notification that something specific has happened, such as when a document finishes loading, a button is pushed or contents of a textbox is changed.
- An event handler is a script that is implicitly executed in response to an event happening.
- Event-driven programming is when parts of the programming are executed in an unpredictable sequence in response to specific events

Events, Attributes and Tags

- The same attribute can appear in several different tags.
- An XHTML element is said to get focus when the user puts the mouse over it and left-clicks or tabs over to the element.
- An element gets blurred when the user move the cursor away and left-clicks or when (s)he tabs away.

EVENTS, ATTRIBUTES, AND TAGS

WEB Programming

<u>Event</u>	<u>Tag Attribute</u>
blur	onblur
change	onchange
click	onclick
dblclick	ondblclick
focus	onfocus
keydown	onkeydown
keypress	onkeypress
keyup	onkeyup
load	onload

<u>Event</u>	<u>Tag Attribute</u>
mousedown	Onmousedown
mousemove	Onmousemove
mouseout	Onmouseout
mouseover	Onmouseover
mouseup	Onmouseup
reset	Onreset
select	Onselect
submit	Onsubmit
unload	Onunload

In many cases, the same attribute can appear in several different tags. The circumstances under which an event is created are related to a tag and an attribute, and they can be different for the same attribute when it appears in different tags

Attribute	Tag	Description
onblur	<a>	The link loses the input focus
	<button>	The button loses the input focus
	<input>	The input element loses the input focus
	<textarea>	The text area loses the input focus
	<select>	The selection element loses the input focus
onchange	<input>	The input element is changed and loses the input focus
	<textarea>	The text area is changed and loses the input focus
	<select>	The selection element is changed and loses the input focus
onclick	<a>	The user clicks on the link
	<input>	The input element is clicked
ondblclick	Most elements	The user double-clicks the left mouse button
onfocus	<a>	The link acquires the input focus
	<input>	The input element receives the input focus
	<textarea>	A text area receives the input focus
	<select>	A selection element receives the input focus
onkeydown	<body>, form elements	A key is pressed down
onkeypress	<body>, form elements	A key is pressed down and released
onkeyup	<body>, form elements	A key is released
onload	<body>	The document is finished loading
onmousedown	Most elements	The user clicks the left mouse button
onmousemove	Most elements	The user moves the mouse cursor within the element
onmouseout	Most elements	The mouse cursor is moved away from being over the element
onmouseover	Most elements	The mouse cursor is moved over the element
onmouseup	Most elements	The left mouse button is unclicked
onreset	<form>	The reset button is clicked
onselect	<input>	Any text in the content of the element is selected
	<textarea>	Any text in the content of the element is selected
onsubmit	<form>	The Submit button is pressed
onunload	<body>	The user exits the document

As mentioned previously, there are two ways to register an event handler in the DOM 0 event model. One of these is by assigning the event handler script to an event tag attribute, as in the following example:

```
<input type = "button" id = "myButton"
      onclick = "alert('You clicked my button!');" />
```

In many cases, the handler consists of more than a single statement. In these cases, often a function is used and the literal string value of the attribute is the call to the function. Consider the example of a button element:

```
<input type = "button" id = "myButton"
      onclick = "myButtonHandler();" />
```

An event handler function could also be registered by assigning its name to the associated event property on the button object, as in the following example:

```
document.getElementById("myButton").onclick =
      myButtonHandler;
```

HANDLING EVENTS FROM BODY ELEMENTS

The events most often created by body elements are `load` and `unload`. As our first example of event handling, we consider the simple case of producing an alert message when the body of the document has been loaded. In this case, we use the `onload` attribute of `<body>` to specify the event handler:

```
<?xml version = "1.0" encoding = "utf-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- load.html
      A document for load.js
-->
<html xmlns = "http://www.w3.org/1999/xhtml">
  <head>
    <title> load.html </title>
    <script type = "text/javascript" src = "load.js" >
    </script>
  </head>
  <body onload="load_greeting();">
    <p />
  </body>
</html>
```

```
// load.js
//   An example to illustrate the load event

// The onload event handler
function load_greeting () {
  alert("You are visiting the home page of \n" +
    "Pete's Pickled Peppers \n" + "WELCOME!!!");
}
```

Output:



The `unload` event is probably more useful than the `load` event. It is used to do some cleanup before a document is unloaded, as when the browser user goes on to some new document. For example, if the document opened a second browser window, that window could be closed by an `unload` event handler.

HANDLING EVENTS FROM BUTTON ELEMENTS

Buttons in a Web document provide an effective way to collect simple input from the browser user. Example:

`//radio_click.html`

```
<html>
<head>
<title> radio_click.html</title>
<script type = "text/javascript" src = "radio_click.js">
</script>

</head>
<body>
```

WEB Programming

```
<h4> Choose your favourite Director in Kannada Film Industry</h4>
<form id = "myForm" action = " ">
```

```
</p>
```

```
<label><input type = "radio" name = "dButton" value = "1" onclick =
"dChoice(1)"/> Yogaraj Bhat</label><br/>
```

```
<label><input type = "radio" name = "dButton" value = "2" onclick =
"dChoice(2)"/> Suri</label><br/>
```

```
<label><input type = "radio" name = "dButton" value = "3" onclick =
"dChoice(3)"/> Guru Prasad</label><br/>
```

```
<label><input type = "radio" name = "dButton" value = "4" onclick =
"dChoice(4)"/> Prakash</label>
```

```
</form>
```

```
</body>
```

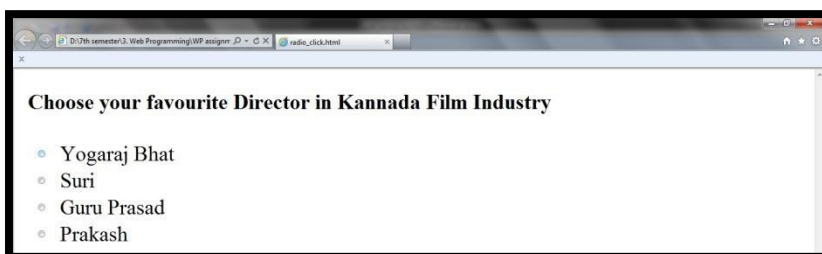
```
</html>
```

```
//radio_click.js function dChoice(ch)
{
switch(ch)
{
case 1: alert("Mungaaru Male");
break;
case 2: alert("Duniya");
break;
case 3: alert("Eddelu Manjunatha");
break;
case 4: alert("Milana");
break;
default: alert("Ooops..Invalid choice :0");
break;

}
}
```

Output:

```
case 1: alert("Mungaaru Male"); break;
case 2: alert("Duniya"); break;
case 3: alert("Eddelu Manjunatha"); break;
case 4: alert("Milana"); break;
default: alert("Ooops..Invalid choice :0");
```



HANDLING EVENTS FROM TEXT BOX AND PASSWORD ELEMENTS

Text boxes and passwords can create four different events: `blur`, `focus`, `change`, and `select`

THE FOCUS EVENT

/ nochange.html

```
<html>
<head><title>nochange.html</title>
<script type = "text/javascript" src = "nochange.js">
</script>
</head>
<body>
<form action = " ">
<h3> Non-Veg Items Order Form</h3>
<table border="border">
<tr>
<th>Item</th>
<th>Price</th>
<th>Quantity</th>
</tr>
<tr>
<th>Chicken Kabab (full)</th>
<td>Rs. 150</td>
<td><input type = "text" id = "chicken" size = "2"/></td>
</tr>
<tr>
<th>Mutton Kaima (half)</th>
<td>Rs. 250</td>
<td><input type = "text" id = "mutton" size = "2"/></td>
</tr>
<tr>
<th>Fish Fry (2 pieces)</th>
<td>Rs. 100</td>
<td><input type = "text" id = "fish" size = "2"/></td>
</tr>
</table>
<p>
<input type = "button" value = "Total Cost" onclick = "computeCost();" />
<input type = "text" size = "5" id = "cost" onfocus = "this.blur();" />
</p>
<p>
<input type = "submit" value = "Submit Order" />
<input type = "reset" value = "Clear Order Form" />
</p>
</form>
</body>
</html>
```

WEB Programming

```
//nochange.js
```

```
function computeCost()
{
var chicken = document.getElementById("chicken").value; var mutton =
document.getElementById("mutton").value; var fish =
document.getElementById("fish").value;

document.getElementById("cost").value = totalCost = chicken*150
+ mutton*250 + fish*100;
}
```

Output:

Output:

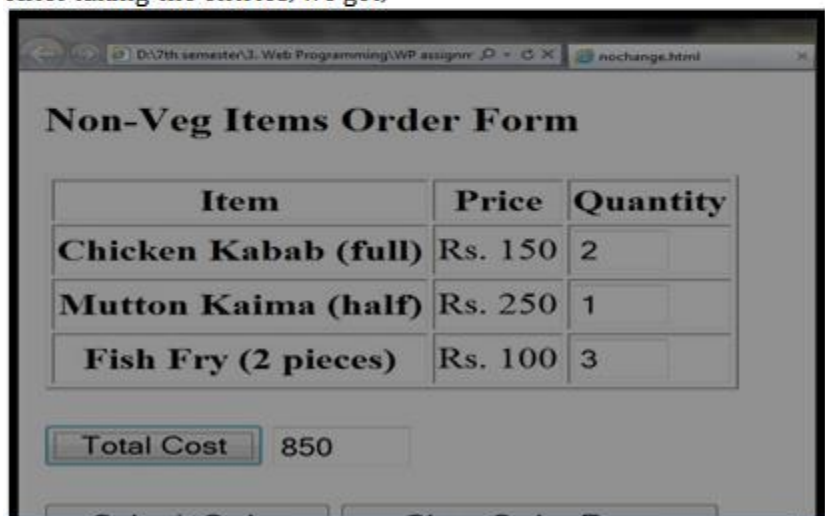


The screenshot shows a web browser window with the title 'nochange.html'. The page contains a form titled 'Non-Veg Items Order Form'. It features a table with three columns: 'Item', 'Price', and 'Quantity'. The table lists three items: 'Chicken Kabab (full)' with a price of 'Rs. 150', 'Mutton Kaima (half)' with a price of 'Rs. 250', and 'Fish Fry (2 pieces)' with a price of 'Rs. 100'. Each item has an empty quantity input field. Below the table, there is a 'Total Cost' label followed by an empty input field. At the bottom, there are two buttons: 'Submit Order' and 'Clear Order Form'.

Item	Price	Quantity
Chicken Kabab (full)	Rs. 150	<input type="text"/>
Mutton Kaima (half)	Rs. 250	<input type="text"/>
Fish Fry (2 pieces)	Rs. 100	<input type="text"/>

Total Cost

After taking the entries, we get,



The screenshot shows the same web browser window as before, but now the form is filled with data. The 'Quantity' column has values: '2' for 'Chicken Kabab (full)', '1' for 'Mutton Kaima (half)', and '3' for 'Fish Fry (2 pieces)'. The 'Total Cost' input field now displays the value '850'. The 'Submit Order' and 'Clear Order Form' buttons are still present at the bottom.

Item	Price	Quantity
Chicken Kabab (full)	Rs. 150	2
Mutton Kaima (half)	Rs. 250	1
Fish Fry (2 pieces)	Rs. 100	3

Total Cost

VALIDATING FORM INPUT

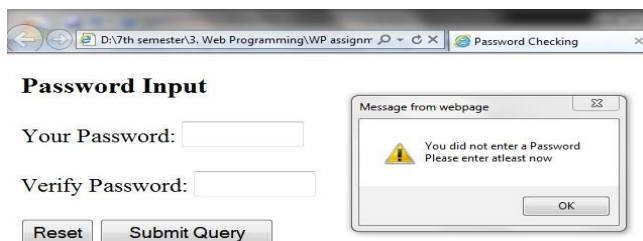
After taking the entries, we get,

- One of the common uses of JavaScript is to check the values provided in forms by users to determine whether the values are sensible.

WEB Programming

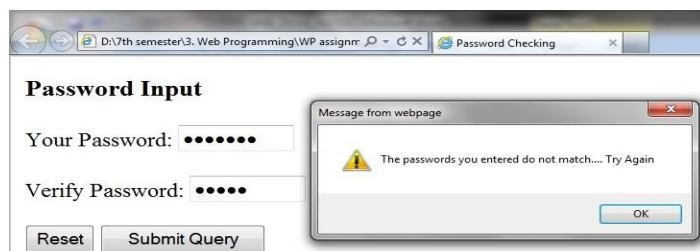
- When a user fills in a form input element incorrectly and a JavaScript event handler function detects the error, the function should produce an alert message indicating the error to the user and informing the user of the correct format for the input.
- The form in the next example includes the two password input elements, along with Reset and Submit buttons.
- The JavaScript function that checks the passwords is called either when the Submit button is pressed, using the onsubmit event to trigger the call, or when the second text box loses focus, using the blur event.
- The function performs two different tests.
 - oFirst, it determines whether the user typed the initial password (in the first input box) by testing the value of the element against the empty string. If no password has been typed into the first field, the function calls alert to produce an error message and returns false.
 - oThe second test determines whether the two typed passwords are the same. If they are different, once again the function calls alert to generate an error message and returns false.
- If they are the same, it returns true.

```
//pswd_chk.html
<html>
<head>
<title>Password Checking</title>
<script type = "text/javascript" src = "pswd_chk.js">
</script>
</head>
<body>
<h3>Password Input</h3>
<form id="myForm" action=" ">
<p>
<label>Your Password: <input type="password" id="initial"
size="10"/></label><br/><br/>
<label>Verify Password: <input type="password" id="final"
size="10"/></label><br/><br/>
<input type="reset" name="Reset"/>
<input type="submit" name="Submit"/>
</p>
</form>
<script type = "text/javascript" src = "pswd_chkr.js">
</script>
</body>
</html>
Output:
```



```
//pswd_chk.js function chkPass( )
{
var init=document.getElementById("initial"); var
fin=document.getElementById("final"); if(init.value=="")
{
alert("You did not enter a Password\n" + "Please          enter atleast now");
init.focus( ); return false;
}

if(init.value!=fin.value)
{
alert("The passwords you entered do not match    Try
Again");
init.focus( ); init.select( ); return false;
}
Else return true;
}
```



THE DOM 2 EVENT MODEL

The DOM 2 model is a modularized interface. One of the DOM 2 modules is Events, which includes several sub-modules. The ones most commonly used are HTMLEvents and MouseEvents. The interfaces and events defined by these modules are as follows:

EVENT PROPAGATION:

- A browser which understands DOM, on receiving the XHTML document from the server, creates a tree known as document tree.
- The tree constructed consists of elements of the document except the HTML
- The root of the document tree is document object itself

- The other elements will form the node of the tree
- In case of DOM2, the node which generates an event is known as target node
- Once the event is generated, it starts the propagation from root node
- During the propagation, if there are any event handlers on any node and if it is enabled then event handler is executed
- The event further propagates and reaches the target node.
- When the event handler reaches the target node, the event handler gets executed
- After this execution, the event is again re-propagated in backward direction
- During this propagation, if there are any event handlers which are enabled, will be executed.
- The propagation of the even from the root node towards the leaf node or the target node is known as capturing phase.
- The execution of the event handler on the target node is known as execution phase.
- This phase is similar to event handling mechanism in DOM – 0
- The propagation of the event from the leaf or from the target node is known as bubbling phase
- All events cannot be bubbled for ex: load and unload event
- If user wants to stop the propagation of an event, then stop propagation has to be executed.

EVENT REGISTRATION:

- In case of DOM2, the events get registered using an API known as addEventListener
- The first arg is the eventName. Ex: click, change, blur, focus
- The second arg is the event handler function that has to be executed when there is an event
- The third arg is a Boolean argument that can either take a true or false value
- If the value is true, it means event handler is enabled in capturing phase
- If the event value is off (false), then event handler is enabled at target node
- The addEventListener method will return event object to eventhandler function. The event object can be accessed using the keyword “Event”
- The address of the node that generated event will be stored in current target, which is property of event object

AN EXAMPLE OF THE DOM 2 EVENT MODEL

The next example is a revision of the validator.html document and validator.js script from previous example, which used the DOM 0 event model. Because this version uses the DOM 2 event model, it does not work with IE8.

//validator2.html

```
<html>
<head>
<title>Illustrate form input validation with DOM 2</title>
<script type = "text/javascript" src = "validator2.js">
</script>
</head>
<body>
<h3>enter your details</h3>
<form action="">
<p>
<label><input type="text" id="custName"/>Name(last name, first name, middle
initial)</label><br/><br>
<label><input type="text" id="custPhone"/>Phone (ddd-ddddddd)</label><br/><br>
<input type="reset" />
<input type="submit" id="submitButton"/>
</p>
</form>
<script type = "text/javascript" src = "validator2r.js"/>
</body>
</html>
```

//validator2.js

```
function chkName(event)
{
var myName = event.currentTarget;
var pos = myName.value.search(/^[A-Z][a-z]+, ?[A-Z][a-z]+, ?[A-Z]\.?$$/); if(pos != 0)
{
alert("The name you entered (" + myName.value + ") is not in the correct form.\n" + "The correct form is:
" + "last-name, first-name, middle-initial \n" +
"Please go and fix your name"); myName.focus();
myName.select();
}
}

function chkPhone(event)
```

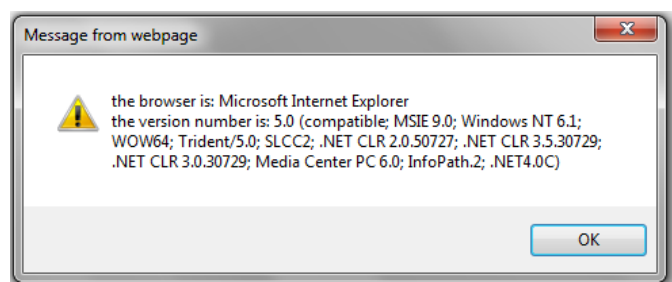
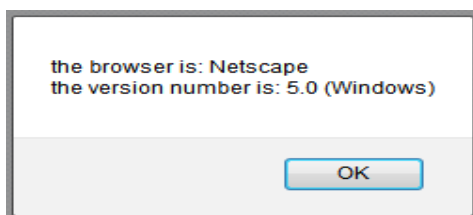
WEB Programming

```
{  
var myPhone = event.currentTarget;  
var pos = myPhone.value.search(/^\d{3}-\d{8}$/); if(pos != 0)  
{  
alert("The phone you entered (" + myPhone.value + ") is not in the correct form.\n" + "The correct form  
is: " + "ddd-ddddddd \n" +  
"Please go and fix your phone number"); myPhone.focus();  
myPhone.select();  
}  
}
```

//validator2r.js

```
var c = document.getElementById("custName");  
var p = document.getElementById("custPhone");  
c.addEventListener("change",chkName,false);  
p.addEventListener("change",chkPhone,false);
```

--	--



DOM TREE TRAVERSAL AND MODIFICATION

DOM TREE TRAVERSAL

The parentNode property has the DOM address of the parent node of the node through which it is referenced. The childNodes property is an array of the child nodes of the node through which it is referenced. The previousSibling property has the DOM address of the previous sibling node of the node through which it is referenced. The nextSibling property has the DOM address of the next sibling node of the node through which it is referenced. The firstChild and lastChild properties have the DOM addresses of the first and last child nodes, respectively, of the node through which they are referenced. The nodeType property has the type of the node through which it is referenced.

DOM TREE MODIFICATION

A number of methods allow JavaScript code to modify an existing DOM tree structure. The insertBefore(newChild, refChild) method places the newChild node before the refChild node. The replaceChild(newChild, oldChild) method replaces the oldChild node with the newChild node. The removeChild(oldChild) method removes the specified node from the DOM structure. The appendChild(newChild) method adds the given node to the end of the list of siblings of the node through which it is called.

Positioning elements

For DHTML content developers, the most important feature of CSS is the ability to use ordinary CSS style attributes to specify the visibility, size, and precise position of individual elements of a document. In order to do DHTML programming, it is important to understand how these style attributes work.

We can position elements in three ways namely

- 1. Absolute position**
- 2. Relative position**
- 3. Static position**

1. Absolute position

This value allows you to specify the position of an element relative to its containing element.

Absolutely positioned elements are positioned independently of all other elements and are not part of the flow of statically positioned elements.

An absolutely positioned element is positioned either relative to the <body> of the document or, if it is nested within another absolutely positioned element, relative to that element. This is the most commonly used positioning type for DHTML

Elements with fixed positioning do not scroll with the rest of the document and thus can be used to achieve frame-like effects. Like absolutely positioned elements, fixed-position elements are independent of all others and are not part of the document flow.

```
<p style="position: absolute; left:100px;top:20px">  
Hi girls and boys  
</p>
```

2. Relative positions

When the position attribute is set to relative, an element is laid out according to the normal flow, and its position is then adjusted relative to its position in the normal flow.

The space allocated for the element in the normal document flow remains allocated for it, and the elements on either side of it do not close up to fill in that space, nor are they "pushed away" from the new position of the element.

Relative positioning can be useful for some static graphic design purposes, but it is not commonly used for DHTML effects.

Relative positions are used to create subscripts and superscripts by placing the values to be raised or lower in ,<div> tags.

```
<div style="position: absolute; left: 100px; top: 100px;">
```

3. Static position

This is the default value and specifies that the element is positioned according to the normal flow of document content (for most Western languages, this is left to right and top to bottom.)

Statically positioned elements are not DHTML elements and cannot be positioned with the top, left, and other attributes. To use DHTML positioning techniques with a document element, you must first set its position attribute to one of the other three values.

XML

What Is XML ?

XML (Extensible Markup Language) is a data description language. XML has many strengths, which account for its great popularity and large availability:

- It has a simple text syntax based on elements identified by start and end tags. One can read or create XML easily with just a simple text editor.
- It supports pluggable grammars (called DTDs or XSD schemas), which enable the creation of tag vocabularies for any domain.
- It is very strictly standardized by the W3C (World Wide Web Consortium), which makes XML the ideal vehicle to exchange data among heterogeneous systems.

SYNTAX OF XML AND CREATING XML DOCUMENT

- ★ XML imposes two distinct levels of syntax:
 - There is a general low level syntax that is appreciable on all XML documents
 - The other syntactic level is specified by DTD (Document Type Definition) or XML schemas.
- ★ The DTDs and XML schemas specify a set of tag and attribute that can appear in a particular document or collection of documents.
- ★ They also specify the order of occurrence in the document.
- ★ The XML documents consist of data elements which form the statements of XML document.
- ★ The XML document might also consist of markup declaration, which act as instructions to the XML parser
- ★ All XML documents begin with an XML declaration. This declaration identifies that the document is an XML document and also specifies version number of XML standard.
- ★ It also specifies encoding standard.

<?xml version = "1.0" encoding = "utf-8"?>

- ★ Comments in XML is similar to HTML
- ★ XML names are used to name elements and attributes.
- ★ XML names are case-sensitive.
- ★ There is no limitation on the length of the names.
- ★ All XML document contains a single root element whose opening tag appears on first line of the code
- ★ All other tags must be nested inside the root element
- ★ As in case of XHTML, XML tags can also have attributes
- ★ The values for the attributes must be in single or double quotation

Example:

1. `<?xml version = "1.0" encoding = "utf-8"?>`

```
<student>
  <name>Santhosh B S</name>
  <usn>1RN10CS090</usn>
</student>
```

2. Tags with attributes

The above code can be also written as

```
<student name = "Santhosh B S" usn = "1RN10CS090">
</student>
```

XML DOCUMENT STRUCTURE

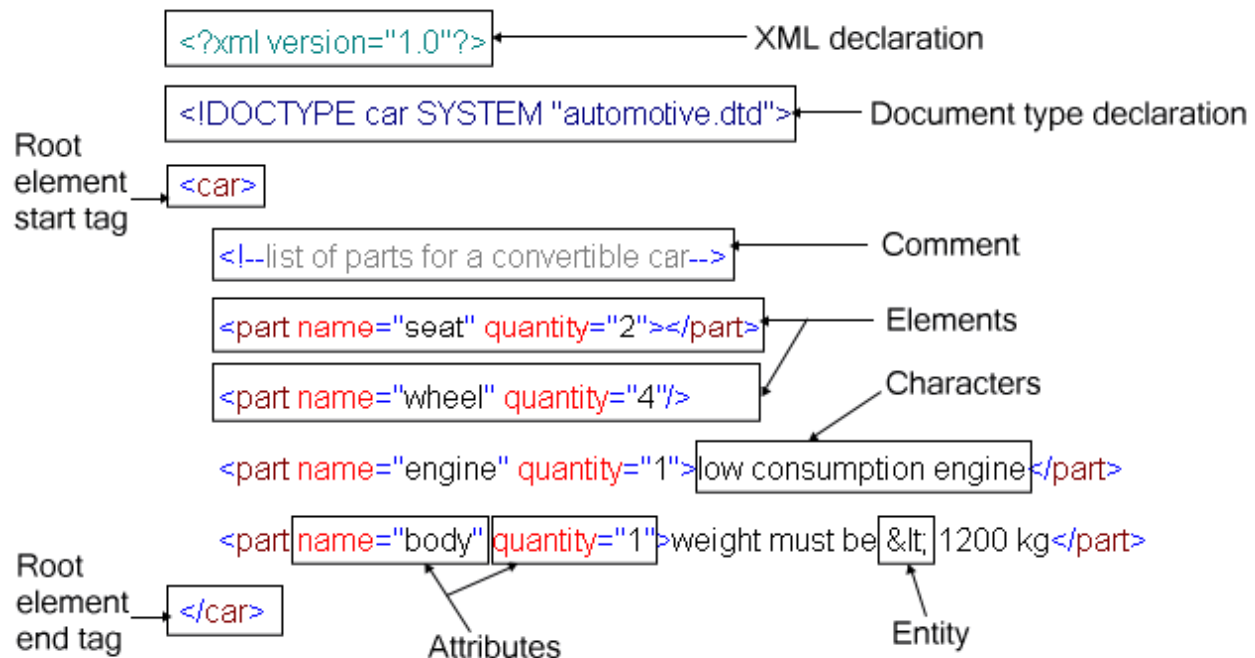
- ♥ An XML document often consists of 2 files:
 - One of the document – that specifies its tag set
 - The other specifies the structural syntactic role and one that contains a style sheet to describe how content of the document is to be printed
- ☐ The structural roles are given as either a DTD or an XML schema
- ☐ An XML document consists of logically related collection of information known as entities
- ☐ The *document entity* is the physical file that represent the document itself
- ☐ The document is normally divided into multiple entities.
- ☐ One of the advantage dividing document into multiple entities is managing the document becomes simple

Many documents include information that cannot be represented as text. Ex: images

- ☐ Such information units are stored as binary data
- ☐ These binary data must be a separate unit to be able to include in XML document
- ☐ These entities are called as *Binary entities*
- ☐ When an XML processor encounters the name of a non-binary entity in a document, it replaces the name with value it references
- ☐ Binary entities can be handled only by browsers
- ☐ XML processor or parsers can only deal with text
- ☐ Entity names can be of any length. They must begin with a letter, dash or a colon
- ☐ A reference to an entity is its name with a prepended ampersand and an appended semicolon
- ☐ Example: if `stud_name` is the name of entity, `&stud_name;` is a reference to it
- ☐ One of the use of entities is to allow characters used as markup delimiters to appear as themselves
- ☐ The entity references are normally placed in CDATA section
- ☐ Syntax: `<![CDATA[content]]>`
- ☐ For example, instead of
The last word of the line is >>> here <<<.
the following could be used:
`<![CDATA[The last word of the line is >>> here <<<]]>`

Anatomy of an XML Document

XML documents are composed of elements, delimited by a start tag (of the form `<element_name>`) and an end tag (of the form `</element_name>`). Elements can contain either other elements or free text. Every XML document has one and only one root element, which contains all the other elements. The following XML sample is provided to illustrate the anatomy of an XML document.



The following syntactic constructs are the most common in XML documents.

The XML declaration

It identifies the document as XML. Note the mandatory XML version number (1.0) and the description of the encoding (UTF-8) used by the document. XML documents can use any encoding, provided that the XML parser used to process the document knows how to deal with the encoding.

The document type declaration

It identifies the DTD (tag vocabulary) used by the XML document and the tag name for the root element ("car"). This vocabulary is stored in an external file with the .dtd extension (automotive.dtd), though it can also be in-lined in the document. The document type declaration is optional and tends to be gradually replaced by another equivalent but more powerful mechanism, called XSD schemas.

The root element

An XML has one and only one root element (called "car" here).

Elements

An element is identified by a start tag and an end tag. If the element has no sub-elements, the syntax can optionally be abbreviated to just one tag, as for the "wheel" element. Notice the trailing '/' in this case.

Attributes

The start tag can contain attributes, which are (name, value) pairs qualifying the element. Attribute names are unique within the tag scope. Attribute values appear within quotes. "body" is the value of the attribute called "name" of the "part" element.

Characters

XML elements can also contain free text.

Entities

XML use entities to escape reserved characters or specify characters not supported by the document code page. The "<" entity is use here to escape the "<" reserved character.

Comments

XML documents can be annotated with comments.

Other syntactic constructs (CDATA sections, processing instructions, etc.) can occasionally appear in XML documents. For a complete description of the XML syntax, please refer to

XML DTDs

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 linked 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*.

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>
```

Let us go through the above code –

Start Declaration – Begin the XML declaration with the 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 (J>). 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 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**.

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:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```



```
<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 –

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" />
```

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"/>
```

XSLT STYLE SHEETS

The eXtensible Stylesheet Language (XSL) is a family of recommendations for defining the presentation and transformations of XML documents.

It consists of three related standards:

- o XSL Transformations (XSLT),
- o XML Path Language (XPath), and

- o XSL Formatting Objects (XSL-FO).

XSLT style sheets are used to transform XML documents into different forms or formats, perhaps using different DTDs.

One common use for XSLT is to transform XML documents into XHTML documents, primarily for

display. In the transformation of an XML document, the content of elements can be moved, modified, sorted, and converted to attribute values, among other things.

- ☐ XSLT style sheets are XML documents, so they can be validated against DTDs.
- ☐ They can even be transformed with the use of other XSLT style sheets.
- ☐ The XSLT standard is given at <http://www.w3.org/TR/xslt>.
- ☐ XPath is a language for expressions, which are often used to identify parts of XML documents, such as specific elements that are in specific positions in the document or elements that have particular attribute values.
- ☐ XPath is also used for XML document querying languages, such as XQL, and to build new XML document

structures with XPointer. The XPath standard is given at <http://www.w3.org/TR/xpath>.

OVERVIEW OF XSLT

- ☐ XSLT is actually a simple functional-style programming language.
- ☐ Included in XSLT are functions, parameters, names to which values can be bound, selection constructs, and conditional expressions for multiple selection.
- ☐ XSLT processors take both an XML document and an XSLT document as input. T

- The XSLT document is the program to be executed; the XML document is the input data to the program.
- Parts of the XML document are selected, possibly modified, and merged with parts of the XSLT document to form a new document, which is sometimes called an XSL document.
- The transformation process used by an XSLT processor is shown in Figure 7.5.

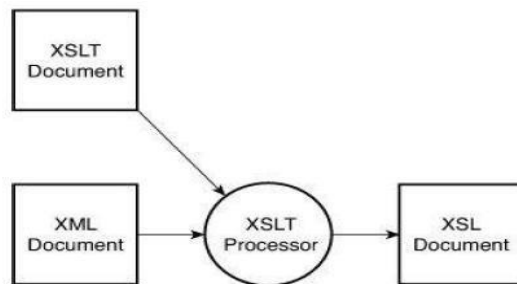


Figure 7.5 XSLT processing

An XSLT document consists primarily of one or more templates.

- Each template describes a function that is executed whenever the XSLT processor finds a match to the template's pattern.
- One XSLT model of processing XML data is called the template-driven model, which works well when the data consists of multiple instances of highly regular data collections, as with files containing records.
- XSLT can also deal with irregular and recursive data, using template fragments in what is called the data-driven model.
- A single XSLT style sheet can include the mechanisms for both the template- and data-driven models

XSL TRANSFORMATIONS FOR PRESENTATION

Consider a sample program:

//6b.xml

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<?xml-stylesheet type="text/xsl" href="6b.xsl"?>
```

```
<vtu>
```

```
<student>
  <name>Santhosh B S</name>
  <usn>1RN10CS090</usn>
  <collegeName>RNSIT</collegeName>
  <branch>CSE</branch>
  <year>2010</year>
  <email> santhosh.b.suresh@gmail.com </email>
</student>
<student>
  <name>Akash Bangera</name>
  <usn>1RN10CS003</usn>
  <collegeName>RNSIT</collegeName>
  <branch>CSE</branch>
  <year>2010</year>
  <email>akash.bangera@gmail.com</email>
</student>
<student>
  <name>Manoj Kumar</name>
  <usn>1RN10CS050</usn>
  <collegeName>RNSIT</collegeName>
  <branch>CSE</branch>
  <year>2010</year>
  <email>manoj.kumar@gmail.com</email>
</student>
</vtu>
```

An XML document that is to be used as data to an XSLT style sheet must include a processing instruction to inform the XSLT processor that the style sheet is to be used. The form of this instruction is as follows:

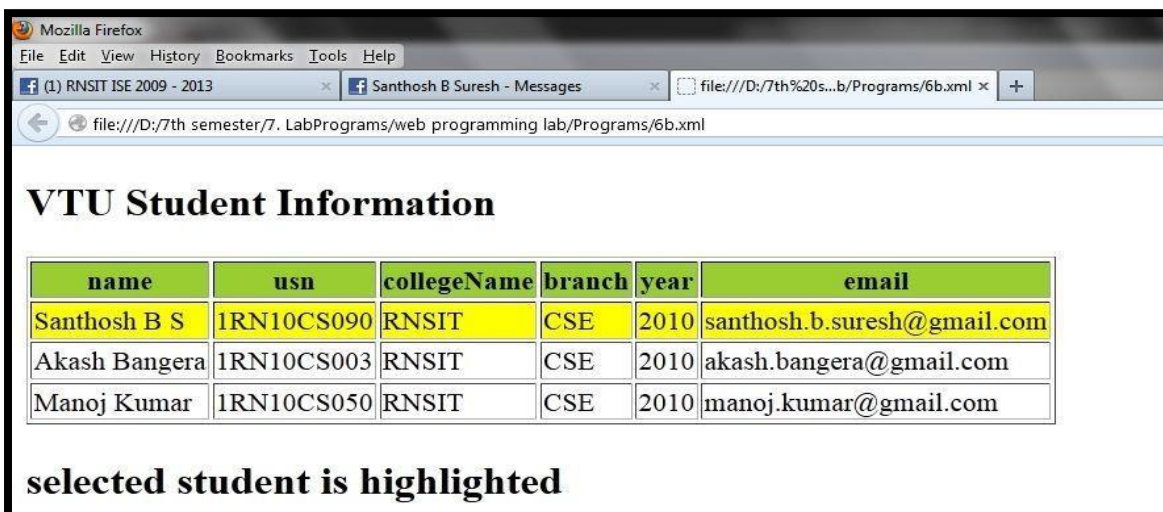
```
<?xml-stylesheet type = "text/xsl" href =
    "XSL_stylesheet_name" ?>
```

//6b.xsl

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<body>
<h2>VTU Student Information</h2>
<table border="1">
<tr bgcolor="#99cd32">
<th>name</th>
<th>usn</th>
<th>collegeName</th>
<th>branch</th>
<th>year</th>
<th>email</th>
</tr>
```

WEB Programming

```
<xsl:for-each select="vtu/student">
<xsl:choose>
<xsl:when test="name = 'Santhosh B S'">
<tr bgcolor="yellow">
<td><xsl:value-of select="name"/></td>
<td><xsl:value-of select="usn"/></td>
<td><xsl:value-of select="collegeName"/></td>
<td><xsl:value-of select="branch"/></td>
<td><xsl:value-of select="year"/></td>
<td><xsl:value-of select="email"/></td>
</tr>
</xsl:when>
<xsl:otherwise>
<tr >
<td><xsl:value-of select="name"/></td>
<td><xsl:value-of select="usn"/></td>
<td><xsl:value-of select="collegeName"/></td>
<td><xsl:value-of select="branch"/></td>
<td><xsl:value-of select="year"/></td>
<td><xsl:value-of select="email"/></td>
</tr>
</xsl:otherwise>
</xsl:choose>
</xsl:for-each>
</table>
<h2>selected student is highlighted</h2>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```



An XSLT style sheet is an XML document whose root element is the special-purpose element `stylesheet`. The `stylesheet` tag defines namespaces as its

attributes and encloses the collection of elements that defines its transformations. It also identifies the document as an XSLT document.

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

In many XSLT documents, a template is included to match the root node of the XML document.

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

In many cases, the content of an element of the XML document is to be copied to the output document. This is done with the `value-of` element, which uses a `select` attribute to specify the element of the XML document whose contents are to be copied.

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

The `select` attribute can specify any node of the XML document. This is an advantage of XSLT formatting over CSS, in which the order of data as stored is the only possible order of display

XML PROCESSORS

The XML processor takes the XML document and DTD and processes the information so that it may then be used by applications requesting the information. The processor is a software module that reads the XML document to find out the structure and content of the XML document. The structure and content can be derived by the processor because XML documents contain self-explanatory data.

THE PURPOSES OF XML PROCESSORS

- ★ First, the processor must check the basic syntax of the document for well-formedness.
- ★ Second, the processor must replace all references to entities in an XML document with their definitions.
- ★ Third, attributes in DTDs and elements in XML schemas can specify that their values in an XML document have default values, which must be copied into the XML document during processing.
- ★ Fourth, when a DTD or an XML schema is specified and the processor includes a validating parser, the structure of the XML document must be checked to ensure that it is legitimate.

THE SAX APPROACH

- The Simple API for XML (SAX) approach to processing is called event processing.
- The processor scans the XML document from beginning to end.
- Every time a syntactic structure of the document is recognized, the processor signals an event to the application by calling an event handler for the particular structure that was found.
- The syntactic structures of interest naturally include opening tags, attributes, text, and closing tags.
- The interfaces that describe the event handlers form the SAX API.

THE DOM APPROACH

- The Document Object Model (DOM) is an application programming interface (API) for HTML and XML documents.
- It defines the logical structure of documents and the way a document is accessed and manipulated
- Properties of DOM
 - Programmers can build documents, navigate their structure, and add, modify, or delete elements and content.
 - Provides a standard programming interface that can be used in a wide variety of environments and applications.
 - structural isomorphism.
- The DOM representation of an XML document has several advantages over the sequential listing provided by SAX parsers.
- First, it has an obvious advantage if any part of the document must be accessed more than once by the application.
- Second, if the application must perform any rearrangement of the elements of the document, that can most easily be done if the whole document is accessible at the same time.
- Third, accesses to random parts of the document are possible.
- Finally, because the parser sees the whole document before any processing takes place, this approach avoids any processing of a document that is later found to be invalid.

WEB SERVICES

A Web service is a method that resides and is executed on a Web server, but that can be called from any computer on the Web. The standard technologies to support Web services are WSDL, UDDI, SOAP, and XML.

WSDL - It is used to describe the specific operations provided by the Web service, as well as the protocols for the messages the Web service can send and receive.

UDDI - also provides ways to query a Web services registry to determine what specific services are available.

SOAP - was originally an acronym for Standard Object Access Protocol, designed to describe data objects.

XML - provides a standard way for a group of users to define the structure of their data documents, using a subject-specific mark-up language.

