[Unit 1: Introduction]

Web Technology (CSC-353)

Jagdish Bhatta

Central Department of Computer Science & Information Technology

Tribhuvan University

Introduction:

Web technologies related to the interface between web servers and their clients. This information includes markup languages, programming interfaces and languages, and standards for document identification and display. In general web technology incorporates tools and techniques for web development.

Web Development is a broad term for the work involved in developing a web site for World Wide Web. This can include ***web design, web content development, client liaison, client-side/server-side scripting, web server and network security configuration, and e- commerce development.*** However, among web professionals, "web development" usually refers to the main non-design aspects of building web sites: writing markup and coding. Web development can range from developing the simplest static single page of plain text to the most complex web-based internet applications, electronic businesses, or social network services.

Web design is a broad term used to encompass the way that content (usually hypertext or hypermedia) is delivered to an end-user through the World Wide Web, using a web browser or other web-enabled software is displayed. The intent of web design is to create a website—a collection of online content including documents and applications that reside on a web servers. A website may include text, images, sounds and other content, and may be interactive.

For the typical web sites, the basic aspects of design are:

- **The *content:*** the substance, and information on the site should be relevant to the site and should target the area of the public that the website is concerned with.

- **The *usability:*** the site should be user-friendly, with the interface and navigation simple and reliable.

- **The *appearance:*** the graphics and text should include a single style that flows throughout, to show consistency. The style should be professional, appealing and relevant.

- **The *structure:*** of the web site as a whole.

Internet and its Evolution:

Internet is a short form of the technical term ***internetwork***, the result of interconnecting computer networks with special gateways or routers. The Internet is also often referred to as *the Net*. The Internet is a massive network of networks, a networking infrastructure. It connects millions of computers together globally, forming a network in which any computer can communicate with any other computer as long as they are both connected to the Internet. Information that travels over the Internet does so via a variety of languages known as protocols. **The Internet is loosely connected compared with the randomized graph.**

The Internet is a globally distributed network comprising many voluntarily interconnected autonomous networks. It operates without a central governing body. However, to maintain interoperability, all technical and policy aspects of the underlying core infrastructure and the principal name spaces are administered by the **Internet Corporation for Assigned Names and Numbers (ICANN).**

The **history of the Internet** starts in the 1950s and 1960s with the development of computers. This began with point-to-point communication between mainframe computers and terminals, expanded to point-to-point connections between computers and then early research into packet switching.

Since the mid-1990s the Internet has had a drastic impact on culture and commerce, including the rise of near instant communication by electronic mail, instant messaging, Voice over Internet Protocol (VoIP) "phone calls", two-way interactive video calls, and the World Wide Web with its discussion forums, blogs, social networking, and online shopping sites. **(Just go through the brief history yourself)**

World Wide Web:

WWW is a system of interlinked hypertext documents accessed via the Internet. The World Wide Web, or simply Web, is a way of accessing information over the medium of the Internet. It is an information-sharing model that is built on top of the Internet. The Web uses the HTTP protocol, only one of the languages spoken over the Internet, to transmit data. Web services, which use HTTP to allow applications to communicate in order to exchange business logic, use the Web to share information. The Web also utilizes browsers, such as Internet Explorer or Firefox, to access Web documents called Web pages that are linked to each other via hyperlinks. Web documents also contain graphics, sounds, text and video.

The Web is one of the services that runs on the Internet. It is a collection of textual documents and other resources, linked by hyperlinks and URLs, transmitted by web browsers and web servers. The Web is just one of the ways that information can be disseminated over the Internet, so the Web is just a portion of the Internet. In short, the Web can be thought of as an application "running" on the Internet

What is Hypertext?

Hypertext provides the links between different documents and different document types. In a hypertext document, links from one place in the document to another are included with the text. By selecting a link, you are able to jump immediately to another part of the document or even to a different document. In the WWW, links can go not only from one document to another, but from one computer to another

World Wide Consortium:

The **World Wide Web Consortium** (**W3C**) is the main international standards organization for the World Wide Web. W3C was created to ensure compatibility and agreement among industry members in the adoption of new standards. Prior to its creation, incompatible versions of HTML were offered by different vendors, increasing the potential for inconsistency between web pages. The consortium was created to get all those vendors to agree on a set of core principles and components which would be supported by everyone.

Web Page:

A **web page** is a document or information resource that is suitable for the World Wide Web and can be accessed through a web browser and displayed on a monitor or mobile device. This information is usually in HTML or XHTML format, and may provide navigation to other web pages via hypertext links. Web pages frequently subsume other resources such as style sheets, scripts and images into their final presentation.

Web pages may be retrieved from a local computer or from a remote web server. The web server may restrict access only to a private network, e.g. a corporate intranet, or it may publish pages on the World Wide Web. Web pages are requested and served from web servers using Hypertext Transfer Protocol (HTTP).

Web pages may consist of files of static text and other content stored within the web server's file system (static web pages), or may be constructed by server-side software when they are requested (dynamic web pages). Client-side scripting can make web pages more responsive to user input once on the client browser.

Web Site:

A **website** or simply **site**, is a collection of related web pages containing images, videos or other digital assets. A website is hosted on at least one web server, accessible via a network such as the Internet or a private local area network through an Internet address known as a Uniform Resource Locator. All publicly accessible websites collectively constitute the World Wide Web. *Web sites can be static or dynamic.*

Static Website:

A static website is one that has web pages stored on the server in the format that is sent to a client web browser. It is primarily coded in Hypertext Markup Language, HTML.

Simple forms or marketing examples of websites, such as *classic website*, a *five-page website* or a *brochure website* are often static websites, because they present pre-defined, static information to the user. This may include information about a company and its products and services via text, photos, animations, audio/video and interactive menus and navigation.

This type of website usually displays the same information to all visitors. Similar to handing out a printed brochure to customers or clients, a static website will generally provide consistent, standard information for an extended period of time. Although the website owner may make updates periodically, it is a manual process to edit the text, photos and other content and may require basic website design skills and software.

In summary, visitors are not able to control what information they receive via a static website, and must instead settle for whatever content the website owner has decided to offer at that time.

Dynamic Website:

A dynamic website is one that changes or customizes itself frequently and automatically, based on certain criteria.

Dynamic websites can have two types of dynamic activity: Code and Content. Dynamic code is invisible or behind the scenes and dynamic content is visible or fully displayed.

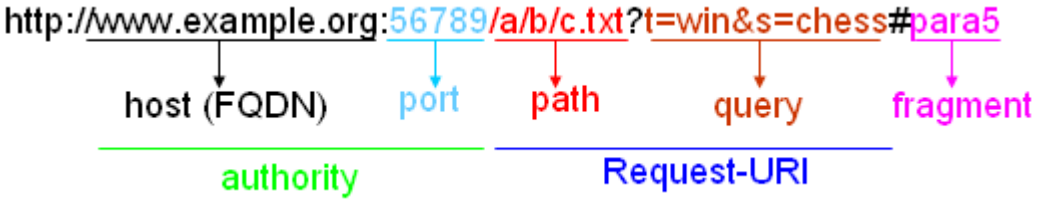
The first type is a web page with dynamic code. The code is constructed dynamically on the fly using active programming language instead of plain, static HTML.

The second type is a website with dynamic content displayed in plain view. Variable content is displayed dynamically on the fly based on certain criteria, usually by retrieving content stored in a database

Domain Names, DNS, and URLs:

- IP addresses are not convenient for users to remember easily. So an IP address can be represented by a natural language convention called a **domain name**
    - **Domain name system (DNS)** translates domain names into IP addresses. DNS is the "phone book" for the Internet, it maps between host names and IP addresses.
- A **uniform resource locator (URL)**, which is the address used by a Web browser to identify the location of content on the Web, also uses a domain name as part of the URL.

- **Syntax: scheme: scheme-depend-part.** Example: In http://www.example.com/, the scheme is http.



HTTP:

- HTTP is based on the request-response communication model:
  - Client sends a request
  - Server sends a response
  - HTTP is a stateless protocol: where the protocol does not require the server to remember anything about the client between requests.

- Normally implemented over a TCP connection (80 is standard port number for HTTP)

- The following is the typical browser-server interaction using HTTP:
  1. User enters Web address in browser
  2. Browser uses DNS to locate IP address
  3. Browser opens TCP connection to server
  4. Browser sends HTTP request over connection
  5. Server sends HTTP response to browser over connection
  6. Browser displays body of response in the client area of the browser window

Client/Server Computing:

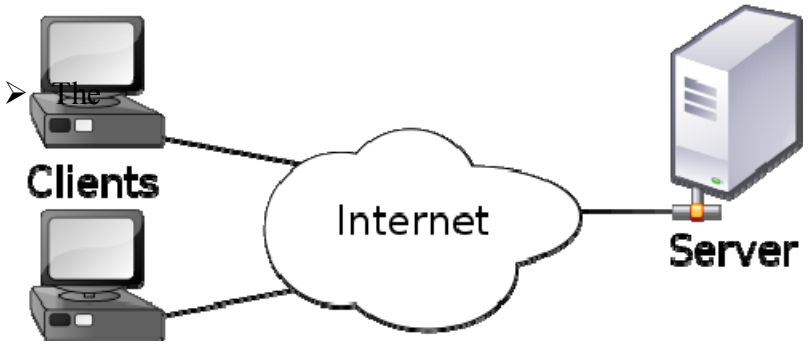
- A model of computing in which powerful personal computers are connected in a network together with one or more servers

- **Client** is a powerful personal computer that is part of a network; service requester

- **Server** is a networked computer dedicated to common functions that the client computers on the network need; service provider

- Web is based on client/server technology. Web servers are included as part of a larger package of internet and intranet related programs for serving e-mail, downloading requests for FTP files and building and publishing web pages. Typically the e-commerce customer is the client and the business is the server. In the client/ server model single machine can be both client and the server The client/ server model utilises a database server in which RDBMS user queries can be answered directly by the server.

- The client/ server



architecture reduces network traffic by providing a query response to the user rather than transferring total files. The client/ server model improves multi-user updating through a graphical user interface (GUI) front end to the shared database. In client/ server architectures client and server typically communicate through statements made in structured query language (SQL).

Fig: Client/ Server Model

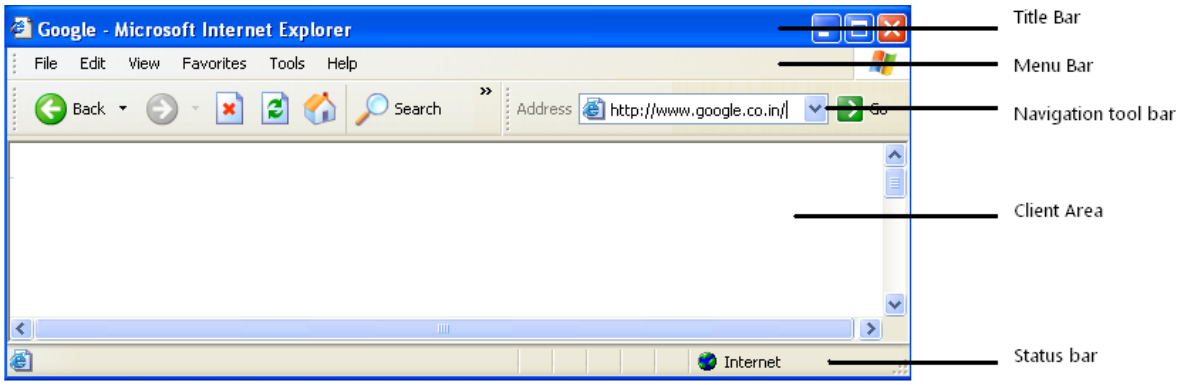
Web Clients:

It typically refers to the Web browser in the user's machine. It is a software application for retrieving, presenting, and traversing information resources on the web server. It is used to create a HTTP request message and for processing the HTTP response message.

User agent: Any web client is designed to directly support user access to web servers is known as user agent. Web browsers can run on desktop or laptop computers. Some of the browsers are: Internet Explorer, Mozilla, FireFox, Chrome, Safari, Opera, Netscape Navigator.

Web Browsers:

Browsers are software programs that allow you to search and view the many different kinds of information that's available on the World Wide Web. The information could be web sites, video or audio information.



Status Bar: You will find the status bar at the very bottom of your browser window. It basically tells you what you are doing at the moment. Mainly, it shows you load speed and the URL address of whatever your mouse is hovering over.

Title Bar: You will find this bar at the absolute top of your browser and in will be the colour blue for the major browsers. The purpose of the Title bar is to display the title of the web page that you are currently viewing.

Menu Bar: The menu bar contains a set of dropdown menus

Navigational Tool: A bar contains standard push button controls that allow the user to return to a previously viewed page, to reverse and refresh the page, to display the home page and to print the page etc.

Toolbar Icons: You will find the Toolbar directly under the Title Bar. The Toolbar is where you will find the back button, home button and the refresh button etc.

Client Area: It is a display window which is the space in which you view the website.

Scroll Bars: The Scroll bars, usually located to the right of the Display Window, allows you to "scroll" (move down or up the web page) so you can view information that is below or above what is currently in the Display Window.

Web Servers:

Basic functionality:

- It receives HTTP request via TCP
- It maps Host header to specific virtual host (one of many host names sharing an IP address)
- It maps Request-URI to specific resource associated with the virtual host
  - File: Return file in HTTP response
  - Program: Run program and return output in HTTP response

- It maps type of resource to appropriate MIME type and use to set Content-Type header in HTTP response
- It Logs information about the request and response
- All e-commerce site require basic Web server software to answer requests from customers like ;
  - **Apache**
    - Leading Web server software (47% of market)
    - Works with UNIX, Linux , Windows OSs
  - **Microsoft's Internet Information Server (IIS)**
    - Second major Web server software (25% of market)
    - Windows-based

Client-Side Scripting:

- Client-side scripting generally refers to writing the class of computer programs (scripts) on the web that are executed at *client-side*, by the user's web browser, instead of *server-side* (on the web server). Usually scripts are embedded in the HTML page itself.

- JavaScript , VBScript, Jscript, Java Applets etc. are the examples of client side scripting technologies. JavaScript is probably the most widely used client-side scripting language.

- Client-side scripts have greater access to the information and functions available on the user's browser, whereas server-side scripts have greater access to the information and functions available on the server. Upon request, the necessary files are sent to the user's computer by the web server (or servers) on which they reside. The user's web browser executes the script, then displays the document, including any visible output from the script.

- Client-side scripts may also contain instructions for the browser to follow in response to certain user actions, (e.g., clicking a button). Often, these instructions can be followed without further communication with the server.

Server-Side Scripting:

- Includes writing the applications executed by the server at run-time to process client input or generate document in response to client request. So server side script consists the directives embedded in Web page for *server* to process before passing page to requestor.

- It is usually used to provide interactive web sites that interface to databases or other data stores.

- This is different from client-side scripting where scripts are run by the viewing web browser, usually in JavaScript. The primary advantage to server-side scripting is

the ability to highly customize the response based on the user's requirements, access rights, or queries into data stores.

- PHP, JSP, ASP…. etc, are the server side scripting technologies.

Web 2.0:

The term **Web 2.0** is associated with web applications that facilitate participatory *information sharing, interoperability, user-centered design, and collaboration* on the World Wide Web. A Web 2.0 site allows users to interact and collaborate with each other in a social media dialogue as creators of user-generated content in a virtual community, in contrast to websites where users are limited to the passive viewing of content that was created for them. Examples of Web 2.0 include *social networking sites, blogs, wikis, video sharing sites, hosted services, web applications.*

I think following portion you have studied in Data Communication (So Self Study)

SMTP:

Simple Mail Transfer Protocol (**SMTP**) is an Internet standard for electronic mail (e- mail) transmission across Internet Protocol (IP) networks.

POP:

In computing, the **Post Office Protocol** (**POP**) is an application-layer Internet standard protocol used by local e-mail clients to retrieve e-mail from a remote server over a TCP/IP connection.

HTML

HTML stands for **hypertext markup language**. It is not a programming language. A markup language specifies the *layout and style* of a document. A markup language consists of a set of **markup tags**. HTML uses markup tags to describe web pages. HTML tags are keywords surrounded by **angle brackets** like <html>. Most HTML tags normally come in pairs like and . The first tag is called the **start tag** (or **opening tag**) and the second tag is called the **end tag** (or **closing tag**). HTML documents describe Web pages. HTML documents contain HTML tags and plain text. HTML documents are also called Web pages. A web browser read HTML documents and displays them as Web pages. The browser does not display the HTML tags, but uses the tags to interpret the content of the page. A simple HTML document is given below:

<html>

```
<head>
        <title>This is my first web page</title>
</head>
<body>
        <h1>My first heading</h1>
        <p>My first paragraph</p>
</body>
 </html>
```

Save this page with **.html** or **.htm** extension. However, it is good practice to use **.htm** extension.

HTML Elements

HTML documents are defined by HTML elements. An HTML element is everything from the start tag to the end tag. For example, My first paragraph . An HTML element consists of start tag, end tag, and element content. The element content is everything between the start tag and end tag. Empty elements are closed in the start tag. Most HTML elements can have attributes. For example, **src** attribute of **img** tag.

HTML Attributes

Attributes provide additional information about HTML elements. Attributes are always specified in the start tag. Attributes come in name/value pair like name = "value". For example, HTML links are defined with **<a>** tag and the link address is provided as an attribute **href** like **cdcsit**.

Note: Always quote attribute values and use lowercase attributes.

HTML Headings

HTML headings are defined with the <h1> to <h6> tags. <h1> displays largest text and <h6> smallest. For example, <h1>My first heading</h1>.

HTML Paragraphs

HTML paragraphs are defined with tag. For example, My first paragraph .

HTML Rules (Lines)

We use <hr /> tag to create horizontal line.

HTML Comments

We use comments to make our HTML code more readable and understandable. Comments are ignored by the browser and are not displayed. Comments are written between <!-- and --->. For example, <!-- This is a comment -->.

HTML Line Breaks

If you want a new line (line break) without starting a new paragraph, use <br /> tag.

HTML Formatting Tags

We use different tags for formatting output. For example, <b> is used for bold and <i> is used for italic text. Some other tags are <big>, <small>, <sup>, <sub> etc.

HTML Styles

It is a new HTML attribute. It introduces CSS to HTML. The purpose of style attribute is to provide a common way to style all HTML elements. For example, <body style =

“background-color:yellow”>, <p style = “font-family:courier new; color:red; font-size:20px”>, <h1 style = “text-align:center”> etc.

HTML Links

A link is the address to a resource on the web. HTML links are defined using an anchor tag (**<a>**). We can use this tag to point to a resource (an HTML page, an image, a sound file, a movie etc.) and an address inside a document.

We can use **href** attribute to define the link address. For example, cdcsit.

We can use **target** attribute to define where the linked document will be opened. For example, cdcsit will open the document in a new window.

We can use **name** attribute to define a named anchor inside a HTML document. Named anchor are invisible to the reader. For example, Any content defines a named anchor and we use the syntax Any content to link to the named anchor.
We can also use named anchor to link to some content within another document. For example, Jump to the Useful Tips section.

HTML Images

HTML images are defined with **<img>** tag. To display an image on a page, you need to use the **src** attribute. We can also use **width** and **height** attributes with img tag. For example, .

We can use alt attribute to define an alternate text for an image. For example, . The "alt" attribute tells the reader what he or she is missing on a page if the browser can't load images. The browser will then display the alternate text instead of the image. It is a good practice to include the "alt" attribute for each image on a page, to improve the display and usefulness of your document for people who have text-only browsers.

HTML Tables

Tables are defined with the <table> tag. A table is divided into rows (with the <tr> tag), and each row is divided into data cells (with the <td> tag). The letters td stands for "table data," which is the content of a data cell. A data cell can contain text, images, lists, paragraphs, forms, horizontal rules, tables, etc. For example,

```
<table border="1">
<tr>
<td>row 1, cell 1</td>
<td>row 1, cell 2</td>
```

```
</tr>
<tr>
<td>row 2, cell 1</td>
<td>row 2, cell 2</td>
</tr>
</table>
```

Output:

row 1, cell 1	row 1, cell 2
row 2, cell 1	row 2, cell 2

We use border attribute to display table with border as shown in the above example. Headings in a table are defined with **<th>** tag. For example,

```
<table border="1">
<tr>
<th>Heading</th>
<th>Another Heading</th>
</tr>
<tr>
<td>row 1, cell 1</td>
<td>row 1, cell 2</td>
</tr>
<tr>
<td>row 2, cell 1</td>
<td>row 2, cell 2</td>
</tr>
</table>
```

Output:

Heading	Another Heading
row 1, cell 1	row 1, cell 2
row 2, cell 1	row 2, cell 2

We can use <caption> tag inside a <table> to display caption for a table. We can define table cells that span more than one row or one column using **colspan** and **rowspan** attributes respectively. For example, <td colspan = "2">Data</td>. We can use **cellpadding** and **cellspacing** attributes to create white space between the cell content and its borders, and to increase the distance between cells respectively. For example, <table border="1" cellpadding="10"> and <table border="1" cellspacing="10">. We can use align attribute to align the contents of a cell. For example, <td align = "left">Data</td>.

HTML Lists

HTML supports *ordered*, *unordered* and *definition lists*. Ordered lists items are marked with numbers, letter etc. We use **<ol>** tag for ordered list and each list item starts with **<li>** tag. For example,

```
<ol type="A">
        <li>Apples</li>
  <li>Bananas</li>
  <li>Lemons</li>
  <li>Oranges</li>
```

```
</ol>
```

Output:

A. Apples
B. Bananas
C. Lemons
D. Oranges

If we do not use type attribute, items are marked with numbers. We use **type = "a"** for lowercase letters list, **type = "I"** for roman numbers list, and **type = "i"** for lowercase numbers list.

Unordered lists items are marked with bullets. We use **<ul>** tag for unordered list and each list item starts with **<li>** tag. For example,

```
<ul type="disc">
        <li>Apples</li>
        <li>Bananas</li>
        <li>Lemons</li>
        <li>Oranges</li>
</ul>
```

Output:

- Apples
- Bananas
- Lemons
- Oranges

If we do not use type attribute, items are marked with discs. We use **type = "circle"** for circle bullets list, and **type = "square"** for square bullets list.

Definition list is the list of items with a description of each item. We use **<dl>** tag for definition list, **<dt>** for definition term, and **<dd>** for definition description. For example,

```
<dl>
        <dt>Coffee</dt>
  <dd>Black hot drink</dd>
        <dt>Milk</dt>
  <dd>White cold drink</dd>
</dl>
```

Output:

Coffee
    Black hot drink
Milk
    White cold drink

HTML Forms

Forms are used to select different types of user input. A form is an area that contains different form elements (like text fields, text area fields, drop-down menus, radio buttons,

checkboxes etc.). Form elements are elements that allow the user to enter information in a form. A form is defined with the **<form>** tag. For example,

```
<form>
   input elements
</form>
```

The most commonly used form tag is **<input>** tag. The type of input is specified with the **type attribute** within the <input> tag. For example,

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

Output: First name: Last name:

Another input type is **radio button**. Radio buttons are used when you want the user to select one of a limited number of choices. For example,

```
<form>
<input type="radio" name="sex" value="male" /> Male
<br />
<input type="radio" name="sex" value="female" /> Female
</form>
```

Output:

Male
Female

Another input type is **checkboxes**. Checkboxes are used when you want to select one or more options of a limited number of choices. For example,

```
<form>
I have a bike:
<input type="checkbox" name="vehicle" value="Bike" />
<br />
I have a car:
<input type="checkbox" name="vehicle" value="Car" />
<br />
I have an airplane:
```

```
<input type="checkbox" name="vehicle" value="Airplane" />
</form>
```

Output:

I have a bike: I have a car:

I have an airplane:

Another input type is **submit button**. When the user clicks on the "Submit" button, the content of the form is sent to the server. The form's **action attribute** defines the name of the file to send the content to. The file defined in the action attribute usually does something with the received input. For example,

```
<form name="input" action=" submit.php" method="get"> Username:
  <input type="text" name="user" />
  <input type="submit" value="Submit" />
</form>
```

Output:

Username:

Submit

If you type some characters in the text field above, and click the "Submit" button, the browser will send your input to a page called "submit.php". The page will show you the received input.

Note: *You can use other different form elements as well.*

The **method** attribute of <form> tag specifies how to send form-data (the form-data is sent to the page specified in the action attribute). We can use **"get"** and **"post"** as values of method attribute. When we use get, form-data can be sent as URL variables and when we use post, form-data are sent as HTTP post.

Notes on the "get" method:

- This method appends the form-data to the URL in name/value pairs
- There is a limit to how much data you can place in a URL (varies between browsers), therefore, you cannot be sure that all of the form-data will be correctly transferred
- Never use the "get" method to pass sensitive information! (password or other sensitive information will be visible in the browser's address bar)

Notes on the "post" method:

- This method sends the form-data as an HTTP post transaction

- The "post" method is more robust and secure than "get", and "post" does not have size limitations

We can create a simple drop-down box on an HTML page. A drop-down box is a selectable list. See code below:

```
<select name="cars">
<option value="volvo">Volvo</option>
<option value="saab">Saab</option>
<option value="fiat">Fiat</option>
<option value="audi">Audi</option>
</select>
```

Output:

HTML Color

HTML colors are displayed using RED, GREEN, and BLUE light. Colors are defined using hexadecimal (hex) notation for combination of red, green, and blue color values (RGB). The lowest value that can be given to one of the light sources is 0 (hex 00) and the highest values is 255 (hex FF). We can use HEX (e.g. **#2000FF**) as well as RGB (e.g. **rgb(32, 0, 255)**) values to define different colors.

The combination of Red, Green and Blue values from 0 to 255 gives a total of more than 16 million different colors to play with (256 x 256 x 256).

We can also use color names instead of hex and rgb values. The World Wide Web Consortium (W3C) has listed 16 valid color names for HTML and CSS: aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white, and yellow. Some examples are given below:

```
<body style = "background:rgb(12, 32, 255)">
<body style = "background:#0008FF>
<body style = "background:red">
```

HTML Frames

We can use frames to display more than one web page in the same browser window. Each HTML document is called a frame, and each frame is independent of the others. The disadvantages of using frames are:

- The web developer must keep track of more HTML documents
- It is difficult to print the entire page

We use **<frameset>** tag to define how to divide the window into frames. Each frameset defines a set of rows or columns. Within frameset, we use **<frame>** tag to define what HTML document to put into each frame.

If a frame has visible borders, the user can resize it by dragging the border. To prevent a user from doing this, you can add noresize="noresize" to the <frame> tag. Add the <noframes> tag for browsers that do not support frames.

Important: You cannot use the <body></body> tags together with the <frameset></frameset> tags. However, if you add a <noframes> tag containing some text for browsers that do not support frames, you will have to enclose the text in <body></body> tags.

Example 1:

```
<frameset cols="25%,50%,25%">
  <frame src="frame_a.htm" noresize="noresize"/>
  <frame src="frame_b.htm"/>
  <frame src="frame_c.htm"/>
<noframes>
<body>Your browser does not handle frames!</body>
</noframes>
</frameset>
```

Example 2:

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

Example 3: Mixed Frameset

```
<frameset rows="50%,50%">
  <frame src="frame_a.htm"/>
  <frameset cols="25%,75%">
            <frame src="frame_b.htm"/>
            <frame src="frame_c.htm"/>
 </frameset>
</frameset>
```

HTML Fonts

The **<font>** tag in HTML is deprecated. It is supposed to be removed in a future version of HTML. For example,

```
<p>
        <font size="2" face="Verdana" color = "red">
   This is a paragraph.
        </font>
</p>
```

HTML Character Entities

Character entities are replaced with reserved characters. A character entity looks *&entity_name* **OR** *&#entity_number*. Some commonly used character entities are:

Result	Description	Entity Name	Entity Number
	non-breaking space		
<	less than	<	<
>	greater than	>	>
&	Ampersand	&	&
¢	Cent	¢	¢
£	Pound	£	£
¥	Yen	¥	¥
€	Euro	€	€
©	Copyright	©	©
®	registered trademark	®	®

HTML Head

The head element contains general information, also called meta-information, about a document. The elements inside the head element should not be displayed by a browser. According to the HTML standard, only a few tags are legal inside the head section. These are: <base>, <link>, <meta>, <title>, <style>, and <script>.

You must use this element and it should be used just once. It must start immediately after the opening <html> tag and end directly before the opening <body> tag.

HTML Meta

HTML includes a meta element that goes inside the head element. The purpose of the meta element is to provide meta-information about the document. Meta elements are purely used for search engine's use and to provide some additional information about your pages. We use three attributes (**name**, **content**, and **http-equiv**) with **<meta>** tag.

We use **name = "keywords"** to provide information for a search engine. If the keywords you have chosen are the same as the ones they have put in, you come up in the search engine's result pages. For example,

<meta name="keywords" content="HTML, DHTML, CSS, XML, XHTML, JavaScript" />

We use **name = "description"** to define a description of your page. It is sort summary of the content of the page. Depending on the search engine, this will be displayed along with the title of your page in an index. For example,

<meta name="description" content="Free Web tutorials on HTML, CSS, XML, and XHTML" />

We use **name = "generator"** to define a description for the program you used to write your pages. For example,

<meta name="generator" content="Homesite 4.5" />

We use **name = "author"** and **name = "copyright"** for author and copyright details. For example,

<meta name="author" content="W3schools" />

<meta name="copyright" content="W3schools 2005" />

We use **name = "expires"** to give the browsers a data, after which the page is deleted from the browsers cache, and must be downloaded again. This is useful if you want to make sure your visitors are reading the most current version of a page. For example,

<meta name="expires" content="13 July 2008" />

We use **http-equiv = "expires"** to refresh itself to the most current version or change to another location (page) entirely after some time. This is useful if you've moved a page to a new url and want any visitors to the old address to be quietly sent to the new location. For example,

<meta http-equiv = "refresh" content="5;url=http://www.tu.edu.np" />

Here, the number is the number of seconds to wait before changing to the new page. Setting it to 0 results in an instant redirect.

HTML Div

The ** ** element defines logical divisions within the document. When you use a element, you are indicating that the enclosed content is specific section of the page and you can format the section with CSS (Cascading Style Sheet). For example,

```
<div style="background-color:orange;text-align:center">
  <p>Navigation section</p>
</div>
<div style="border:1px solid black">
  <p>Content section</p>
</div>
```

HTML Events

Events trigger actions in the browser, like starting a JavaScript when a user clicks on an HTML element. Below is a list of attributes that can be inserted to HTML tags to define event actions. These HTML events are given below:

Window Events (**Only valid in body and frameset elements)**

Attribute	Value	Description
Onload	*Script*	Script to be run when a document loads
Onunload	*Script*	Script to be run when a document unloads

Form Element Events **(Only valid in form elements)**

Attribute	Value	Description
Onchange	*Script*	Script to be run when the element changes
Onsubmit	*Script*	Script to be run when the form is submitted
Onreset	*Script*	Script to be run when the form is reset
Onselect	*script*	Script to be run when the element is selected
Onblur	*script*	Script to be run when the element loses focus
Onfocus	*script*	Script to be run when the element gets focus

Keyboard Events **(Not valid in *base, bdo, br, frame, frameset, head, html, iframe, meta, param, script, style,* and *title* elements)**

Attribute	Value	Description
Onkeydown	*script*	What to do when key is pressed
Onkeypress	*script*	What to do when key is pressed and released
Onkeyup	*script*	What to do when key is released

Mouse Events **(Not valid in *base, bdo, br, frame, frameset, head, html, iframe, meta, param, script, style, title* elements)**

Attribute	Value	Description
Onclick	*script*	What to do on a mouse click
Ondblclick	*script*	What to do on a mouse double-click
Onmousedown	*script*	What to do when mouse button is pressed
Onmousemove	*script*	What to do when mouse pointer moves
Onmouseout	*Script*	What to do when mouse pointer moves out of an element

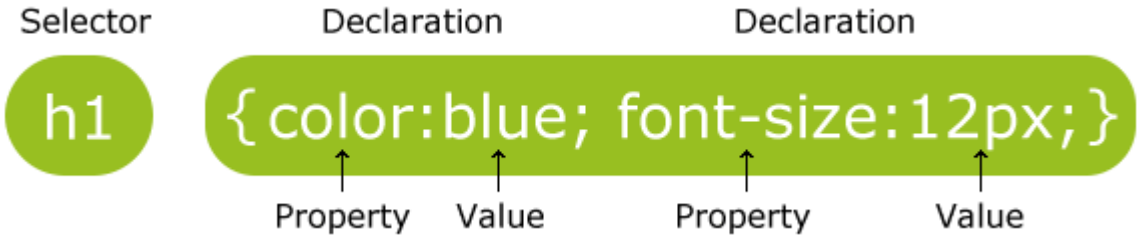
Onmouseover	*Script*	What to do when mouse pointer moves over an element
Onmouseup	*script*	What to do when mouse button is released

CSS (Cascading Style Sheets)

CSS stands for cascading style sheets. It was first developed in 1997, as a way for Web developers to define the **look and feel** of their Web pages. It was intended to allow developers to separate content from design and layout so that HTML could perform more of the function without worry about the design and layout. It is used to separate style from content.

Syntax

A CSS rule has two main parts: a *selector* and one or more *declarations*. **Selector** is normally the HTML element you want to style and each **declaration** consists of a *property* and *value*. The **property** is the style attribute we want to use and each property has a **value** associated with it.



Example:

```
p {color:red;text-align:center;}
```

Inserting CSS

We can use style sheets in three different ways in out HTML document. There are **external style sheet**, **internal style sheet** and **inline style**.

External Style Sheet

If we want to apply the same style to many pages, we use external style sheet. With an external style sheet, you can change the look of an entire Web site by changing one style sheet file. Each page must link to the style sheet using the **<link>** tag. The <link> tag goes inside the head section. For example,

```
<head>
<link rel="stylesheet" type="text/css" href="mystyle.css" />
</head>
```

An external style sheet can be written in any text editor. The file should not contain any html tags. Your style sheet should be saved with a **.css** extension. An example of a style sheet file is shown below:

```
hr {color:sienna;}
p {margin-left:20px;} /*Note: Do not leave space between property value and units*/
body {background-image:url("images/back40.gif");}
```

Internal Style Sheet

If you want a unique style to a single document, an internal style sheet should be used. You define internal styles in the head section of an HTML page, by using the **<style>** tag. For example,

```
<head>
<style type="text/css">
hr {color:red;}
p {margin-left:20px;}
body {background-image:url("images/back40.gif");}
</style>
</head>
```

Inline Styles

If you want a unique style to a single element, an inline style sheet should be used. An inline style loses many of the advantages of style sheets by mixing content with presentation. To use inline styles you use the **style attribute** in the relevant tag. The style attribute can contain any CSS property. For example,

```
<p style="color:yellow;margin-left:20px">This is a paragraph.</p>
```

Comments

Comments are used to explain your code, and may help you when you edit the source code at a later date. Comments are ignored by browsers. A CSS comment begins with "/*", and ends with "*/".

Id and Class Selectors

The **id** selector is used to specify a style for a single, unique element. The id selector uses id attribute of the HTML element and is defined with **"#"**. For example,

```
<head>
<style type="text/css">
#para1
{
text-align:center;
color:red;
}
</style>
</head>
<body>
<p id="para1">Hello World!</p>
<p>This paragraph is not affected by the style.</p>
</body>
```

The **class** selector is used to specify a style for a group of elements. Unlike the id selector, the class selector is most often used on several elements. This allows you to set a particular style for any HTML elements with the same class. The class selector uses the HTML class attribute, and is defined with a ".". For example,

```
<head>
<style type="text/css">
.center
{
text-align:center;
}
</style>
</head>
<body>
<h1 class="center">Center-aligned heading</h1>
<p class="center">Center-aligned paragraph.</p>
</body>
```

You can also specify that only specific HTML elements should be affected by a class. For example,

```
<head>
<style type="text/css">
p.center
{
text-align:center;
}
</style>
</head>
<body>
<h1 class="center">This heading will not be affected</h1>
<p class="center">This paragraph will be center-aligned.</p>
</body>
```

Multiple Styles Will Cascade into One

Styles can be specified:

- inside an HTML element
- inside the head section of an HTML page
- in an external CSS file

Tip: Even multiple external style sheets can be referenced inside a single HTML document.

Cascading order

What style will be used when there is more than one style specified for an HTML element?

Generally speaking we can say that all the styles will "cascade" into a new "virtual" style sheet by the following rules, where number four has the highest priority:

1. Browser default
2. External style sheet
3. Internal style sheet (in the head section)
4. Inline style (inside an HTML element)

So, an inline style (inside an HTML element) has the highest priority, which means that it will override a style defined inside the <head> tag, or in an external style sheet, or in a browser (a default value).

Note: If the link to the external style sheet is placed after the internal style sheet in HTML <head>, the external style sheet will override the internal style sheet!

CSS Background

Background properties are used to define the background effects of an HTML element.
CSS properties used to define background effects are: **background-color**, **background- image**, **background-repeat**, **background-attachment**, and **background-position**.

Background Image

The background-image property specifies an image to use as the background of an element. By default, the image is repeated so it covers the entire element.

The background image for a page can be set like this:

```
body {background-image:url('paper.gif');}
```

Background Image - Repeat Horizontally or Vertically

By default, the background-image property repeats an image both horizontally and vertically. Some images should be repeated only horizontally or vertically, or they will look strange, like this:

Example

```
body
{
background-image:url('gradient2.png');
}
```

If the image is repeated only horizontally (repeat-x), the background will look better:

Example

```
body
{
background-image:url('gradient2.png');
background-repeat:repeat-x;
}
```

Background Image - Set position and no-repeat

When using a background image, use an image that does not disturb the text. Showing the image only once is specified by the background-repeat property:

Example

```
body
{
background-image:url('img_tree.png');
background-repeat:no-repeat;
}
```

In the example above, the background image is shown in the same place as the text. We want to change the position of the image, so that it does not disturb the text too much.

The position of the image is specified by the background-position property:

Example

```
body
{
background-image:url('img_tree.png'); background-
repeat:no-repeat; background-position:right top;
}
```

Shorthand Property

To shorten the code, it is also possible to specify all the properties in one single property.
This is called a shorthand property. The shorthand property for background is simply "background". When using the shorthand property the order of the property values are: background-color, background-image, background-repeat, background-attachment, and background-position. For example,

```
body {background:#ffffff url('img_tree.png') no-repeat right top;}
```

Grouping Selectors

In style sheets there are often elements with the same style.

```
h1
{
color:green;
}
h2
{
color:green;
}
p
{
color:green;
}
```

To minimize the code, you can group selectors. Separate each selector with a comma. In the example below we have grouped the selectors from the code above:

Example

```
h1,h2,p
{
color:green;
}
```

CSS Display and Visibility

The display property specifies if/how an element is displayed, and the visibility property specifies if an element should be visible or hidden.

Hiding an Element - display:none or visibility:hidden

Hiding an element can be done by setting the display property to "none" or the visibility property to "hidden". However, notice that these two methods produce different results:

visibility: hidden hides an element, but it will still take up the same space as before. The element will be hidden, but still affect the layout.

Example

```
h1.hidden {visibility:hidden;}
```

display: none hides an element, and it will not take up any space. The element will be hidden, and the page will be displayed as the element is not there:

Example

```
h1.hidden {display:none;}
```

CSS Display - Block and Inline Elements

A block element is an element that takes up the full width available, and has a line break before and after it.

Examples of block elements:

- <h1>
- <p>
-

An inline element only takes up as much width as necessary, and does not force line breaks.

Examples of inline elements:

-
- <a>

Changing How an Element is Displayed

Changing an inline element to a block element, or vice versa, can be useful for making the page look a specific way, and still follow web standards.

The following example displays list items as inline elements:

Example

```
li {display:inline;}
```

The following example displays span elements as block elements:

Example

```
span {display:block;}
```

Changing the display type of an element changes only how the element is displayed, NOT what kind of element it is. For example: An inline element set to display:block is not allowed to have a block element nested inside of it.

CSS Padding Property:

Example

Set the padding of a p element:

```
p
{
padding:2cm 4cm 3cm 4cm;
}
```

Definition and Usage

The padding shorthand property sets all the padding properties in one declaration. This property can have from one to four values.

Examples:

- **padding:10px 5px 15px 20px;**
  - top padding is 10px

- right padding is 5px

  - bottom padding is 15px
  - left padding is 20px
- **padding:10px 5px 15px;**
  - top padding is 10px
  - right and left padding are 5px
  - bottom padding is 15px
- **padding:10px 5px;**
  - top and bottom padding are 10px
  - right and left padding are 5px
- **padding:10px;**
  - all four paddings are 10px

Note: Negative values are not allowed.

CSS Float:

With CSS float, an element can be pushed to the left or right, allowing other elements to wrap around it. Float is very often used for images, but it is also useful when working with layouts.

How Elements Float

Elements are floated horizontally; this means that an element can only be floated left or right, not up or down. A floated element will move as far to the left or right as it can. Usually this means all the way to the left or right of the containing element. The elements after the floating element will flow around it. The elements before the floating element will not be affected. If an image is floated to the right, a following text flows around it, to the left.

Example

```
img
{
float:right;
}
```

Floating Elements Next to Each Other

If you place several floating elements after each other, they will float next to each other if there is room. Here we have made an image gallery using the float property:

Example

```
.thumbnail
{
```

```
float:left; width:110px;
height:90px; margin:5px;
}
```

Turning off Float - Using Clear

Elements after the floating element will flow around it. To avoid this, use the clear property.

The clear property specifies which sides of an element other floating elements are not allowed.

Add a text line into the image gallery, using the clear property:

Example

```
.text_line
{
clear:both;
}
```

JavaScript

- JavaScript was designed to add interactivity to HTML pages
- JavaScript is a scripting language
- A scripting language is a lightweight programming language
- JavaScript is usually embedded directly into HTML pages
- JavaScript is an interpreted language (means that scripts execute without preliminary compilation)
- Everyone can use JavaScript without purchasing a license

Are Java and JavaScript the same?

NO! Java and JavaScript are two completely different languages in both concept and design! Java (developed by Sun Microsystems) is a powerful and much more complex programming language - in the same category as C and C++.

What can a JavaScript do?

- **JavaScript gives HTML designers a programming tool -** HTML authors are normally not programmers, but JavaScript is a scripting language with a very simple syntax! Almost anyone can put small "snippets" of code into their HTML pages
- **JavaScript can put dynamic text into an HTML page -** A JavaScript statement like this: document.write("<h1>" + name + "</h1>") can write a variable text into an HTML page
- **JavaScript can react to events -** A JavaScript can be set to execute when something happens, like when a page has finished loading or when a user clicks on an HTML element
- **JavaScript can read and write HTML elements -** A JavaScript can read and change the content of an HTML element
- **JavaScript can be used to validate data -** A JavaScript can be used to validate form data before it is submitted to a server. This saves the server from extra processing
- **JavaScript can be used to detect the visitor's browser** - A JavaScript can be used to detect the visitor's browser, and - depending on the browser - load another page specifically designed for that browser
- **JavaScript can be used to create cookies** - A JavaScript can be used to store and retrieve information on the visitor's computer

The Real Name is ECMAScript

- JavaScript's official name is ECMAScript.
- ECMAScript is developed and maintained by the ECMA organization.
- ECMA-262 is the official JavaScript standard.
- The language was invented by Brendan Eich at Netscape (with Navigator 2.0), and has appeared in all Netscape and Microsoft browsers since 1996.
- The development of ECMA-262 started in 1996, and the first edition of was adopted by the ECMA General Assembly in June 1997.
- The standard was approved as an international ISO (ISO/IEC 16262) standard in 1998.
- The development of the standard is still in progress.
- The HTML <script> tag is used to insert a JavaScript into an HTML page. The example

below shows how to use JavaScript to write text on a web page:

```
<html>
<body>
<script type="text/javascript">
document.write("Hello World!");
</script>
</body>
</html>
```

The example below shows how to add HTML tags to the JavaScript:

```
<html>
<body>
<script type="text/javascript">
document.write("<h1>Hello World!</h1>");
</script>
</body>
</html>
```

To insert a JavaScript into an HTML page, we use the <script> tag. Inside the <script> tag we use the type attribute to define the scripting language.

So, the <script type="text/javascript"> and </script> tells where the JavaScript starts and ends:

```
<html>
<body>
<script type="text/javascript">
...
</script>
</body>
</html>
```

The **document.write** command is a standard JavaScript command for writing output to a page.

By entering the document.write command between the <script> and </script> tags, the browser will recognize it as a JavaScript command and execute the code line. In this case the browser will write Hello World! to the page:

```
<html>
<body>
<script type="text/javascript">
document.write("Hello World!");
</script>
</body>
</html>
```

Note: If we had not entered the <script> tag, the browser would have treated the document.write("Hello World!") command as pure text, and just write the entire line on the page.

How to Handle Simple Browsers

Browsers that do not support JavaScript, will display JavaScript as page content. To prevent them from doing this, and as a part of the JavaScript standard, the HTML comment
tag should be used to "hide" the JavaScript.

Just add an HTML comment tag <!-- before the first JavaScript statement, and a --> (end of comment) after the last JavaScript statement, like this:

```
<html>
<body>
<script type="text/javascript">
<!--
document.write("Hello World!");
//-->
</script>
</body>
</html>
```

The two forward slashes at the end of comment line (//) is the JavaScript comment symbol. This prevents JavaScript from executing the --> tag.

JavaScripts can be put in the body and in the head sections of an HTML page.

Where to Put the JavaScript

JavaScripts in a page will be executed immediately while the page loads into the browser. This is not always what we want. Sometimes we want to execute a script when a page loads, or at a later event, such as when a user clicks a button. When this is the case we put the script inside a function, you will learn about functions in a later chapter.

Scripts in <head>

Scripts to be executed when they are called, or when an event is triggered, are placed in functions. Put your functions in the head section, this way they are all in one place, and they do not interfere with page content.

Example

```
<html>
<head>
<script type="text/javascript">
function message()
{
alert("This alert box was called with the onload event");
}
</script>
</head>

<body onload="message()">
</body>
</html>
```

Scripts in <body>

If you don't want your script to be placed inside a function, or if your script should write page content, it should be placed in the body section.

Example

```
<html>
<head>
</head>

<body>
<script type="text/javascript">
document.write("This message is written by JavaScript");
</script>
</body>
</html>
```

Scripts in <head> and <body>

You can place an unlimited number of scripts in your document, so you can have scripts in both the body and the head section.

Example

```
<html>
<head>
<script type="text/javascript">
function message()
{
alert("This alert box was called with the onload event");
}
</script>
</head>

<body onload="message()">
<script type="text/javascript">
document.write("This message is written by JavaScript");
</script>
</body>

</html>
```

Using an External JavaScript

If you want to run the same JavaScript on several pages, without having to write the same

script on every page, you can write a JavaScript in an external file.

Save the external JavaScript file with a .js file extension.

Note: The external script cannot contain the <script></script> tags!

To use the external script, point to the .js file in the "src" attribute of the <script> tag:

```
<html>
<head>
<script type="text/javascript" src="xxx.js"></script>
</head>
<body>
</body>
</html>
```

Note: Remember to place the script exactly where you normally would write the script! JavaScript is a sequence of statements to be executed by the browser.

JavaScript is Case Sensitive

Unlike HTML, JavaScript is case sensitive - therefore watch your capitalization closely when you write JavaScript statements, create or call variables, objects and functions.

JavaScript Statements

A JavaScript statement is a command to a browser. The purpose of the command is to tell the browser what to do.

This JavaScript statement tells the browser to write "Hello Dolly" to the web page:

```
document.write("Hello Dolly");
```

It is normal to add a semicolon at the end of each executable statement. Most people think this is a good programming practice, and most often you will see this in JavaScript examples on the web.

The semicolon is optional (according to the JavaScript standard), and the browser is supposed to interpret the end of the line as the end of the statement. Because of this you will often see examples without the semicolon at the end.

Note: *Using semicolons makes it possible to write multiple statements on one line.*

JavaScript Code

JavaScript code (or just JavaScript) is a sequence of JavaScript statements. Each statement is executed by the browser in the sequence they are written. Following example will write a heading and two paragraphs to a web page:

Example

```
<script type="text/javascript"> document.write("<h1>This is a
heading</h1>"); document.write("<p>This is a paragraph.</p>");
document.write("<p>This is another paragraph.</p>");
</script>
```

JavaScript Blocks

JavaScript statements can be grouped together in blocks. Blocks start with a left curly bracket {, and ends with a right curly bracket }. The purpose of a block is to make the sequence of statements execute together. Following example will write a heading and two paragraphs to a web page:

Example

```
<script type="text/javascript">
{
document.write("<h1>This is a heading</h1>");
document.write("<p>This is a paragraph.</p>");
document.write("<p>This is another paragraph tested at pmc.</p>");
}
</script>
```

The example above is not very useful. It just demonstrates the use of a block. Normally a block is used to group statements together in a function or in a condition (where a group of statements should be executed if a condition is met).

JavaScript Variables

As with algebra, JavaScript variables are used to hold values or expressions. A variable can have a short name, like x, or a more descriptive name, like carname.

Rules for JavaScript variable names:

- Variable names are case sensitive (y and Y are two different variables)
- Variable names must begin with a letter or the underscore character

Note: Because JavaScript is case-sensitive, variable names are case-sensitive.

Declaring (Creating) JavaScript Variables

Creating variables in JavaScript is most often referred to as "declaring" variables. You can declare JavaScript variables with the **var statement**:

```
var x;
var carname;
```

After the declaration shown above, the variables are empty (they have no values yet). However, you can also assign values to the variables when you declare them:

```
var x=5;
```

var carname="Volvo";

After the execution of the statements above, the variable **x** will hold the value **5**, and **carname** will hold the value **Volvo**.

Note: *When you assign a text value to a variable, use quotes around the value.*

Assigning Values to Undeclared JavaScript Variables

If you assign values to variables that have not yet been declared, the variables will automatically be declared. These statements:

x=5;
carname="Volvo";

have the same effect as:

var x=5;
var carname="Volvo";

Redeclaring JavaScript Variables

If you redeclare a JavaScript variable, it will not lose its original value.
var x=5;
var x;

After the execution of the statements above, the variable x will still have the value of 5. The value of x is not reset (or cleared) when you redeclare it.

JavaScript Arithmetic

As with algebra, you can do arithmetic operations with JavaScript variables:
y=x-5;
z=y+5;

Comparison Operators

Comparison operators are used in logical statements to determine equality or difference between variables or values.

Given that **x=5**, the table below explains the comparison operators:

Operator	Description	Example
= =	is equal to	x==8 is false
= = =	is exactly equal to (value and type)	x===5 is true x==="5" is false
!=	is not equal	x!=8 is true

>	is greater than	x>8 is false
<	is less than	x<8 is true
>=	is greater than or equal to	x>=8 is false
<=	is less than or equal to	x<=8 is true

Logical Operators

Logical operators are used to determine the logic between variables or values. Given that **x=6 and y=3**, the table below explains the logical operators:

Operator	Description	Example
&&	And	(x < 10 && y > 1) is true
||	Or	(x==5 || y==5) is false
!	Not	!(x==y) is true

Conditional Operator

JavaScript also contains a conditional operator that assigns a value to a variable based on some condition.

Syntax

variablename=(condition)?value1:value2

Example

greeting=(visitor=="PRES")?"Dear President ":"Dear ";

If the variable **visitor** has the value of "PRES", then the variable **greeting** will be assigned the value "Dear President " else it will be assigned "Dear".

Flow Control

- Conditional statements are used to perform different actions based on different conditions.
- In JavaScript we have the following conditional statements:
- **if statement** - use this statement to execute some code only if a specified condition is true
- **if...else statement** - use this statement to execute some code if the condition is true and another code if the condition is false
- **if...else if....else statement** - use this statement to select one of many blocks of code to be executed

- **switch statement** - use this statement to select one of many blocks of code to be executed

Looping Structures

- Often when you write code, you want the same block of code to run over and over again in a row. Instead of adding several almost equal lines in a script we can use loops to perform a task like this.
- In JavaScript, there are two different kind of loops:
- **for** - loops through a block of code a specified number of times
- **while** - loops through a block of code while a specified condition is true

The for Loop

- The for loop is used when you know in advance how many times the script should run.

Syntax

```
for (var=startvalue;var<=endvalue;var=var+increment)
{
code to be executed
}
```

Example

```
<html>
<body>
<script type="text/javascript">
var i=0;
for (i=0;i<=5;i++)
{
document.write("The number is " + i);
document.write("<br />");
}
</script>
```

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

JavaScript While Loop

- The while loop loops through a block of code while a specified condition is true.

Syntax

- while (var<=endvalue)

```
{
code to be executed
}
```

Example

```
<html>
<body>
<script type="text/javascript">
var i=0;
while (i<=5)
{
document.write("The number is " + i);
document.write("<br />");
i++;
}
</script>
</body>
</html>
```

Javascript do while loop

The do...while loop is a variant of the while loop. This loop will execute the block of code ONCE, and then it will repeat the loop as long as the specified condition is true. Syntax

```
do
        {
```

```
        code to be executed
  }
while (var<=endvalue);
```

Example

The example below uses a do...while loop. The do...while loop will always be executed at least once, even if the condition is false, because the statements are executed before the condition is tested:

```
<html>
<body>
<script type="text/javascript">
var i=0;
do
  {
  document.write("The number is " + i);
  document.write("<br />");
        i++;
  }
while (i<=5);
</script>
</body>
</html>
```

The Break Statement

The break statement will break the loop and continue executing the code that follows after the loop (if any).

```
<html>
<body>
<script type="text/javascript">
var i=0;
for (i=0;i<=10;i++)
```

```
{
if (i==3)
 {
 break;
 }
document.write("The number is " + i);
document.write("<br />");
}
        </script>
</body>
        </html>
```

Javascriptfor…..instatement

The for...in statement loops through the elements of an array or through the properties of an object.

Syntax

for (*variable* in *object*)
{
code to be executed
}

Note: The code in the body of the for...in loop is executed once for each element/property.

Example: Use the for...in statement to loop through an array:

```
<html>
<body>
<script type="text/javascript">
var x;
var mycars = new Array();
mycars[0] = "Saab";
```

```
mycars[1] = "Volvo"; mycars[2] =
"BMW"; for (x in mycars)
 {
 document.write(mycars[x] + "<br />");
 }
</script>
</body>
</html>
```

Functions

- A function is simply a block of code with a name, which allows the block of code to be called by other components in the scripts to perform certain tasks.
- Functions can also accept parameters that they use complete their task.
- JavaScript actually comes with a number of built-in functions to accomplish a variety of tasks.

Creating Custom Functions

- In addition to using the functions provided by javaScript, you can also create and use your own functions.
- General syntax for creating a function in JavaScript is as follows:

```
function name_of_function(argument1,argument2,…arguments)
{
……
//Block of Code
……
}
```

Calling functions

- There are two common ways to call a function: From an *event handler* and from another function.

- Calling a function is simple. You have to specify its name followed by the pair of parenthesis.

```
<SCRIPT  TYPE="TEXT/JAVASCRIPT">
    name_of_function(argument1,argument2,…arguments)
 </SCRIPT>
```

Example

```
<html>
```

```
<head> <title>PMC</title>
        <Script Language="JavaScript">
        function welcomeMessage()
                {
                        document.write("Welcome to Patan Campus!");
                       }
                </Script>
             </head>
              <body>
                    <h1>Patan Multiple Campus CSIT</h1>
                     <h3>Testing the function in PMC</h3>
                    <Script Language="JavaScript">
                                welcomeMessage();
                    </Script>
              </body>
         </html>
```

Popup Boxes

Alert Box:

An alert box is often used if you want to make sure information comes through to the user. When an alert box pops up, the user will have to click "OK" to proceed.

Syntax

```
alert("sometext");
```

Example

```
<html>
<head>
<script type="text/javascript">
function show_alert()
{
alert("I am an alert box!");
}
</script>
</head>
<body>
<input type="button" onclick="show_alert()" value="Show alert box" />
</body>
</html>
```

Confirmation Box:

A confirm box is often used if you want the user to verify or accept something. When a confirm box pops up, the user will have to click either "OK" or "Cancel" to proceed. If the user clicks "OK", the box returns true. If the user clicks "Cancel", the box returns false.

Syntax

confirm("sometext");

Example

```
<html>
<head>
<script type="text/javascript">
function show_confirm()
{
```

```
var r=confirm("Press a button");
if (r==true)
  {
  document.write("You pressed OK!");
  }
            else
  {
  document.write("You pressed Cancel!");
  }
}
</script>
</head>
<body>
<input type="button" onclick="show_confirm()" value="Show confirm box" />
</body>
</html>
```

Prompt Box:

A prompt box is often used if you want the user to input a value before entering a page.

When a prompt box pops up, the user will have to click either "OK" or "Cancel" to proceed after entering an input value. If the user clicks "OK" the box returns the input value. If the user clicks "Cancel" the box returns null.

Syntax

prompt("sometext","defaultvalue");

Example

```
<html>
<head>
<script type="text/javascript">
var name=prompt("Please enter your name",“Rajendra");
</script>
</head>
```

```
<body>
<script type="text/javascript">
document.write("Hello  "+name + "You have worked will with variables");
</script>
</body>
</html>
```

JavaScript objects

JavaScript is an Object Oriented Programming (OOP) language. An OOP language allows you to define your own objects and make your own variable types. An object is just a special kind of data. An object has properties and methods.

Properties: Properties are the values associated with an object.
Methods: Methods are the actions that can be performed on objects.

Array Object in JavaScript

An array is a special variable, which can hold more than one value, at a time. An array can hold all your variable values under a single name. And you can access the values by referring to the array name. Each element in the array has its own ID so that it can be easily accessed. The following code creates an Array object called myCars:

```
var myCars=new Array();
```

There are three ways of adding values to an array (you can add as many values as you need to define as many variables you require).

1.) **Conventional array:** The classic conventional array looks like the following:

```
var myCars=new Array();
myCars[0]="Saab";
```

```
myCars[1]="Volvo";
myCars[2]="BMW";
```

You can expand and contract the array as desired, by adding new array elements. Note that like in most other programming languages, the first array element should have an index number of 0.

With a conventional array, you have the option of presetting the array's length when defining it, by passing in a numeric integer into the Array() constructor:

```
var myCars=new Array(3);
myCars[0]="Saab"; myCars[1]="Volvo";
myCars[2]="BMW";
```

2.) Condensed array: The second way of defining an array is called a condensed array, and differs from the above simply in that it allows you to combine the array and array elements definitions into one step:

```
var myCars=new Array("Saab","Volvo","BMW");
```

This is convinient if you know all the array element values in advance.

3.) Literal array: Finally, we arrive at literal arrays. Introduced in JavaScript1.2 and support by all modern browsers (IE/NS4+), literal arrays sacrafice intuitiveness somewhat in exchange for tremendous robustness. The syntax looks like:

```
var myCars=["Saab","Volvo","BMW"];
```

Literal array with 5 elements (middle 3 with undefined values). var mystudents=["giri", , , "tulsi"]

As you can see, enclose all the array elements within an outter square bracket ([ ]), each separated by a comma (,). To create array elements with an initial undefined value just enter a comma (,) as shown in the second example above.

Literal arrays really shine when it comes to defining multi-dimensional arrays. It is as easy as adding containing brackets ([ ]) within the outermost bracket. For example:

```
var myarray=[["Subash", "Pandey", "Gautam"], Kalanki, Sanepa]
```

Here the first array element is actually a two dimensional array in itself containing various cities names. To access LA, then, you'd use the syntax:

```
myarray[0][1] //returns "Pandey"
```

Note: If you specify numbers or true/false values inside the array then the type of variables will be numeric or Boolean instead of string.

Accessing the Array

You can refer to a particular element in an array by referring to the name of the array and the index number. The index number starts at 0. In above initialized array, the code line *document.write(myCars[0]);* will result in the following output: Saab

To modify a value in an existing array, just add a new value to the array with a specified index number:

myCars[0]="Opel"; Now, the following code line:

document.write(myCars[0]); will result in the following output: Opel.

Some methods associated with array

- **concat( ):** Joins two or more arrays, and returns a copy of the joined arrays
- **join( ):** Joins all elements of an array into a string
- **pop( ):** Removes the last element of an array, and returns that element
- **push( ):** Adds new elements to the end of an array, and returns the new length
- **reverse( ):** Reverses the order of the elements in an array

- **shift( ):** Removes the first element of an array, and returns that element
- **sort( ):** Sorts the elements of an array
- **toString( ):** Converts an array to a string, and returns the result
- **unshift( ):** Adds new elements to the beginning of an array, and returns the new length

Example

Concat( ) : Joining Two Arrays

```
<script type="text/javascript">
var parents = ["Giri", "Pari"];
var children = ["Cactus", "Rose"];
var family = parents.concat(children);
document.write(family);
</script>
```

The output will be :

Giri, Pari, Cactus, Rose

String Object in JavaScript

The String object is used to manipulate a stored piece of text. String objects are created with new String().

Syntax

var txt = new String(string);or more simply:

var txt = string;

Some methods associated with String object:

- **toLowerCase( ):** Converts a string to lowercase letters
- **toUpperCase( ):** Converts a string to uppercase letters

- **concat( ):** Joins two or more strings, and returns a copy of the joined strings
- **charAt( ):** Returns the character at the specified index
- **indexOf( ):** Returns the position of the first found occurrence of a specified value in a string
- **replace( ):** Searches for a match between a substring (or regular expression) and a string, and replaces the matched substring with a new substring

Examples

In the following example we are using the length property of the String object to return the number of characters in a string:

```
<script type="text/javascript"> var txt="Hello
World!"; document.write(txt.length);
</script>
```

The output of the code above will be: 12

In the following example we are using the toUpperCase( ) method of the String object to display a text in uppercase letters:

```
<script type="text/javascript"> var str="hello its me
webtech!"; document.write(str.toUpperCase());
</script>
```

The output of the code above will be: HELLO ITS ME
WEBTECH

Example: IndexOf ( ) method

The indexOf( ) method returns the position of the first occurrence of a specified value in a string. This method returns -1 if the value to search for never occurs. The indexOf( ) method is case sensitive.

Syntax

string.indexOf(searchstring, start)

searchstring: Required. The string to search for.

start: Optional. The start position in the string to start the search. If omitted, the search starts from position 0

```
<script type="text/javascript"> var str="Patan world!";
document.write(str.indexOf("d") + "<br />");
document.write(str.indexOf("WORLD") + "<br />");
document.write(str.indexOf("world"));

</script>
```

Output

10

-1

6

Math Object in Javascript

The Math object allows you to perform mathematical tasks. The Math object includes several mathematical constants and methods. For example

```
var pi_value=Math.PI;
var sqrt_value=Math.sqrt(16);
```

Note: Math is not a constructor. All properties and methods of Math can be called by using Math as an object without creating it.

Properties

- **Math.E:** Returns Euler's number (approx. 2.718)
- **Math.LN2:** Returns the natural logarithm of 2 (approx. 0.693)
- **Math.LN10:** Returns the natural logarithm of 10 (approx. 2.302)
- **Math.LOG2E:** Returns the base-2 logarithm of E (approx. 1.442)
- **Math.LOG10E:** Returns the base-10 logarithm of E (approx. 0.434)
- **Math.PI:** Returns PI (approx. 3.14159)
- **Math.SQRT1_2:** Returns the square root of 1/2 (approx. 0.707)
- **Math.SQRT2:** Returns the square root of 2 (approx. 1.414)

Methods

- **abs(x):** Returns the absolute value of x
- **ceil(x):** Returns x, rounded upwards to the nearest integer
- **floor(x):** Returns x, rounded downwards to the nearest integer
- **log(x):** Returns the natural logarithm (base E) of x
- **max(x,y,z,...,n):** Returns the number with the highest value
- **min(x,y,z,...,n):** Returns the number with the lowest value
- **pow(x,y):** Returns the value of x to the power of y
- **sqrt(x):** Returns the square root of x
- **random( ):** Returns a random number between 0 and 1
- **round(x):** Rounds x to the nearest integer
- **sin(x):** Returns the sine of x (x is in radians)
- **cos(x):** Returns the cosine of x (x is in radians)
- **tan(x):** Returns the tangent of an angle

Examples

document.write(Math.round(4.7)); Output: 5

document.write(Math.random()); Output:
0.19733826867061233

document.write(Math.floor(Math.random()*6)); Output: 3

Date Object in Javascript

The Date object is used to work with dates and times. Date objects are created with the Date( ) constructor. We can easily manipulate the date by using the methods available for the Date object. In the example below we set a Date object to a specific date (14th January
2010):

```
var myDate=new Date();
myDate.setFullYear(2010,0,14);
```

And in the following example we set a Date object to be 5 days into the future:

```
var myDate=new Date();
myDate.setDate(myDate.getDate()+5);
```

Note: If adding five days to a date shifts the month or year, the changes are handled automatically by the Date object itself!

Methods

- getDate() Returns the day of the month (from 1-31)
- getDay() Returns the day of the week (from 0-6)
- getFullYear() Returns the year (four digits)
- getHours() Returns the hour (from 0-23)
- getMilliseconds() Returns the milliseconds (from 0-999)

- getMinutes() Returns the minutes (from 0-59)
- getMonth() Returns the month (from 0-11)
- getSeconds() Returns the seconds (from 0-59)
- setDate() Sets the day of the month (from 1-31)
- setFullYear() Sets the year (four digits)
- setHours() Sets the hour (from 0-23)
- setMilliseconds() Sets the milliseconds (from 0-999)
- setMinutes() Set the minutes (from 0-59)
- setMonth() Sets the month (from 0-11)
- setSeconds() Sets the seconds (from 0-59)
- toString() Converts a Date object to a string

Examples

The Date object is also used to compare two dates. The following example compares today's date with the 14th January 2010:

```
var myDate=new Date();
myDate.setFullYear(2010,0,14); var today = new
Date();
if (myDate>today)
 {
 alert("Today is before 15th December 2011");
 }
        else
 {
 alert("Today is after 15th January 2011");
 }
```

Examples

```
<html>
```

<head>

```
<script type="text/javascript">
function displayDate()
{
document.getElementById("demo").innerHTML=Date();
}
</script>
</head>
<body>

<h1>My First Web Page</h1>
<p id="demo">This is a paragraph.</p>

<button type="button" onclick="displayDate()">Display Date</button>

</body>
</html>


<html>
```

```
<body>
<script type="text/javascript"> var d=new Date();
document.write(d);
</script>
</body>
 </html>
```

Example: Displaying the clock

```
<html>
        <head>
        <script type="text/javascript">
```

```
function startTime()
{
var today=new Date(); var
h=today.getHours(); var
m=today.getMinutes(); var
s=today.getSeconds();
// add a zero in front of numbers<10
//m=checkTime(m);
//s=checkTime(s); document.getElementById('txt').innerHTML=h+":"+m+":"+s;
t=setTimeout('startTime()',1000);
}

//to concat 0 if i is not double digit
/*function checkTime(i)
{
if (i<10)
  {
  i="0" + i;
  }
return i;
} */
</script>
</head>
<body onload="startTime()">
<div id="txt"></div>
</body>
</html>
```

With JavaScript, it is possible to execute some code after a specified time-interval. This is called timing events It's very easy to time events in JavaScript. The two key methods that are used are:

- *setTimeout() - executes a code some time in the future*
- *clearTimeout() - cancels the setTimeout()*

Note: *The setTimeout() and clearTimeout() are both methods of the HTML DOM Window object.*

The setTimeout() method returns a value. In the syntax defined above, the value is stored in a variable called t. If you want to cancel the setTimeout() function, you can refer to it using the variable name. The first parameter of setTimeout() can be a string of executable code, or a call to a function. The second parameter indicates how many milliseconds from now you want to execute the first parameter.

Note: *There are 1000 milliseconds in one second.*

In above example the function startTime( ) get executed after each second, showing the content of div tag getting refreshed each time so as to display the clock.

User defined objects in JavaScript:

We have seen that JavaScript has several built-in objects, like String, Date, Array, and more. In addition to these built-in objects, you can also create your own.

An object is just a special kind of data, with a collection of properties and methods.

Let's illustrate with an example: A person is an object. Properties are the values associated with the object. The persons' properties include name, height, weight, age, skin tone, eye color, etc. All persons have these properties, but the values of those properties will differ from person to person. Objects also have methods. Methods are the actions that can be performed on objects. The persons' methods could be eat(), sleep(), work(), play(), etc.

The syntax for accessing a property of an object is:

objName.propName

You can call a method with the following syntax:

objName.methodName()

Note: Parameters required for the method can be passed between the parentheses. There are

different ways to create a new object:

1. Create a direct instance of an object

The following code creates an new instance of an object, and adds four properties to it:

```
personObj=new Object();
personObj.firstname="Jyoti";
personObj.lastname="Joshi";
personObj.age=25;
personObj.eyecolor="black";
```

alternative syntax (using object literals):

```
personObj={firstname:"Jyoti", lastname:"Joshi", age:25, eyecolor:"black"};
```

Adding a method to the personObj is also simple. The following code adds a method called eat() to the personObj:

```
personObj.eat=eat;


function eat( )
{
// code for the function
}
```

2. Create an object constructor

Create a function that constructs objects:

```
function person(firstname,lastname,age,eyecolor)
{
this.firstname=firstname;
this.lastname=lastname; this.age=age;
this.eyecolor=eyecolor;
}
```

Inside the function you need to assign things to this.propertyName. The reason for all the "this" stuff is that you're going to have more than one person at a time (which person you're dealing with must be clear). That's what "this" is: the instance of the object at hand.

Once you have the object constructor, you can create new instances of the object, like this:

```
var myFather=new person("Ramesh","Joshi",50,"black");
var myMother=new person("Gita","Joshi",48,"blue");
```

You can also add some methods to the person object. This is also done inside the function:

```
function person(firstname,lastname,age,eyecolor)
{ this.firstname=firstname;
this.lastname=lastname; this.age=age;
this.eyecolor=eyecolor;

this.newlastname=newlastname;
}
```

Note that methods are just functions attached to objects. Then we will have to write the newlastname( ) function:

```
function newlastname(new_lastname)
{
this.lastname=new_lastname;
}
```

The newlastname( ) function defines the person's new last name and assigns that to the person. JavaScript knows which person you're talking about by using "this." . So, now you can write: myMother.newlastname("Joshi").

Example: Creating a circle object

```
<html>
    <head>
    <script type="text/javascript">
    // mycircle object defined function
    mycircle(r) { this.radius = r;
            this.retArea = getTheArea;
    }
    function getTheArea( )
    {
      return ( Math.PI * this.radius * this.radius );
```

```
}
function createcircle ( )
{
//create a mycircle called testcircle wtih radius 10 var testcircle =
new mycircle(10);
window.alert( 'The area of the circle is ' + testcircle.retArea );
}
</script>
</head>

<body onLoad="createcircle()"> </body>
</html>
```

HTML Document Object Model

The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page. DOM provides a language-independent, object-based model for accessing / modifying and adding to these tags.

The HTML DOM defines a standard set of objects for HTML, and a standard way to access and manipulate HTML documents. All HTML elements, along with their containing text and attributes, can be accessed through the DOM. The contents can be modified or deleted, and new elements can be created. The HTML DOM is platform and language independent. It can be used by any programming language like Java, JavaScript, and VBScript.

When an HTML page is rendered in a browser, the browser assembles all the elements (objects) that are contained in the HTML page, downloaded from web-server in its memory. Once done the browser then renders these objects in the browser window as text,

forms, input boxes, etc. Once the HTML page is rendered in web-browser window, the browser can no longer recognize individual HTML elements (Objects).

Since the JavaScript enabled browser uses the **D**ocument **O**bject **M**odel (DOM), after the page has been rendered, JavaScript enabled browsers are capable of recognizing individual objects in an HTML page.

The HTML objects, which belong to the DOM, have a descending relationship with each other.

The topmost object in the DOM is the **Navigator** (*i.e.* Browser) itself. The next level in the DOM is the browser's **Window**, and under that are the **Documents** displayed in Browser's Window.

DOM

```
|-> Window
    |-> Document
          |-> Anchor
          |-> Link
          |-> Form
               |-> Text-box
               |-> Text Area
               |-> Radio Button
               |-> Check Box
               |-> Select
               |-> Button

               ..........
```

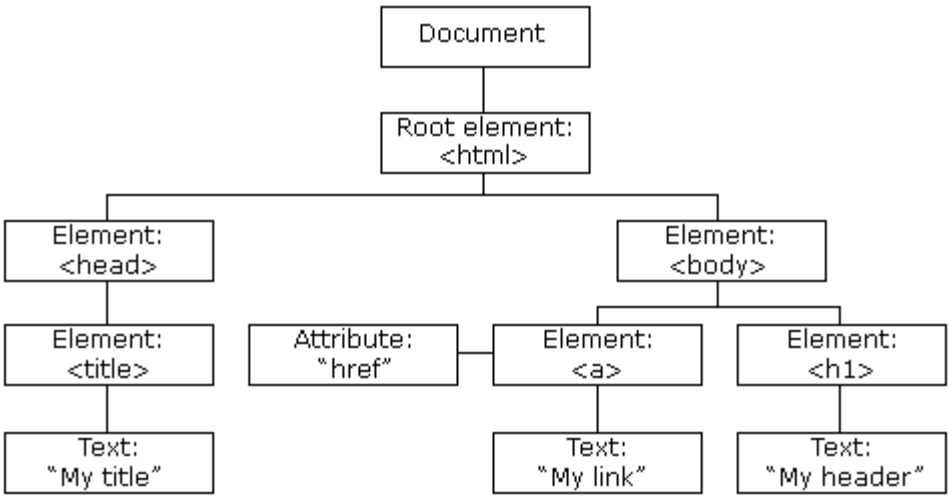


Fig: HTML DOM Tree Example

The Form Object:

The Form object represents an HTML form. For each <form> tag in an HTML document, a Form object is created. Forms are used to collect user input, and contain input elements like text fields, checkboxes, radio-buttons, submit buttons and more. A form can also contain select menus, textarea, fieldset, legend, and label elements. Forms are used to pass data to a server.

Form Object Collections

Collection	Description
elements[]	Returns an array of all elements in a form

Form Object Properties

Property	Description
acceptCharset	Sets or returns the value of the accept-charset attribute in a form
action	Sets or returns the value of the action attribute in a form
enctype	Sets or returns the value of the enctype attribute in a form
length	Returns the number of elements in a form
method	Sets or returns the value of the method attribute in a form
name	Sets or returns the value of the name attribute in a form
target	Sets or returns the value of the target attribute in a form

Form Object Methods

Method	Description
reset()	Resets a form
submit()	Submits a form

Form Object Events

Event	The event occurs when...
onreset	The reset button is clicked
onsubmit	The submit button is clicked

Form Method Property

The method property sets or returns the value of the method attribute in a form. The method attribute specifies how to send form-data (the form-data is sent to the page specified in the action attribute).

formObject.method=*value*

The method property can have one of the following values:

Value	Description
get	Appends the form-data to the URL: URL?name=value&name=value (this is default)
post	Sends the form-data as an HTTP post transaction

RegExp Object:

A regular expression is an object that describes a pattern of characters. When you search in a text, you can use a pattern to describe what you are searching for. A simple pattern can be one single character. A more complicated pattern can consist of more characters, and can be used for parsing, format checking, substitution and more.

Regular expressions are used to perform powerful pattern-matching and "search-and- replace" functions on text.

Syntax

var patt=new RegExp(pattern,modifiers);

or more simply:

var patt=/pattern/modifiers;

- pattern specifies the pattern of an expression
- modifiers specify if a search should be global, case-sensitive, etc.

Modifiers: Modifiers are used to perform case-insensitive and global searches. The ***i*** modifier is used to perform case-insensitive matching. The ***g*** modifier is used to perform a global match (find all matches rather than stopping after the first match).

For example:

```
<html>
<body>

<script type="text/javascript"> var str = "Visit
W3Schools"; var patt1 = /w3schools/i;
document.write(str.match(patt1));
</script>

</body>
</html>
```

The output: W3Schools

```
<html>
<body>

<script type="text/javascript">

var str="Is this all there is?"; var patt1=/is/g;
document.write(str.match(patt1));

</script>

</body>
</html>
```

The output : is, is

```
<html>
<body>

<script type="text/javascript">
```

```
var str="Is this all there is?"; var patt1=/is/gi;
document.write(str.match(patt1));

</script>

</body>
</html>
```

The output : Is,is,is

test()

The test() method searches a string for a specified value, and returns true or false, depending on the result. The following example searches a string for the character "e":

```
<html>
<body>

<script type="text/javascript">
var patt1=new RegExp("e");

document.write(patt1.test("The best things in life are free"));
</script>

</body>
</html>
```

exec()

The exec() method searches a string for a specified value, and returns the text of the found value. If no match is found, it returns *null*. The following example searches a string for the character "e":

```
<html>
<body>

<script type="text/javascript">
var patt1=new RegExp("e");

document.write(patt1.exec("The best things in life are free"));
</script>

</body>
</html>
```

A caret (^) at the beginning of a regular expression indicates that the string being searched must start with this pattern.

- The pattern ^foo can be found in "food", but not in "barfood".

A dollar sign ($) at the end of a regular expression indicates that the string being searched must end with this pattern.

- The pattern foo$ can be found in "curfoo", but not in "food"

Number of Occurrences (? + * {})

The following symbols affect the number of occurrences of the preceding character: ?, +, *, and {}.

A questionmark (?) indicates that the preceding character should appear zero or one times in the pattern.

- The pattern foo? can be found in "food" and "fod", but not "faod".

A plus sign (+) indicates that the preceding character should appear one or more times in the pattern.

- The pattern fo+ can be found in "fod", "food" and "foood", but not "fd".

A asterisk (*) indicates that the preceding character should appear zero or more times in the pattern.

- The pattern fo*d can be found in "fd", "fod" and "food".

Curly brackets with one parameter ({n}) indicate that the preceding character should appear exactly n times in the pattern.

- The pattern fo{3}d can be found in "foood" , but not "food" or "fooood".

Curly brackets with two parameters ({n1,n2}) indicate that the preceding character should appear between n1 and n2 times in the pattern.

- The pattern fo{2,4}d can be found in "food","foood" and "fooood", but not "fod" or "foooood".

Curly brackets with one parameter and an empty second paramenter ({n,}) indicate that the preceding character should appear at least n times in the pattern.

- The pattern fo{2,}d can be found in "food" and "foooood", but not "fod".

Common Characters (. \d \D \w \W \s \S)

A period (.) represents any character except a newline.

- The pattern fo.d can be found in "food", "foad", "fo9d", and "fo*d". Backslash-d (\d)

represents any digit. It is the equivalent of [0-9].

- The pattern fo\dd can be found in "fo1d", "fo4d" and "fo0d", but not in "food" or "fodd".

Backslash-D (\D) represents any character except a digit. It is the equivalent of [^0-9].

- The pattern fo\Dd can be found in "food" and "foad", but not in "fo4d".

Backslash-w (\w) represents any word character (letters, digits, and the underscore (_)).

- The pattern fo\wd can be found in "food", "fo_d" and "fo4d", but not in "fo*d". Backslash-W (\W

) represents any character except a word character.

- The pattern fo\Wd can be found in "fo*d", "fo@d" and "fo.d", but not in "food". Backslash-s (\s)

represents any whitespace character (e.g, space, tab, newline, etc.).

- The pattern fo\sd can be found in "fo d", but not in "food". Backslash-S (\S)

represents any character except a whitespace character.

- The pattern fo\Sd can be found in "fo*d", "food" and "fo4d", but not in "fo d".

Form Validation:

Form validation is the process of checking that a form has been filled in correctly before it is processed. For example, if your form has a box for the user to type their email address, you might want your form handler to check that they've filled in their address before you deal with the rest of the form.

There are two main methods for validating forms: ***server-side*** (using CGI scripts, ASP, etc), and ***client-side*** (usually done using JavaScript). Server-side validation is more secure but often more tricky to code and it also increases load of server computer, whereas client- side (JavaScript) validation is easier to do and quicker too (the browser doesn't have to

connect to the server to validate the form, so the user finds out instantly if they've missed out that required field!) and it also decreases the load of server computer and hence server computer can focus on business logic processing.

Form Validation - Checking for Non-Empty

This has to be the most common type of form validation. You want to be sure that your visitors enter data into the HTML fields you have "required" for a valid submission. Below is the JavaScript code to perform this basic check to see if a given HTML input is empty or not.

```
<script type='text/javascript'>
function notEmpty()
{
        var v= document.getElementById('elem').value;
        if(v.length == 0)
        {
                alert("Field should not be empty:");
                document.getElementById('elem').value=" ";
                document.getElementById('elem').focus();
        }
}
</script>
<form>
Required Field: <input type='text' id='elem'/>
<input type='button' onclick="notEmpty()" value='Check'/>
</form>
```

Form Validation - Checking for All Numbers

If someone is entering a credit card, phone number, zip code, similar information you want to be able to ensure that the input is all numbers. The quickest way to check if an input's

string value is all numbers is to use a regular expression /^[0-9]+$/ that will only *match* if the string is all numbers and is at least one character long.

```
<script type='text/javascript'>
function validate()
{
        var patt=/^[0-9]+$/;
        var v= document.getElementById('elem').value;
        if(v.match(patt))
        {

                                                                        }
                                                                        else
                                                                        {

                                                                        }
}
```

```
alert("valid entry");



alert("Invalid entry:"); document.getElementById('elem').value="";
document.getElementById('elem').focus();
 </script>
 <form>
 Required Field: <input type='text' id='elem'/>
 <input type='button' onclick="validate()" value='Check'/>
 </form>
```

Form Validation - Checking for All Letters

If we wanted to see if a string contained only letters we need to specify an expression that allows for both lowercase and uppercase letters: /^[a-zA-Z]+$/ .

```
<script type='text/javascript'>
function validate()
{
        var patt=/^[a-zA-Z]+$/;
```

```
var v= document.getElementById('elem').value;
if(v.match(patt))
{

                                                                    }
                                                                    else
                                                                    {



                                                                    }
}
```

```
alert("valid entry");



alert("Invalid entry:"); document.getElementById('elem').value="";
document.getElementById('elem').focus();
 </script>
 <form>
 Required Field: <input type='text' id='elem'/>
 <input type='button' onclick="validate()" value='Check'/>
 </form>
```

Form Validation - Restricting the Length

Being able to restrict the number of characters a user can enter into a field is one of the best ways to prevent bad data. Below we have created a function that checks for length of input.

```
<script type='text/javascript'>
function validate()
{
        var minlen=6;
        var v= document.getElementById('elem').value;
        if(v.length<6)
        {
```

```
}
else
{
```

```
alert("User ID must have at least 6 Characters");
document.getElementById('elem').value="";
document.getElementById('elem').focus();



alert("Valid entry:");

        }
}
</script>
<form>
User ID: <input type='text' id='elem'/>
<input type='button' onclick="validate()" value='Check'/>
</form>
```

Form Validation - Selection Made

To be sure that someone has actually selected a choice from an HTML select input you can use a simple trick of making the first option as helpful prompt to the user and a red flag to you for your validation code. By making the first option of your select input something like "Please Choose" you can spur the user to both make a selection and allow you to check to see if the default option "Please Choose" is still selected when he/she submit the form.

```
<script type='text/javascript'>
function validate()
{
        var si=document.getElementById('con').selectedIndex; var v=
        document.getElementById('con').options[si].text; if(v=="Please Choose")
        {
                alert("You must choose the country");
```

```
}
else
{
```

```
alert("Your Country is:"+v);

        }
}
</script>
<form>
Select Country: <select id='con'>
        <option>Please Choose</option> <option>Nepal</option>
        <option>India</option> <option>China</option>
</select>
<input type='button' onclick='validate()' value='Check'/>
</form>
```

Validating radio buttons

Radio buttons are used if we want to choose any one out of many options such as gender. In such case any one of the radio button must be selected. We can validate radio button selection as below:

```
<script type='text/javascript'>
function validate()
{
        var sex=document.getElementsByName("gen");
        if(sex[0].checked==false && sex[1].checked==false)
        {

                                                                }
                                                                else
                                                                {
```

```
alert("You must choose Gender");
```

```
            if(sex[0].checked==true)
            alert("Male"); else alert("Female");
        }
}
</script>
<form>
Select Gender:
        <input type=radio name='gen'>Male
        <input type=radio name='gen'>Female
        <input type='button' onclick='validate()' value='Check'/>
</form>
```

Form Validation - Email Validation

How to check to see if a user's email address is valid? Every email is made up for 5 parts:

1. A combination of letters, numbers, periods, hyphens, plus signs, and/or underscores
2. The at symbol @
3. A combination of letters, numbers, hyphens, and/or periods
4. A period
5. The top level domain (com, net, org, us, gov, ...) Valid Examples:

- jagdish@ntc.net
- jagdish+bhatta@gmail.com
- jagdish-bhatta@patan.edu.np

Invalid Examples:

- @deleted.net - no characters before the @
- free!dom@bravehe.art - invalid character !
- shoes@need_shining.com - underscores are not allowed in the domain name

```
<script type='text/javascript'>
function validate()
{
        var patt=/^[\w\-\.\+]+\@[a-zA-Z0-9\.\-]+\.[a-zA-z0-9]{2,4}$/; var v=
        document.getElementById('elem').value; if(v.match(patt))
        {

                                                                    }
                                                                    else
                                                                    {


                                                                    }
}
```

```
alert("valid Email");

alert("Invalid Email"); document.getElementById('elem').value="";
document.getElementById('elem').focus();
 </script>
 <form>
        Email ID: <input type='text' id='elem'/>
        <input type='button' onclick="validate()" value='Check'/>
 </form>
```

Handling Cookies in JavaScript:

A cookie is a variable that is stored on the visitor's computer. Each time the same computer requests a page with a browser, it will send the cookie too. With JavaScript, you can both create and retrieve cookie values. A cookie is nothing but a small text file that's stored in your browser. It contains some data:

1. A name-value pair containing the actual data
2. An expiry date after which it is no longer valid
3. The domain and path of the server it should be sent to

As soon as you request a page from a server to which a cookie should be sent, the cookie is added to the HTTP header. Server side programs can then read out the information and decide that you have the right to view the page you requested. So every time you visit the site the cookie comes from, information about you is available. This is very nice sometimes, at other times it may somewhat endanger your privacy. Cookies can be read by JavaScript too. They're mostly used for storing user preferences.

Examples of cookies:

- Name cookie - The first time a visitor arrives to your web page, he or she must fill in her/his name. The name is then stored in a cookie. Next time the visitor arrives at your page, he or she could get a welcome message like "Welcome John Doe!" The name is retrieved from the stored cookie
- Password cookie - The first time a visitor arrives to your web page, he or she must fill in a password. The password is then stored in a cookie. Next time the visitor arrives at your page, the password is retrieved from the cookie
- Date cookie - The first time a visitor arrives to your web page, the current date is stored in a cookie. Next time the visitor arrives at your page, he or she could get a message like "Your last visit was on Tuesday August 11, 2005!" The date is retrieved from the stored cookie
    - And so on.

document.cookie:

Cookies can be created, read and erased by JavaScript. They are accessible through the property document.cookie. Though you can treat document.cookie as if it's a string, it isn't really, and you have only access to the name-value pairs. If you want to set a cookie for this domain with a name-value pair 'ppkcookie1=testcookie' that expires in seven days from the moment you should write this sentence,

document.cookie = "ppkcookie1=testcookie; expires=Thu, 2 Aug 2001 20:47:11 UTC; path=/"

1. First the name-value pair ('ppkcookie1=testcookie')
2. then a semicolon and a space

3. then the expiry date in the correct format ('expires=Thu, 2 Aug 2001 20:47:11 UTC')
4. again a semicolon and a space
5. then the path (path=/)

Example:

```
function createCookie(name, value, days) {
   if (days) {
      var date = new Date();
      date.setTime(date.getTime() + (days * 24 * 60 * 60 * 1000));
      var expires = "; expires=" + date.toGMTString();
   }
   else var expires = "";
   document.cookie = name + "=" + value + expires + "; path=/";
}

function getCookie(c_name) {
   if (document.cookie.length > 0) {
      c_start = document.cookie.indexOf(c_name + "=");
      if (c_start != -1) {
         c_start = c_start + c_name.length + 1;
         c_end = document.cookie.indexOf(";", c_start);
         if (c_end == -1) {
                        c_end = document.cookie.length;
         }
         return unescape(document.cookie.substring(c_start, c_end));
      }
   }
   return "";
}
```

More we can set cookie as below with the proper paths, domain and other parameters;

```
function setCookie(name, value, expires, path, domain)
{

      /* Some characters - including spaces - are not allowed in cookies so we escape to change the value
      we have entered into a form acceptable to the cookie.*/
```

```
        var thisCookie = name + "=" + escape(value) +
        ((expires) ? "; expires=" + expires.toGMTString() : "") + ((path) ? "; path="
        + path : "") +
        ((domain) ? "; domain=" + domain : "") ;

        document.cookie = thisCookie;

}
```

Simply we can display cookie in alert box as;

```
function showCookie()
{
        alert(unescape(document.cookie));
}
```

More Example;

```
<html>
<head>
<script type="text/javascript">
function setCookie()
{
    var name="Cookie1"; var
    value="Jagdish"; var ed=new Date();
    ed.setDate(ed.getDate() +2);
    document.cookie = name + "=" + value+" ;expires="+ed.toGMTString()+" ;path=/";
}
function getCookie()
{
            var l=document.cookie.length;
            setCookie();
```

```
var ind=document.cookie.indexOf("Cookie1=");
if(ind==-1)
{
                                                                    }
                                                                    else
                                                                    {


                                                                    }
}
```

```
alert("Cookie not found");

var n=document.cookie.substring(ind+8,l);
alert("Wel come:"+n);
 </script> </head>
 <body>
        <input type=button value="setcookie" onclick="setCookie()">
        <input type=button value="getcookie" onclick="getCookie()">
 </body> </html>
```

Handling runtime errors in JavaScript:

An exception is an error that occurs at **runtime** due to an illegal operation during execution. Examples of exceptions include trying to reference an undefined variable, or calling a non-existent method. S**yntax** errors occur when there is a problem with your JavaScript syntax. Consider the following examples of syntax errors versus exceptions:

```
alert("I am missing a closing parenthesis //syntax error alert(x)
//exception assuming "x" isn't defined yet undefinedfunction()
//exception
```

It is almost impossible for a programmer to write a program without errors. Programming languages include exceptions, or errors, that can be tracked and controlled. Exception handling is a very important concept in programming technology. In earlier versions of

JavaScript, the exceptions handling was not so efficient and programmers found it difficult to use. Later versions of JavaScript resolved this difficulty with exceptions handling features like try..catch handlers, which presented a more convenient solution for programmers. Normally whenever the browser runs into an exception somewhere in a JavaScript code, it displays an error message to the user while aborting the execution of the remaining code. There are mainly two ways of trapping errors in JavaScript.

- Using try…catch statement
- Using onerror event

Using try…catch statement:

The try..catch statement has two blocks in it: try block and catch block. In the try block, the code contains a block of code that is to be tested for errors. The catch block contains the code that is to be executed if an error occurs. The general syntax of try..catch statement is as follows:

```
try
{



}
catch (err)
{



}
```

……

……//Block of code which is to be tested for errors

……

…… //Block of code which is to be executed if an error occurs

When, in the above structure, an error occurs in the try block then the control is immediately transferred to the catch block with the error information also passed to the catch block. Thus, the try..catch block helps to handle errors without aborting the program and therefore proves user-friendly.

<html>

<head>

```
<script type="text/javascript">
var txt="";
function message()
{
try
{


}
```

```
adddlert("Welcome guest!");
alert("test");
        catch(err)
        {
                txt="There was an error on this page.\n\n"; txt+="Click OK to
                continue viewing this page,\n"; txt+="or Cancel to return to the home
                page.\n\n"; if(!confirm(txt))
                {

                                                                                }
                                                        }
  }
 </script>
 </head>
 <body>
```

```
document.location.href="http://www.w3schools.com/";

</body>
</html>
```

```
<input type="button" value="View message" onclick="message()" />
```

There is another statement called throw available in JavaScript that can be used along with. try…catch statements to throw exceptions and thereby helps in generating. General syntax of this throw statement is as follows:

```
throw(exception)
```

```
<html>
 <body>
        <script type="text/javascript">
        try
        {
                var a=10; var b=0; if(b==0)
                {
                        throw "Division by zero!!!!"
                }
        }

        catch(err)
        {
                alert(err);
        }
        </script>
</body>
</html>
```

Although finally is not used as often as catch, it can often be useful. The finally clause is guaranteed to be executed if any portion of the try block is executed, regardless of how the code in the try block completes. It is generally used to clean up after the code in the try clause. If an exception occurs in the try block and there is an associated catch block to handle the exception, control transfers first to the catch block and then to the finally block. If there is no local catch block to handle the exception, control transfers first to the finally.

```
<head>
<script type="text/javascript">
<!--
        function myFunc()
        {
          var a = 100;
          try
          {
            alert("Value of variable a is : " + a );
          }
```

```
        catch ( e )
        {
            alert("Error: " + e.description );
        }
        finally
        {
            alert("Finally block will always execute!" );
        }
        }
        //-->
</script>
</head>
<body>
<p>Click the following to see the result:</p>
<form>
<input type="button" value="Click Me" onclick="myFunc()" />
</form>
</body>
</html>
```

Using onerror event

The onerror event fires when a page has a script error. This onerror event occurs in JavaScript when an image or document causes an error during loading. This does not mean that it is a browser error. This event handler will only be triggered by a JavaScript error, not a browser error. The general syntax of onerror event is as follows:

```
onerror=functionname()
function functionname()
{
  //Error Handling Code
}
```

Example:

```
<html>
<head>
<script type="text/javascript">
        onerror=exfoerr var text1=""
        function exfoerr(msg,url,line)
        {
            text1="Error Displayed\n\n" text1+="Error: " + msg + "\n"
            text1+="URL: " + url + "\n" text1+="Line Number: " +
            line + "\n\n" text1+="Click OK to continue.\n\n"
            alert(text1)
            return true
        }
        function display()
        {
            addxlert("Click to Proceed!!!!")
        }
    </script>
  </head>
   <body>
     <input type="button" value="View message"
      onclick="display()" />
  </body>
</html>
```

In the above example program, the function display() has an error in it (the addalert is typed wrongly as addxlert). When the program reads this error, the onerror event handler

fires and the function exfor( ) is called with the three parameters passed to it (the error message, the url of the page and the line number of error 18)

[Unit 2: Issues of Web Technology]

Web Technology (CSC-353)

Jagdish Bhatta

Central Department of Computer Science & Information Technology

Tribhuvan University

Architectural Issues of Web Layer:

The **web layer** is also referred to as the UI layer. The web layer is primarily concerned with presenting the user interface and the behavior of the application (handling user interactions/events). While the web layer can also contain logic, core application logic is usually located in the services layer. T**he three Layers within the Web Layer are:**

- **HTML-The Content Layer:** The content layer is where you store all the content that your customers want to read or look at. This includes text and images as well as multimedia. It's also important to make sure that every aspect of your site is represented in the content layer. That way, your customers who have JavaScript turned off or can't view CSS will still have access to the entire site, if not all the functionality.

- **CSS - the Styles Layer:** Store all your styles for your Web site in an external style sheet. This defines the way the pages should look, and you can have separate style sheets for various media types. Store your CSS in an external style sheet so that you can get the benefits of the style layer across the site.

- **JavaScript - the Behavior Layer:** JavaScript is the most commonly used language for writing the behavior layer; ASP, CGI and PHP can also generate Web page behaviors. However, when most developers refer to the behavior layer, they mean that layer that is activated directly in the Web browser - so JavaScript is nearly always the language of choice. You use this layer to interact directly with the DOM or Document Object Model.

When you're creating a Web page, it is important to keep the layers separate. Using external style sheets is the best way to separate your content from your design. And the same is true for using external JavaScript files. Some of the benefits of separating the layers are:

- **Shared resources:** When you write an external CSS file or JavaScript file, you can use that file by any page on your Web site. There is no duplication of effort, and whenever the file changes, it changes for every page that uses it without you making more than one change.

- **Faster downloads:** Once the script or stylesheet has been downloaded by your customer the first time, it is cached. Then every other page that is downloaded loads more quickly in the browser window.

- **Multi-person teams:** If you have more than one person working on a Web site at once, you can divide up the workload without worrying about permissions or content management. You can also hire people who are style/design experts to work on the CSS while your scripters work on the JavaScript, and your writers work in the content files.
- **Accessibility:** External style sheets and script files are more accessible to more browsers, because they can be ignored more easily, and because they provide more options. For example, you can set up a style sheet that is displayed only for screen readers or a script library that's only used by people on cell phones.

- **Backwards compatibility:** When you have a site that is designed with the development layers, it will be more backwards compatible because browsers that can't use technology like CSS and JavaScript can still view the HTML.

HTTP (Hypertext Transfer Protocol):

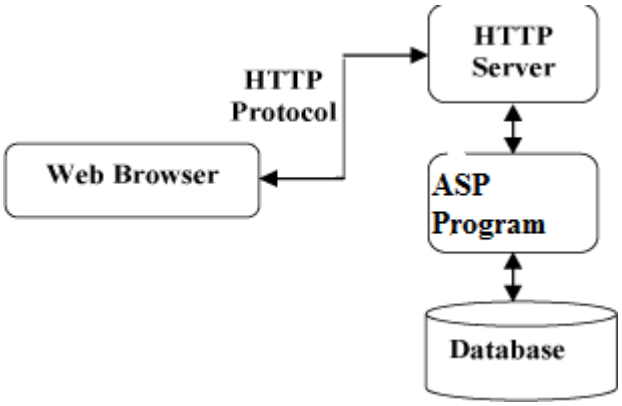
HTTP stands for **Hypertext Transfer Protocol**. It is a TCP/IP based communication protocol which is used to deliver virtually all files and other data, collectively called resources, on the World Wide Web. These resources could be HTML files, image files, query results, or anything else. A browser is works as an HTTP client because it sends requests to an HTTP server which is called Web server. The Web Server then sends responses back to the client. The standard and default port for HTTP servers to listen on is 80 but it can be changed to any other port like 8080 etc. There are three important things about HTTP of which you should be aware:

- **HTTP is connectionless:** After a request is made, the client disconnects from the server and waits for a response. The server must re-establish the connection after it processes the request.

- **HTTP is media independent:** Any type of data can be sent by HTTP as long as both the client and server know how to handle the data content. How content is handled is determined by the MIME specification.

- **HTTP is stateless:** This is a direct result of HTTP's being connectionless. The server and client are aware of each other only during a request. Afterwards, each forgets the other. For this reason neither the client nor the browser can retain information between different requests across the web pages.

Following diagram shows where HTTP Protocol fits in communication;



Like most network protocols, HTTP uses the client-server model: An HTTP client opens a connection and sends a request message to an HTTP server; the server then returns a response message, usually containing the resource that was requested. After delivering the response, the server closes the connection. The format of the request and response messages is similar and will have following structure:

- An initial line CRLF
- Zero or more header lines CRLF
- A blank line i.e. a CRLF
- An optional message body like file, query data or query output.

CR and LF here mean ASCII values 13 and 10. The initial line is different for the request than for the response. A request line has three parts, separated by spaces: An HTTP Method Name, the local path of the requested resource, the version of HTTP being used. Example of initial line for Request Message is: "GET /path/to/file/index.html HTTP/1.0". The initial response line, called the status line, also has three parts separated by spaces: The version of HTTP being used, a response status code that gives the result of the request, an English reason phrase describing the status code. Example, HTTP/1.0 200 OK or "HTTP/1.0 404 Not Found"

Header lines provide information about the request or response, or about the object sent in the message body. The header lines are in the usual text header format, which is: one line per header, of the form "Header-Name: value", ending with CRLF. Example of Header Line is "User-agent: Mozilla/3.0Gold" or "Last-Modified: Fri, 31 Dec 1999 23:59:59
GMT".

An HTTP message may have a body of data sent after the header lines. In a response, this is where the requested resource is returned to the client (the most common use of the message body), or perhaps explanatory text if there's an error. In a request, this is where user-entered data or uploaded files are sent to the server.

HTTP: header fields

HTTP header fields are components of the message header of requests and responses in the Hypertext Transfer Protocol (HTTP). They define the operating parameters of an HTTP transaction.

The header fields are transmitted after the request or response line, the first line of a message. Header fields are colon-separated name-value pairs in clear-text string format, terminated by a carriage return (CR) and line feed (LF) character sequence. The end of the header fields is indicated by an empty field, resulting in the transmission of two consecutive CR-LF pairs. Long lines can be folded into multiple lines; continuation lines are indicated by presence of space (SP) or horizontal tab (HT) as first character on next line. Few fields can also contain comments (i.e. in. User-Agent, Server, Via fields), which can be ignored by software.

There are no limits to size of each header field name or value, or number of headers in standard itself. However most servers, clients and proxy software, impose some limits for practical and security reasons. For example; Apache 2.3 server by default limits each header size to 8190 bytes, and there can be at most 100 headers in single request.

HTTP Session:

An HTTP session is a sequence of network request-response transactions. An HTTP client initiates a request by establishing a Transmission Control Protocol (TCP) connection to a particular port on a server (typically port 80). An HTTP server listening on that port waits for a client's request message. Upon receiving the request, the server sends back a status line, such as "HTTP/1.1 200 OK", and a message of its own, the body of which is perhaps the requested resource, an error message, or some other information

File Transfer Protocol:

File Transfer Protocol (FTP) lives up to its name and provides a method for transferring files over a network from one computer to another. More generally, it provides for some simple file management on the contents of a remote computer. It is an old protocol and is used less than it was before the World Wide Web came along. Today, its primary use is uploading files to a Web site. It can also be used for downloading from the Web but, more often than not, downloading is done via HTTP. Sites that have a lot of downloading (software sites, for example) will often have an FTP server to handle the traffic. If FTP is involved, the URL will have *ftp:* at the front.

The File Transfer Protocol is used to send files from one system to another under user commands. Both text and binary files are accommodated and the protocol provides features for controlling user access. When a user wishes to engage in File transfer, FTP sets up a TCP connection to the target system for the exchange of control messages. These allow used ID and password to be transmitted and allow the user to specify the file and file action desired. Once file transfer is approved, a second TCP connection is set up for data transfer. The file is transferred over the data connection, without the overhead of headers,

or control information at the application level. When the transfer is complete, the control connection is used to signal the completion and to accept new file transfer commands.

FTP can be run in active or passive mode, which determines how the data connection is established. In active mode, the client sends the server the IP address and port number on which the client will listen, and the server initiates the TCP connection. at the condition when the client is behind a firewall and unable to accept incoming TCP connections, passive mode may be used. In this mode the client sends a PASV command to the server and receives an IP address and port number in return. The client uses these to open the data connection to the server. Data transfer can be done in any of three modes:

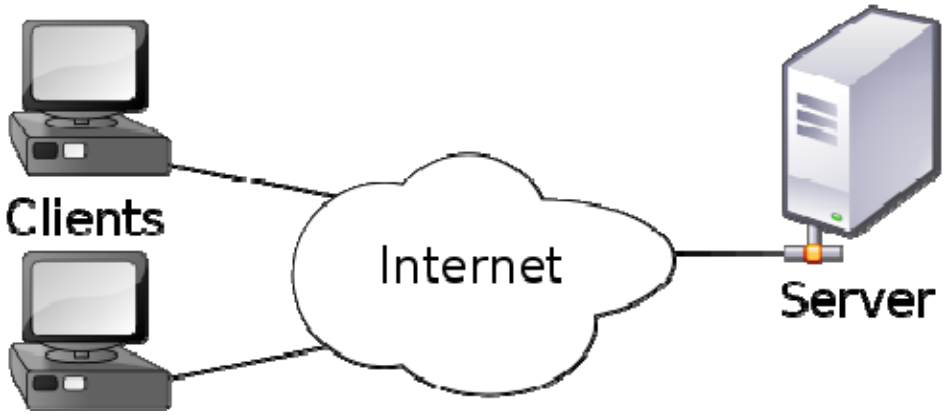
- Stream mode: Data is sent as a continuous stream, relieving FTP from doing any processing. Rather, all processing is left up to TCP. No End-of-file indicator is needed, unless the data is divided into records.
- Block mode: FTP breaks the data into several blocks (block header, byte count, and data field) and then passes it on to TCP.
- Compressed mode: Data is compressed using a single algorithm (usually run- length encoding).

Client/Server Model:

The **client–server model** is a computing model that acts as distributed application which partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients. Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system. A server machine is a host that is running one or more server programs which share their resources with clients. A client does not share any of its resources, but requests a server's content or service function. Clients therefore initiate communication sessions with servers which await incoming requests.

Client/Server Architecture:

Client server network architecture consists of two kinds of computers: clients and servers. Clients are the computers that that do not share any of its resources but requests data and other services from the server computers and server computers provide services to the client computers by responding to client computers requests. Normally servers are powerful computers and clients are less powerful personal computers. Web servers are included as part of a larger package of internet and intranet related programs for serving e- mail, downloading requests for FTP files and building and publishing web pages.



Advantages

- The client/ server architecture reduces network traffic by providing a query response to the user rather than transferring total files.
- The client/ server model improves multi-user updating through a graphical user interface (GUI) front end to the shared database.
- Easy to implement security policies, since the data are stored in central location
- Simplified network administration

Disadvantages

- Failure of the server causes whole network to be collapsed
- Expensive than P2P, Dedicated powerful servers are needed
- Extra effort are needed for administering and managing the server.

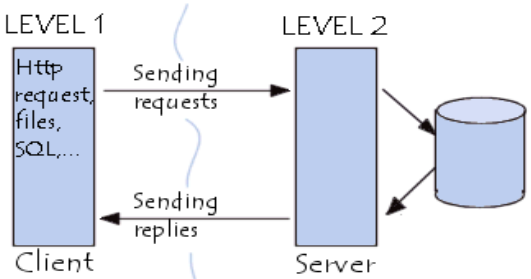
Client/Sever architecture can be of different model based on the number of layers it holds. Some of them are;

- **2-Tier Architecture**

  2-tier architecture is used to describe client/server systems where the client requests resources and the server responds directly to the request, using its own resources. This means that the server does not call on another application in order to provide part of the service. It runs the client processes separately from the server processes, usually on a different computer:

  - The client processes provide an interface for the customer, and gather and present data usually on the customer's computer. This part of the application is the presentation layer
  - The server processes provide an interface with the data store of the business. This part of the application is the data layer

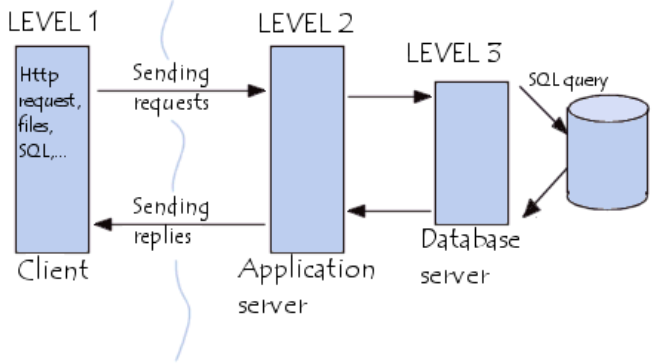
- The business logic that validates data, monitors security and permissions, and performs other business rules can be housed on either the client or the server, or split between the two.
  - Fundamental units of work required to complete the business process
  - Business rules can be automated by an application program.



- **3-Tier Architecture**

In 3-tier architecture, th ure is an intermediary level, meaning that archi tec generally split up between:

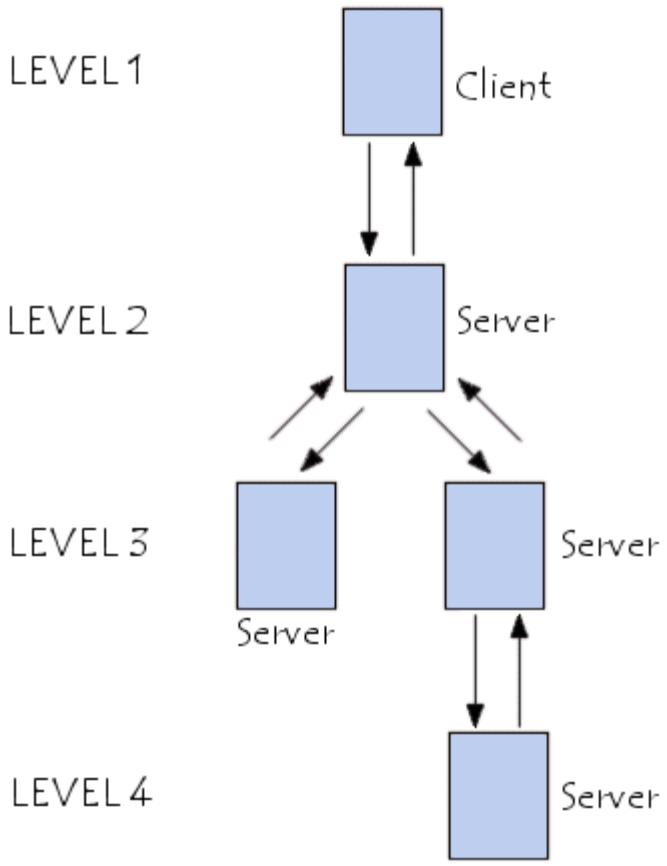
- A client, i.e. the computer, which requests the resources, equipped with a user interface (usually a web browser) for presentation purposes
- The application server (also called **middleware**), whose task it is to provide the requested resources, but by calling on another server
- The data server, which provides the application server with the data it requires



- **N-Tier Architecture (multi-tier)**

N-tier architecture (with N more than 3) is really 3 tier architectures in which the middle tier is split up into new tiers. The application tier is broken down into separate parts. What these parts are differs from system to system. The following picture shows it:

The primary advantage of N-tier architectures is that they make load balancing possible. Since the application logic is distributed between several servers, processing can then be more evenly distributed among those servers. N-tiered architectures are also more easily scalable, since only servers experiencing high demand, such as the application server, need be upgraded. The primary disadvantage of N-tier architectures is that it is also more difficult to program and test an N-tier architecture due to its increased complexity.



Advantages of Multi-Tier Client/Server architectures include:

- Changes to the user interface or to the application logic are largely independent from one another, allowing the application to evolve easily to meet new requirements.
- Network bottlenecks are minimized because the application layer does not transmit extra data to the client, only what is needed to handle a task.
- The client is insulated from database and network operations. The client can access data easily and quickly without having to know where data is or how many servers are on the system.
- Database connections can be 'pooled' and thus shared by several users, which greatly reduces the cost associated with per-user licensing.
- The organization has database independence because the data layer is written using standard SQL which is platform independent. The enterprise is not tied to vendor- specific stored procedures.
- The application layer can be written in standard third or fourth generation languages, such as ASP, PHP with which the organization's in-house programmers are experienced.

What kind of systems can benefit?

Generally, any Client/Server system can be implemented in an 'N-Tier' architecture, where application logic is partitioned among various servers. This application partitioning creates an integrated information infrastructure which enables consistent, secure, and global access to critical data. A significant reduction in network traffic, which leads to faster netwo rk communications, greater reliability, and greater overall performance is also made possible in a 'N-Tier' Client/Server architecture.

[Unit 3: XML]

Web Technology (CSC-353)

Jagdish Bhatta

Central Department of Computer Science & Information Technology

Tribhuvan University

Introduction::

As we have studied in unit one that HTML is designed to display data. In contrast, XML is designed to transport and store data. XML stands for EXtensible Markup Language and is much like HTML. XML was designed to carry data, not to display data. XML tags are not predefined. You must define your own tags. XML is designed to be self-descriptive. **Extensible Markup Language** (**XML**) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

XML is not a replacement for HTML. XML and HTML were designed with different goals:

- XML was designed to transport and store data, with focus on what data is
- HTML was designed to display data, with focus on how data looks

HTML is about displaying information, while XML is about carrying information.

Maybe it is a little hard to understand, but XML does not DO anything. XML was created to structure, store, and transport information. The following example is a note to Tulsi, from Giri, stored as XML:

```
<note>
<to>Tulsi</to>
<from>Giri</from>
<heading>Reminder</heading>
<body>Don't forget to bunk web tech class at Patan!</body>
</note>
```

The note above is quite self descriptive. It has sender and receiver information, it also has a heading and a message body. But still, this XML document does not DO anything. It is just information wrapped in tags. Someone must write a piece of software to send, receive or display it.

The tags in the example above (like <to> and <from>) are not defined in any XML standard. These tags are "invented" by the author of the XML document. That is because the XML language has no predefined tags. However, the tags used in HTML are predefined. HTML documents can only use tags defined in the HTML standard (like <p>,
<h1>, etc.). In contrast, XML allows the author to define his/her own tags and his/her own
document structure. The XML processor can not tell us which elements and attributes are

valid. As a result we need to define the XML markup we are using. To do this, we need to define the markup language's grammar. There are numerous "tools" that can be used to build an XML language – some relatively simple, some much more complex. They include DTD (Document Type Definition), RELAX, TREX, RELAX NG, XML Schema, Schmatron, etc.

The design goals for XML are:

1. XML shall be straightforwardly usable over the Internet.
2. XML shall support a wide variety of applications.
3. XML shall be compatible with SGML.
4. It shall be easy to write programs which process XML documents.
5. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
6. XML documents should be human-legible and reasonably clear.
7. The XML design should be prepared quickly.
8. The design of XML shall be formal and concise.
9. XML documents shall be easy to create.

XML Usages

XML is used in many aspects of web development, often to simplify data storage and sharing.

XML Separates Data from HTML: If you need to display dynamic data in your HTML document, it will take a lot of work to edit the HTML each time the data changes. With XML, data can be stored in separate XML files. This way you can concentrate on using HTML for layout and display, and be sure that changes in the underlying data will not require any changes to the HTML. With a few lines of JavaScript code, you can read an external XML file and update the data content of your web page.

XML Simplifies Data Sharing: In the real world, computer systems and databases contain data in incompatible formats. XML data is stored in plain text format. This provides a software- and hardware-independent way of storing data. This makes it much easier to create data that can be shared by different applications.

XML Simplifies Data Transport: One of the most time-consuming challenges for developers is to exchange data between incompatible systems over the Internet. Exchanging data as XML greatly reduces this complexity, since the data can be read by different incompatible applications.

XML Simplifies Platform Changes: Upgrading to new systems (hardware or software platforms), is always time consuming. Large amounts of data must be converted and incompatible data is often lost. XML data is stored in text format. This makes it easier to expand or upgrade to new operating systems, new applications, or new browsers, without losing data.

XML Makes Your Data More Available: Different applications can access your data, not only in HTML pages, but also from XML data sources. With XML, your data can be available to all kinds of "reading machines" (Handheld computers, voice machines, news feeds, etc), and make it more available for blind people, or people with other disabilities.

XML Used to Create New Internet Languages: A lot of new Internet languages are created with XML. Here are some examples:

- XHTML
- WSDL (Web Services Description Language) for describing available web services
- WAP and WML (Wireless Markup Language) as markup languages for handheld devices
- RSS (Really Simple Syndication / Rich Site Summary) languages for news feeds
- RDF (Resource Description Framework), a family of w3c spec, and OWL (Web Ontology Language) for describing resources and ontology
- SMIL (Synchronized Multimedia Integration Language) for describing multimedia for the web

XML Tree

XML documents form a tree structure that starts at "the root" and branches to "the leaves". XML documents use a self-describing and simple syntax:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
        <to>Tulsi</to>
        <from>Giri</from>
         <heading>Reminder</heading>
        <body>Don't forget to bunk the web tech class at Patan!</body>
</note>
```

The first line is the XML declaration. It defines the XML version (1.0) and the encoding used (ISO-8859-1 = Latin-1/West European character set). The next line describes the **root element** of the document (like saying: "this document is a note"):

<note>

The next 4 lines describe 4 **child elements** of the root (to, from, heading, and body):

<to>Tulsi</to>
<from>Giri</from>
<heading>Reminder</heading>
<body>Don't forget to bunk the web tech class at Patan!</body>

And finally the last line defines the end of the root element:

</note>

You can assume, from this example, that the XML document contains a note to Tulsi from Giri.

Thus, XML documents must contain a **root element**. This element is "the parent" of all other elements. The elements in an XML document form a document tree. The tree starts at the root and branches to the lowest level of the tree. All elements can have sub elements (child elements):

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

The terms parent, child, and sibling are used to describe the relationships between elements. Parent elements have children. Children on the same level are called siblings (brothers or sisters). All elements can have text content and attributes (just like in HTML).

XML Syntax Rules

The syntax rules of XML are very simple and logical. The rules are easy to learn, and easy to use.

1. **All XML Elements Must Have a Closing Tag**. In HTML, some elements may not have to have a closing tag, like;

   ```
   <p>This is a paragraph.
   <br>
   ```

   In XML, it is illegal to omit the closing tag. All elements must have a closing tag:

   ```
   <p>This is a paragraph.</p>
   <br />
   <hello> This is hello </hello>
   ```

2. **XML tags are case sensitive**. The tag <Letter> is different from the tag <letter>. Opening and closing tags must be written with the same case:

   ```
   <Message>This is incorrect</message>
   <message>This is correct</message>
   ```

3. **XML Elements Must be Properly Nested**. In HTML, you might see improperly nested elements:

```
<b><i>This text is bold and italic</b></i>
```

In XML, all elements must be properly nested within each other:

```
<b><i>This text is bold and italic</i></b>
```

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

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

5. **XML Attribute Values Must be Quoted.** XML elements can have attributes in name/value pairs just like in HTML. In XML, the attribute values must always be quoted. Study the two XML documents below. The first one is incorrect, the second is correct:

```
<note date=06/01/2012>
  <to>Tulsi</to>
  <from>Giri</from>
</note>

<note date="06/01/2012">
  <to>Tulsi</to>
  <from>Giri</from>
</note>
```

The error in the first document is that the date attribute in the note element is not quoted.

6. **Entity Reference.** Some characters have a special meaning in XML. If you place a character like "<" inside an XML element, it will generate an error because the parser interprets it as the start of a new element. This will generate an XML error:

```
<message>if salary < 1000 then</message>
```

To avoid this error, replace the "<" character with an **entity reference**:

```
<message>if salary &lt; 1000 then</message> There are 5
```

predefined entity references in XML:

&lt;	<	less than
&gt;	>	greater than
&amp;	&	ampersand
&apos;	'	apostrophe
&quot;	"	quotation mark

7. **Comments in XML.** The syntax for writing comments in XML is similar to that of HTML.

```
<!-- This is a comment -->
```

8. **White-space is preserved in XML**. HTML truncates multiple white-space characters to one single white-space:

HTML: Hello Tulsi
Output: Hello Tulsi

With XML, the white-space in a document is not truncated.

XML Elements

An XML document contains XML Elements. An XML element is everything from (including) the element's start tag to (including) the element's end tag. An element can

contain:

- other elements
- text
- attributes
- or a mix of all of the above... Consider an

example;

```
<bookstore>

  <book category="CHILDREN">
  <title>Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>

<book category="WEB">
  <title>Learning XML</title>
  <author>Erik T. Ray</author>
```

```
  <year>2003</year>
  <price>39.95</price>
 </book>

</bookstore>
```

In the example above, <bookstore> and <book> have **element contents**, because they contain other elements. <book> also has an **attribute** (category="CHILDREN"). <title>, <author>, <year>, and <price> have **text content** because they contain text.

XML Naming Rules

XML elements must follow these naming rules:

- Names can contain letters, numbers, and other characters
- Names cannot start with a number or punctuation character
- Names cannot start with the letters xml (or XML, or Xml, etc)
- Names cannot contain spaces
- Any name can be used, no words are reserved.

Best Naming Practices

- Make names descriptive. Names with an underscore separator are nice: <first_name>, <last_name>.
- Names should be short and simple, like this: <book_title> not like this: <the_title_of_the_book>.
- Avoid "-" characters. If you name something "first-name," some software may think you want to subtract name from first.
- Avoid "." characters. If you name something "first.name," some software may think that "name" is a property of the object "first."
- Avoid ":" characters. Colons are reserved to be used for something called namespaces (more later).
- XML documents often have a corresponding database. A good practice is to use the naming rules of your database for the elements in the XML documents.
- Non-English letters like éòá are perfectly legal in XML, but watch out for problems if your software vendor doesn't support them.

XML Elements are Extensible

XML elements can be extended to carry more information. Look at the following XML example:

```
<note>
<to>Tulsi</to>
<from>Giri</from>
<body>Don't forget to bunk the web tech class at Patan!</body>
</note>
```

Let's imagine that we created an application that extracted the <to>, <from>, and <body> elements from the XML document to produce this output:

MESSAGE To: Tulsi
From: Giri

Don't forget to bunk the web tech class at Patan!

Suppose the XML document has been modified by adding some extra information to it like:

```
<note>
<date>2012-01-06</date>
<to>Tulsi</to>
<from>Giri</from>
<heading>Reminder</heading>
<body>Don't forget to bunk thee web tech class at Patan!</body>
</note>
```

Should the application break or crash?

No. The application should still be able to find the <to>, <from>, and <body> elements in the XML document and produce the same output. Thus, one of the beauties of XML, is that it can be extended without breaking applications.

XML Attributes

XML elements can have attributes, just like HTML. Attributes provide additional information about an element. In HTML, attributes provide additional information about elements:

```
<img src="computer.gif">
<a href="demo.asp">
```

Attributes often provide information that is not a part of the data. In the example below, the file type is irrelevant to the data, but can be important to the software that wants to manipulate the element:

<file type="gif">computer.gif</file>

Attribute values must always be quoted. Either single or double quotes can be used. For a person's sex, the person element can be written like this:

```
<person sex="male">
```

or like this:

```
<person sex='male'>
```

If the attribute value itself contains double quotes you can use single quotes, like in this example:

```
<gangster name='Chota "Shotgun" Chetan'>
```

or you can use character entities:

```
<gangster name="Chota &quot;Shotgun&quot; Chetan">
```

XML Elements vs. Attributes

Take a look at these examples:

```
<person sex="male">
    <firstname>Jagdish</firstname>
    <lastname>Bhatta</lastname>
</person>

<person>
    <sex>male</sex>
    <firstname>Jagdish</firstname>
    <lastname>Bhatta</lastname>
</person>
```

In the first example sex is an attribute. In the last, sex is an element. Both examples provide the same information. There are no rules about when to use attributes or when to use elements. Attributes are handy in HTML. In XML my advice is to avoid them. Use elements instead.

Writing in different ways

The following three XML documents contain exactly the same information: A date attribute is

used in the first example:

```
<note date="10/01/2008">
  <to>Tulsi</to>
  <from>Giri</from>
  <heading>Reminder</heading>
  <body>Don't forget to bunk the web tech class at Patan!</body>
</note>
```

A date element is used in the second example:

```
<note>
  <date>10/01/2008</date>
  <to>Tulsi</to>
  <from>Giri</from>
  <heading>Reminder</heading>
  <body>Don't forget to bunk the web tech class at Patan!</body>
</note>
```

An expanded date element is used in the third:

```
<note>
  <date>
                <day>10</day>
                <month>01</month>
    <year>2008</year>
  </date>
  <to>Tulsi</to>
  <from>Giri</from>
  <heading>Reminder</heading>
  <body>Don't forget to bunk the web tech class at Patan!</body>
</note>
```

Restrictions with XML Attributes

Some of the problems with using attributes are:

- attributes cannot contain multiple values (elements can)
- attributes cannot contain tree structures (elements can)
- attributes are not easily expandable (for future changes)

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

XML Attributes for Metadata

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

```
<messages>
            <note id="501">
  <to>Tulsi</to>
  <from>Giri</from>
  <heading>Reminder</heading>
  <body>Don't forget to bunk the web tech class at Patan!</body>
 </note>

            <note id="502">
  <to>Giri</to>
  <from>Tulsi</from>
  <heading>Re: Reminder</heading>
  <body>Ok Giri dai !!</body>
 </note>
</messages>
```

The id attributes above are for identifying the different notes. It is not a part of the note itself. In other words, metadata (data about data) should be stored as attributes, and the data itself should be stored as elements.

XML Validation

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

Well Formed XML Documents

A "Well Formed" XML document has correct XML syntax. The syntax rules as described in previous sections are:

- XML documents must have a root element
- XML elements must have a closing tag
- XML tags are case sensitive
- XML elements must be properly nested
- XML attribute values must be quoted

Consider the earlier example;

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>Tulsi</to>
<from>Giri</from>
<heading>Reminder</heading>
<body>Don't forget to bunk the web tech class at Patan!</body>
</note>
```

Now, a "Valid" XML document is a "Well Formed" XML document, which also conforms to the rules of a Document Type Definition (DTD):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE note SYSTEM "Note.dtd">
<note>
<to>Tulsi</to>
<from>Giri</from>
<heading>Reminder</heading>
<body>Don't forget to bunk the web tech class at Patan!</body>
</note>
```

The DOCTYPE declaration in the example above, is a reference to an external DTD file. The purpose of a DTD is to define the structure of an XML document. It defines the structure with a list of legal elements. For above example the DTD seems like;

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

W3C supports an XML-based alternative to DTD, called XML Schema:

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

<xs:complexType>
            <xs:sequence>
              <xs:element name="to" type="xs:string"/>
              <xs:element name="from" type="xs:string"/>
  <xs:element name="heading" type="xs:string"/>
              <xs:element name="body" type="xs:string"/>
            </xs:sequence>
```

</xs:complexType>

</xs:element>

XML schema:

An **XML schema** is a description of a type of XML document, typically expressed in terms of constraints on the structure and content of documents of that type, above and beyond the basic syntactical constraints imposed by XML itself. These constraints are generally expressed using some combination of grammatical rules governing the order of elements, Boolean predicates that the content must satisfy, data types governing the content of elements and attributes, and more specialized rules such as uniqueness and referential integrity constraints.

Technically, a **schema** is an abstract collection of metadata, consisting of a set of **schema components**: chiefly element and attribute declarations and complex and simple type definitions. These components are usually created by processing a collection of **schema documents**, which contain the source language definitions of these components. In popular usage, however, a schema document is often referred to as a schema.

Schema documents are organized by namespace: all the named schema components belong to a target namespace, and the target namespace is a property of the schema document as a whole. A schema document may *include* other schema documents for the same namespace, and may *import* schema documents for a different namespace.

There are languages developed specifically to express XML schemas. The **Document Type Definition (DTD)** language, which is native to the XML specification, is a schema language that is of relatively limited capability, but that also has other uses in XML aside from the expression of schemas. Two more expressive XML schema languages in widespread use are **XML Schema** (with a capital *S*) and **RELAX NG** (REgular LAnguage for XML Next Generation).

There is some confusion as to when to use the capitalized spelling "Schema" and when to use the lowercase spelling. The lowercase form is a generic term and may refer to any type of schema, including DTD, XML Schema (aka XSD), RELAX NG, or others, and should always be written using lowercase except when appearing at the start of a sentence. The form "Schema" (capitalized) in common use in the XML community always refers to W3C XML Schema.

XML Namespace:

XML Namespaces provide a method to avoid element name conflicts. In XML, element names are defined by the developer. This often results in a conflict when trying to mix XML documents from different XML applications. Consider following examples;

This XML carries HTML table information:

```
<table>
        <tr>
  <td>Apples</td>
  <td>Bananas</td>
        </tr>
</table>
```

This XML carries information about a table (a piece of furniture):

```
<table>
 <name>African Coffee Table</name>
 <width>80</width>
 <length>120</length>
</table>
```

If these XML fragments were added together, there would be a name conflict. Both contain a <table> element, but the elements have different content and meaning.

An XML parser will not know how to handle these differences.

Thus, **xmlns** tagged **XML namespaces** are used for providing uniquely named elements and attributes in an XML document. They are defined in a W3C recommendation. An XML instance may contain element or attribute names from more than one XML vocabulary. If each vocabulary is given a namespace, the ambiguity between identically named elements or attributes can be resolved. The XML namespace is a special type of *reserved XML attribute* that you place in an XML tag. The reserved attribute is actually more like a prefix that you attach to any namespace you create. This *attribute prefix* is "**xmlns:**", which stands for XML NameSpace. The colon is used to separate the prefix from your namespace that you are creating.

A *namespace name* is a uniform resource identifier (URI). Typically, the URI chosen for the namespace of a given XML vocabulary describes a resource under the control of the author or organisation defining the vocabulary, such as a URL for the author's Web server. However, the namespace specification does not require nor suggest that the namespace URI be used to retrieve information; it is simply treated by an XML parser as a string. For example, the document at http://www.w3.org/1999/xhtml itself does not contain any code

The name conflicts in above mentioned example can be handled by using the concept of namespace as a name prefix, as below ;

This XML carries information about an HTML table, and a piece of furniture:

```
<h:table>
        <h:tr>
  <h:td>Apples</h:td>
  <h:td>Bananas</h:td>
```

```
    </h:tr>
</h:table>

<f:table>
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>
```

When using prefixes in XML, a so-called **namespace** for the prefix must be defined. The namespace is defined by the **xmlns attribute** in the start tag of an element. The namespace declaration has the following syntax. xmlns:*prefix*="*URI*".

```
<root>

<h:table xmlns:h="http://www.w3.org/TR/html4/">
    <h:tr>
      <h:td>Apples</h:td>
      <h:td>Bananas</h:td>
    </h:tr>
</h:table>

<f:table xmlns:f="http://www.w3schools.com/furniture">
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>

</root>
```

In the example above, the xmlns attribute in the <table> tag give the h: and f: prefixes a qualified namespace. When a namespace is defined for an element, all child elements with the same prefix are associated with the same namespace.

Namespaces can be declared in the elements where they are used or in the XML root element:

```
<root xmlns:h="http://www.w3.org/TR/html4/"
xmlns:f="http://www.w3schools.com/furniture">

<h:table>
    <h:tr>
      <h:td>Apples</h:td>
      <h:td>Bananas</h:td>
    </h:tr>
```

```
</h:table>

<f:table>
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>

</root>
```

The namespace URI is not used by the parser to look up information. The purpose is to give the namespace a unique name. However, often companies use the namespace as a pointer to a web page containing namespace information.

XML schema languages

- **DTD**
- **XML Schema**

Document Type Definition (DTD)

DTD is an approach for defining the structure of XML Document. It is an XML schema language whose purpose is to define legal building blocks of an XML document. A DTD defines the document structure with a list of legal elements and attributes.

Document Type Definition (**DTD**) is a set of *markup declarations* that define a *document type* for SGML-family markup languages (SGML, XML, HTML). DTDs were a precursor to XML schema and have a similar function, although different capabilities.

DTDs use a terse formal syntax that declares precisely which elements and references may appear where in the document of the particular type, and what the elements' contents and attributes are. DTDs also declare entities which may be used in the *instance* document. XML uses a subset of SGML DTD.

We use DTD because with a DTD, each of your XML files can carry a description of its own format. With a DTD, independent groups of people can agree to use a standard DTD for interchanging data. Your application can use a standard DTD to verify that the data you receive from the outside world is valid. You can also use a DTD to verify your own data.

A Document Type Declaration associates a DTD with an XML document. Document Type Declarations appear in the syntactic fragment *doctypedecl* near the start of an XML

document. The declaration establishes that the document is an instance of the type defined by the referenced DTD.

DTDs make two sorts of declaration:

- an optional *external subset*
- an optional *internal subset*

The declarations in the internal subset form part of the Document Type Declaration in the document itself. The declarations in the external subset are located in a separate text file.

If the DTD is declared inside the XML file, it should be wrapped in a DOCTYPE definition with the following syntax:

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

Example XML document with an internal DTD:

```
<?xml version="1.0"?>
<!DOCTYPE note [
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
<note>
<to>Tulsi</to>
<from>Giri</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend</body>
</note>
```

The DTD above is interpreted like this:

- **!DOCTYPE note** defines that the root element of this document is note
- **!ELEMENT note** defines that the note element contains four elements: "to,from,heading,body"
- **!ELEMENT to** defines the to element to be of type "#PCDATA"
- **!ELEMENT from** defines the from element to be of type "#PCDATA"
- **!ELEMENT heading** defines the heading element to be of type "#PCDATA"
- **!ELEMENT body** defines the body element to be of type "#PCDATA"

If the DTD is declared in an external file, it should be wrapped in a DOCTYPE definition. Here, DTD is present in separate file and a reference is placed to its location in

the document. External DTD's are easy to apply to multiple documents. In case, a modification is to be made in future, it could be done in just one file and the onerous task of doing it for all the documents is omitted. External DTDs are of two types: **private** and **public**.

Private external DTDs are identified by the keyword `SYSTEM`, and are intended for use by a single author or group of authors. Its syntax is:

<!DOCTYPE root-element SYSTEM "DTD location">.

For Example, the listed below is the same XML document as above, but with an external DTD.

```
<?xml version="1.0"?>
<!DOCTYPE note SYSTEM "note.dtd">
<note>
  <to>Tulsi</to>
  <from>Girii</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

And this is the file "note.dtd" which contains the DTD:

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

Public external DTDs are identified by the keyword `PUBLIC` and are intended for broad use. Its syntax is:
<!DOCTYPE root_element PUBLIC "DTD_name" "DTD_location">.

The DTD_name follows the syntax: "prefix//owner_of_the_DTD//description_of_the_DTD//ISO 639_language_identifier".

For example,

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
"http://www.w3.org/TR/REC-html40/loose.dtd">

The following prefixes are allowed in the DTD name:

Prefix:	Definition:
ISO	The DTD is an ISO standard. All ISO standards are approved.
+	The DTD is an approved non-ISO standard.
-	The DTD is an unapproved non-ISO standard.

Defining Elements:

Elements are the main building blocks of XML documents. In a DTD, elements are declared with an ELEMENT declaration with the following syntax.

<!ELEMENT element-name category> Or

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

Empty elements are declared with the category keyword EMPTY. Its syntax is: <!ELEMENT element-name EMPTY>. For example, <!ELEMENT br EMPTY>.

Elements with only parsed character data are declared with #PCDATA inside parentheses. Its syntax is: <!ELEMENT element-name (#PCDATA)>. For example,

<!ELEMENT from (#PCDATA)>.

Elements with any content are declared with the category keyword ANY, can contain any combination of parsable data. Its syntax is: <!ELEMENT element-name ANY>. For example, <!ELEMENT note ANY>.

Elements with one or more children are declared with the name of the children elements inside parentheses. Its syntax is <!ELEMENT element-name (child1, child2,…)>. For example, <!ELEMENT note (to,from,body)>.

When children are declared in a sequence separated by commas, the children must appear in the same sequence in the document. In a full declaration, the children must also be declared, and the children can also have children.

We can declare **only one occurrence of an element**. Its syntax is: <!ELEMENT element- name (child-name)>. For example, <!ELEMENT note (message)>. This example declares that the child element "message" must occur once, and only once inside the "note" element.

We can also declare **minimum one occurrence of an element**. Its syntax is <!ELEMENT element-name (child-name+)>. For example, <!ELEMENT note (message+)>. The + sign in the example above declares that the child element "message" must occur one or more times inside the "note" element.

Note: We can use * in place of + to declare zero or more occurrence of an element. We can use ? in place of + to declare zero or one occurrence of an element

We can also declare **either/or content.** For example, <!ELEMENT note (to,from,header,(message|body))>. This example declares that the "note" element must contain a "to" element, a "from" element, a "header" element, and either a "message" or a "body" element.

We can declare **mixed content**. For example, <!ELEMENT note (#PCDATA|to|from|header|message)*>. This example declares that the "note" element can contain zero or more occurrences of parsed character data, "to", "from", "header", or "message" elements.

Defining Attributes

In a DTD, attributes are declared with an **ATTLIST** declaration. An attribute declaration has the following syntax:

<!ATTLIST element-name attribute-name attribute-type default-value> For example,

<!ATTLIST payment type CDATA "check"> And its XML

example is

<payment type="check" />

The **attribute-type** can be one of the following:

Type	Description
CDATA	The value is character data (text that doesn't contain

	markup)
(*en1*|*en2*|..)	The value must be one from an enumerated list
ID	The value is a unique id
IDREF	The value is the id of another element
IDREFS	The value is a list of other ids
NMTOKEN	The value is a valid XML name
NMTOKENS	The value is a list of valid XML names separated by whitespace
ENTITY	The name of an entity (which must be declared in the DTD)
ENTITIES	The value is a list of entities, separated by whitespace
NOTATION	The value is a name of a notation (which must be declared in the DTD)
xml:	The value is a predefined xml value

The **default-value** can be one of the following:

Value	**Explanation**
Value	The default value of the attribute. For example, <!ATTLIST square width CDATA "0">
#REQUIRED	The attribute is required. For example, <!ATTLIST person number CDATA #REQUIRED>
#IMPLIED	The attribute is not required (optional). For example, <!ATTLIST contact fax CDATA #IMPLIED>
#FIXED *value*	The attribute value is fixed. For example, <!ATTLIST sender company CDATA #FIXED "Microsoft">

A Default attribute value:

Example:
DTD
<!ELEMENT square EMPTY>
<!ATTLIST square width CDATA "0">

Valid XML:
<square width="100" />

In the example above, the "square" element is defined to be an empty element with a "width" attribute of type CDATA. If no width is specified, it has a default value of 0.

#REQUIRED:

Syntax:
<!ATTLIST element-name attribute-name attribute-type #REQUIRED>

Example:
DTD:
<!ATTLIST person number CDATA #REQUIRED>

Valid XML:
<person number="5677" />

Invalid XML:
<person />

Use the #REQUIRED keyword if you don't have an option for a default value, but still want to force the attribute to be present.

#IMPLIED:

Syntax:
<!ATTLIST element-name attribute-name attribute-type #IMPLIED>

Example:
DTD:
<!ATTLIST contact fax CDATA #IMPLIED>

Valid XML:
<contact fax="555-667788" />

Valid XML:
<contact />

Use the #IMPLIED keyword if you don't want to force the author to include an attribute, and you don't have an option for a default value.

#FIXED:

Syntax:

<!ATTLIST element-name attribute-name attribute-type #FIXED "value">

Example:

DTD:
<!ATTLIST sender company CDATA #FIXED "Microsoft">

Valid XML:
<sender company="Microsoft" />

Invalid XML:
<sender company="W3Schools" />

Use the #FIXED keyword when you want an attribute to have a fixed value without allowing the author to change it. If an author includes another value, the XML parser will return an error.

Enumerated Attribute Values:

Syntax:

<!ATTLIST element-name attribute-name (en1|en2|..) default-value>

Example:

DTD:
<!ATTLIST payment type (check|cash) "cash">

XML example:
<payment type="check" />
or
<payment type="cash" />

Use enumerated attribute values when you want the attribute value to be one of a fixed set of legal values.

DTD Examples:

```
<!DOCTYPE NEWSPAPER [

<!ELEMENT NEWSPAPER (ARTICLE+)>
<!ELEMENT ARTICLE (HEADLINE,BYLINE,LEAD,BODY,NOTES)>
<!ELEMENT HEADLINE (#PCDATA)>
<!ELEMENT BYLINE (#PCDATA)>
<!ELEMENT LEAD (#PCDATA)>
<!ELEMENT BODY (#PCDATA)>
<!ELEMENT NOTES (#PCDATA)>

<!ATTLIST ARTICLE AUTHOR CDATA #REQUIRED>
<!ATTLIST ARTICLE EDITOR CDATA #IMPLIED>
<!ATTLIST ARTICLE DATE CDATA #IMPLIED>
<!ATTLIST ARTICLE EDITION CDATA #IMPLIED>

]>
```

XML Schema

XML Schema is a XML schema language which is an alternative to DTD. XML Schema is an XML-based alternative to DTD. Unlike DTD, XML Schemas has support for data types and namespaces. The XML Schema language, also referred to as XML Schema Definition (XSD), is used to define XML schema.

An XML Schema:

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

XML Schemas are the successors of DTDs. In near future, XML Schemas will be used in most Web applications as a replacement for DTDs because of the following reasons;

- **XML Schemas are extensible to future additions**
- **XML Schemas are richer and more powerful than DTDs**
- **XML Schemas are written in XML**
- **XML Schemas support data types**
- **XML Schemas support namespaces**

DTDs are better for text-intensive applications, while schemas have several advantages for data-intensive workflows. Schemas are written in XML and thusly follow the same rules, while DTDs are written in a completely different language.

The <schema> Element:

The <schema> element is the root element of every XML Schema.

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

<xs:schema>
...
...
</xs:schema>
```

The <schema> element may contain some attributes. A schema declaration often looks something like this:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.w3schools.com"
xmlns="http://www.w3schools.com" elementFormDefault="qualified">
...
...

</xs:schema>
```

The code fragment **xmlns:xs="http://www.w3.org/2001/XMLSchema"** indicates that the elements and data types used in the schema come from the "http://www.w3.org/2001/XMLSchema" namespace. It also specifies that the elements and data types that come from the "http://www.w3.org/2001/XMLSchema" namespace should be prefixed with **xs: .**

The code fragment **targetNamespace="http://www.w3schools.com"** indicates that the elements defined by this schema (note, to, from, heading, bod y.) come from the "http://www.w3schools.com" namespace.

The code fragment **xmlns="http://www.w3schools.com"** indicates that the default namespace is "http://www.w3schools.com".

The code fragment **elementFormDefault="qualified"** indicates that any elements used by the XML instance document which were declared in this schema must be namespace qualified.

Referencing a Schema in an XML Document:

XML documents can have a reference to an XML Schema. For example consider the following "note.xml" file. This file has a reference the "note.xsd" schema.

```
<?xml version="1.0"?>
<note xmlns="http://www.w3schools.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.w3schools.com note.xsd">
<to>Tulsi</to>
<from>Giri</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

The code fragment **xmlns="http://www.w3schools.com"** specifies the default namespace declaration. This declaration tells the schema-validator that all the elements used in this XML document are declared in the "http://www.w3schools.com" namespace.

The code fragment **xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"** is the namespace.

In the code fragment **xsi:schemaLocation="http://www.w3schools.com note.xsd"**, there are two attribute values. The first value is the namespace to use. The second value is the location of the XML schema to use for that namespace.

The following example is an XML Schema file called "note.xsd" that defines the elements of the XML document above ("note.xml"):

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.w3schools.com"
xmlns="http://www.w3schools.com" elementFormDefault="qualified">
<xs:element name="note">
   <xs:complexType>
            <xs:sequence>
   <xs:element name="to" type="xs:string"/>
   <xs:element name="from" type="xs:string"/>
   <xs:element name="heading" type="xs:string"/>
   <xs:element name="body" type="xs:string"/>
            </xs:sequence>
   </xs:complexType>
</xs:element>
</xs:schema>
```

Here, the note element is a **complex type** because it contains other elements. The other elements (to, from, heading, body) are **simple types** because they do not contain other elements.

XSD Simple Type: Consists of simple elements and attributes.

XSD Simple Elements:

A simple element is an XML element that can contain only text. It cannot contain any other elements or attributes. The text can be of many different types. It can be one of the types included in the XML Schema definition (Boolean, string, date, etc.), or it can be a custom type that you can define yourself. You can also add restrictions (facets) to a data type in order to limit its content, or you can require the data to match a specific pattern.

The syntax for defining a simple element is:

<xs:element name="xxx" type="yyy"/> , where xxx is the name of the element and yyy is the data type of the element. XML Schema has a lot of built-in data types. The most common types are:

- xs:string

- xs:decimal
- xs:integer
- xs:boolean
- xs:date
- xs:time

For Example;

Consider the XML elements;

```
<lastname>Bhatta</lastname>
<age>42</age>
<dateborn>1970-03-27</dateborn>
```

And here are the corresponding simple element definitions:

```
<xs:element name="lastname" type="xs:string"/>
<xs:element name="age" type="xs:integer"/>
<xs:element name="dateborn" type="xs:date"/>
```

Default and Fixed Values for Simple Elements:

Simple elements may have a default value OR a fixed value specified. A default value is automatically assigned to the element when no other value is specified In the following example the default value is "red":

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

A fixed value is also automatically assigned to the element, and you cannot specify another value. In the following example the fixed value is "red":

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

XSD Attributes:

Simply attributes are associated with the complex elements. If an element has attributes, it is considered to be of a complex type. Simple elements cannot have attributes. But the attribute itself is always declared as a simple type. All attributes are declared as simple types.

The syntax for defining an attribute is:

<xs:attribute name="xxx" type="yyy"/> , where xxx is the name of the attribute and yyy specifies the data type of the attribute.

XML Schema has a lot of built-in data types. The most common types are:

- xs:string
- xs:decimal
- xs:integer
- xs:boolean
- xs:date
- xs:time

Example

Here is an XML element with an attribute:

<lastname lang="EN">Smith</lastname> And here is the

corresponding attribute definition:

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

Default and Fixed Values for Attributes:

Attributes may have a default value OR a fixed value specified. A default value is automatically assigned to the attribute when no other value is specified. In the following example the default value is "EN":

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

A fixed value is also automatically assigned to the attribute, and you cannot specify another value.

In the following example the fixed value is "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.

With XML Schemas, you can also add your own restrictions to your XML elements and attributes. These restrictions are called facets.

XSD Restrictions/ Facets:

1. **Restrictions on Values**

The following example defines an element called "age" with a restriction. 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>
```

2. **Restrictions on a Set of Values**

To limit the content of an XML element to a set of acceptable values, we would use the enumeration constraint. 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>
```

The example above could also have been written like this:

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

Note: *In this case the type "carType" can be used by other elements because it is not a part of the "car" element.*

3. Restrictions on a Series of Values

To limit the content of an XML element to define a series of numbers or letters that can be used, we would use the pattern constraint.

The example below defines an element called "letter" with a restriction. The only acceptable value is ONE of the LOWERCASE letters from a to z:

```
<xs:element name="letter">
            <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[a-z]"/>
    </xs:restriction>
            </xs:simpleType>
</xs:element>
```

The next example defines an element called "initials" with a restriction. The only acceptable value is THREE of the UPPERCASE letters from a to z:

```
<xs:element name="initials">
            <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[A-Z][A-Z][A-Z]"/>
    </xs:restriction>
            </xs:simpleType>
</xs:element>
```

The next example also defines an element called "initials" with a restriction. The only acceptable value is THREE of the LOWERCASE OR UPPERCASE letters from a to z:

```
<xs:element name="initials">
            <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[a-zA-Z][a-zA-Z][a-zA-Z]"/>
```

```
            </xs:restriction>
          </xs:simpleType>
</xs:element>
```

The next example defines an element called "choice" with a restriction. The only acceptable value is ONE of the following letters: x, y, OR z:

```
<xs:element name="choice">
          <xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:pattern value="[xyz]"/>
            </xs:restriction>
          </xs:simpleType>
</xs:element>
```

The next example defines an element called "zipcode" with a restriction. The only acceptable value is FIVE digits in a sequence, and each digit must be in a range from 0 to 9:

```
<xs:element name="zipcode">
          <xs:simpleType>
  <xs:restriction base="xs:integer">
    <xs:pattern value="[0-9][0-9][0-9][0-9][0-9]"/>
            </xs:restriction>
          </xs:simpleType>
</xs:element>
```

4. Restrictions on Whitespace Characters

To specify how whitespace characters should be handled, we would use the whiteSpace constraint. This example defines an element called "address" with a restriction. The whiteSpace constraint is set to "preserve", which means that the XML processor WILL NOT remove any white space characters:

```
<xs:element name="address">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:whiteSpace value="preserve"/>
          </xs:restriction>
</xs:simpleType>
</xs:element>
```

This example also defines an element called "address" with a restriction. The whiteSpace constraint is set to "replace", which means that the XML processor WILL REPLACE all white space characters (line feeds, tabs, spaces, and carriage returns) with spaces:

```
<xs:element name="address">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:whiteSpace value="replace"/>
            </xs:restriction>
</xs:simpleType>
</xs:element>
```

This example also defines an element called "address" with a restriction. The whiteSpace constraint is set to "collapse", which means that the XML processor WILL REMOVE all white space characters (line feeds, tabs, spaces, carriage returns are replaced with spaces, leading and trailing spaces are removed, and multiple spaces are reduced to a single space):

```
<xs:element name="address">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:whiteSpace value="collapse"/>
            </xs:restriction>
</xs:simpleType>
</xs:element>
```

5. Restrictions on Length:

To limit the length of a value in an element, we would use the length, maxLength, and minLength constraints. This example defines an element called "password" with a restriction. The value must be exactly eight characters:

```
<xs:element name="password">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:length value="8"/>
            </xs:restriction>
</xs:simpleType>
</xs:element>
```

This example defines another element called "password" with a restriction. The value must be minimum five characters and maximum eight characters:

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

```
</xs:simpleType>
</xs:element>
```

Restrictions for Data types

Constraint	Description
Enumeration	Defines a list of acceptable values
fractionDigits	Specifies the maximum number of decimal places allowed. Must be equal to or greater than zero
Length	Specifies the exact number of characters or list items allowed. Must be equal to or greater than zero
maxExclusive	Specifies the upper bounds for numeric values (the value must be less than this value)
maxInclusive	Specifies the upper bounds for numeric values (the value must be less than or equal to this value)
maxLength	Specifies the maximum number of characters or list items allowed. Must be equal to or greater than zero
minExclusive	Specifies the lower bounds for numeric values (the value must be greater than this value)
minInclusive	Specifies the lower bounds for numeric values (the value must be greater than or equal to this value)
minLength	Specifies the minimum number of characters or list items allowed. Must be equal to or greater than zero
Pattern	Defines the exact sequence of characters that are acceptable
totalDigits	Specifies the exact number of digits allowed. Must be greater than zero
whiteSpace	Specifies how white space (line feeds, tabs, spaces, and carriage returns) is handled

XSD Complex Types:

A complex element is an XML element that contains other elements and/or attributes. There are four kinds of complex elements:

- empty elements
- elements that contain only other elements
- elements that contain only text
- elements that contain both other elements and text

Note: Each of these elements may contain attributes as well!

Examples of Complex Elements

A complex XML element, "product", which is empty:

```
<product pid="1345"/>
```

A complex XML element, "employee", which contains only other elements:

```
<employee>
  <firstname>Jagdish</firstname>
  <lastname>Bhatta</lastname>
</employee>
```

A complex XML element, "food", which contains only text:

```
<food type="dessert">Ice cream</food>
```

A complex XML element, "description", which contains both elements and text:

```
<description>
It happened on <date lang="Nepali">03/09/2099</date> ....
</description>
```

How to Define a Complex Element

Look at this complex XML element, "employee", which contains only other elements:

```
<employee>
        <firstname>Jagdishfirstname>
        <lastname>Smith</lastname>
</employee>
```

We can define a complex element in an XML Schema two different ways:

1. The "employee" element can be declared directly by naming the element, like this:

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

If you use the method described above, only the "employee" element can use the specified complex type. Note that the child elements, "firstname" and "lastname", are surrounded by the <sequence> indicator. This means that the child elements must appear in the same order as they are declared.

2. The "employee" element can have a type attribute that refers to the name of the complex type to use:

```
<xs:element name="employee" type="personinfo"/>

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

If you use the method described above, several elements can refer to the same complex type, like this:

```
<xs:element name="employee" type="personinfo"/>
<xs:element name="student" type="personinfo"/>
<xs:element name="member" type="personinfo"/>

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

You can also base a complex element on an existing complex element and add some elements, like this:

```
<xs:element name="employee" type="fullpersoninfo"/>

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

<xs:complexType name="fullpersoninfo">
  <xs:complexContent>
```

```
<xs:extension base="personinfo">
            <xs:sequence>
   <xs:element name="address" type="xs:string"/>
   <xs:element name="city" type="xs:string"/>
   <xs:element name="country" type="xs:string"/>
  </xs:sequence>
          </xs:extension>
</xs:complexContent>
        </xs:complexType>
```

Types of XSD Complex Elements

1. **XSD Empty Element**

An empty complex element cannot have contents, only attributes. Consider an empty XML element:

```
<product prodid="1345" />
```

The "product" element above has no content at all. To define a type with no content, we must define a type that allows elements in its content, but we do not actually declare any elements, like this:

```
<xs:element name="product">
            <xs:complexType>
  <xs:complexContent>
   <xs:restriction base="xs:integer">
                <xs:attribute name="prodid" type="xs:positiveInteger"/>
               </xs:restriction>
  </xs:complexContent>
            </xs:complexType>
</xs:element>
```

In the example above, we define a complex type with a complex content. The complexContent element signals that we intend to restrict or extend the content model of a complex type, and the restriction of integer declares one attribute but does not introduce any element content.

However, it is possible to declare the "product" element more compactly, like this:

```
<xs:element name="product">
            <xs:complexType>
  <xs:attribute name="prodid" type="xs:positiveInteger"/>
            </xs:complexType>
</xs:element>
```

Or you can give the complexType element a name, and let the "product" element have a type attribute that refers to the name of the complexType (if you use this method, several elements can refer to the same complex type):

```
<xs:element name="product" type="prodtype"/>

<xs:complexType name="prodtype">
  <xs:attribute name="prodid" type="xs:positiveInteger"/>
</xs:complexType>
```

2. XSD Elements only

An "elements-only" complex type contains an element that contains only other elements. Consider an XML element "person", that contains only other elements:

```
<person>
  <firstname>Jagdish</firstname>
  <lastname>Bhatta</lastname>
</person>
```

You can define the "person" element in a schema, like this:

```
<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>
```

Notice the <xs:sequence> tag. It means that the elements defined ("firstname" and "lastname") must appear in that order inside a "person" element.

Or you can give the complexType element a name, and let the "person" element have a type attribute that refers to the name of the complexType (if you use this method, several elements can refer to the same complex type):

```
<xs:element name="person" type="persontype"/>

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

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

3. XSD Text only Elements

A complex text-only element can contain text and attributes. This type contains only simple content (text and attributes), therefore we add a simpleContent element around the content. When using simple content, you must define an extension OR a restriction within the simpleContent element, like this:

```
<xs:element name="somename">
        <xs:complexType>
          <xs:simpleContent>
    <xs:extension base="basetype">
              ....
              ....
            </xs:extension>
          </xs:simpleContent>
        </xs:complexType>
</xs:element> OR

<xs:element name="somename">
        <xs:complexType>
          <xs:simpleContent>
    <xs:restriction base="basetype">
              ....
              ....
            </xs:restriction>
          </xs:simpleContent>
        </xs:complexType>
</xs:element>
```

Note: You can use the extension/restriction element to expand or to limit the base simple type for the element.

Here is an example of an XML element, "shoesize", that contains text-only:

```
<shoesize country="france">35</shoesize>
```

The following example declares a complexType, "shoesize". The content is defined as an integer value, and the "shoesize" element also contains an attribute named "country":

```
<xs:element name="shoesize">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:integer">
        <xs:attribute name="country" type="xs:string" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

We could also give the complexType element a name, and let the "shoesize" element have a type attribute that refers to the name of the complexType (if you use this method, several elements can refer to the same complex type):

```
<xs:element name="shoesize" type="shoetype"/>

<xs:complexType name="shoetype">
  <xs:simpleContent>
    <xs:extension base="xs:integer">
      <xs:attribute name="country" type="xs:string" />
                </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

4. XSD Mixed Content (that contain other element and text)

A mixed complex type element can contain attributes, elements, and text. Consider an XML element, "ordernote", that contains both text and other elements:

```
<ordernnote>
  Dear Mr.<name>Jagdish Bhatta</name>.
  Your gift order for the valentine day with order id
<orderid>9999</orderid>
  will be shipped on <shipdate>2012-02-13</shipdate>.
</ordernnote>
```

The following schema declares the "ordernote" element:

```
<xs:element name="ordernote">
  <xs:complexType mixed="true">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="orderid" type="xs:positiveInteger"/>
      <xs:element name="shipdate" type="xs:date"/>
    </xs:sequence>
```

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

Note: To enable character data to appear between the child-elements of "ordernote", the mixed attribute must be set to "true". The <xs:sequence> tag means that the elements defined (name, orderid and shipdate) must appear in that order inside a "ordernote" element.

We could also give the complexType element a name, and let the "ordernote" element have a type attribute that refers to the name of the complexType (if you use this method, several elements can refer to the same complex type):

```
<xs:element name="ordernote" type="ordertype"/>

<xs:complexType name="ordertype" mixed="true">
  <xs:sequence>
            <xs:element name="name" type="xs:string"/>
    <xs:element name="orderid" type="xs:positiveInteger"/>
            <xs:element name="shipdate" type="xs:date"/>
  </xs:sequence>
</xs:complexType>
```

XSD Indicators:

XSD indicators are used to control how elements are to be used in documents with indicators. There are seven indicators:

1. ***Order indicators:*** They contain;
   - All
   - Choice
   - Sequence
2. ***Occurrence indicators:*** They include;
   - maxOccurs
   - minOccurs
3. ***Group indicators:*** They contain;
   - Group name
   - attributeGroup name

1. **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>
```

2. 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>
```

The example above indicates that the "child_name" element can occur a minimum of one time (the default value for minOccurs is 1) and a maximum of ten times in the "person" 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>
```

The example above indicates that the "child_name" element can occur a minimum of zero times and a maximum of ten times in the "person" element.

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

Consider an example;

An XML file called "Myfamily.xml":

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<persons xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="family.xsd">

<person>
  <full_name>Anjolina</full_name>
  <child_name>Janet</child_name>
</person>

<person>
  <full_name>Dhritrasta</full_name>
  <child_name>Duryodhan</child_name>
  <child_name>Dushasan</child_name>
  <child_name>Kushashan</child_name>
  <child_name>Sushasan</child_name>
</person>

<person>
  <full_name>Bhismapitamaha</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="ISO-8859-1"?>
<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>
```

3. 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: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>

<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: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>

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

XSD The <any> Element :

The <any> element enables us to extend the XML document with elements not specified by the schema. The following example is a fragment from an XML schema called "family.xsd". It shows a declaration for the "person" element. By using the <any> element we can extend (after <lastname>) the content of "person" with any element:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
      <xs:any minOccurs="0"/>
                </xs:sequence>
  </xs:complexType>
</xs:element>
```

Now we want to extend the "person" element with a "children" element. In this case we can do so, even if the author of the schema above never declared any "children" element.

Look at this schema file, called "children.xsd":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.w3schools.com"
xmlns="http://www.w3schools.com"
elementFormDefault="qualified">

<xs:element name="children">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="childname" type="xs:string"
      maxOccurs="unbounded"/>
                </xs:sequence>
  </xs:complexType>
</xs:element>

</xs:schema>
```

The XML file below (called "Myfamily.xml"), uses components from two different schemas; "family.xsd" and "children.xsd":

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<persons xmlns="http://www.microsoft.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.microsoft.com family.xsd
http://www.w3schools.com children.xsd">

<person>
  <firstname>Ram</firstname>
  <lastname>Bhagwan</lastname>
  <children>
```

```
    <childname>Luv</childname>
  </children>
</person>

<person>
  <firstname>Harry</firstname>
  <lastname>Porter</lastname>
</person>

</persons>
```

The XML file above is valid because the schema "family.xsd" allows us to extend the "person" element with an optional element after the "lastname" element.

The <any> and <anyAttribute> elements are used to make EXTENSIBLE documents! They allow documents to contain additional elements that are not declared in the main XML schema.

XSD The <anyAttribute> Element :

The <anyAttribute> element enables us to extend the XML document with attributes not specified by the schema. The following example is a fragment from an XML schema called "family.xsd". It shows a declaration for the "person" element. By using the <anyAttribute> element we can add any number of attributes to the "person" element:

```
<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:anyAttribute/>
            </xs:complexType>
</xs:element>
```

Now we want to extend the "person" element with a "gender" attribute. In this case we can do so, even if the author of the schema above never declared any "gender" attribute.

Look at this schema file, called "attribute.xsd":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.w3schools.com"
xmlns="http://www.w3schools.com" elementFormDefault="qualified">
```

```
<xs:attribute name="gender">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="male|female"/>
                </xs:restriction>
  </xs:simpleType>
</xs:attribute>

</xs:schema>
```

The XML file below (called "Myfamily.xml"), uses components from two different schemas; "family.xsd" and "attribute.xsd":

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<persons xmlns="http://www.microsoft.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:SchemaLocation="http://www.microsoft.com family.xsd
http://www.w3schools.com attribute.xsd">

<person gender="female">
  <firstname>Sita</firstname>
  <lastname>Mata</lastname>
</person>

<person gender="male">
  <firstname>Ram</firstname>
  <lastname>Bhagwan</lastname>
</person>

</persons>
```

The XML file above is valid because the schema "family.xsd" allows us to add an attribute to the "person" element.

The <any> and <anyAttribute> elements are used to make EXTENSIBLE documents! They allow documents to contain additional elements that are not declared in the main XML schema.

XSL Language:

XSL stands for E**X**tensible **S**tylesheet **L**anguage. The World Wide Web Consortium (W3C) started to develop XSL because there was a need for an XML-based Stylesheet Language.

XML does not use predefined tags (we can use any tag-names we like), and therefore the meaning of each tag is **not well understood**. A <table> tag could mean an HTML table, a piece of furniture, or something else - and a browser **does not know how to display it**.

XSL describes how the XML document should be displayed, however its more than a Style Sheet Language. XSL consists of three parts:

- XSLT - a language for transforming XML documents
- XPath - a language for navigating in XML documents
- XSL-FO - a language for formatting XML documents

XSLT (Extensible Stylesheet Language):

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 documents

XSLT is the most important part of XSL. XSLT is used to transform an XML document into another XML document, or another type of document that is recognized by a browser, like HTML and XHTML. Normally XSLT does this by transforming each XML element into an (X)HTML element. With XSLT you can add/remove elements and attributes to or from the output file. You can also rearrange and sort elements, perform tests and make decisions about which elements to hide and display, and a lot more.

A common way to describe the transformation process is to say that **XSLT transforms an XML source-tree into an XML result-tree**.

XSLT uses XPath to find information in an XML document. XPath is used to navigate through elements and attributes in XML documents. In the transformation process, XSLT uses XPath to define parts of the source document that should match one or more predefined templates. When a match is found, XSLT will transform the matching part of the source document into the result document.

XSLT Transformation:

Style Sheet Declaration

The root element that declares the document to be an XSL style sheet is <xsl:stylesheet> or <xsl:transform>. <xsl:stylesheet> and <xsl:transform> are completely synonymous and either can be used. The correct way to declare an XSL style sheet according to the W3C XSLT Recommendation is:

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

or:

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

To get access to the XSLT elements, attributes and features we must declare the XSLT namespace at the top of the document.

The xmlns:xsl="http://www.w3.org/1999/XSL/Transform" points to the official W3C XSLT namespace. If you use this namespace, you must also include the attribute version="1.0".

Consider an example below, which we want to **transform** the following XML document ("cdcatalog.xml") into XHTML:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <country>USA</country>
    <company>Columbia</company>
    <price>10.90</price>
    <year>1985</year>
            </cd>
</catalog>
```

We can create an XSL Style Sheet ("cdcatalog.xsl") with a transformation template:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<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>
```

Now link the XSL Style Sheet to the XML Document. For this, add the XSL style sheet reference to the above mentioned XML document ("cdcatalog.xml") as:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="cdcatalog.xsl"?>
        <catalog>
 <cd>
  <title>Empire Burlesque</title>
  <artist>Bob Dylan</artist>
  <country>USA</country>
  <company>Columbia</company>
  <price>10.90</price>
  <year>1985</year>
 </cd>
        </catalog>
```

If you have an XSLT compliant browser it will nicely **transform** your XML into XHTML.

XSLT <xsl:template> Element:

An XSL style sheet consists of one or more set of rules that are called templates. A template contains rules to apply when a specified node is matched. The <xsl:template> element is used to build templates.

The **match** attribute is used to associate a template with an XML element. The match attribute can also be used to define a template for the entire XML document. The value of the match attribute is an XPath expression (i.e. match="/" defines the whole document).

Consider the example of cdcatalog.xsl, discussed above, it can be explained as;

Since an XSL style sheet is an XML document, it always begins with the XML declaration: **<?xml version="1.0" encoding="ISO-8859-1"?>**.

The next element, **<xsl:stylesheet>**, defines that this document is an XSLT style sheet document (along with the version number and XSLT namespace attributes).

The **<xsl:template>** element defines a template. The **match="/"** attribute associates the template with the root of the XML source document.

The content inside the <xsl:template> element defines some HTML to write to the output. The last two lines define the end of the template and the end of the style sheet.

The <xsl:value-of> Element:

The <xsl:value-of> element can be used to extract the value of an XML element and add it to the output stream of the transformation. In the above example cdcatalog.xsl, we have used it as;

```
<xsl:value-of select="catalog/cd/title"/>
<xsl:value-of select="catalog/cd/artist"/>
```

The **select** attribute in the example above, contains an XPath expression. An XPath expression works like navigating a file system; a forward slash (/) selects subdirectories.

XSLT <xsl:for-each> Element:

The <xsl:for-each> element allows you to do looping in XSLT. It can be used to select every XML element of a specified node-set. In the cdcatalog.xsl, we have

```
<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>
```

The value of the **select** attribute is an XPath expression. An XPath expression works like navigating a file system; where a forward slash (/) selects subdirectories.

We can also filter the output from the XML file by adding a criterion to the select attribute in the <xsl:for-each> element.

<xsl:for-each select="catalog/cd[artist='Bob Dylan']">

Legal filter operators are:

- = (equal)
  - != (not equal)
  - < less than
- > greater than

In the above example of cdcatalog.xsl; we can use restriction as;

```
<xsl:for-each select="catalog/cd[artist='Bob Dylan']">
        <tr>
  <td><xsl:value-of select="title"/></td>
          <td><xsl:value-of select="artist"/></td>
        </tr>
 </xsl:for-each>
```

XSLT <xsl:sort> Element:

To sort the output, simply add an <xsl:sort> element inside the <xsl:for-each> element in the XSL file as;

```
      <xsl:for-each select="catalog/cd">
  <xsl:sort select="artist"/>
  <tr>
    <td><xsl:value-of select="title"/></td>
    <td><xsl:value-of select="artist"/></td>
  </tr>
 </xsl:for-each>
```

XSLT <xsl:if> Element:

The <xsl:if> element is used to put a conditional test against the content of the XML file. To add a conditional test, add the <xsl:if> element inside the <xsl:for-each> element in the XSL file:

```
<xsl:if test="expression">
  ...some output if the expression is true...
</xsl:if>
```

In above example cdcatalog.xsl, we can write it as;

```
<xsl:for-each select="catalog/cd">
        <xsl:if test="price &gt; 10">
    <tr>
      <td><xsl:value-of select="title"/></td>
```

```
<td><xsl:value-of select="artist"/></td>
            </tr>
          </xsl:if>
 </xsl:for-each>
```

XSLT <xsl:choose> Element:

The <xsl:choose> element is used in conjunction with <xsl:when> and <xsl:otherwise> to express multiple conditional tests. The syntax is as;

```
<xsl:choose>
  <xsl:when test="expression">
                  ... some output ...
  </xsl:when>
  <xsl:otherwise>
    ... some output ....
                </xsl:otherwise>
</xsl:choose>
```

In the above example of cdcatalog.xsl, we can embed this element as;

```
<xsl:for-each select="catalog/cd">
    <tr>
      <td><xsl:value-of select="title"/></td>
                    <xsl:choose>
        <xsl:when test="price &gt; 10">
          <td bgcolor="#ff00ff">
          <xsl:value-of select="artist"/></td>
                      </xsl:when>
        <xsl:otherwise>
          <td><xsl:value-of select="artist"/></td>
        </xsl:otherwise>
                    </xsl:choose>
    </tr>
                  </xsl:for-each>
```

We can put sequence of the <xsl:when> as;

```
<xsl:for-each select="catalog/cd">
    <tr>
      <td><xsl:value-of select="title"/></td>
                    <xsl:choose>
        <xsl:when test="price &gt; 10">
          <td bgcolor="#ff00ff">
          <xsl:value-of select="artist"/></td>
```

```
    </xsl:when>
    <xsl:when test="price &gt; 9">
      <td bgcolor="#cccccc">
      <xsl:value-of select="artist"/></td>
    </xsl:when>
    <xsl:otherwise>
      <td><xsl:value-of select="artist"/></td>
    </xsl:otherwise>
            </xsl:choose>
  </tr>
  </xsl:for-each>
```

XPath:

XPath is a language for finding information in an XML document. It is a syntax for defining parts of an XML document which uses path expressions to navigate in XML documents. XPath contains a library of standard functions. It is a major element in XSLT.

XML documents are treated as trees of nodes. The topmost element of the tree is called the root element. XPath uses path expressions to select nodes or node-sets in an XML document. The node is selected by following a path or steps.

We will use the following XML document in the examples below.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<bookstore>

<book>
  <title lang="eng">Harry Potter</title>
  <price>29.99</price>
</book>

<book>
  <title lang="eng">Learning XML</title>
  <price>39.95</price>
</book>

</bookstore>
```

Selecting Nodes:

XPath uses path expressions to select nodes in an XML document. The node is selected by following a path or steps. The most useful path expressions are listed below:

Expression	Description
nodename	Selects all child nodes of the named node
/	Selects from the root node
//	Selects nodes in the document from the current node that match the selection no matter where they are
.	Selects the current node
..	Selects the parent of the current node
@	Selects attributes

In the table below we have listed some path expressions and the result of the expressions:

Path Expression	Result
bookstore	Selects all the child nodes of the bookstore element
/bookstore	Selects the root element bookstore **Note:** If the path starts with a slash (/) it always represents an absolute path to an element!
bookstore/book	Selects all book elements that are children of bookstore
//book	Selects all book elements no matter where they are in the document bookstore//book
Selects all book elements that are descendant of the bookstore element, no matter where they are under the bookstore element	
//@lang	Selects all attributes that are named lang

Predicates:

Predicates are used to find a specific node or a node that contains a specific value. Predicates are always embedded in square brackets. In the table below we have listed some path expressions with predicates and the result of the expressions:

Path Expression	Result
/bookstore/book[1]	Selects the first book element that is the child of the bookstore element. **Note:** IE5 and later has implemented that [0] should be the first node, but according to the W3C standard it should have been [1]!!
/bookstore/book[last()]	Selects the last book element that is the child of the bookstore element
/bookstore/book[last()-1]	Selects the last but one book element that is the child of the bookstore element
/bookstore/book[position()<3]	Selects the first two book elements that are children of

	the bookstore element
//title[@lang]	Selects all the title elements that have an attribute named lang
//title[@lang='eng']	Selects all the title elements that have an attribute named lang with a value of 'eng'
/bookstore/book[price>35.00]	Selects all the book elements of the bookstore element that have a price element with a value greater than 35.00
/bookstore/book[price>35.00]/title	Selects all the title elements of the book elements of the bookstore element that have a price element with a value greater than 35.00

Selecting Unknown Nodes:

XPath wildcards can be used to select unknown XML elements.

Wildcard Description

* Matches any element node

@* Matches any attribute node node() Matches any node of any kind

In the table below we have listed some path expressions and the result of the expressions:

Path Expression Result

/bookstore/*	Selects all the child nodes of the bookstore element
//*	Selects all elements in the document
//title[@*]	Selects all title elements which have any attribute

Selecting Several Paths:

By using the | operator in an XPath expression you can select several paths. In the table below we have listed some path expressions and the result of the expressions:

Path Expression	**Result**
//book/title | //book/price	Selects all the title AND price elements of all book elements
//title | //price	Selects all the title AND price elements in the document
/bookstore/book/title | //price	Selects all the title elements of the book element of the bookstore element AND all the price elements in the document

XQuery:

XQuery is to XML what SQL is to database tables. XQuery is designed to query XML data - not just XML files, but anything that can appear as XML, including databases. XQuery is a language for finding and extracting elements and attributes from XML documents. Some basic

syntax rules:

- XQuery is case-sensitive
- XQuery elements, attributes, and variables must be valid XML names
- An XQuery string value can be in single or double quotes
- An XQuery variable is defined with a $ followed by a name, e.g. $bookstore
- XQuery comments are delimited by (: and :), e.g. (: XQuery Comment :)

Consider a XML example;

```
<bookstore>

<book category="CHILDREN">
  <title lang="en">Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>

<book>

……

……..

</book>


</bookstore
```

XQuery to extract the data can be written as in following example;

```
for $x in doc("books.xml")/bookstore/book return if
($x/@category="CHILDREN") then
<child>{data($x/title)}</child>
else <adult>{data($x/title)}</adult>
```

Notes on the "if-then-else" syntax: parentheses around the if expression are required. else is required, but it can be just else ().

The result of the example above will be:

<child>Harry Potter</child>

XML DOM:

The XML DOM is:

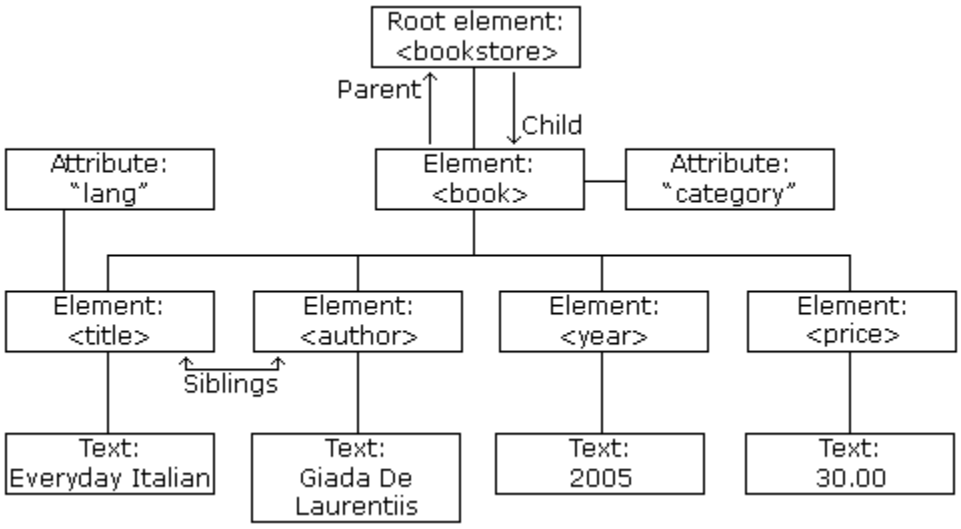
- A standard object model for XML
- A standard programming interface for XML
- Platform- and language-independent
- A W3C standard

The XML DOM defines the **objects and properties** of all XML elements, and the **methods** (interface) to access them. In other words: **The XML DOM is a standard for how to get, change, add, or delete XML elements.**

The XML DOM views an XML document as a tree-structure. All elements can be accessed through the DOM tree. Their content (text and attributes) can be modified or deleted, and new elements can be created. The elements, their text, and their attributes are all known as nodes. The node tree shows the set of nodes, and the connections between them. The tree starts at the root node and branches out to the text nodes at the lowest level of the tree.

All modern browsers have a build-in XML parser that can be used to read and manipulate XML. With the XML DOM properties and methods, you can access every node in an XML document. The XML DOM contains methods (functions) to traverse XML trees, access, insert, and delete nodes. However, before an XML document can be accessed and manipulated, it must be loaded into an XML DOM object. An XML parser reads XML, and converts it into an XML DOM object that can be accessed with JavaScript. Most browsers have a built-in XML parser.

Following shows a DOM tree example;



What will be the equivalent XML file for this??

SAX:

SAX (**Simple API for XML**) is an event-based sequential access parser API developed by the XML-DEV mailing list for XML documents. SAX provides a mechanism for reading data from an XML document that is an alternative to that provided by the Document Object Model (DOM). Where the DOM operates on the document as a whole, SAX parsers operate on each piece of the XML document sequentially.

Unlike DOM, there is no *formal* specification for SAX. The Java implementation of SAX is considered to be normative. It is used for state-independent processing of XML documents, in contrast to StAX (**Streaming API for XML)** that processes the documents state-dependently.

SAX parsers have certain benefits over DOM-style parsers. The quantity of memory that a SAX parser must use in order to function is typically much smaller than that of a DOM parser. DOM parsers must have the entire tree in memory before any processing can begin, so the amount of memory used by a DOM parser depends entirely on the size of the input data. The memory footprint of a SAX parser, by contrast, is based only on the maximum depth of the XML file (the maximum depth of the XML tree) and the maximum data stored in XML attributes on a single XML element. Both of these are always sma ller than the size of the parsed tree itself.

Because of the event-driven nature of SAX, processing documents can often be faster than DOM-style parsers. Memory allocation takes time, so the larger memory footprint of the DOM is also a performance issue. Due to the nature of DOM, streamed reading from disk is impossible. Processing XML documents larger than main memory is also impossible with DOM parsers, but can be done with SAX parsers. However, DOM parsers may make use of disk space as memory to sidestep this limitation.

[Unit 4/5: Server side scripting with ASP.NET]

Web Technology (CSC-353)

Jagdish Bhatta

Central Department of Computer Science & Information Technology

Tribhuvan University

.NET an Overview: ASP.net and VB.net

ASP.NET stands for Active Server Pages .NET, and VB.NET stands for Visual Basic.NET. VB.NET, put simply, is a programming language, and ASP.NET is a technology used to render dynamic web content. An ASP.NET web site is typically made up of code written in either VB.NET or C# (C Sharp). When creating a web site with VB.NET, you are actually creating an ASP.NET application *using* VB.NET. This is different from a traditional Active Server Page (ASP) page, in that an ASP.NET application is written using fully-featured programming languages with full functionality, like VB.NET, instead of scripting languages like Visual Basic Script (VBScript).

Microsoft .net

Microsoft .NET is a package of software that consists of clients, servers, and development tools. This package includes the Microsoft .NET Framework, development tools such as Visual Studio 2008, a set of server applications such as Microsoft Windows Server 2003 and Microsoft SQL Server, and client-side applications such as Windows XP and Microsoft Office. Microsoft .NET Framework includes many other subcomponents that allow software that has been written in different languages to work together by establishing rules for language independence. Using the Microsoft .NET Framework as a base, software development toolmakers can create development tools for different languages such as COBOL or C++. Microsoft itself used the .NET Framework to create VS, which is a development tool used to create software using the VB or C# programming languages.

The Microsoft .NET Framework also provides many common functions that previously needed to be built by the developer. This includes access to the file system, access to the registry, and easier development when using the Windows Application Programming Interfaces (API) to access operating system–level functionality. This allows the developer to concentrate more on business problems, instead of worrying how to access low-level windows functionality.

The Common Language Runtime

The Microsoft **Common Language Runtime** (CLR) is one of the components within the .NET Framework. The **CLR provides runtime services, including loading and execution of code**. The CLR essentially takes the language-specific code that was written and translates it **Microsoft Intermediate Language** (MSIL) code. The resulting code is the same, no matter what language the original code was written in. This is what allows code written with VB to work with code written in C#. This is also the most important aspect of the .NET Framework for a software development company, because one developer can write code in VB and another developer can write code with C#, but the application will still work without a problem, allowing companies to use their existing skill sets. Without this framework and the MSIL, an entire application would need to be built using the same language. This would require a software development company to have a full staff of developers that know a specific development language, such as VB. A single program, written in multiple languages, works mainly because the framework contains a set of common data types that must be used by all languages building applications with the .NET Framework. This set of data types is the **Common Type System (CTS)**, which defines how types are declared, used, and managed. To accommodate the CLR, some of the data types within languages such as VB needed to be changed so they could work better with data types from other languages such as C++.

Assemblies

An assembly is the main component of a .NET Framework application and is a collection of all of the functionality for the particular application. The assembly is created as either a .dll file for web sites or an .exe file for Windows applications, and it contains all of the MSIL code to be used by the framework. Without the assembly there is no application. The creation of an assembly is automatically performed by VS. It is possible to create applications for the .NET Framework without VS—however, you need to use the various tools that come with the .NET Framework Software Development Kit (SDK) to create the assemblies and perform other tasks that are automatically done by VS.

How Web Servers Execute ASP Files

When a site visitor requests a Web page address, the browser contacts the Web server specified in the address URL and makes a request for the page by formulating a HTTP request, which is sent to the Web server. The Web server on receiving the request determines the file type requested and passes processing to the appropriate handler. ASP.NET files are compiled, if necessary, into .NET Page classes and then executed, with the results sent to the client's browser. Compilation means that on first load ASP.NET applications take longer to display than previous versions of ASP, but once compiled they are noticeably faster. The browser can request
information from and send information to the server using two HTTP methods, GET and POST.

Web Server

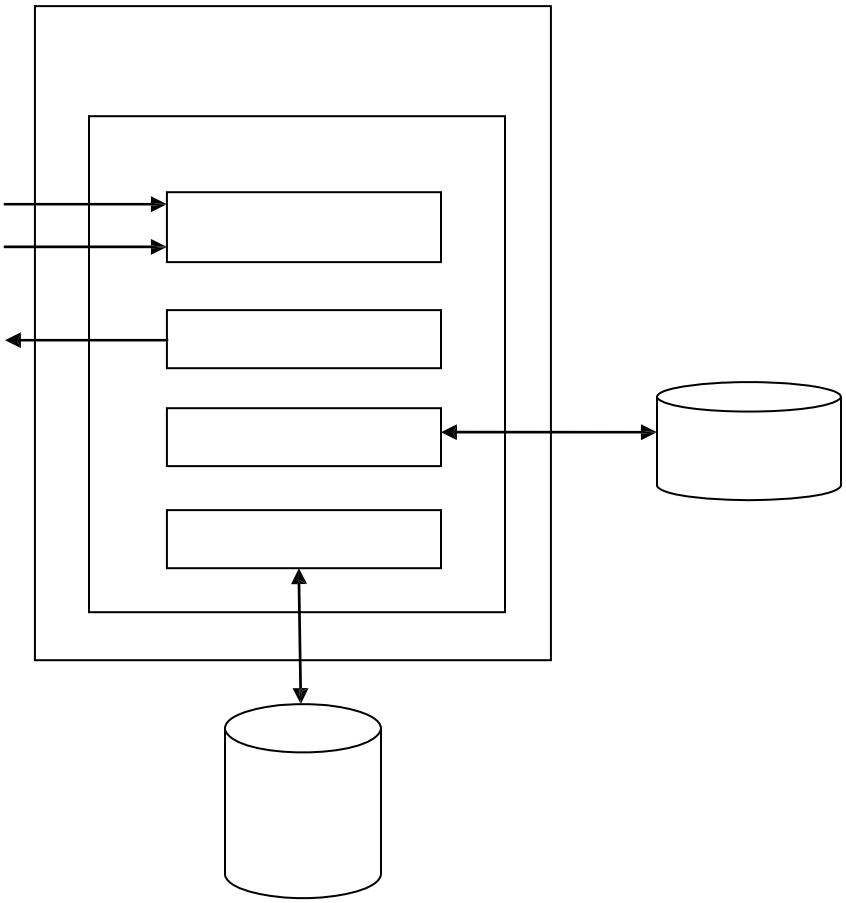
get post

Response

Request

Response

File System



ADO.net

Database

When the server receives this request, it will find the page that was requested using the path information specified, and the relevant system will process the page. When the response is complete, it is flushed back out to the user's browser, usually as HTML but not necessarily, and

the browser renders this page as it arrives as the page on screen. The process of compiling and delivering ASP.NET pages goes through the following stages:

1. IIS matches the URL in the request against a file on the physical file system (hard disk) by translating the virtual path (for example, /site/ index.aspx) into a path relative to the site's Web root (for example, d:\domains\thisSite\wwwroot\site\index.aspx).
2. Once the file is found, the file extension (.aspx) is matched against a list of known file types for either sending on to the visitor or for processing.
3. If this is first visit to the page since the file was last changed, the ASP code is compiled into an assembly using the Common Language Runtime compiler, into MSIL, and then into machine-specific binary code for execution.
4. The binary code is a .NET class .dll and is stored in a temporary location.
5. Next time the page is requested the server will check to see if the code has changed. If the code is the same, then the compilation step is skipped and the previously compiled class code is executed; otherwise, the class is deleted and recompiled from the new source.
6. The compiled code is executed and the request values are interpreted, such as form input fields or URL parameters.
7. If the developer has used Web forms, then the server can detect what software the visitor is using and render pages that are tailored to the visitors' requirements, for example, returning Netscape specific code, or Wireless Markup Language (WML) code for mobiles.
8. Any results are delivered back to the visitor's Web browser.
9. Form elements are converted into client side markup and script, HTML and JavaScript for Web browsers, and WML and WMLScript for mobiles, for example.

Request

Server Finds
File

ASP.net
Process

Compilation Errors

Compile

Yes

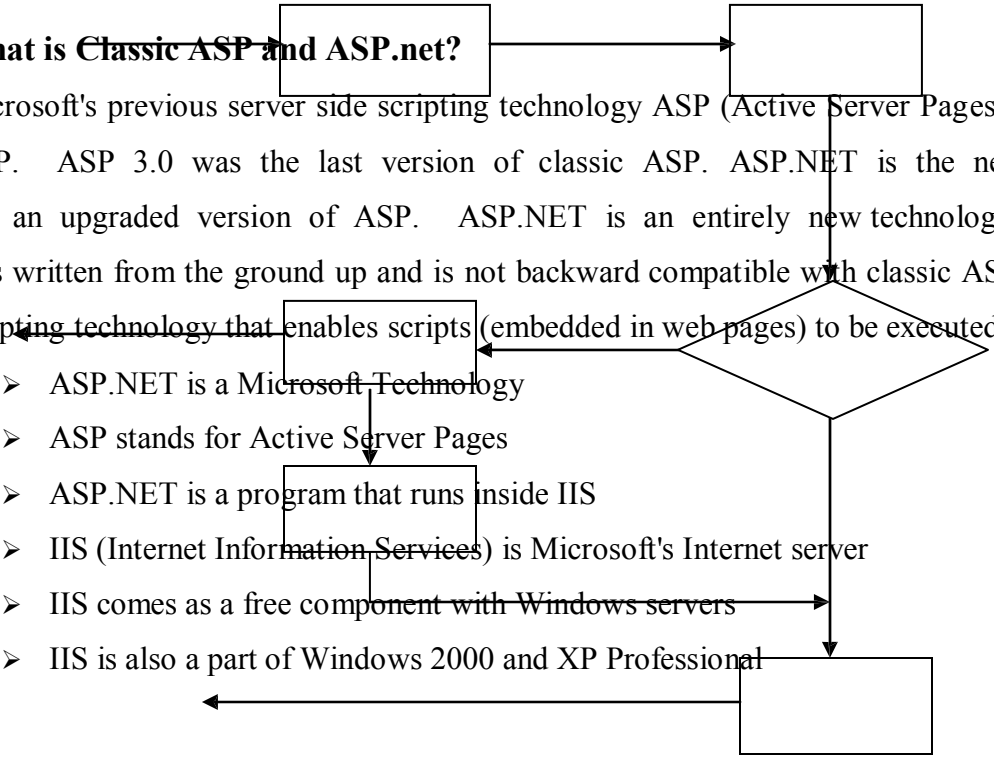
Changed?

No

Save

Response

Execute



What is Classic ASP and ASP.net?

Microsoft's previous server side scripting technology ASP (Active Server Pages) is now often called classic ASP. ASP 3.0 was the last version of classic ASP. ASP.NET is the next generation ASP, but it's not an upgraded version of ASP. ASP.NET is an entirely new technology for server-side scripting. It was written from the ground up and is not backward compatible with classic ASP. ASP.NET is a server side scripting technology that enables scripts (embedded in web pages) to be executed by an Internet server.

- ASP.NET is a Microsoft Technology
- ASP stands for Active Server Pages
- ASP.NET is a program that runs inside IIS
- IIS (Internet Information Services) is Microsoft's Internet server
- IIS comes as a free component with Windows servers
- IIS is also a part of Windows 2000 and XP Professional

The .NET Framework consists of 3 main parts:

Programming languages:

- C# (Pronounced C sharp)
- Visual Basic (VB .NET)
- J# (Pronounced J sharp)

Server technologies and client technologies:

- ASP .NET (Active Server Pages)
- Windows Forms (Windows desktop solutions)
- Compact Framework (PDA / Mobile solutions)

Development environments:

- Visual Studio .NET (VS .NET)
- Visual Web Developer

Features of ASP.net:

- **ASP.NET Controls:** ASP.NET contains a large set of HTML controls. Almost all HTML elements on a page can be defined as ASP.NET control objects that can be controlled by scripts. ASP.NET also contains a new set of object-oriented input controls, like programmable list-boxes and validation controls. A new data grid control supports sorting, data paging, and everything you can expect from a dataset control.
- **Event Aware Controls:** All ASP.NET objects on a Web page can expose events that can be processed by ASP.NET code. Load, Click and Change events handled by code makes coding much simpler and much better organized.
- **ASP.NET Components:** ASP.NET components are heavily based on XML.
- **User Authentication:** ASP.NET supports form-based user authentication, cookie management, and automatic redirecting of unauthorized logins.
- **User Accounts and Roles:** ASP.NET allows user accounts and roles, to give each user (with a given role) access to different server code and executables.
- **High Scalability:** Much has been done with ASP.NET to provide greater scalability. Server-to-server communication has been greatly enhanced, making it possible to scale

an application over several servers. One example of this is the ability to run XML parsers, XSL transformations and even resource hungry session objects on other servers.

- **Compiled Code:** The first request for an ASP.NET page on the server will compile the ASP.NET code and keep a cached copy in memory. The result of this is greatly increased performance.
- **Easy Configuration:** Configuration of ASP.NET is done with plain text files. Configuration files can be uploaded or changed while the application is running. No need to restart the server. No more metabase or registry puzzle.
- **Easy Deployment:** No more server-restart to deploy or replace compiled code. ASP.NET simply redirects all new requests to the new code.
- **Compatibility:** ASP.NET is not fully compatible with earlier versions of ASP, so most of the old ASP code will need some changes to run under ASP.NET. To overcome this problem, ASP.NET uses a new file extension ".aspx". This will make ASP.NET applications able to run side by side with standard ASP applications on the same server.

Creating ASP.NET Application

Simple HTML page that will display "Hello W3Schools" in an Internet browser can be written like this:

```
<html>

</html>
```

```
<body bgcolor="yellow">
<center>
<h2>Hello W3Schools!</h2>
</center>
</body>
```

The simplest way to convert an HTML page into an ASP.NET page is to copy the HTML file to a new file with an **.aspx** extension

```
<html>
        <body bgcolor="yellow">
        <center>
        <h2>Hello W3Schools!</h2>
```

```
</html>
```

```
</center>
</body>
```

Fundamentally an ASP.NET page is just the same as an HTML page. An HTML page has the extension .htm. If a browser requests an HTML page from the server, the server sends the page to the browser without any modifications. An ASP.NET page has the extension .aspx. If a browser requests an ASP.NET page, the server processes any executable code in the page, before the result is sent back to the browser. The ASP.NET page above does not contain any executable code, so nothing is executed. In the next examples we will add some executable code to the page to demonstrate the difference between static HTML pages and dynamic ASP pages.

Dynamic Page in Classic ASP and ASP.net

To demonstrate how ASP can display pages with dynamic content, we have added some executable code (in red) to the previous example:

```
<html>
    <body bgcolor="yellow">
    <center>
    <h2>Hello W3Schools!</h2>
    <p><%Response.Write(now())%></p>
    </center>
    </body>
</html>
```

The code inside the <% --%> tags is executed on the server. Response.Write is ASP code for writing something to the HTML output stream. Now() is a function returning the servers current date and time. This same code can also be used as ASP.NET page. The code above illustrates a limitation in Classic ASP: The code block has to be placed where you want the output to appear. With Classic ASP it is impossible to separate executable code from the HTML itself. This makes the page difficult to read, and difficult to maintain.

ASP.NET - Server Controls

ASP.NET has solved the "spaghetti-code" problem described above with server controls. Server controls are tags that are understood by the server. There are three kinds of server controls:

- HTML Server Controls - Traditional HTML tags
- Web Server Controls - New ASP.NET tags
- Validation Server Controls - For input validation

ASP.NET - HTML Server Controls

HTML server controls are HTML tags understood by the server. HTML elements in ASP.NET files are, by default, treated as text. To make these elements programmable, add a runat="server" attribute to the HTML element. This attribute indicates that the element should be treated as a server control. The id attribute is added to identify the server control. The id reference can be used to manipulate the server control at run time. All HTML server controls must be within a

<form> tag with the runat="server" attribute. The runat="server" attribute indicates that the form should be processed on the server. It also indicates that the enclosed controls can be accessed by server scripts.

In the following example we declare an HtmlAnchor server control in an .aspx file. Then we manipulate the HRef attribute of the HtmlAnchor control in an event handler (an event handler is a subroutine that executes code for a given event). The Page_Load event is one of many events that ASP.NET understands.

```
<script runat="server"> Sub Page_Load
    link1.HRef="http://www.w3schools.com" End Sub
    </script>
    <html>
    <body>
    <form runat="server">
    <a id="link1" runat="server">Visit W3Schools!</a>
    </form>
    </body>
```

```
</html>
```

The executable code itself has been moved outside the HTML.

ASP.NET - Web Server Controls

Web server controls are special ASP.NET tags understood by the server. Like HTML server controls, Web server controls are also created on the server and they require a runat="server" attribute to work. However, Web server controls do not necessarily map to any existing HTML elements and they may represent more complex elements. The syntax for creating a Web server control is:

<asp:control_name id="some_id" runat="server" />

In the following example we declare a Button server control in an .aspx file. Then we create an event handler for the Click event which changes the text on the button:

```
<script runat="server">
    Sub submit(Source As Object, e As EventArgs)
    button1.Text="You clicked me!" End Sub
</script>
<html>
    <body>
    <form runat="server">
    <asp:Button id="button1" Text="Click me!" runat="server" OnClick="submit"/>
    </form>
    </body>
</html>
```

ASP.NET - Validation Server Controls

Validation server controls are used to validate user-input. If the user-input does not pass validation, it will display an error message to the user. Each validation control performs a specific type of validation (like validating against a specific value or a range of values). By default, page validation is performed when a Button, ImageButton, or LinkButton control is

clicked. You can prevent validation when a button control is clicked by setting the CausesValidation property to false. The syntax for creating a Validation server control is:

<asp:control_name id="some_id" runat="server" />

In the following example we declare one TextBox control, one Button control, and one RangeValidator control in an .aspx file. If validation fails, the text "The value must be from 1 to 100!" will be displayed in the RangeValidator control:

```
<html>
<body>
<form runat="server">
<p>Enter a number from 1 to 100:
<asp:TextBox id="tbox1" runat="server" />
<br /><br />
<asp:Button Text="Submit" runat="server" />
</p>
<p>
<asp:RangeValidator  ControlToValidate="tbox1" MinimumValue="1"
MaximumValue="100" Type="Integer" Text="The value must be from 1 to 100!"
runat="server" />
</p>
</form>
</body>
</html>
```

ASP.NET Web Forms

All server controls must appear within a <form> tag, and the <form> tag must contain the runat="server" attribute. The runat="server" attribute indicates that the form should be processed on the server. It also indicates that the enclosed controls can be accessed by server scripts:

```
<form runat="server">
...HTML + server controls
</form>
```

The form is always submitted to the page itself. If you specify an action attribute, it is ignored. If you omit the method attribute, it will be set to method="post" by default. Also, if you do not specify the name and id attributes, they are automatically assigned by ASP.NET. An .aspx page can only contain ONE <form runat="server"> control!. If you select view source in an .aspx page containing a form with no name, method, action, or id attribute specified, you will see that ASP.NET has added these attributes to the form. It looks something like this:

```
<form name="_ctl0" method="post" action="page.aspx" id="_ctl0">
...some code
</form>
```

Submitting a Form

A form is most often submitted by clicking on a button. The Button server control in ASP.NET has the following format:

<asp:Button id="id" text="label" OnClick="sub" runat="server" />

The id attribute defines a unique name for the button and the text attribute assigns a label to the button. The onClick event handler specifies a named subroutine to execute. In the following example we declare a Button control in an .aspx file. A button click runs a subroutine which changes the text on the button:

Maintaining the ViewState

When a form is submitted in classic ASP, all form values are cleared. Suppose you have submitted a form with a lot of information and the server comes back with an error. You will have to go back to the form and correct the information. You click the back button, and what happens.....ALL form values are CLEARED, and you will have to start all over again! The site did not maintain your ViewState. When a form is submitted in ASP .NET, the form reappears in the browser window together with all form values. How come? This is because ASP .NET maintains your ViewState. The ViewState indicates the status of the page when submitted to the server. The status is defined through a hidden field placed on each page with a <form runat="server"> control. The source could look something like this:

```
<form name="_ctl0" method="post" action="page.aspx" id="_ctl0">
<input type="hidden" name="__VIEWSTATE"
```

```
value="dDwtNTI0ODU5MDE1Ozs+ZBCF2ryjMpeVgUrY2eTj79HNl4Q=" />
.....some code
</form>
```

Maintaining the ViewState is the default setting for ASP.NET Web Forms. If you want to NOT maintain the ViewState, include the directive <%@ Page EnableViewState="false" %> at the top of an .aspx page or add the attribute EnableViewState="false" to any control. Look at the following .aspx file. It demonstrates the "old" way to do it. When you click on the submit button, the form value will disappear:

```
<html>
<body>
<form action="demo_classicasp.aspx" method="post"> Your name:
<input type="text" name="fname" size="20">
<input type="submit" value="Submit">
</form>
<%
dim fname fname=Request.Form("fname")
If fname<>"" Then
Response.Write("Hello " & fname & "!") End If
%>
</body>
</html>
```

Here is the new ASP .NET way. When you click on the submit button, the form value will NOT disappear:

```
<script runat="server">
Sub submit(sender As Object, e As EventArgs)
lbl1.Text="Hello " & txt1.Text & "!" End Sub
</script>
<html>
```

```
<body>
<form runat="server">
Your name: <asp:TextBox id="txt1" runat="server" />
<asp:Button OnClick="submit" Text="Submit" runat="server" />
<p><asp:Label id="lbl1" runat="server" /></p>
</form>
</body>
</html>
```

The TextBox Control

The TextBox control is used to create a text box where the user can input text. The TextBox control's attributes and properties are listed in our web controls reference page. The example below demonstrates some of the attributes you may use with the TextBox control:

```
<html>
<body>
<form runat="server"> A basic
TextBox:
<asp:TextBox id="tb1" runat="server" />
<br /><br />
A password TextBox:
<asp:TextBox id="tb2" TextMode="password" runat="server" />
<br /><br />
A TextBox with text:
<asp:TextBox id="tb4" Text="Hello World!" runat="server" />
<br /><br />
A multiline TextBox:
<asp:TextBox id="tb3" TextMode="multiline" runat="server" />
<br /><br />
A TextBox with height:
<asp:TextBox id="tb6" rows="5" TextMode="multiline"
runat="server" />
```

```
<br /><br />
A TextBox with width:
<asp:TextBox id="tb5" columns="30" runat="server" />
</form>
</body>
</html>
```

Add a Script

The contents and settings of a TextBox control may be changed by server scripts when a form is submitted. A form can be submitted by clicking on a button or when a user changes the value in the TextBox control. In the following example we declare one TextBox control, one Button control, and one Label control in an .aspx file. When the submit button is triggered, the submit subroutine is executed. The submit subroutine writes a text to the Label control:

```
<script runat="server">
Sub submit(sender As Object, e As EventArgs)
lbl1.Text="Your name is " & txt1.Text
End Sub
</script>
<html>
<body>
<form runat="server"> Enter your
name:
<asp:TextBox id="txt1" runat="server" />
<asp:Button OnClick="submit" Text="Submit" runat="server" />
<p><asp:Label id="lbl1" runat="server" /></p>
</form>
</body>
</html>
```

In the following example we declare one TextBox control and one Label control in an .aspx file. When you change the value in the TextBox and either click outside the TextBox or press the Tab key, the change subroutine is executed. The submit subroutine writes a text to the Label control:

```
<script runat="server">
Sub change(sender As Object, e As EventArgs)
lbl1.Text="You changed text to " & txt1.Text End Sub
</script>
<html>
<body>
<form runat="server"> Enter your
name:
<asp:TextBox id="txt1" runat="server" text="Hello
World!"
ontextchanged="change" autopostback="true"/>
<p><asp:Label id="lbl1" runat="server" /></p>
</form>
</body>
</html>
```

The Button Control

The Button control is used to display a push button. The push button may be a submit button or a command button. By default, this control is a submit button. A submit button does not have a command name and it posts the page back to the server when it is clicked. It is possible to write an event handler to control the actions performed when the submit button is clicked. A command button has a command name and allows you to create multiple Button controls on a page. It is possible to write an event handler to control the actions performed when the command button is clicked. The example below demonstrates a simple Button control:

```
<html>
<body>
<form runat="server">
<asp:Button id="b1" Text="Submit" runat="server" />
</form>
</body>
```

```
</html>
```

Data Binding

The following controls are list controls which support data binding:

- asp:RadioButtonList
- asp:CheckBoxList
- asp:DropDownList
  - asp:Listbox

The selectable items in each of the above controls are usually defined by one or more asp:ListItem controls, like this:

```
<html>
<body>
<form runat="server">
<asp:RadioButtonList id="countrylist" runat="server">
<asp:ListItem value="N" text="Norway" />
<asp:ListItem value="S" text="Sweden" />
<asp:ListItem value="F" text="France" />
<asp:ListItem value="I" text="Italy" />
</asp:RadioButtonList>
</form>
</body>
</html>
```

However, with data binding we may use a separate source, like a database, an XML file, or a script to fill the list with selectable items. By using an imported source, the data is separated from the HTML, and any changes to the items are made in the separate data source.

The ArrayList object is a collection of items containing a single data value. Items are added to the ArrayList with the Add() method. The following code creates a new ArrayList object named mycountries and four items are added:

```
<script runat="server"> Sub
Page_Load
```

```
if Not Page.IsPostBack then
  dim mycountries=New ArrayList
  mycountries.Add("Norway")
  mycountries.Add("Sweden")
  mycountries.Add("France")
  mycountries.Add("Italy")
end if end sub
</script>
```

By default, an ArrayList object contains 16 entries. An ArrayList can be sized to its final size with the TrimToSize() method:

```
<script runat="server"> Sub
Page_Load
if Not Page.IsPostBack then
  dim mycountries=New ArrayList
  mycountries.Add("Norway")
  mycountries.Add("Sweden")
  mycountries.Add("France")
  mycountries.Add("Italy")
  mycountries.TrimToSize()
end if end sub
</script>
```

An ArrayList can also be sorted alphabetically or numerically with the Sort() method:

```
<script runat="server"> Sub
Page_Load
if Not Page.IsPostBack then
  dim mycountries=New ArrayList
  mycountries.Add("Norway")
  mycountries.Add("Sweden")
  mycountries.Add("France")
```

```
  mycountries.Add("Italy")
  mycountries.TrimToSize()
  mycountries.Sort()
end if end sub
</script>
```

To sort in reverse order, apply the Reverse() method after the Sort() method:

```
script runat="server"> Sub
Page_Load
if Not Page.IsPostBack then
  dim mycountries=New ArrayList
  mycountries.Add("Norway")
  mycountries.Add("Sweden")
  mycountries.Add("France")
  mycountries.Add("Italy")
  mycountries.TrimToSize() mycountries.Sort()
  mycountries.Reverse()
end if end sub
</script>
```

An ArrayList object may automatically generate the text and values to the following controls:

- asp:RadioButtonList
- asp:CheckBoxList
- asp:DropDownList
- asp:Listbox

To bind data to a RadioButtonList control, first create a RadioButtonList control (without any asp:ListItem elements) in an .aspx page:

```
<script runat="server"> Sub
Page_Load
if Not Page.IsPostBack then
```

```
dim mycountries=New ArrayList
mycountries.Add("Norway")
mycountries.Add("Sweden")
mycountries.Add("France")
mycountries.Add("Italy")
mycountries.TrimToSize() mycountries.Sort()
rb.DataSource=mycountries rb.DataBind()
end if end sub
</script>
<html>
<body>
<form runat="server">
<asp:RadioButtonList id="rb" runat="server" />
</form>
</body>
</html>
```

The DataSource property of the RadioButtonList control is set to the ArrayList and it defines the data source of the RadioButtonList control. The DataBind() method of the RadioButtonList control binds the data source with the RadioButtonList control. The data values are used as both the Text and Value properties for the control.

Creating a HashTable

The Hashtable object contains items in key/value pairs. The keys are used as indexes, and very quick searches can be made for values by searching through their keys. Items are added to the Hashtable with the Add() method. The following code creates a Hashtable named mycountries and four elements are added:

```
<script runat="server"> Sub
Page_Load
```

```
if Not Page.IsPostBack then
  dim mycountries=New Hashtable
  mycountries.Add("N","Norway")
  mycountries.Add("S","Sweden")
  mycountries.Add("F","France")
  mycountries.Add("I","Italy")
end if end sub
</script>
```

A Hashtable object may automatically generate the text and values to the following controls:

- asp:RadioButtonList
- asp:CheckBoxList
- asp:DropDownList
- asp:Listbox

To bind data to a RadioButtonList control, first create a RadioButtonList control (without any asp:ListItem elements) in an .aspx page:

```
<html>
<body>
<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True" />
</form>
</body>
</html>
```

Then add the script that builds the list:

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycountries=New Hashtable
  mycountries.Add("N", "Norway")
```

```
  mycountries.Add("S", "Sweden")
  mycountries.Add("F", "France")
  mycountries.Add("I", "Italy")
  rb.DataSource=mycountries
  rb.DataValueField="Key"
  rb.DataTextField="Value" rb.DataBind()
end if end sub
</script>
<html>
<body>
<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True" />
</form>
</body>
</html>
```

Then we add a sub routine to be executed when the user clicks on an item in the RadioButtonList control. When a radio button is clicked, a text will appear in a label:

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycountries=New Hashtable
  mycountries.Add("N", "Norway")
  mycountries.Add("S", "Sweden")
  mycountries.Add("F", "France")
  mycountries.Add("I", "Italy")
  rb.DataSource=mycountries
  rb.DataValueField="Key"
  rb.DataTextField="Value" rb.DataBind()
```

```
end if end sub
sub displayMessage(s as Object,e As EventArgs) lbl1.text="Your favorite
country is: " & rb.SelectedItem.Text end sub
</script>
<html>
<body>
<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True"
onSelectedIndexChanged="displayMessage" />
<p><asp:label id="lbl1" runat="server" /></p>
</form>
</body>
</html>
```

You cannot choose the sort order of the items added to the Hashtable. To sort items alphabetically or numerically, use the SortedList object.

The SortedList Object

The SortedList object contains items in key/value pairs. A SortedList object automatically sort the items in alphabetic or numeric order. Items are added to the SortedList with the Add() method. A SortedList can be sized to its final size with the TrimToSize() method. The following code creates a SortedList named mycountries and four elements are added:

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycountries=New SortedList
  mycountries.Add("N","Norway")
  mycountries.Add("S","Sweden")
  mycountries.Add("F","France")
```

```
  mycountries.Add("I","Italy")
end if end sub
</script>
```

A SortedList object may automatically generate the text and values to the following controls:

- asp:RadioButtonList
- asp:CheckBoxList
- asp:DropDownList
- asp:Listbox

To bind data to a RadioButtonList control, first create a RadioButtonList control (without any asp:ListItem elements) in an .aspx page:

```
<html>
<body>
<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True" />
</form>
</body>
</html>
```

Then add the script that builds the list:

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycountries=New SortedList
  mycountries.Add("N","Norway")
  mycountries.Add("S","Sweden")
  mycountries.Add("F","France")
  mycountries.Add("I","Italy")
  rb.DataSource=mycountries
  rb.DataValueField="Key"
  rb.DataTextField="Value"
```

```
  rb.DataBind()
end if end sub
</script>
<html>
<body>
<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True" />
</form>
</body>
</html>
```

Then we add a sub routine to be executed when the user clicks on an item in the RadioButtonList control. When a radio button is clicked, a text will appear in a label:

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycountries=New SortedList
  mycountries.Add("N","Norway")
  mycountries.Add("S","Sweden")
  mycountries.Add("F","France")
  mycountries.Add("I","Italy")
  rb.DataSource=mycountries
  rb.DataValueField="Key"
  rb.DataTextField="Value" rb.DataBind()
end if end sub
sub displayMessage(s as Object,e As EventArgs) lbl1.text="Your favorite
country is: " & rb.SelectedItem.Text end sub
</script>
```

```
<html>
<body>
<form runat="server">
<asp:RadioButtonList id="rb" runat="server"
AutoPostBack="True" onSelectedIndexChanged="displayMessage" />
<p><asp:label id="lbl1" runat="server" /></p>
</form>
</body>
</html>
```

ASP .NET - XML Files

Here is an XML file named "countries.xml":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<countries>
<country>
  <text>Norway</text>
  <value>N</value>
</country>
<country>
  <text>Sweden</text>
  <value>S</value>
</country>
<country>
  <text>France</text>
  <value>F</value>
</country>
<country>
  <text>Italy</text>
  <value>I</value>
</country>
</countries>
```

Bind a DataSet to a List Control

First, import the "System.Data" namespace. We need this namespace to work with DataSet objects. Include the following directive at the top of an .aspx page:

```
<%@ Import Namespace="System.Data" %>
```

Next, create a DataSet for the XML file and load the XML file into the DataSet when the page is first loaded:

```
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycountries=New DataSet
mycountries.ReadXml(MapPath("countries.xml")) end if
end sub
```

To bind the DataSet to a RadioButtonList control, first create a RadioButtonList control (without any asp:ListItem elements) in an .aspx page:

```
<html>
<body>
<form runat="server">
<asp:RadioButtonList id="rb" runat="server" AutoPostBack="True" />
</form>
</body>
</html>
```

Then add the script that builds the XML DataSet:

```
<%@ Import Namespace="System.Data" %>
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycountries=New DataSet
  mycountries.ReadXml(MapPath("countries.xml"))
  rb.DataSource=mycountries rb.DataValueField="value"
```

```
  rb.DataTextField="text"
  rb.DataBind()
end if end sub
</script>
<html>
<body>
<form runat="server">
<asp:RadioButtonList id="rb" runat="server"
AutoPostBack="True" onSelectedIndexChanged="displayMessage" />
</form>
</body>
</html>
```

Then we add a sub routine to be executed when the user clicks on an item in the RadioButtonList control. When a radio button is clicked, a text will appear in a label:

```
<%@ Import Namespace="System.Data" %>
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycountries=New DataSet
  mycountries.ReadXml(MapPath("countries.xml"))
  rb.DataSource=mycountries rb.DataValueField="value"
  rb.DataTextField="text"
  rb.DataBind()
end if end sub
sub displayMessage(s as Object,e As EventArgs) lbl1.text="Your favorite
country is: " & rb.SelectedItem.Text end sub
</script>
```

```
<html>
<body>
<form runat="server">
<asp:RadioButtonList id="rb" runat="server"
AutoPostBack="True" onSelectedIndexChanged="displayMessage" />
<p><asp:label id="lbl1" runat="server" /></p>
</form>
</body>
</html>
```

The Repeater Control

The Repeater control is used to display a repeated list of items that are bound to the control. The Repeater control may be bound to a database table, an XML file, or another list of items. Here we will show how to bind an XML file to a Repeater control.

We will use the following XML file in our examples ("cdcatalog.xml"):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<catalog>
<cd>
  <title>Empire Burlesque</title>
  <artist>Bob Dylan</artist>
  <country>USA</country>
  <company>Columbia</company>
  <price>10.90</price>
  <year>1985</year>
</cd>
<cd>
  <title>Hide your heart</title>
  <artist>Bonnie Tyler</artist>
  <country>UK</country>
  <company>CBS Records</company>
  <price>9.90</price>
```

```
  <year>1988</year>
 </cd>
 <cd>
  <title>Greatest Hits</title>
  <artist>Dolly Parton</artist>
  <country>USA</country>
  <company>RCA</company>
  <price>9.90</price>
  <year>1982</year>
 </cd>
 <cd>
  <title>Still got the blues</title>
  <artist>Gary Moore</artist>
  <country>UK</country>
  <company>Virgin records</company>
  <price>10.20</price>
  <year>1990</year>
 </cd>
 <cd>
  <title>Eros</title>
  <artist>Eros Ramazzotti</artist>
  <country>EU</country>
  <company>BMG</company>
  <price>9.90</price>
  <year>1997</year>
 </cd>
 </catalog>
```

Next, create a DataSet for the XML file and load the XML file into the DataSet when the page is first loaded:

```
<script runat="server">
sub Page_Load
```

```
if Not Page.IsPostBack then
  dim mycdcatalog=New DataSet
mycdcatalog.ReadXml(MapPath("cdcatalog.xml")) end if
end sub
```

Then we create a Repeater control in an .aspx page. The contents of the <HeaderTemplate> element are rendered first and only once within the output, then the contents of the <ItemTemplate> element are repeated for each "record" in the DataSet, and last, the contents of the <FooterTemplate> element are rendered once within the output:

```
<html>
<body>
<form runat="server">
<asp:Repeater id="cdcatalog" runat="server">
<HeaderTemplate>
...
</HeaderTemplate>
<ItemTemplate>
...
</ItemTemplate>
<FooterTemplate>
...
</FooterTemplate>
</asp:Repeater>
</form>
</body>
</html>
```

Then we add the script that creates the DataSet and binds the mycdcatalog DataSet to the Repeater control. We also fill the Repeater control with HTML tags and bind the data items to the cells in the<ItemTemplate> section with the <%#Container.DataItem("fieldname")%> method:

Example

```
<%@ Import Namespace="System.Data" %>
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycdcatalog=New DataSet
  mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
  cdcatalog.DataSource=mycdcatalog cdcatalog.DataBind()
end if end sub
</script>
<html>
<body>
<form runat="server">
<asp:Repeater id="cdcatalog" runat="server">
<HeaderTemplate>
<table border="1" width="100%">
<tr>
<th>Title</th>
<th>Artist</th>
<th>Country</th>
<th>Company</th>
<th>Price</th>
<th>Year</th>
</tr>
</HeaderTemplate>
<ItemTemplate>
<tr>
<td><%#Container.DataItem("title")%></td>
<td><%#Container.DataItem("artist")%></td>
<td><%#Container.DataItem("country")%></td>
```

```
<td><%#Container.DataItem("company")%></td>
<td><%#Container.DataItem("price")%></td>
<td><%#Container.DataItem("year")%></td>
</tr>
</ItemTemplate>
<FooterTemplate>
</table>
</FooterTemplate>
</asp:Repeater>
</form>
</body>
</html>
```

Output

Title	Artist	Country	Company	Price	Year
Empire Burlesque	Bob Dylan	USA	Columbia	10.90	1985
Hide your heart	Bonnie Tyler	UK	CBS Records	9.90	1988
Greatest Hits	Dolly Parton	USA	RCA	9.90	1982
Still got the blues	Gary Moore	UK	Virgin records	10.20	1990
Eros	Eros Ramazzotti	EU	BMG	9.90	1997

Using the <AlternatingItemTemplate>

You can add an <AlternatingItemTemplate> element after the <ItemTemplate> element to describe the appearance of alternating rows of output. In the following example each other row in the table will be displayed in a light grey color:

```
<%@ Import Namespace="System.Data" %>
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycdcatalog=New DataSet
  mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
  cdcatalog.DataSource=mycdcatalog cdcatalog.DataBind()
```

```
end if end sub
</script>
<html>
<body>
<form runat="server">
<asp:Repeater id="cdcatalog" runat="server">
<HeaderTemplate>
<table border="1" width="100%">
<tr>
<th>Title</th>
<th>Artist</th>
<th>Country</th>
<th>Company</th>
<th>Price</th>
<th>Year</th>
</tr>
</HeaderTemplate>
<ItemTemplate>
<tr>
<td><%#Container.DataItem("title")%></td>
<td><%#Container.DataItem("artist")%></td>
<td><%#Container.DataItem("country")%></td>
<td><%#Container.DataItem("company")%></td>
<td><%#Container.DataItem("price")%></td>
<td><%#Container.DataItem("year")%></td>
</tr>
</ItemTemplate>
<AlternatingItemTemplate>
<tr bgcolor="#e8e8e8">
<td><%#Container.DataItem("title")%></td>
```

```
<td><%#Container.DataItem("artist")%></td>
<td><%#Container.DataItem("country")%></td>
<td><%#Container.DataItem("company")%></td>
<td><%#Container.DataItem("price")%></td>
<td><%#Container.DataItem("year")%></td>
</tr>
</AlternatingItemTemplate>
<FooterTemplate>
</table>
</FooterTemplate>
</asp:Repeater>
</form>
</body>
</html>
```

Output

Title	Artist	Country	Company	Price	Year
Empire Burlesque	Bob Dylan	USA	Columbia	10.90	1985
Hide your heart	Bonnie Tyler	UK	CBS Records	9.90	1988
Greatest Hits	Dolly Parton	USA	RCA	9.90	1982
Still got the blues	Gary Moore	UK	Virgin records	10.20	1990
Eros	Eros Ramazzotti	EU	BMG	9.90	1997

Using the <SeparatorTemplate>

The <SeparatorTemplate> element can be used to describe a separator between each record. The following example inserts a horizontal line between each table row:

```
<%@ Import Namespace="System.Data" %>
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycdcatalog=New DataSet
  mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
  cdcatalog.DataSource=mycdcatalog
```

```
        cdcatalog.DataBind()
end if end sub
</script>
<html>
<body>
<form runat="server">
<asp:Repeater id="cdcatalog" runat="server">
<HeaderTemplate>
<table border="0" width="100%">
<tr>
<th>Title</th>
<th>Artist</th>
<th>Country</th>
<th>Company</th>
<th>Price</th>
<th>Year</th>
</tr>
</HeaderTemplate>
<ItemTemplate>
<tr>
<td><%#Container.DataItem("title")%></td>
<td><%#Container.DataItem("artist")%></td>
<td><%#Container.DataItem("country")%></td>
<td><%#Container.DataItem("company")%></td>
<td><%#Container.DataItem("price")%></td>
<td><%#Container.DataItem("year")%></td>
</tr>
</ItemTemplate>
<SeparatorTemplate>
<tr>
```

```
<td colspan="6"><hr /></td>
</tr>
</SeparatorTemplate>
<FooterTemplate>
</table>
</FooterTemplate>
</asp:Repeater>
</form>
</body> </html>
```

Output

Title	Artist	Country	Company	Price	Year
Empire Burlesque	Bob Dylan	USA	Columbia	10.90	1985
Hide your heart	Bonnie Tyler	UK	CBS Records	9.90	1988
Greatest Hits	Dolly Parton	USA	RCA	9.90	1982
Still got the blues	Gary Moore	UK	Virgin records	10.20	1990
Eros	Eros Ramazzotti	EU	BMG	9.90	1997

ASP.NET - The DataList Control

The DataList control is, like the Repeater control, used to display a repeated list of items that are bound to the control. However, the DataList control adds a table around the data items by default. Bind a DataSet to a DataList Control The DataList control is, like the Repeater control, used to display a repeated list of items that are bound to the control. However, the DataList control adds a table around the data items by default. The DataList control may be bound to a database table, an XML file, or another list of items. Here we will show how to bind an XML file to a DataList control.

Next, create a DataSet for the XML file and load the XML file into the DataSet when the page is first loaded:

```
<script runat="server">
```

```
sub Page_Load
if Not Page.IsPostBack then
  dim mycdcatalog=New DataSet
mycdcatalog.ReadXml(MapPath("cdcatalog.xml")) end if
end sub
```

Then we create a DataList in an .aspx page. The contents of the <HeaderTemplate> element are rendered first and only once within the output, then the contents of the <ItemTemplate> element are repeated for each "record" in the DataSet, and last, the contents of the <FooterTemplate> element are rendered once within the output:

Then we add the script that creates the DataSet and binds the mycdcatalog DataSet to the DataList control. We also fill the DataList control with a <HeaderTemplate> that contains the header of the table, an <ItemTemplate> that contains the data items to display, and a <FooterTemplate> that contains a text. Note that the gridlines attribute of the DataList is set to "both" to display table borders:

```
<%@ Import Namespace="System.Data" %>
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycdcatalog=New DataSet
  mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
  cdcatalog.DataSource=mycdcatalog cdcatalog.DataBind()
end if end sub
</script>
<html>
<body>
<form runat="server">
<asp:DataList id="cdcatalog" gridlines="both" runat="server">
<HeaderTemplate>
```

```
My CD Catalog
</HeaderTemplate>
<ItemTemplate>
"<%#Container.DataItem("title")%>" of
<%#Container.DataItem("artist")%> -
$<%#Container.DataItem("price")%>
</ItemTemplate>
<FooterTemplate> Copyright Hege
Refsnes
</FooterTemplate>
</asp:DataList>
</form>
</body>
</html>
```

My CD Catalog
"Empire Burlesque" of Bob Dylan - $10.90
"Hide your heart" of Bonnie Tyler - $9.90
"Greatest Hits" of Dolly Parton - $9.90
"Still got the blues" of Gary Moore - $10.20
"Eros" of Eros Ramazzotti - $9.90
Copyright Hege Refsnes

You can also add styles to the DataList control to make the output more fancy: Example

```
<%@ Import Namespace="System.Data" %>
<script runat="server">
sub Page_Load
if Not Page.IsPostBack then
  dim mycdcatalog=New DataSet
  mycdcatalog.ReadXml(MapPath("cdcatalog.xml"))
  cdcatalog.DataSource=mycdcatalog cdcatalog.DataBind()
end if
```

```
end sub
</script>
<html>
<body>
<form runat="server">
<asp:DataList id="cdcatalog" runat="server" cellpadding="2" cellspacing="2"
borderstyle="inset" backcolor="#e8e8e8" width="100%" headerstyle-font-name="Verdana" headerstyle-font-
size="12pt" headerstyle-horizontalalign="center" headerstyle-font-bold="true" itemstyle-
backcolor="#778899" itemstyle-forecolor="#ffffff" footerstyle-font-size="9pt" footerstyle-font-italic="true">
<HeaderTemplate> My CD
Catalog
</HeaderTemplate>
<ItemTemplate>
"<%#Container.DataItem("title")%>" of
<%#Container.DataItem("artist")%> -
$<%#Container.DataItem("price")%>
</ItemTemplate>
<FooterTemplate> Copyright Hege
Refsnes
</FooterTemplate>
</asp:DataList>
</form>
</body>
</html>
```

My CD Catalog

"Empire Burlesque" of Bob Dylan - $10.90

"Hide your heart" of Bonnie Tyler - $9.90

"Greatest Hits" of Dolly Parton - $9.90

"Still got the blues" of Gary Moore - $10.20

"Eros" of Eros Ramazzotti - $9.90

Copyright Hege Refsnes

Using SQL Server with ASP.NET

Microsoft SQL Server is based on the client/server architecture, in which data is stored on a centralized computer called a server. Other computers, called clients, can access the data stored on the server through a network. The client/server architecture prevents data inconsistency. You can access data stored on a SQL server through Web Forms. To do so, you can create Web applications that have data access controls. These data access Web controls present the data in a consistent manner irrespective of the actual source, such as Microsoft SQL Server or MS Access. Therefore, while creating a Web application, you do not need to worry about the format of the data. However, before you can access or manipulate data from a SQL server, you need to perform the following steps in the specified sequence:

1. Establish a connection with the SQL Server.
2. Write the actual command to access or manipulate data.
3. Create a result set of the data from the data source with which the application can work.
   This result set is called the data set and is disconnected from the actual source. The application accesses and updates data in the data set, which is later reconciled with the actual data source.

Selecting Data from Table

To achieve this functionality, you first need to import two namespaces, System.Data and System.Data.SqlClient, into your Web Forms page. The syntax is given as follows:

```
<%@ Import Namespace="System.Data.OleDb" %>
```

We need this namespace to work with Microsoft Access and other OLE DB database providers. We will create the connection to the database in the Page_Load subroutine. We create a dbconn variable as a new OleDbConnection class with a connection string which identifies the OLE DB provider and the location of the database. Then we open the database connection:

```
<%@ Import Namespace="System.Data.OleDb" %>
<script runat="server">
sub Page_Load dim dbconn
dbconn=New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;//;;;;data source="
& server.mappath("northwind.mdb"))
```

```
dbconn.Open()
end sub
</script>
```

The connection string must be a continuous string without a line break. To specify the records to retrieve from the database, we will create a dbcomm variable as a new OleDbCommand class. The OleDbCommand class is for issuing SQL queries against database tables:

```
<%@ Import Namespace="System.Data.OleDb" %>
    <script runat="server">
    sub Page_Load
    dim dbconn,sql,dbcomm
    dbconn=New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;
    data source=" & server.mappath("northwind.mdb"))
    dbconn.Open()
    sql="SELECT * FROM customers" dbcomm=New
    OleDbCommand(sql,dbconn) end sub
</script>
```

The OleDbDataReader class is used to read a stream of records from a data source. A DataReader is created by calling the ExecuteReader method of the OleDbCommand object:

```
<%@ Import Namespace="System.Data.OleDb" %>
<script runat="server">
sub Page_Load
dim dbconn,sql,dbcomm,dbread
dbconn=New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;
data source=" & server.mappath("Bank.mdb"))
dbconn.Open()
sql="SELECT * FROM customers" dbcomm=New
OleDbCommand(sql,dbconn)
dbread=dbcomm.ExecuteReader()
end sub
</script>
```

Then we bind the DataReader to a Repeater control:

```
<%@ Import Namespace="System.Data.OleDb" %>
<script runat="server">
sub Page_Load
dim dbconn,sql,dbcomm,dbread
    dbconn = New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;data source=" &
Server.MapPath("Bank.mdb"))
dbconn.Open()
    sql = "SELECT * From customer where Address='Mnr'"
dbcomm=New OleDbCommand(sql,dbconn)
dbread=dbcomm.ExecuteReader() customers.DataSource=dbread
customers.DataBind()
dbread.Close() dbconn.Close()
end sub
</script>
<html>
<body>
<form id="Form1" runat="server">
<asp:Repeater id="customers" runat="server">

<HeaderTemplate>
<table border="1" width="100%">
<tr>
<th>Customer ID</th>
<th>Customer Name</th>
<th>Address</th>
<th>Age</th>
<th>Mobile</th>
<th>Email</th>
```

```
</tr>
</HeaderTemplate>
<ItemTemplate>
<tr>
<td><%#Container.DataItem("Cid")%></td>
<td><%#Container.DataItem("CName")%></td>
<td><%#Container.DataItem("Address")%></td>
<td><%#Container.DataItem("Age")%></td>
<td><%#Container.DataItem("Mobile")%></td>
<td><%#Container.DataItem("Email")%></td>
</tr>
</ItemTemplate>
<FooterTemplate>
</table>
</FooterTemplate>
</asp:Repeater>
</form>
</body>
</html>
```

Creating Table

```
<%@ Import Namespace="System.Data.OleDb" %>
<script runat="server">
sub Page_Load
    Dim dbconn, sql, dbcomm
    dbconn = New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;data source=" &
Server.MapPath("Bank.mdb"))
    dbconn.Open()
    sql = "Create Table Products (ProductID VarChar (4) Primary Key, ProductName
VarChar (20), UnitPrice Money,QtyAvailable Integer)" dbcomm =
    New OleDbCommand(sql, dbconn)
```

```
        dbcomm.ExecuteNonQuery()
dbconn.Close()
end sub
</script>
```

Inserting Data into Table

```
<%@ Import Namespace="System.Data.OleDb" %>
<script runat="server">
sub Page_Load
        Dim dbconn, sql, dbcomm dbconn =
        New
OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;datasource=" &
Server.MapPath("Bank.mdb"))
        dbconn.Open()
        sql = "Insert into customer values(3,'Aaryan','Pkr',34,'9803456789','ar@gmail.com')" dbcomm =
        New OleDbCommand(sql, dbconn)
        dbcomm.ExecuteNonQuery()
       dbconn.Close()
end sub
</script>
```

Deleting Data from Table

```
<%@ Import Namespace="System.Data.OleDb" %>
<script runat="server">
sub Page_Load
        Dim dbconn, sql, dbcomm
        dbconn = New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;data source=" &
Server.MapPath("Bank.mdb"))
        dbconn.Open()
        sql = "Delete from customer where Cid=1" dbcomm = New
        OleDbCommand(sql, dbconn)
```

```
    dbcomm.ExecuteNonQuery()
dbconn.Close()
end sub
</script>
```

Updating Data in the Table

```
<%@ Import Namespace="System.Data.OleDb" %>
<script runat="server">
sub Page_Load
    Dim dbconn, sql, dbcomm
    dbconn = New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;data source=" &
Server.MapPath("Bank.mdb"))
    dbconn.Open()
    sql = "update customer set Address='Ktm' where Cid=2" dbcomm =
    New OleDbCommand(sql, dbconn) dbcomm.ExecuteNonQuery()
dbconn.Close()
end sub
</script>
```

A Sample GUI Based Form with Database Connectivity

```
<%@ Page Language="VB"%>
<%@ Import Namespace="System.Data.OleDb"%>
<html>
<script language="VB" runat=server>
Sub Insert_Click(Src As Object, E As EventArgs)
' Connect to Database
    Dim cnAccess As New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;data source=" &
Server.MapPath("Bank.mdb"))
cnAccess.Open()
dim sID, sFName, sLName, sAge, sInsertSQL as string
```

```
sID = eID.Text
sFName = FName.Text sLName =
LName.Text sAge = Age.Text
'Make the insert statement
      sInsertSQL = "insert into employees values(" & sID & ",'" & sFName & "','" &
sLName & "'," & sAge & ")"
'Make the OleDbCommand object
dim cmdInsert as New OleDbCommand(sInsertSQL,cnAccess)
' This not a query so we do not expect any return data so use
' the ExecuteNonQuery method
cmdInsert.ExecuteNonQuery() response.write
("Data recorded!") End Sub
</script>
<body>
<form id="Form1" runat=server>
<h3><font face="Verdana">Enter Employee Details</font></h3>
<table>
<tr>
<td>ID:</td>
<td><asp:textbox id="eID" runat="server"/></td>
</tr>
<tr>
<td>First Name:</td>
<td><asp:textbox id="FName" runat="server"/></td>
</tr>
<tr>
<td>Last Name:</td>
<td><asp:textbox id="LName" runat="server"/></td>
</tr>
```

```
<tr>
<td>Age:</td>
<td><asp:textbox id="Age" runat="server"/></td>
</tr>
</table>
<asp:button ID="Button1" text="Insert" OnClick="Insert_Click" runat=server/>
<p>
<asp:Label id="Msg" ForeColor="red" Font-Name="Verdana" Font-Size= "10"
runat=server />
</form>
</body>
</html>
```

Handling Session and Cookie in ASP.net

A session is defined as the period of time that a unique user interacts with a Web application. Active Server Pages (ASP) developers who wish to retain data for unique user sessions can use an intrinsic feature known as session state. Programmatically, session state is nothing more than memory in the shape of a dictionary or hash table, e.g. key-value pairs, which can be set and read for the duration of a user's session. For example, a user selects stocks to track and the Web application can store these values in the user's ASP session instance:

```
Session("Stocks") = "MSFT; VRSN; GE"
```

On subsequent pages these values are read and the Web application has access to these values without the user re-entering them:

```
Dim StockString
StockString = Session("Stocks")
```

ASP maintains session state by providing the client with a unique key assigned to the user when the session begins. This key is stored in an HTTP cookie that the client sends to the server on each request. The server can then read the key from the cookie and re-inflate the server session state.

Problems with ASP Session State

ASP developers know session state as a great feature, but one that is somewhat limited. These limitations include:

- **Process dependent.** ASP session state exists in the process that hosts ASP; thus the actions that affect the process also affect session state. When the process is recycled or fails, session state is lost.
- **Server farm limitations.** As users move from server to server in a Web server farm, their session state does not follow them. ASP session state is machine specific. Each ASP server provides its own session state, and unless the user returns to the same server, the session state is inaccessible. While network IP level routing solutions can solve such problems, by ensuring that client IPs are routed to the originating server, some ISPs choose to use a proxy load-balancing solution for their clients. Most infamous of these is AOL. Solutions such as AOL's prevent network level routing of requests to servers because the IP addresses for the requestor cannot be guaranteed to be unique.
- **Cookie dependent.** Clients that don't accept HTTP cookies can't take advantage of session state. Some clients believe that cookies compromise security and/or privacy and thus disable them, which disables session state on the server.

These are several of the problem sets that were taken into consideration in the design of ASP.NET session state.

ASP.NET Session State

ASP.NET session state solves all of the above problems associated with classic ASP session state:

- **Process independent.** ASP.NET session state is able to run in a separate process from the ASP.NET host process. If session state is in a separate process, the ASP.NET process can come and go while the session state process remains available. Of course, you can still use session state in process similar to classic ASP, too.
- **Support for server farm configurations.** By moving to an out-of-process model, ASP.NET also solves the server farm problem. The new out-of-process model allows all servers in the farm to share a session state process. You can implement this by changing the ASP.NET configuration to point to a common server.

- **Cookie independent.** Although solutions to the problem of cookieless state management do exist for classic ASP, they're not trivial to implement. ASP.NET, on the other hand, reduces the complexities of cookieless session state to a simple configuration setting.

Using ASP.NET Session State

Before we use session state, we need an application to test it with. Below is the code for a simple Visual Basic application that writes to and reads from session state, **SessionState.aspx**:

```
<Script runat=server>
  Sub Session_Add(sender As Object, e As EventArgs)
    Session("MySession") = text1.Value
      span1.InnerHtml = "Session data updated! <P> Your session contains: <font color=red>"
+ Session("MySession").ToString() + "</font>"
        End Sub
  Sub CheckSession(sender As Object, e As EventArgs) If
      (Session("MySession") = "") Then
        span1.InnerHtml = "NOTHING, SESSION DATA LOST!" Else
        span1.InnerHtml = "Your session contains: <font color=red>" +
Session("MySession").ToString() + "</font>"
            End If
        End Sub
</Script>
<form id="Form1" runat=server>
  <input id=text1 type=text runat=server>
  <input id="Submit1" type=submit runat=server OnServerClick="Session_Add"
Value="Add to Session State">
  <input id="Submit2" type=submit runat=server OnServerClick="CheckSession"
Value="View Session State">
</form>
<hr size=1>
<font size=6><span id=span1 runat=server/></font>
```

This simple page wires up two server-side events for the **Add** and **View** buttons, and simply sets the session state to the value in the text box. There are four general configuration settings we can look at in more detail: in-process mode, out-of-process mode, SQL Server mode, and Cookieless.

Handling Cookies

A cookie is a small bit of text that accompanies requests and pages as they go between the Web server and browser. The cookie contains information the Web application can read whenever the user visits the site. For example, if a user requests a page from your site and your application sends not just a page, but also a cookie containing the date and time, when the user's browser gets the page, the browser also gets the cookie, which it stores in a folder on the user's hard disk. Later, if user requests a page from your site again, when the user enters the URL the browser looks on the local hard disk for a cookie associated with the URL. If the cookie exists, the browser sends the cookie to your site along with the page request. Your application can then determine the date and time that the user last visited the site. You might use the information to display a message to the user or check an expiration date.

Cookies are associated with a Web site, not with a specific page, so the browser and server will exchange cookie information no matter what page the user requests from your site. As the user visits different sites, each site might send a cookie to the user's browser as well; the browser stores all the cookies separately. Cookies help Web sites store information about visitors. More generally, cookies are one way of maintaining continuity in a Web application—that is, of performing state management. Except for the brief time when they are actually exchanging information, the browser and Web server are disconnected. Each request a user makes to a Web server is treated independently of any other request. Many times, however, it's useful for the Web server to recognize users when they request a page. For example, the Web server on a shopping site keeps track of individual shoppers so the site can manage shopping carts and other user-specific information. A cookie therefore acts as a kind of calling card, presenting pertinent identification that helps an application know how to proceed.

Cookies are used for many purposes, all relating to helping the Web site remember users. For example, a site conducting a poll might use a cookie simply as a Boolean value to indicate whether a user's browser has already participated in voting so that the user cannot vote twice. A

site that asks a user to log on might use a cookie to record that the user already logged on so that the user does not have to keep entering credentials.

Page Directives

Page directives are used to set various attributes about a page. The ASP Engine and the compiler follow these directives to prepare a page. There are many kinds of directives. The most frequently ones are the following: @ Page,@ Import,@ Implements, @ Register,@ OutputCache and @ Assembly directives. These directives can be placed anywhere in a page, however, these are typically placed at the top.

1. @ Page: We may use this directive to declare many page-related attributes about a particular page. For example, we use this directive to declare the language to be used in a page, such as *<%@ Page Language="VB" Debug="true" %>* page.
2. @ Import: We use this directive to import a namespace in the page class file. For example, in the following directive, we are importing the *System.Data.OleDb* namespace in our page: *<%@ Import Namespace="System.Data.OleDb" %>*.
3. @ OutputCache: We can use this directive to specify how to cache the page. In the following example, we are setting the duration that a page or user control is output cached:*<%@ OutputCache Duration="10" /%>*.
4. @ Register: This directive is used to register a custom control in a page. In the following example, we are registering one of our user custom controls in page: *<%@ Register tagprefix ="utoledo" tagname="Time" Src="TimeUserControl.ascx"%>*.
5. @ Assembly We use this directive to link to an assembly to the current page or user control. The following example shows how to link to an assembly-named payroll: *<%@ Assembly Name="Payroll" %>*.
6. @ Implements This directive enables us to implement an interface in our page. In the following example, we are implementing the *IpostBackEventHandler* interface in one of our user controls: *<%@ ImplementsInterface="System.Web.UI.IPostBackEventHandler" %>*.

Tag Libraries

In a Web application, a common design goal is to separate the display code from business logic. Java tag libraries are one solution to this problem. Tag libraries allow you to isolate business logic from the display code by creating a **Tag** class (which performs the business logic) and including an HTML-like tag in your JSP page. When the Web server encounters the tag within your JSP page, the Web server will call methods within the corresponding Java **Tag** class to produce the required HTML content.

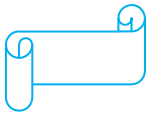
Microsoft® ASP.NET uses Web form controls to serve the same purpose as Java tag libraries. Similar to JSP tags, Web form controls are added to an ASP.NET Web page using an HTML-like syntax. Unlike JSP tags however, a Web form control is actually an object that is contained within your ASP.NET page. This allows you to access information from your Web form control both before and after the page is loaded. **The Microsoft® .NET Framework contains many ready-to-use Web form controls, including a Calendar Web form control and a Crystal Reports Viewer Web control.** If you require different functionality than is provided by these Web form controls, you can either extend the existing Web form controls or create your own Web form controls by implementing various interfaces.

Tag libraries were designed so that Java code could be executed within a JSP page without using Java script blocks, which clutter up the HTML and break the design goal of separating display code from business logic. Instead of script blocks, tag libraries allow you to create custom HTML-like tags that map to a Java class that performs the business logic. Groups of these HTML-like tags are called tag libraries. Creating and using a custom tag library involves three things:

- One or more classes that implement the **javax.servlet.jsp.tagext.Tag** interface. The **Tag** interface defines six methods that allow your JSP page to use the class to create the desired HTML output. There are also classes/interfaces that implement/extend the **Tag** interface, such as **TagSupport** and **BodyTagSupport**, to make it easier for you to develop your custom tag.
- An XML document that describes your tag library. Tag library description files must conform to the JSP tag library description DTD, and generally have an extension of "tld".
- Importing the tag library to the JSP page using the **taglib** directive.

Once the three requirements are met, you can use the tags in your tag library anywhere within your JSP page.

For detail explore: ***http://msdn.microsoft.com/en-us/library/aa478990.aspx***



Unit 5: Introduction to Advanced Server Side Issues

Database connection

A database connection is a facility in computer science that allows client software to communicate with database server software, whether on the same machine or not. A connection is required to send commands and receive answers.Connections are a key concept in data-centric programming. Since some DBMS engines require considerable time to connect connection pooling was invented to improve performance. No command can be performed against a database without an "open and available" connection to it.

Connections are built by supplying an underlying driver or provider with a connection string, which is a way of addressing a specific database or server and instance as well as user authentication credentials (for example, `Server=sql_box;Database=Common;User ID=uid;Pwd=password;`). Once a connection has been built it can be opened and closed at will, and properties (such as the command time-out length, or transaction, if one exists) can be set. The Connection String is composed of a set of key/value pairs as dictated by the data access interface and data provider being used.

Databases, such as PostgreSQL, only allow one operation to be performed at a time on each connection. If a request for data (a SQL Select statement) is sent to the database and a result set is returned, the connection is open but not available for other operations until the client finishes consuming the result set. Other databases, like SQL Server 2005 (and later), do not impose this limitation. However, databases that provide multiple operations per connection usually incur far more overhead than those that permit only a single operation task at a time.

Connection Pooling

Database connections are finite and expensive and can take a disproportionately long time to create relative to the operations performed on them. It is very inefficient for an application to create and close a database connection whenever it needs to update a database.Connection pooling is a technique designed to alleviate this problem. A pool of database connections can be created and then shared among the applications that need to access the database. When an application needs database access, it requests a connection from the pool. When it is finished, it returns the connection to the pool, where it becomes available for use by other applications.

The connection object obtained from the connection pool is often a wrapper around the actual database connection. The wrapper understands its relationship with the pool, and hides the details of the pool from the application. For example, the wrapper object can implement a "close" method that can be called just like the "close" method on the database connection. Unlike the method on the database connection, the method on the wrapper may not actually close the database connection, but instead return it to the pool. The application need not be aware of the connection pooling when it calls the methods on the wrapper object.
This approach encourages the practice of opening a connection in an application only when needed, and closing it as soon as the work is done, rather than holding a connection open for the entire life of the application. In this manner, a relatively small number of connections can service a large number of requests. This is also called **multiplexing.**In a client/server architecture, on the other hand, a persistent connection is typically used so that server state can be managed. This "state" includes server-side cursors, temporary

products, connection-specific functional settings, and so on.
It is desirable to set some limit on the number of connections in the pool. Using too many connections may just cause thrashing rather than get more useful work done. In case an operation is attempted and all connections are in use, the operation can block until a connection is returned to the pool, or an error may be returned.

ActiveX Data Objects

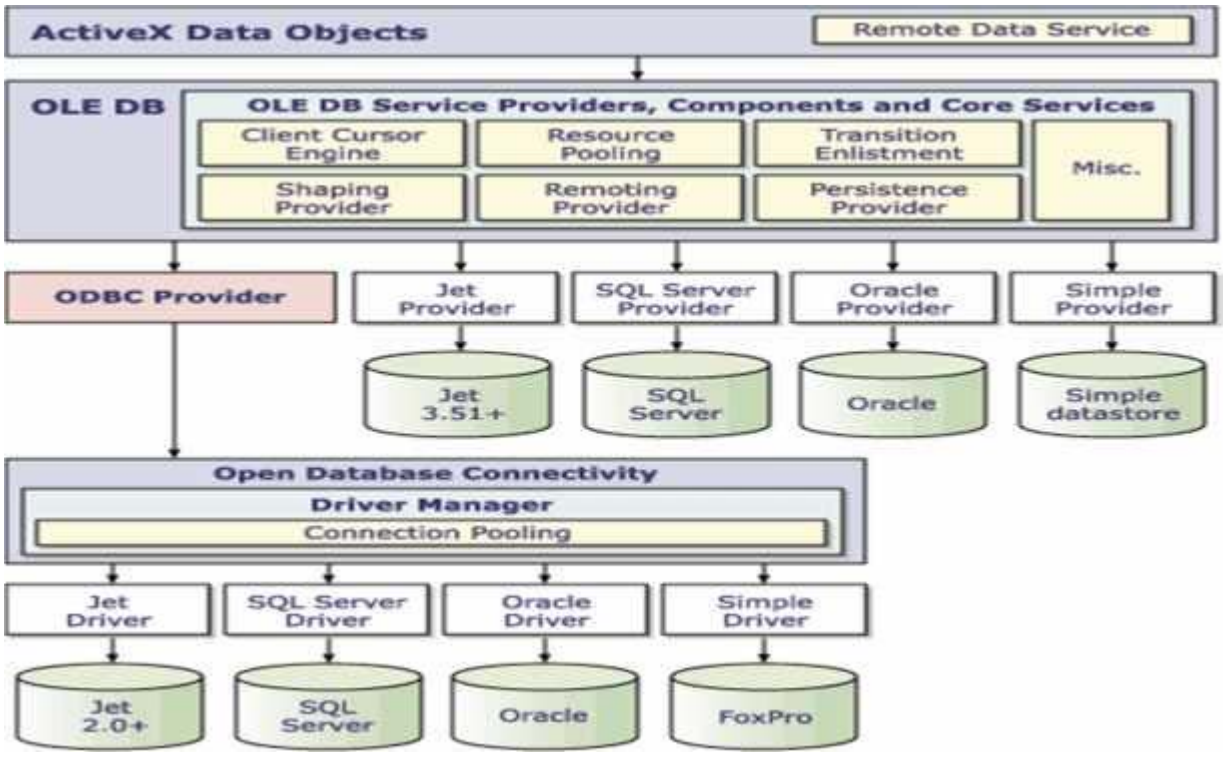
ADO itself is a COM (Microsoft Component Object Model) component. ADO is designed to give a single method for accessing data to everybody.

There are three main objects on which the ADO object model is built:

- Connection Object
- Command Object
- Recordset Object

Microsoft's ActiveX Data Objects (ADO) is a set of Component Object Model (COM) objects for accessing data sources. A part of MDAC, it provides a middleware layer between programming languages and OLE DB
(a means of accessing data stores, whether they be databases or otherwise, in a uniform manner). ADO allows a developer to write programs that access data without knowing how the database is implemented. You must be aware of your database for connection only. No knowledge of SQL is required to access a database when using ADO, although one can use ADO to execute SQL commands. The disadvantage of this (i.e. using SQL directly) is that it introduces a dependency upon the type of database used.

It is positioned as a successor to Microsoft's earlier object layers for accessing data sources, including RDO (Remote Data Objects) and DAO (Data Access Objects). ADO was introduced by Microsoft in October 1996.



Some basic steps are required in order to be able to access and manipulate data using ADO :

1. Create a connection object to connect to the database.
2. Create a recordset object in order to receive data in.
3. Open the connection
4. Populate the recordset by opening it and passing the desired table name or SQL statement as a parameter to open function.
5. Do all the desired searching/processing on the fetched data.
6. Commit the changes you made to the data (if any) by using Update or UpdateBatch methods.
7. Close the recordset

8. Close the connection

OLE DB (Object Linking and Embedding, Database)

OLE-DB allows programs to access information in any type of data store(databases, spreadsheets, graphs, emails and so on), where ODBC allows only access to a database. It's a collection of the COM interfac e encapsulating various database management system services. A service is a component that extends the functionality of data providers, by providing interfaces not natively supported by the data store.

OLE DB (Object Linking and Embedding, Database) is an API designed by Microsoft for accessing data from a variety of sources in a uniform manner. It is a set of interfaces implemented using the Component Object Model (COM); it is otherwise unrelated to OLE. It was designed as a higher-level replacement for, and successor to, ODBC, extending its feature set to support a wider variety of non-relational databases, such as object databases and spreadsheets that do not necessarily implement SQL.

OLE DB separates the data store from the application that needs access to it through a set of abstractions that include the datasource, session, command, and rowsets. This was done because different applications need access to different types and sources of data, and do not necessarily want to know how to access functionality with technology-specific methods. OLE DB is conceptually divided into consumers and providers. The consumers are the applications that need access to the data, and the providers are the software components that implement the interface and thereby provides the data to the consumer. OLE DB is part of the Microsoft Data Access Components (MDAC) stack. MDAC is a group of Microsoft technologies that interact together as a framework that allows programmers a uniform and comprehensive way of developing applications for accessing almost any data store.

ODBC (Open Database Connectivity)

ODBC –Where everybody can get the information they want without having to worry about the method of data storage. CGI(Common Gateway Interface) w/ ODBC was used before ADO. The web server itself would use ODBC to communicate with the database. CGI isn't used as much anymore due to it being slower and having to use languages like Perl and C++.

Open Database Connectivity (ODBC) provides a standard software interface for accessing database management systems (DBMS). The designers of ODBC aimed to make it independent of programming languages, database systems, and operating systems. Thus, any application can use ODBC to query data from a database, regardless of the platform it is on or DBMS it uses. This is accomplished by using an ODBC driver as a translation layer between the application and the DBMS. The application thus only needs to know ODBC syntax, and the driver can then pass the query to the DBMS in its native format, returning the data in a format the application can understand.

ADO.NET (ActiveX Data Object for .NET) is a set of computer software components that programmers can use to access data and data services. It is a part of the base class library that is included with the Microsoft .NET Framework. It is commonly used by programmers to access and modify data stored in relational database systems, though it can also access data in non-relational sources. ADO.NET is sometimes considered an evolution of ActiveX Data Objects (ADO) technology, but was changed so extensively that it can be considered an entirely new product.

ADO.NET: Explicit and

Factored

The functionality that the ADO **Recordset** provides has been factored into the following explicit objects in
ADO.NET: the **DataReader**, which provides fast, forward-only, read-only access to query results; the DataSet,

which provides an in-memory relational representation of data; and the **DataAdapter**, which provides a bridge between the **DataSet** and the data source. The ADO.NET Command object also includes explicit functionality such as the **ExecuteNonQuery** method for commands that do not return rows, and the ExecuteScalar method for queries that return a single value rather than a row set.To better understand how the design of ADO.NET is made up of objects that are optimized to perform explicit behavior, consider the following tasks that are common when working with data.

Forward-Only Read-Only Data Streams

Applications, particularly middle-tier applications, often process a series of results programmatically, requiring no user interaction and no updating of or scrolling back through the results as they are read. In ADO, this type of data retrieval is performed using a **Recordset** with a forward-only cursor and a read-only lock. In ADO.NET, however, the **DataReader** object optimizes this type of data retrieval by providing a non-buffered, forward- only, read-only stream that provides the most efficient mechanism for retrieving results from the database. Much of this efficiency is gained as a result of the **DataReader** having been designed solely for this purpose, without having to support scenarios where data is updated at the data source or cached locally as with the ADO **Recordset**.

Returning a Single Value

Often the only data to be retrieved from a database is a single value (for example, an account balance). In ADO, you perform this type of data retrieval by creating a **Recordset** object, reading through the results, retrieving the single value, and then closing the **Recordset**. In ADO.NET, however, the Command object supports this function through the **ExecuteScalar** method, which returns the single value from the database without having to introduce an additional object to hold the results.

Disconnected Access to Data

A frequent case for exposing data is a representation in which a user can navigate the data in an ad-hoc manner without holding locks or tying up resources on the server. Some examples of this scenario are binding data to a control or combining data from multiple data sources and/or XML. The ADO **Recordset** provides some support for these scenarios, using a client-side cursor location. However, in ADO.NET the **DataSet** is explicitly designed for such tasks.

The **DataSet** provides a common, completely disconnected data representation that can hold results from a variety of different sources. Because the **DataSet** is completely independent of the data source, it provides the same performance and semantics regardless of whether the data is loaded from a database, loaded from XML, or is generated by the application. A single **DataSet** may contain tables populated from several different databases and other non-database sources; to the consumer of the **DataSet** it all looks and behaves exactly the same. Within the **DataSet** you can define relations to navigate from a table populated from one database (for example, "Customers"), to a related table populated from an entirely different database (for example, "Orders"), and from there to a third table (for example, "OrderDetails") containing values loaded from XML. The relational capabilities of the **DataSet** provide an advantage over the **Recordset**, which is limited to exposing the results from multiple tables either as a single joined result, or by returning multiple distinct result sets, requiring the developer to handle and relate the results manually. Though the **Recordset** has the ability to return and navigate hierarchical results (using the MSDataShape provider), the DataSet provides much greater flexibility when dealing with related result sets. The **DataSet** also provides

the ability to transmit results to and from a remote client or server in an open XML format, with the schema defined using the XML Schema definition language (XSD).

Retrieving and Updating Data from a Data Source

Based on customer feedback and common use cases it is clear that in most application development scenarios (with the exception of ad-hoc tools and generic data components) the developer knows certain things about the data at design time that technologies like ADO attempt to derive at run time. For example, in most middle-tier applications the developer knows, at the time of application development, the type of database to be accessed, what queries will be executed, and how the results will be returned. ADO.NET gives you the ability to apply this knowledge at design time in order to provide better run-time performance and predictability.

As an example, when using batch updating with ADO **Recordset** objects, you must submit changes to the database by executing appropriate INSERT, UPDATE, and DELETE statements for each row that has changed. ADO generates these statements implicitly, at run time, based on metadata that is often expensive to obtain. ADO.NET, however, enables you to explicitly specify INSERT, UPDATE, and DELETE commands, as well as custom business logic such as a stored procedure, that will be used to resolve changes in a **DataSet** back to the data source using the **DataAdapter**. This model provides you with greater control over how application data is returned and updated, and removes the expense of gathering the metadata at run time.

The DataAdapter provides the bridge between the **DataSet** and the data source. A **DataAdapter** is used to populate a DataSet with results from a database, and to read changes out of a **DataSet** and resolve those changes back to the database. Using a separate object, the **DataAdapter**, to communicate with the database allows the **DataSet** to remain completely generic with respect to the data it contains, and gives you more control over when and how commands are executed and changes are sent to the database. ADO performs much of this behavior implicitly; however the explicit design of ADO.NET enables you to fine-tune your interaction with a data source for best performance and scalability.
The implicit update behavior of ADO is also available in ADO.NET using a CommandBuilder object that, based on a single table SELECT, automatically generates the INSERT, UPDATE, and DELETE commands used for queries by the **DataAdapter**. However, the compromise for this convenience is slower performance and less control over how changes are propagated to the data source because, as with ADO, the commands are generated from metadata collected at run time.

Data Types

In ADO, all results are returned in a standard OLE Automation Variant type. This can hinder performance because, in addition to conversion overhead, variants are allocated using task-allocated system memory, which causes contention across the system. When retrieving results from a **DataReader** in ADO.NET, however, you can retrieve columns in their native data type, as a common Object class, without going through expensive conversions. Data values can either be exposed as .NET Framework types, or can be placed in a proprietary structure in the .NET Framework to preserve the fidelity of the native type. An example of this is the SQL Server .NET Data Provider, which can be used to expose Microsoft® SQL Server™ data as .NET Framework types, or as proprietary types defined by the classes in the System.Data.SqlTypes namespace

Comparison of ADO and ADO.NET

The following is a comparison of two different database access technologies from Microsoft, namely, ActiveX Data Objects (ADO) and ADO.NET. Before comparing the two technologies, it is essential to get an overview of Microsoft Data Access Components (MDAC) and the .NET Framework. Microsoft Data Access Components provide a uniform and comprehensive way of developing applications for accessing almost any data store entirely from unmanaged code. The .NET Framework is an application virtual machine-based software environment that provides security mechanisms, memory management, and exception handling and is designed so that developers need not consider the capabilities of the specific CPU that will execute the .NET application. The .NET application virtual machine turns intermediate language (IL) into machine code. High-level language compilers for C#, VB.NET and C++ are provided to turn source code into IL. ADO.NET is shipped with the Microsoft NET Framework.

ADO relies on COM whereas ADO.NET relies on managed-providers defined by the .NET CLR. ADO.NET does not replace ADO for the COM programmer; rather, it provides the .NET programmer with access to relational data sources, XML, and application data.

	ADO	ADO.NET
Business Model	Connection-oriented Models used mostly	Disconnected models are used:Message-like Models.
Disconnected Access	Provided by Record set	Provided by Data Adapter and Data set
XML Support	Limited	Robust Support
Connection Model	Client application needs to be connected always	

to data-server while working on the data, unless using client-side cursors or a disconnected Record set

	Client disconnected as soon as the data is processed. DataSet is disconnected at all times.	
Data Passing	ADO objects communicate in binary mode.	ADO.NET uses XML for passing the data.
Control of data access behaviors		
Design-time support		

Includes implicit behaviors that may not always be required in an application and that may therefore limit performance.

Derives information about data implicitly at run time, based on metadata that is often expensive to obtain.

Provides well-defined, factored components with predictable behavior, performance, and semantics.

Leverages known metadata at design time in order to provide better run-time performance and more consistent run-time behavior.

JDBC (Java Database Connectivity)

Java DataBase Connectivity, commonly referred to as JDBC, is an API for the Java programming language that defines how a client may access a database. It provides methods for querying and updating data in a database. JDBC is oriented towards relational databases. A JDBC-to-ODBC bridge enables connections to any ODBC- accessible data source in the JVM host environment.The method Class.forName(String) is used to load the JDBC driver class. The line below causes the JDBC driver from some jdbc vendor to be loaded into the application. (Some JVMs also require the class to be instantiated with .newInstance().)

```
Class.forName( "com.somejdbcvendor.TheirJdbcDriver"
);
```

In JDBC 4.0, it's no longer necessary to explicitly load JDBC drivers using Class.forName(). See JDBC 4.0
Enhancements in Java SE 6.When a Driver class is loaded, it creates an instance of itself and registers it with the DriverManager. This can be done by including the needed code in the driver class's static block. e.g. DriverManager.registerDriver(Driver driver)

Now when a connection is needed, one of the DriverManager.getConnection() methods is used to create a JDBC
connectio
n.

```
Connection conn = DriverManager.getConnection(
    "jdbc:somejdbcvendor:other data needed by some jdbc
    vendor", "myLogin",
    "myPassword" );
tr
y
{
    /* you use the connection here */
}
finally
{
   //It's important to close the connection when you are done with it
   try { conn.close(); } catch (Throwable ignore) { /* Propagate the original
exception instead of this one that you may want just logged */ }
}
```

The URL used is dependent upon the particular JDBC driver. It will always begin with the "jdbc:" protocol, but the rest is up to the particular vendor. Once a connection is established, a statement must be created.

```
Statement stmt =
conn.createStatement();
tr
y
{
   stmt.executeUpdate( "INSERT INTO MyTable( name ) VALUES ( 'my name' ) " );
}
finally
{
   //It's important to close the statement when you are done with it
```

```
    try { stmt.close(); } catch (Throwable ignore) { /* Propagate the original
exception instead of this one that you may want just logged */ }
}
```

5.2 Creating an SQL Statement : Select , Insert, Update, and Delete

```
using System;
using System.Data;
using System.Data.OracleClient;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
public class UserInfo
{       public UserInfo()

        {

                //

                // TODO: Add constructor logic here

                //

public string AddUser(string user_id, string User_Name, string Password, string User_level, string user_department)

  {

string sql = "insert into user_setup values ('" + user_id + "','" + User_Name + "','" + Password + "','" + User_level + "','" +
user_department + "')";

    try

    {

      string ConStr;
      ConStr = ConfigurationManager.AppSettings["Constring"];
      OracleConnection conn = new OracleConnection();
      conn.ConnectionString = ConStr;
      conn.Open();
      OracleCommand cmd = conn.CreateCommand();
      cmd.CommandText = sql;
      cmd.ExecuteNonQuery();
      conn.Close();
      return "information add in user_setup in your Database";

    }

    catch (System.Exception err)
    {
      return err.ToString();
    }
```

```
}


public string checkuser(string user_name, string password)

{

  string sql = "select * from user_setup  where User_Id = '" + user_name + "'and User_Password = '" + password + "' ";

   //try
   //{
      //string ConStr;
      ////Constr = "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\\project\\EMSFINAL\\Database\\EmployeeAdministration.mdb";
      //ConStr = ConfigurationManager.AppSettings["Constring"];
      //OleDbConnection conn = new OleDbConnection();
      //conn.ConnectionString = ConStr;
      //conn.Open();
      //OleDbCommand cmd = conn.CreateCommand();
      //cmd.CommandText = sql;
   string ConStr;
   ConStr =
   ConfigurationManager.AppSettings["Constring"];
   OracleConnection conn = new OracleConnection();
   conn.ConnectionString = ConStr;
   conn.Open();
   OracleCommand cmd =
   conn.CreateCommand(); cmd.CommandText =
   sql; cmd.ExecuteNonQuery();
    OracleDataReader  dr = cmd.ExecuteReader();
      if(dr.HasRows)
      {

dr.Read();

        tr
        y

        {

          if (dr.GetValue(3).ToString().Trim() == "ADMIN")
            return "Admin";
          else if (dr.GetValue(3).ToString().Trim() == "USER")
            return "No Admin";
          else
            return "simple";
        }
        catch
```

```
{
   return "No Admin";
```

```
        }

          return "user valid";
        }
        else
          return "INVALID USERNAME OR PASSWORD";

    //}
      //  catch (System .Exception err)
      //    {
       //   return err.ToString ();
         // }

        conn.Close();

  }

  public string UpdateUser(string User_id, string User_name, string User_Password, string User_Level, string
    User_Department)

  {

    string sql;

     sql = "update user_setup set User_Id='" + User_id + "', User_Name='" + User_name + "',User_Password ='" +
User_Password + "',User_Level='" + User_Level + "',User_Department='" + User_Department + "' where
user_setup.User_Id = '" + User_id + "' ";

     try

     {

        string ConStr;

        ConStr = ConfigurationManager.AppSettings["Constring"];
        OracleConnection conn = new OracleConnection();
        conn.ConnectionString = ConStr;
        conn.Open();
        OracleCommand cmd = conn.CreateCommand();
        cmd.CommandText = sql; cmd.ExecuteNonQuery();
        conn.Close();
        return "user updated";
    }
        catch (System.Exception err)

        {
          return err.ToString ();

        }
```

```
        }

public string deletUser(string user_Id)
{
   string sql;
  // sql = "delete from  user_setup  where  User_Id = '" + user_Id + "'and User_Name='" + user_name + "'";
   sql = "delete from  user_setup  where  User_Id = '" + user_Id + "'";
   try
   {
      string ConStr;
      ConStr = ConfigurationManager.AppSettings["Constring"];
      OracleConnection conn = new OracleConnection();
      conn.ConnectionString = ConStr;
      conn.Open();
      OracleCommand cmd = conn.CreateCommand();
      cmd.CommandText = sql; cmd.ExecuteNonQuery();
      conn.Close();
      return "user deleted";
   }
   catch (System.Exception err)
   {
      return err.ToString();

   }

}
public string addlog(string user_id, string time_stamp)
{
  string sql = "insert into hr_log  values ('" + user_id + "','" + time_stamp + "')";
   try
   {        string ConStr;
      ConStr = ConfigurationManager.AppSettings["Constring"];
      OracleConnection conn = new OracleConnection();
      conn.ConnectionString = ConStr;
      conn.Open();
      OracleCommand cmd = conn.CreateCommand();
      cmd.CommandText = sql; cmd.ExecuteNonQuery();
      conn.Close();
      return null;
   }
   catch (System.Exception err)

   {        return err.ToString();

   }
}

}
```

Connection String Setting in Web.Config file

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

<!--
    Note: As an alternative to hand editing this file you can use
    the web admin tool to configure settings for your application.
    Use
    the Website->Asp.Net Configuration option in Visual
    Studio. A full list of settings and comments can be found
    in machine.config.comments usually located in
    \Windows\Microsoft.Net\Framework\v2.x\Config
-->
<configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
     <appSettings>
  <add key="Constring" value="Data Source=EMS;user id=hem; password=pass" />
  <add key="CrystalImageCleaner-AutoStart" value="true" />
  <add key="CrystalImageCleaner-Sleep" value="60000" />
  <add key="CrystalImageCleaner-Age" value="120000" />
 </appSettings>
     <connectionStrings>
          <add name="ConnectionString" connectionString="Data
Source=EMS;Persist Security Info=True;User ID=ascol;Password=pass;Unicode=True"
providerName="System.Data.OracleClient"/>
          <add name="ConnectionString2" connectionString="Data
Source=EMS;Persist Security Info=True;User
ID=ascol;Password=pass;Unicode=True" providerName="System.Data.OracleClient"/>
          <add name="ConnectionString3" connectionString="Data
Source=EMS;Persist Security Info=True;User
ID=ascol;Password=pass;Unicode=True" providerName="System.Data.OracleClient"/>
          <add name="emsConnStr" connectionString="Data Source=ems;Persist
Security Info=True;User
ID=hem;Password=pass;Unicode=True"
providerName="System.Data.OracleClient"/>
     </connectionStrings>
     <system.web>
          <!--
           Set compilation debug="true" to insert debugging
           symbols into the compiled page. Because this
           affects performance, set this value to true
           only during development.
        -->
          <httpHandlers>
               <add path="Reserved.ReportViewerWebControl.axd" verb="*"
type="Microsoft.Reporting.WebForms.HttpHandler,
Microsoft.ReportViewer.WebForms, Version=8.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a" validate="false"/>
               <add verb="GET" path="CrystalImageHandler.aspx"
type="CrystalDecisions.Web.CrystalImageHandler,
CrystalDecisions.Web, Version=10.2.3600.0, Culture=neutral,
PublicKeyToken=692fbea5521e1304"/></httpHandlers>
          <compilation debug="true">
               <assemblies>
                    <add assembly="System.Data.OracleClient,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=B77A5C561934E089"/>
                    <add assembly="System.Windows.Forms,
```

```
Version=2.0.0.0, Culture=neutral, PublicKeyToken=B77A5C561934E089"/>
```

```
                    <add assembly="CrystalDecisions.CrystalReports.Engine,
Version=10.2.3600.0, Culture=neutral, PublicKeyToken=692fbea5521e1304"/>
                    <add assembly="CrystalDecisions.ReportSource,
Version=10.2.3600.0, Culture=neutral, PublicKeyToken=692fbea5521e1304"/>
                    <add assembly="CrystalDecisions.Shared,
Version=10.2.3600.0, Culture=neutral, PublicKeyToken=692fbea5521e1304"/>
                    <add assembly="CrystalDecisions.Web, Version=10.2.3600.0,
Culture=neutral, PublicKeyToken=692fbea5521e1304"/>
                    <add assembly="CrystalDecisions.ReportAppServer.ClientDoc,
Version=10.2.3600.0, Culture=neutral, PublicKeyToken=692fbea5521e1304"/>
                    <add assembly="CrystalDecisions.Enterprise.Framework,
Version=10.2.3600.0, Culture=neutral, PublicKeyToken=692fbea5521e1304"/>
                    <add assembly="CrystalDecisions.Enterprise.InfoStore,
Version=10.2.3600.0, Culture=neutral, PublicKeyToken=692fbea5521e1304"/>
                    <add assembly="System.Web.Extensions, Version=1.0.61025.0,
Culture=neutral, PublicKeyToken=31BF3856AD364E35"/></assemblies>
                <buildProviders>
                    <add extension=".rdlc"
type="Microsoft.Reporting.RdlBuildProvider, Microsoft.ReportViewer.Common,
Version=8.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"/>
                </buildProviders>
            </compilation>
            <!--
             The <authentication> section enables configuration
             of the security authentication mode used by
             ASP.NET to identify an incoming user.
        -->
            <authentication mode="Windows"/>
            <customErrors mode="Off">
            </customErrors>
     </system.web>
</configuration>
```