AJWT

[Advanced Java & Web Technologies]

LECTURE NOTES

Course Details

Course Name: Advanced Java & Web Technologies (AJWT)

Course Code: R32055

Course Material: http://www.startertutorials.com/ajwt/
Material like lecture notes, lab manual, lab exercises, assignments, previous question papers etc are available.

Course Prerequisites: Knowledge of programming in any programming language is required and thorough knowledge of core java concepts is a must.

Course Objectives: To make the students get acquainted with skills for creating websites and web apps through learning various technologies like HTML, CSS, JavaScript, PHP, MYSQL, XML, JavaBeans, Servlets, JSP, JDBC, AJAX and Web Services.

Course Outcomes: Students will be acquainted with necessary fundamental skills for creating websites and web apps. This course alone is **not sufficient** for creating high-end web apps like Facebook, Google etc.

Syllabus:

UNIT I: HTML tags, Lists, Tables, Images, forms, Frames, Cascading style sheets, Introduction to Java script, objects in Java Script, Dynamic HTML with Java Script.

UNIT II: PHP Programming: Introducing PHP: creating PHP script, running PHP script. Working with variables and constants: using variables, using constants, data types, operators, Controlling program flow: conditional statements, control statements, Arrays, functions, working with forms and database.

UNIT III: Working with XML: Document Type Definition, XML schemas, Document object model, XSLT, DOM and SAX.

UNIT-IV: Java Beans: Introduction to Java Beans, Advantages of Java Beans, BDK Introspection, Using Bound properties, Bean Info Interface, Constrained properties Persistence, Customizes, Java Beans API, Introduction to EJB's.

UNIT-V: Web Servers and Servlets: Tomcat web server, Introduction to Servlets: Lifecycle of a Servlet, JSDK, The Servlet API, The javax.servlet Package, Reading Servlet parameters,

and Reading Initialization parameters. The javax.servlet HTTP package, Handling Http Request & Responses, Using Cookies-Session Tracking, Security Issues,

UNIT-VI: Introduction to JSP: The Problem with Servlet. The Anatomy of a JSP Page, JSP Processing. JSP application design with MVC.
JSP Application Development: Generating Dynamic Content, Using Scripting Elements Implicit JSP Objects, Conditional Processing – Displaying Values Using an Expression to Set an Attribute, Declaring Variables and Methods Error Handling and Debugging Sharing Data Between JSP pages, Requests, and Users Passing Control and Date between Pages – Sharing Session and Application Data – Memory Usage Considerations

UNIT VII: Database Access: Database Programming using JDBC, studying javax.sql.* package, accessing a database from a JSP page, application specific database actions, deploying Java Beans in a JSP Page, Introduction to struts framework..

UNIT VIII: AJAX A New Approach: Introduction to AJAX, Integrating PHP and AJAX, Consuming WEB services in AJAX**:** (SOAP, WSDL, UDDI)

Text Books:

- Programming the World Wide Web - Robert W. Sebesta - 7th edition - Pearson
- Web Technologies - Uttam K. Roy - Oxford

Web / Other References:

- Wikipedia.org (for information on various concepts related to AJWT)
- php.net (for documentation/help on PHP language)
- w3schools.com (for code examples of various concepts related to AJWT)
- ajax.org (for help and tutorials on ajax)
- jquery.com (a javascript framework)
- tizag.com (tutorials on various languages and technologies)

Overview

The Introduction section introduces several fundamental concepts like Internet, World Wide Web (WWW), Web Browsers, Web Servers, URLs (Uniform Resource Locators), HTTP (HyperText Transfer Protocol) and Security related concepts.

Chapter 1 introduces fundamental technologies with which the web pages are developed. You will learn HTML (HyperText Markup Language) which is used to describe the content in a web page, CSS (Cascading Style Sheets) which is used to specify styling information for the content in a webpage and JavaScript which is a famous client-side scripting language for creating dynamic web pages.

Chapter 2 introduces PHP (Hypertext Pre Processor) and MYSQL, which are technologies on which majority of the web sites are based on, as they are free. PHP is a server-side scripting language and MYSQL is a DBMS (Data Base Management System) like Oracle, Microsoft SQL Server, IBM's DB2 etc.

Chapter 3 introduces XML (eXtensible Markup Language) which is used to specify topic related tags (user defined tags) for representing and transmitting information from one system to another.

Chapter 4 introduces JavaBeans which is a Java related technology for creating platform independent software components and embedding/using them in web pages or tools.

Chapter 5 introduces Servlets which is a Java related technology for server-side programming that includes database connectivity, cookies management, session management and other computational tasks.

Chapter 6 introduces JSP (Java Server Pages) which is a Java related technology for server-side programming and is a simplification over Servlets.

Chapter 7 introduces JDBC (Java Data Base Connectivity) which a Java specification for connecting to databases and transmitting data from one end to another.

Chapter 8 introduces AJAX (Asynchronous JavaScript And XML) which is used to reload a part of the webpage, and Web Services which allow different software components on multiple systems to interoperate with each other.

Introduction

Perhaps the ground breaking discovery made in the past three decades is the World Wide Web (WWW) and Internet. The WWW has changed the life of most people. All the daily mundane tasks like railway ticket reservation, movie ticket reservation, news paper reading, watching television and many other things are being done through Internet and WWW. Like every coin has two sides, as there is good, there is also bad. The advantages (good) of the Internet outweigh the disadvantages (bad).

Information sharing has become so simple that you can send information to another person on the other side of the globe within a few minutes. The information in the web is stored in web pages.

Internet

The Internet is a huge collection of computers across the world connected through a communications network. Generally each computer in the Internet will not be connected to every other computer physically. Computers in an organization will be connected to one another known as a Local Area Network (LAN) and one of the computers in the LAN will be connected to the outside world. So, Internet can also be described as a network of networks.

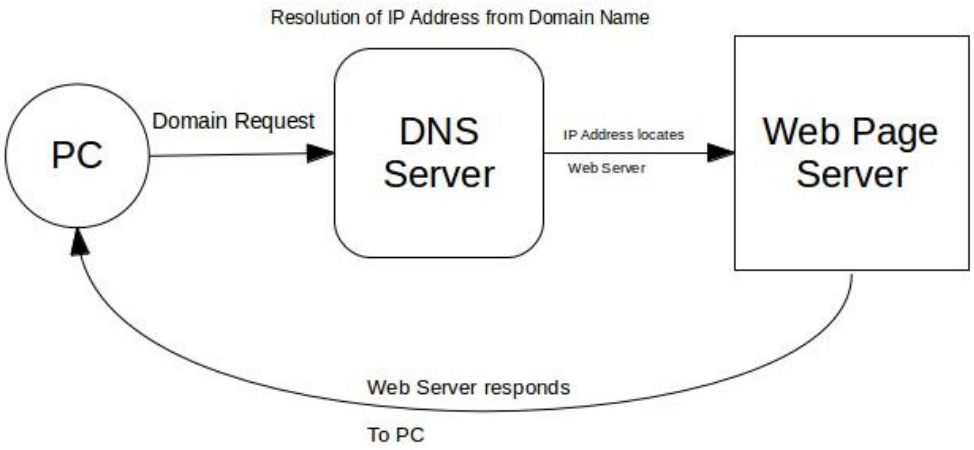
The interface between various computers in the Internet for communicating with one another is a low level protocol known as the ***TCP/IP***. In order to communicate with one another each device in the Internet should be assigned an address which is known as the ***IP address***. Widely used IP address format is the IPv4 which contains 32 bits. An IP address contains 4 octets. For example 192.168.220.15 is an IP address. As the available IP addresses are very less, a new standard known as IPv6 has been approved in 1998 which contains 128 bits.

As it is quite difficult to remember the numerical IP addresses, they are converted to names using the ***DNS (Domain Name System)***, which are easy to remember. Names and IP addresses are analogous to variable names and memory addresses in programming.

As the IP addresses are by the Internet internally, there should be some mechanism for converting a name (domain name) to its corresponding IP address. This is done by a software system known as the ***name server*** which implements the ***DNS***. Below figure illustrates the abstract representation how a request is handled in the Internet.

By mid 1980's many high-level protocols like telnet, ftp (file transfer protocol), Usenet (bulletin board), mailto (mail transfer) and others were developed. So, in order to gain all the advantages of the Internet, a user should learn all interfaces related to the protocols mentioned before.

By late 1990, a new better approach came into existence which is known as the World Wide Web (WWW) for using the Internet effectively.



World Wide Web (WWW)

In 1989 a small group of people led by Tim Berners-Lee at CERN (European Organization for Nuclear Research) proposed the idea of World Wide Web(WWW) where the scientists across the world can share scientific documents irrespective of the underlying equipment. In 1991 WWW was practically implemented and released to the public.

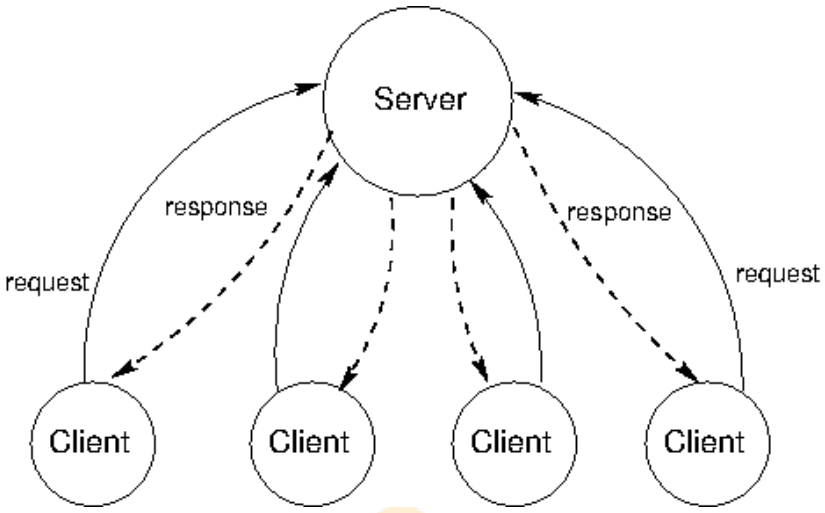
The basic units of information in the WWW are *documents* or *pages* or *resources*. The documents contain text known as *hypertext*. Hypertext is text which can link to other documents on the Web. Primarily the documents shared through Internet using WWW used to contain text only. Later, images, audio, video and other forms of content were also included. This collection of content in a document is known as *hypermedia*.

World Wide Web and Internet same or different?

It is important to know that the Internet existed even before WWW was invented. Internet is a collection of computers and other devices which allows them to communicate with each other. WWW is a collection of software and protocols available on almost all of the computers in the Internet. Internet was useful through telnet, FTP, mailto and other protocols even before the invention of WWW. WWW just made it easier to access the services through Internet.

Web Browsers

The communication model that the Web (WWW) and Internet follows is the client-server model. Below figure illustrates the client-server model:



In the client-server model, a client, generally a web browser sends requests to one or more servers. A request might be for a web page or to execute an application directly on the server.

Web browser is a software application which runs on the client machine and sends requests to the server. It is named after its functionality of browsing the web pages. A web browser supports many protocols among which the most general supported protocol is HTTP (HyperText Transfer Protocol) which allows the client and server to communicate with each other.

The first web browser with a Graphical User Interface (GUI) was Mosaic developed in 1993. Popular web browsers at present are Google Chrome, Mozilla Firefox, Internet Explorer, Opera, Safari etc.

Web Server

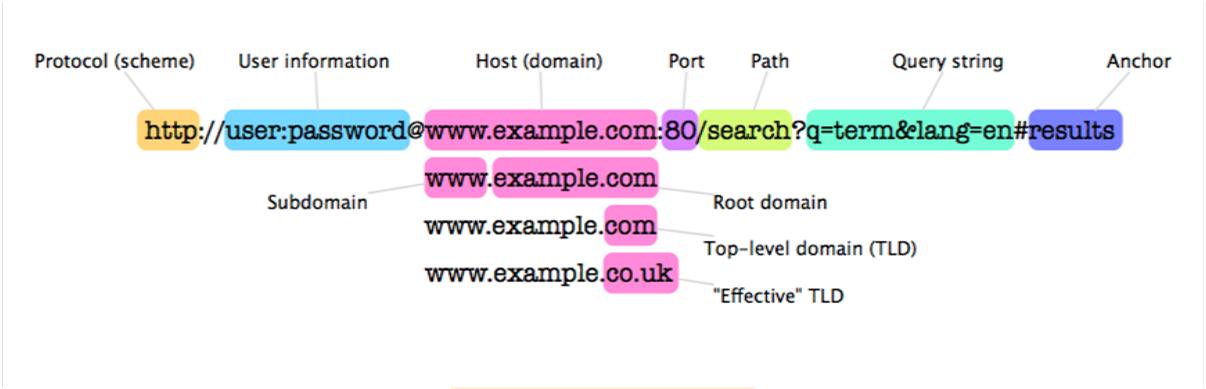
A web server is a software application which accepts requests from the clients, process them and send a response back. All the communications between a web browser and web server are carried out through HTTP. In general, a web server monitors a port on its host machine for HTTP requests from clients, performs operations and returns responses back to the clients. Famous examples of a web server are Apache web server and Microsoft's IIS (Internet Information Service).

Besides the underlying hardware and operating system, all the web servers share the same characteristics. The file structure of a web server contains two separate directories known as *document root* and *server root*. The document root contains the web documents (web pages) that will be served as responses to the client's requests and the server root contains the server and its support software.

Many servers allow multiple web sites to be maintained on a single computer which decreases the cost of each website and its maintenance. Such secondary hosts are known as *virtual hosts*. Some servers can serve documents that are in the *document root* of another server. Such servers are known as *proxy servers*.

Uniform Resource Locator (URL)

The resources provided by the web servers are identified through Uniform Resource Locators (URLs). The format of an URL is shown below:



Above URL is an example for identifying resources through HTTP protocol which is generally used to request and send XHTML (eXtensible HTML) pages. The default port for HTTP is 80. If the web server uses any other port, it should be appended at the end of the domain name.

Like http, we have *file* scheme which denotes that the resource is available on the local host or system. If we want to access a file on the local host it is sufficient to write file:///path_to_file.

URLs are of two types. First one is absolute URL, where the entire path including the domain name is specified. Second one is relative URL, where the domain name is skipped and the rest of the path is specified. For example, the domain name is *www.xyz.com* and the *document root* contains the folder *images*. The images folder contains a file named sun.jpg.

Considering that the image file (sun.jpg) is being referenced from another file in the *document root*, the absolute URL for sun.jpg will be *http://www.xyz.com/images/sun.jpg* and the relative URL will be *images/sun.jpg*.

HyperText Transfer Protocol (HTTP)

All web communications use the same protocol HTTP. Latest version of HTTP is 1.1 released in 1999. A HTTP communication consists of two phases: a request (from client to server) and a response (from server to client). In both the request and response phases, the unit of communication contains two parts: one is the *header* and the other is the *body* part.

The format for a **HTTP request** is shown below:
HTTP-request-method Resource-path HTTP-version
Header fields
Blank line
Body of the request

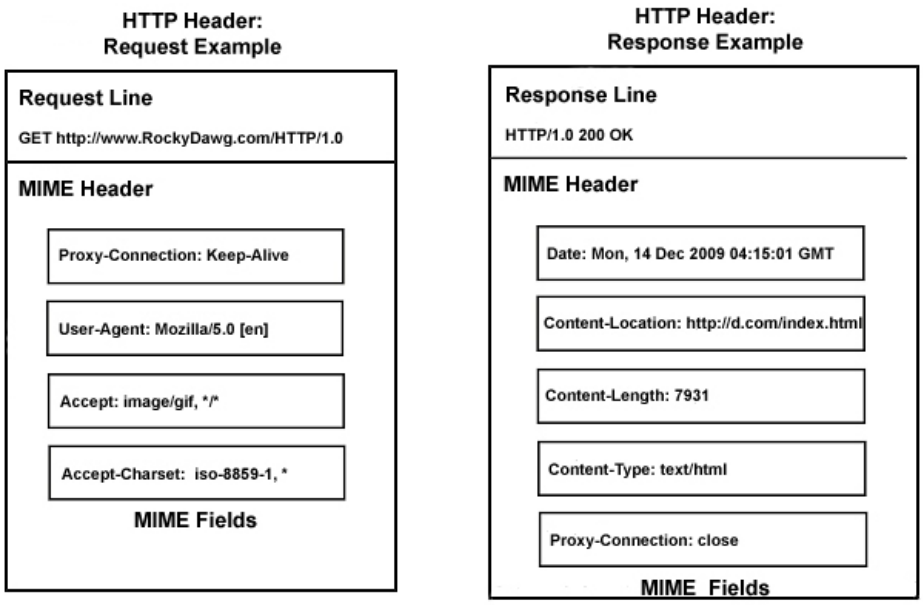
According to HTTP 1.1 there are several request methods, among which some important methods are listed below:

Method	Description
GET	Retrieves the content of the required resource
POST	Executes the specified resource, using the enclosed data
HEAD	Same as GET but only retrieves the header without the body
PUT	Replaces the specified resource with the enclosed resource
DELETE	Deletes the specified resource

Among the HTTP request methods mentioned above, GET and POST are the most frequently used methods. After the first line, the request message contains *request header*, which contains different *fields* known as the *header fields*. Each header field is a key-value pair. The format of a header field is header field name followed by a colon and a value.

Some of the frequently used header fields are: Accept (which specifies the MIME types supported by the browser), Host (which specifies the name of the host), Content-length (which specifies the no of characters in the body of the request message).

Below is an example which shows a live example of HTTP request message and HTTP response messages:



The general format of a **HTTP response** is shown below:
Status line
Header fields
Blank line
Body of the response

The status line contains information like HTTP version, Status code and Short message corresponding to the status code. Well known status codes and corresponding status messages are 200 (success), 301 (redirection), 404 (Page not found) and 500 (Internal server error).

After the first line (status line), a response message contains a set of *fields* which is known as the *response header*. Frequently used fields in the response header are: Server (which specifies information about the web server), Last-modified (which specifies the date on

which the requested resource was last modified), Content-length (which specifies the length of the content in the body of the response message) and Content-type (which specifies the MIME type of the content in the body part of the response message).

Security

Security in the web relate to protecting your sensitive data (like passwords, credit card numbers, PINs etc) from being accessed or manipulated by the people whom you think are not deemed to do so. Most of the security concerns arise due to the vulnerabilities in the Internet and related technologies. To understand what are the security issues, consider an example of a transaction where you send your username and password to login to a website. Security issues for this transaction are as follows:

Privacy - It should not be possible for a third-party to steal your data while it is being sent to a server.
Integrity - It should not be possible for a third-party to modify your data before reaching the server.
Authentication - Both sides of the communication should be able to identify each other's identity.
Non-repudiation - Both sides must be able to prove legally that the message was sent and received.

First two issues, privacy and integrity can be supported by using **encryption** which is way to convert human readable information to non-human readable. There are several algorithms widely available for encryption among which the popular ones are RSA, DES, AES and Triple DES.

There is another dimension which raises various problems regarding the security of the data. It is through malicious programs. Hackers (people with evil intensions) create malicious programs known as viruses, worms, time bombs and others. A virus is a malicious program which is transmitted through attachments of an e-mail or as a part of a software download that when executed on the victim's computer, attaches itself with the other programs and makes them unusable or deletes data available in the memory or hard disk. A worm, unlike a virus, has the ability of self propagation. Viruses and worms allows an attacker to gain control of the victim's computer which in turn are used as a part of attacks known as DoS (Denial of Service) attacks.

In Chapter 1 we will look at the three most widely used client-side languages for creating websites: HTML, CSS and JavaScript.

Chapter 1 - HTML, CSS & JavaScript

HTML

Introduction

HTML is a mark up language used to describe the content in a web document. HTML (HyperText Markup Language) was defined using SGML (Standard Generalized Markup Language) which is an ISO (International Organization for Standardization) standard for describing text-formatting languages. One must remember that HTML is not a programming language. A file with HTML code is saved with .html extension.

History

The initial version of HTML was designed by Tim Berners-Lee along with the Web in the initial 90s. The first major standardized version of HTML was 2.0 released in 1995. HTML 3.2 was published in early 1997. HTML 4.0 was released in late 1997. This version offers three variations: *Strict*, *Transitional* and *Frameset*. HTML 4.01 was published in 1999 and finally the latest HTML 5 was published in October 2014.

HTML Syntax

The basic syntactic units in any HTML document are: *tags*, *elements* and *attributes*. HTML describes information in a web document with the help of ***tags***, which provide a way for marking the text. The syntax of a tag is shown below:

< tag-name >

For example, using HTML, we can display paragraphs in a web document using the ** ** tag. Every browser will have default presentation effects for each tag. Almost all of the tags are written in pairs (a opening tag and a closing tag) as shown below:

< tag-name > content here.... </ tag-name >

In HTML, the content in a web document is described using the tags. The collection of content and the enclosing tags is known as an ***element***. For example, a paragraph element looks as shown below:

 This is a paragraph...

In general, all tags in HTML support ***attributes*** using which additional information is specified to the browser to render the content of the element. For example, an old way of aligning the text in a paragraph to the center of the document is shown below:

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

In the above example, *align* is the attribute-name and *center* is its value. General syntax for writing attributes is:

attribute-name = "value"

Comments

Almost all languages which allows users to write code supports ***comments***. Developers often use comments to write explanations about certain segments of code, to-dos, personal details and so forth. General syntax for writing comments in HTML is shown below:

```
<!-- Comment text goes here... -->
```

Basic or Fundamental HTML Tags

Every HTML document should consist of a set of four tags which are known as the basic or fundamental HTML tags. The basic tags along with their descriptions are given below:

<html> - The root tag for every HTML document
<head> - Specifies the *head* section in a web document which usually contains meta data
<body> - Specifies the *body* section which contains the actual content of a web document
<title> - Specifies the title of the web document to be displayed on the browser's title bar

In general, every web document contains a *head* section and a *body* section which will be enclosed in between <html> tags. Certain tags in HTML can be nested inside other tags. General structure of a web document is shown below:

```
<html>
        <head>
                <title>Title of the web page</title>
        </head>
        <body>
                Web page content here...
        </body>
</html>
```


Although indentation is not necessary, it makes your HTML code more readable.

General Elements in *body* Section

What does the body section of a web page contain? Although many elements can be listed, a non-exhaustive list of elements is given below:

1. Comments

2. Block elements
3. Inline elements
4. Character entities
5. Tables
6. Forms
7. Hyperlinks
8. Images
9. Scripts

Block Elements

As the name implies, the block elements are displayed like blocks of text. Browser renders the block elements as text followed with a line break and some padding automatically. Some of the well known block elements are: paragraphs (<p>), headings (<h1> to <h6>), quotes (<blockquote>), pre formatted text (<pre>), lists (<ul>, <ol> and <dl>), divisions ( ).

Inline Elements

Unlike block elements, there is no automatic line break or padding for inline elements. As the name implies, inline elements are displayed along with other normal text or other inline elements. Inline elements are always enclosed inside block elements.

Some of the well known examples of inline elements are: bold (<b>), italics (<i>), big text (<big>), small text (<small>), emphasized (), strong text (), teletype text (<tt>).

Character Entities

The character entities are used for special characters which cannot be keyed in directly from the keyboard or that may cause ambiguity for the browser interpreter (for example writing '<' might be interpreted as a tag). Such special characters can be referred in a web document by using special coded characters known as *character entities*. All the character entities follow specific syntax which is shown below:

& mnemonic-code ;

For example, to print a non breaking space in a web document, we can use the following character entity:

Scripts

As HTML does not support programming, we need other languages like JavaScript to implement computational behaviour on client-side. Such scripts (code) can be embedded in a web document using the <script> tags. A small example for embedding JavaScript in a web document is shown below:

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

</script>

The *type* attribute in the above code snippet is optional.

General Elements in *head* Section

The elements available in the head section of a web document specifies metadata and other information which might be useful for the browser or third-party agents like search engines (Google, Yahoo etc...). Some of the well known elements that are a part of the head section (<head> tags) are given below:

1. **Title element:** This element is used to display the title of the web document on the title bar of a browser. Implemented using the <title> tags.
2. **Meta elements:** As the name implies, the meta elements are used to specify metadata like author name, software used to generate the web page, redirection information, cache information, content type, description of the page and others. Meta elements are implemented using <meta> tags.
3. **Base element:** This element is used to specify the base path (URL) for all other relative URLs that will be used in the body section of the web document. Base element is implemented using the <base> tag.
4. **Script elements:** As in the body section, scripts can be specified in the head section also. Script elements are implemented using <script> tags.
5. **Style elements:** The presentation details for a web document are specified using a language know as Cascading Style Sheets (CSS). To specify such styling information in the head section, we use style elements which are implemented using <style> tags.
6. **Link elements:** To refer style information in other documents, we use the link elements. Link elements are implemented using <link> tags.

Empty Tags

The tags which does not have corresponding closing tags for opening tags i.e tags that don't have any content are known as empty tags. Empty tags are written as shown below:

<tag-name />

Some well known examples of empty tags are: <meta>, <base>, <link>,
, <img>, <hr> etc. Below example shows the usage of <link> tag below:

<link rel="stylesheet" type="text/css" href="style.css" />

Images

General content in a web document is text. Other than text, most frequently used content are images. We can display an image in a web document by using the <img> tag along with certain attributes. An example for displaying an image is shown below:

The *src* attribute specifies the location (source) of the image. The value given to the *src* attribute is an URL of the image. In the above example, the given URL is a relative URL. At the location where the web document is residing, there should be a folder named *images* and in that folder there should be an image file named *flower.jpg*.

There are other attributes which are frequently used along with *src* attribute like: *alt*, *title*, *height* and *width*.

Hyperlinks

The popularity of the Web is because of the hyperlinks. They enable a client to navigate from one web document on one server to another web document on another server. Hyperlinks can be created in a web document by using the anchor (<a>) tags. Example for creating a hyper link is shown below:

<a href="http://www.google.com">Go to Google<a>

General output for the above HTML code is, user sees the underlined text *Go to Google*. When the user hovers the mouse pointer over the text, the pointer changes from an arrow to hand symbol which denotes that it is a link to some other resource. When the user clicks the link, the browser displays the target webpage, which is Google's homepage as specified by the *href* attribute.

We can use images in place of text for linking with other web documents. Example for using images to link with other web documents is given below:

<a href="http://www.google.com"> <a>

Internal Navigation

What if the current web document has more than thousand lines of text and other content? One cannot simply navigate from top to bottom and bottom to top of the document or to other sections of the document using the scrollbar. To solve this, we can use internal navigation by using the anchor (<a>) tags. The target location within the web document is marked with *id* attribute as shown below. Let's assume that a h1 element is at the top within the document:

<h1 id="top">Main Heading</h1>

Now, somewhere in the middle or bottom of the document, we can use the anchor tag like shown below:

Go to Top

When the user clicks on the hyperlink which is at the bottom of the web document, the browser immediately moves it to the location of the h1 element without any need to scroll the document.

Lists

As the name implies, a HTML list is used to display a list of items as in the list of food items in the menu at a restaurant or a list of topics in the table of contents at the front of a text book. HTML contains three types of lists: ordered lists, unordered lists and definition lists.

Ordered Lists

An ordered list is used to display a list of items where the order is of the items is mandatory. An ordered list can be created in HTML using the <ol> and <li> tags. An example for creating a ordered list is given below:

```
<ol>
        <li>HTML</li>
        <li>CSS</li>
        <li>JavaScript</li>
</ol>
```

Unordered Lists

An unordered list is used display a list of items where the order of the items is not necessary. An unordered list can be created in HTML using the <ul> and <li> tags. An example for creating an unordered list is given below:

```
<ul>
        <li>HTML</li>
        <li>CSS</li>
        <li>JavaScript</li>
</ul>
```

Definition Lists

A definition list is used to display a list of items where each item is a combination of a term and the definition of that term. A definition list can be created using the <dl>, <dt> and <dd> tags. An example for creating a definition list is given below:


```
<dl>
        <dt>HTML</dt>
        <dd>HyperText Markup Language</dd>
        <dt>CSS</dt>
        <dd>Cascading Style Sheets</dd>
        <dt>JS</dt>
        <dd>JavaScript</dd>
</dl>
```

Nested Lists

If there is a need to display a sub list of items within a list element, we can use a nested list. Ordered and unordered lists can be nested within each other. The inner list should be placed after the <li> tag of the outer list as shown below:

```
<ul>
        <li>HTML
        <ol>
                <li>Introduction</li>
                <li>History of HTML</li>
                <li>Tables</li>
                <li>Forms</li>
        </ol>
        </li>
        <li>CSS
        <ol>
                <li>Introduction</li>
                <li>History of CSS</li>
                <li>Types of CSS</li>
                <li>CSS Selectors</li>
        </ol>
        </li>

</ul>
```

Tables

The most common way of organizing the data in a web document is through lists or tables. A table organizes data in the form of rows and columns. The ***basic tags*** for creating a table are:

- <table> - which specifies a table
- <tr> - which specifies a table row
- <td> - which specifies a table cell in a row
- <th> - same as <td> but only used to specify column headings or row headings

A basic example for creating a table is shown below:

```
<table>
        <tr>
                <th>Maths</th>
                <th>Physics</th>
                <th>Chemistry</th>
        </tr>
        <tr>
                <td>90</td>
                <td>80</td>
                <td>70</td>
```

```
        </tr>
        <tr>
                <td>80</td>
                <td>75</td>
                <td>85</td>
        </tr>
        <tr>
                <td>60</td>
                <td>70</td>
                <td>80</td>
        </tr>
</table>
```

A caption can be specified for the above table by using the <caption> tags. The caption element should be written immediately after the opening <table> tag.

rowspan and colspan Attributes

When you want to create uneven tables i.e., tables with unequal number of cells in either rows or columns of a table, we can use *rowspan* (for joining one or more rows) and *colspan* (for joining one or more columns) attributes along with <th> or <td> tags. Below example demonstrates *colspan* attribute:

```
<table>
        <tr>
                <th colspan="3">List of Subjects</th>
        </tr>
        <tr>
                <th>Maths</th>
                <th>Physics</th>
                <th>Chemistry</th>
        </tr>
        <tr>
                <td>90</td>
                <td>80</td>
                <td>70</td>
        </tr>
</table>
```


In the above example, the value 3 for *colspan* attribute specifies the number of columns to span in the row.

cellpadding and cellspacing Attributes:

The two frequently used attributes along with the <table> tag are: *cellpadding* and *cellspacing*. The *cellpadding* attribute is used to specify white space between the text in the cell and the border of the cell. The *cellspacing* attribute is used to specify white space between the border of the cell and the border of the table. An example usage of these attributes is shown below:

```
<table cellpadding="20" cellspacing="5">
	<tr>
		<th colspan="3">List of Subjects</th>
	</tr>
	<tr>
		<th>Maths</th>
		<th>Physics</th>
		<th>Chemistry</th>
	</tr>
	<tr>
		<td>90</td>
		<td>80</td>
		<td>70</td>
	</tr>
</table>
```

Table Sections

The data across the table can be divided into different sections. Sections can be created in a table using <thead>, <tbody> and <tfoot> tags. The <thead> denotes the table head section which contains the table's column headings. The <tbody> denotes the body section which contains the table data. The <tfoot> denotes the footer section which may contain data like the total of data in a cloumn. The body sections can be repeated multiple times. Some browser may display a thick line which separates multiple body sections.

Forms

A form is a mechanism supported by HTML which allows data to be sent from client to the server for processing. Modelled around paper forms, a HTML form contains several objects to gather data from the user (client) which are known as *controls* or *widgets*. Examples of form controls are textbox, textarea, checkbox, radiobutton, combobox, listbox etc.

A form can be created in a web document by using <form> tags. Two important attributes of this tag are *action* and *method*. The *action* attribute specifies the address (URL) of the resource (script file) on the server which processes the data received from the client when the user clicks on the *Submit* button. The *method* attribute specifies the HTTP request type. Possible values are GET, POST, PUT, DELETE etc., among which GET is the default value.

Inside the <form> tags, we can write <input> tags for creating most of the form controls. All <input> tags require *type* attribute, which specifies the type of form control. Another most frequently used attribute is *name*, which allows the value of the control to be accessible on the server.

HTML code for creating various form elements or controls is given below:

Textbox Control:

A textbox is a text control which accepts a single line of text from the user.

User name: <input type="text" name="txtusername" size="25" maxlength="20" />

In the above code, *size* specifies the size of the textbox and *maxlength* specifies the maximum number of characters that a user can enter in the textbox.

Password Control:

A password control is same as the textbox control except that the text entered by the user is visible only as bullets or asterisks instead of plain normal characters entered by the user.

Password: <input type="password" name="txtpassword" size="25" maxlength="20" />

Checkbox Control:

A checkbox control is rendered as a small square. A user can check or uncheck the checkbox by clicking on the square. A user can check or uncheck only a single checkbox among multiple checkboxes if they have different values for their *name* attribute.

<input type="checkbox" name="chklangtel" value="telugu" />Telugu
<input type="checkbox" name="chklangeng" value="english" checked="checked" />English
<input type="checkbox" name="chklanghin" value="hindi" />Hindi

The *value* attribute specifies the value of the checkbox that will be received on the server side when the user checks the checkbox. By default, all the checkboxes will be in a off (unchecked) state. To make a checkbox on (checked) by default, make the value of *checked* attribute as checked. In the above example, by default, the *English* checkbox will be checked. To display some text beside the checkbox (square), simply write the text you want to display after the *input* checkbox tag.

Radio button Control:

The radio button control is same as checkbox control except that user can select only one of multiple radio buttons where as multiple checkboxes can be selected in a group of checkboxes. When a user selects one of the radio buttons, the other radio buttons will be automatically turned off (unchecked).

<input type="radio" name="gender" value="m" checked="checked" />Male
<input type="radio" name="gender" value="f" />Female

Observe that the value of *name* attribute is same for both radio buttons. If the values are different, then the user will be able to check both radio buttons which is not logically correct.

Combo box Control:

When there are many choices among which a user can select one or more choices, use a combo box which is also known as a dropdown box or select box. A combo box will display a list of choices among which the user can select only one choice.

A combo box is created by using the <select> tags and the each choice/option is displayed by using the <option> tags. Example for creating a combo box is shown below:

```
<select>
        <option value="India">India</option>
        <option value="USA">USA</option>
        <option value="UK">UK</option>
        <option value="CA">Canada</option>
</select>
```

Sometimes, there will be a need to display two or more items at an instant to the user. Such control is known as a *list box* which is a variation of combo box. A list box can be created by using the *size* attribute along with the <select> tag. Set the value of *size* attribute to more than 1 to display multiple choices to the user instead of a drop down menu.

```
<select size="2">
        <option value="India">India</option>
        <option value="USA">USA</option>
        <option value="UK">UK</option>
        <option value="CA">Canada</option>
</select>
```

Finally, a user can select more than once choice in a list box by using the *multiple* attribute in the <select> tag. An example is shown below:

```
<select size="2" multiple>
        <option value="India">India</option>
        <option value="USA">USA</option>
        <option value="UK">UK</option>
        <option value="CA">Canada</option>
</select>
```

Text area Control:

Multiple lines of text can be taken as input from the user by using the *text area* control. One good example for accepting multiple lines of text as input is accepting the address of a person. A text area can be created using the <textarea> tags. Any text you want to display in the text area is written as the content of the textarea element.

```
<textarea rows="5" cols="10" name="txtaddress">Enter your address...</textarea>
```

The above code displays a small rectangular input box with 5 pixels height and 10 pixels width.

Action Buttons:

There are two action buttons in HTML forms: *submit* and *reset*. The submit button is used to submit the data entered by the user to the server-side resource mentioned as value to the *action* attribute in the <form> tag. The reset button is used to reset the values of the form controls to their defaults.

```
<input type="submit" value="Submit" />
<input type="reset" value="Clear" />
```

The *value* attribute for submit and reset buttons is used to display the required text on the button.

A normal button can be displayed by setting the *type* attribute to *button* as shown below:

```
<input type="button" value="Click Here" />
```

Frames

Generally a web browser can display only one web document at a time. If there is a need for displaying contents from multiple web documents in a single page, use frames. A frame is a HTML mechanism which has the capability of displaying a web document on its own. We can create multiple frames to display multiple web documents in a single page. For creating frames we use <frameset> and <frame> tags. Remember that a web document must contain either <body> tags or <frameset> tags, but not both.

A <frameset> tag specifies the skeleton of the webpage in which multiple frames can be displayed by using the <frame> tags. An example for <frameset> and <frame> tags is given below:

```
<frameset rows="20% , *">
        <frame src="topframe.html" />
        <frame src="bottomframe.html" />
</frameset>
```

The above code divides the entire browser display area into two rows (which is specified using the *rows* attribute of frameset tag.), first row with 20% height of the display area and the second row with remaining (80%) height.

The first frame (first row) displays contents from "topframe.html" as specified by the *src* attribute of the <frame> tag. The second frame (second row) displays contents from "bottomframe.html". Similarly use *cols* attribute along with the <frameset> tag to divide the browser display area into two or more frames.

Nested Frames

Complex structure of frames can be created by nesting <frameset> tags inside one another. Below example displays two frames in which the second frame is divided again into two sub frames:

```
<frameset cols="20%,*">
        <frame src="left.html" />
        <frameset rows="80%,*">
                <frame src="top.html" />
                <frame src="bottom.html" />
        </frameset>
</frameset>
```

Inline Frames

A web document can display content from another web document with inline frames. Generally, inline frames are used to display additional information that will help or is necessary to understand the content in the current web document. Inline frames can be created by using <iframe> tags. An example for creating an inline frame is shown below:

<iframe src="htttp://www.google.com" height="500" width="600"></iframe>

Frequently used attributes with <iframe> tag are *src* for specifying the address of the target document, *height* for specifying the height of the internal frame and *width* for specifying the width of the internal frame's display area.

Note: Although frames are useful, due to various disadvantages, frames are deprecated in HTML 5 and in XTHML 1.1. Internal frames are still supported in HTML 5. Functionality of frames can be achieved with the help of divisions ( ) and JavaScript.

eXtensible HyperText Markup Language (XHTML)

The direct descendents of SGML are HTML, XML, WML etc. XHTML is an implementation of XML (eXtensible Markup Language) in which the syntax rules are stringent. Difference between HTML and XHTML are:

- HTML is an implementation of SGML, whereas XHTML is an implementation of XML.
- HTML syntax is non-rigid, whereas XHTML's syntax is rigid.
- In HTML, tags and attribute names are case insensitive, whereas in XHTML, they are case sensitive.
- Unlike HTML, every opening tag should have corresponding closing tag in XHTML.
- Unlike HTML, all attribute values must be enclosed in double quotes.
- Unlike HTML, all attribute values must be mentioned explicitly. In HTML it is valid to write *border*, *checked*, *multiple* and *selected* attributes without any values, whereas in XHTML it is invalid.
- Following are rules to be followed for nesting elements:
    1. Text cannot be directly nested in body or form elements.
    2. Block elements cannot be nested in inline elements.
    3. If an element appears inside another element, the closing tag of the inner element must occur before the closing tag of outer element.
    4. Anchor element cannot be nested in another anchor element.
    5. List element cannot be directly nested in another list element.

  HTML doesn't enforce the above rules, whereas XHTML does.

HTML TAGS [w.r.t version 4.01]

<!DOCTYPE>

Purpose: Written as the first line in a HTML document. The DOCTYPE tag is not a HTML tag. This tag specifies the browser what type of content is present in the web page. In HTML 4.01 this tag refers to a DTD (Document Type Definition) as HTML 4.01 is based on SGML. As HTML5 is not based on SGML, the DOCTYPE tag in HTML5 does not contain any reference to a DTD. A DTD specifies a set of rules that the web page should follow.

Below are the DOCTYPE declarations for different versions of HTML:

HTML 2.0
<!DOCTYPE html PUBLIC "-//IETF//DTD HTML 2.0//EN">

HTML 3.2
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">

HTML 4.01: Strict, Transitional and Frameset
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">

XHTML 1.0: Strict, Transitional and Frameset
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">

XHTML 1.1
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

HTML5
<!DOCTYPE html>

Attributes: None

<html>

Purpose: The <html> tag is the root tag in a HTML document. It acts as a container for other HTML elements. Also this specifies where the HTML markup begins and ends. A HTML document should start with <html> tag and end with </html> tag.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"

<head>

Purpose: This tag is used to indicate the head section of the HTML document. This tag contains various other tags which specify the information related to the HTML document like: title, description, keywords etc. The information specified in the <head> tags is generally useful for search engines.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"

<body>

Purpose: This tag specifies the main content of the web page that is visible to the user. This tag should be placed after the head section.

Attributes:

Attribute Name	Description
background	Specifies the URI/URL of a background image. This attribute is now deprecated.
bgcolor	Specifies the background color. This attribute is now deprecated.
text	Specifies the text color. This attribute is now deprecated.
link	Specifies the color of unvisited hyperlink text. This attribute is now deprecated.
vlink	Specifies the color of visited hyperlink text. This attribute is now deprecated.
alink	Specifies the color of active hyperlink text (i.e. when the user clicks on it). This attribute is now deprecated.
class	Document wide identifier.
id	Document wide identifier.
lang	Language code
dir	Specifies the direction of the text
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

<title>

Purpose: Specifies the title of the HTML document. The text in the <title> tags is used by search engines and in bookmarks. The text in the <title> tags is displayed on the title bar of a browser. The <title> tags can be placed only in the head section.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"

<p>

Purpose: Used to create a paragraph in a web page.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)
align	Specifies the alignment of text inside the paragraph. Allowed values are: left, right, center or justify.

<h1>, <h2>, <h3>, <h4>, <h5>, <h6>

Purpose: Used for specifying level 1 – level 6 headings. Level 1 headings are the most important and the level 6 headings are the least important.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)
align	Specifies the alignment of text inside the paragraph. Allowed values

	are: left, right, center or justify.

<blockquote>

Purpose: Used to display quotations in a web page.

Attributes:

Attribute Name	Description
cite	Indicates the source of the quotation.
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

<pre>

Purpose: Used for displaying preformatted text. The whitespaces and line brakes are not eliminated or removed by the browser by default.

Attributes:

Attribute Name	Description
width	Specifies the maximum number of characters per line.
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

Purpose: Used to specify sections in a HTML document. This tag is typically used in CSS for styling the elements in a division. The tag is one of the container tags in HTML.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english

dir	Specifies the direction of text. Valid values are “ltr” and “rtl”
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)
align	Specifies the alignment of text inside the paragraph. Allowed values are: left, right, center or justify.

Purpose: Used for specifying an unordered list (bulleted list).

Attributes:

Attribute Name	Description
type	Specifies the style of the bullet. Possible values are: disc, square, circle.
compact	Renders the list in more compact way. This is deprecated.
lang	Specifies the language of the element’s content. Example value is “en” for english
dir	Specifies the direction of text. Valid values are “ltr” and “rtl”
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

Purpose: Used for specifying an ordered list (numbered list).

Attributes:

Attribute Name	Description
type	Specifies the style of character appearing before the list item. This attribute is deprecated.
start	Specifies the starting number for the list. This attribute is deprecated.
compact	Renders the list in more compact way. This is deprecated.
lang	Specifies the language of the element’s content. Example value is “en” for english
dir	Specifies the direction of text. Valid values are “ltr” and “rtl”
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").

style	Inline style (CSS)

<dl>

Purpose: Used for specifying a definition list.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

Purpose: This tag defines a list item. It is used in unordered lists (<ul>) and ordered lists (<ol>).

Attributes:

Attribute Name	Description
type	Specifies the type of bullet. Possible values are: A, a, I, i, 1, disc, square, circle.
value	Specifies the number of the current list item. Useful only in ordered lists.
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

<b>

Purpose: Is used for specifying bold text.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

Purpose: Is used to indicate stronger emphasis than the *em* tag. Displays the text in bold format.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

<i>

Purpose: Is used to display italicized text.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").

style	Inline style (CSS)

Purpose: Is used to give emphasis to the enclosed text.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

<u>

Purpose: Is used to display underlined text. This tag is deprecated in HTML 4.01.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

<ins>

Purpose: Is used to represent inserted text in multiple versions of the document. The text will be underlined when rendered in the browser.

Attributes:

Attribute Name	Description
cite	Indicates a source that should indicate the reason for the change.
datetime	Date and time of change.
lang	Specifies the language of the element's content. Example value is "en" for english.
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"

class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

<strike>

Purpose: Is used to display strike through text. This tag is deprecated in HTML 4.01.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

<del>

Purpose: Is used to display deleted text. On the browser text appears with a line passing through the text which is similar to <strike> tag.

Attributes:

Attribute Name	Description
cite	Indicates a source that should indicate the reason for the change.
datetime	Date and time of change.
lang	Specifies the language of the element's content. Example value is "en" for english.
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

<big>

Purpose: Is used to display big text.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

<small>

Purpose: Is used to display small text.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

<tt>

Purpose: Is used for displaying teletype (or monospaced) text.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

Purpose: Is used for grouping and applying styles on inline elements.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)
align	Specify the alignment of the text: left, right, center or justify

Purpose: Used to embed images in a webpage.

Attributes:

Attribute Name	Description
name	Assigns a name to an image. It is used to refer an image in style sheets and scripts.
longdesc	Specifies a URI/URL of a long description which will elaborate on a shorter description specified with the *alt* attribute.
src	Location of the image.
alt	Alternate text for the image.
ismap	Used in image maps.
usemap	Used in image maps.
width	Specifies the width of an image
height	Specifies the height of an image
border	Size of the image border. Zero for no border.
hspace	Amount of white space to the left and right of the image.
vspace	Amount of white space to be inserted above and below the web page.
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)
align	Specify the alignment of the text: left, right, center or justify

<meta>

Purpose: Used for declaring the meta data for the HTML document.

Attributes:

Attribute Name	Description
name	Specifies the name of the property. Valid values are description, keywords, author, include, revised, generator etc…
content	Specifies the property's value
scheme	Specifies a scheme to interpret the property's value
http-equiv	Used for http response message headers. Valid values are content-type, expires, refresh, set-cookie, pragma etc…
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"

<base>

Purpose: Is used to specify the base URL for relative links.

Attributes:

Attribute Name	Description
href	Specifies the URI/URL to use
target	Specifies the target frame/window

<script>

Purpose: Is used to embed scripts within a webpage.

Attributes:

Attribute Name	Description
name	Specifies the name of the parameter
src	Specifies the URL of the external javascript file
type	Specifies the MIME type
language	Specifies the scripting language used. This attribute is deprecated
defer	Describes that the script will not generate any content
charset	Defines the character encoding that the script uses

<style>

Purpose: Is used for embedding CSS in the head section of the web pages. Multiple style tags are allowed in the head section.

Attributes:

Attribute Name	Description
name	Defines the name of the parameter
type	Specifies the MIME type
media	Specifies the device on which the document will be displayed. Valid values are all, braille, print, projection, screen, speech etc…
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").

<link>

Purpose: Is used to define a link to an external document. Generally used to link external stylesheet with a webpage.

Attributes:

Attribute Name	Description
href	Specifies the URL of the resource document
hreflang	Language code of the destination URL
type	Specifies the MIME type
media	Specifies the device on which the document will be displayed. Valid values are all, braille, print, projection, screen, speech etc…
rel	Specifies the relationship between the current document and destination document. Valid values are alternate, appendix, bookmark, chapter, contents, copyright, glossary, help, home, index, next, prev, section, start, stylesheet, subsection
rev	Specifies the relationship between the destination and the current document. Valid values are alternate, appendix, bookmark, chapter, contents, copyright, glossary, help, home, index, next, prev, section, start, stylesheet, subsection
target	Specifies the target frame/window to load the page into. Possible values are _blank, _self, _top, _parent
charset	Defines the character encoding of the linked document
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

Purpose: Is used to specify a line break

Attributes:

Attribute Name	Description
clear	Specifies where the next line should appear. Possible values are none, left, right, all
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

<hr>

Purpose: Is used to create a horizontal rule (line).

Attributes:

Attribute Name	Description
align	Specifies the alignment. Valid values are left, right, center. This attribute is deprecated
noshade	Removes the shading effect. This attribute is deprecated.
size	Specifies the height of the rule
width	Specifies the width of the rule
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

<font>

Purpose: Is used to specify font effects for the text.

Attributes:

Attribute Name	Description
size	Specifies the font size
color	Specifies the font color
face	Specifies the name of the font to use
lang	Specifies the language of the element's content. Example value is

	“en” for english
dir	Specifies the direction of text. Valid values are “ltr” and “rtl”
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

<abbr>

Purpose: To indicate an abbreviation in a webpage.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element’s content. Example value is “en” for english
dir	Specifies the direction of text. Valid values are “ltr” and “rtl”
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

<acronym>

Purpose: Used to indicate an acronym in a webpage. This tag is deprecated.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element’s content. Example value is “en” for english
dir	Specifies the direction of text. Valid values are “ltr” and “rtl”
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

<cite>

Purpose: Used for indicating a citation

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

<dfn>

Purpose: Used to indicate a definition term.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

<kbd>

Purpose: Used to indicate the text entered by the user through keyboard.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

<samp>

Purpose: Used to indicate the sample output of a computer program or script.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

<var>

Purpose: Used to indicate a variable or parameter in a program or script.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

<sup>

Purpose: Is used to define superscript text.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a

	"tool tip").
style	Inline style (CSS)

<sub>

Purpose: It is used to define subscript text.

Attributes:

Attribute Name	Description
lang	Specifies the language of the element's content. Example value is "en" for english
dir	Specifies the direction of text. Valid values are "ltr" and "rtl"
class	Document wide identifier.
id	Document wide identifier.
title	Specifies a title to associate with the element. Many browsers will display this when the cursor hovers over the element (similar to a "tool tip").
style	Inline style (CSS)

Cascading Style Sheets (CSS)

Introduction

Although HTML or XHTML tags can specify basic presentation details by adding different attributes, they are all deprecated in favour of CSS. All the presentation details (like layout, font, colour etc...) of a web document are specified in a style sheet. With a *style sheet* we can have more control on how the content in a web document is presented to the user. Style information can be specified at three levels and this information will be cascaded to get the end presentation details. Hence the name *Cascading Style Sheets (CSS)*. A style sheet created with CSS is saved with .css extension.

History

CSS was developed by Wium Lie and Bert Bos and the first version CSS1 was released as a recommendation by W3C (World Wide Web Consortium) in 1996. Next version CSS2 was published in 1998. CSS2.1 which is a revision of CSS2 was published in 2011. After CSS2 was published, different features in CSS2 were divided into documents called *modules* and work on CSS level 3 started separately for each module and is in progress. Although there is no single monolithic CSS4 version, some modules transitioned from level 3 to level 4 while the remaining are in level 3.

CSS Syntax

A style sheet contains several rules (CSS rules). A CSS rule is made of two elements: a *selector* and one or more *declarations* enclosed in braces ({}). The syntax for a CSS rule is shown below:


```
selector
{
        declaration1;
        declaration2;
        ...
}
```

The *selector* specifies the HTML element to which the presentation effects should be applied and the *declaration* contains two parts: *property-name* and *property-value*. A HTML element can have several presentational properties and corresponding property values. An example for applying green colour to a paragraph element is shown below:

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

CSS Levels

Based on where you are writing the style information, there are three levels of CSS: *inline, document* and *external*. When a HTML element has presentation details specified across multiple levels of CSS, they will be cascaded giving highest priority to *inline* followed by *document* and finally *external* CSS.

The ***inline*** CSS allows style information to be specified for individual HTML elements. The style information is specified using the *style* attribute in the opening tag. The advantage of inline CSS is that there is finer grain of control as we can specify style information for individual elements. The disadvantages of inline CSS are: the uniform presentation effect for multiple elements is no longer sustained and it is difficult to modify the style information for multiple elements in a web document (when it is large). An example for specifying inline CSS is given below:

```
<p style="color:green">Paragraph text here...</p>
```

The ***document*** level CSS also known as ***embedded*** CSS allows us to specify style information within the document in the *head* section. Style information is specified using the <style> tags which are enclosed by <head> tags. For the <style> tag, we use the MIME type "text/css". An example for specifying document level CSS is given below:

```
<style type="text/css">
p
{
        color:green;
}
</style>
```

The advantage of document level CSS is that uniform presentation of multiple elements is sustained and the separation of style information from HTML. The disadvantage is, it is still difficult to modify style information across multiple documents.

The ***external*** CSS allows style information to be specified outside the web document. This external style sheet is linked with the web document using the <link> tag which is enclosed in <head> tags. The advantage of using external CSS is, style information is separated completely from the HTML markup. Another advantage is, single external style sheet can be linked to multiple web documents. An example for external CSS is shown below:

```
style.css (file)
p
{
        color:green;
}

page.html (web document - head section)
<link rel="stylesheet" type="text/css" href="style.css" />
```

There is another alternative way for linking the external style sheet. We can use the *import* directive for this purpose. The *import* directive has to be written inside the <style> tags in the

head section and the lines with the *import* directive should be at the beginning if they are followed by other style rules. Multiple *import* directives can be written. An example for importing external style sheets is shown below:

```
<style>
@import url(style1.css);
@import url(style2.css);
p
{
        color:green;
}
</style>
```

All the above example for external CSS assume that the web document and the external style sheet files style.css, style1.css etc., are in the same path.

Note: In favour of separating style information from HTML markup, inline CSS has been deprecated in XHTML 1.1.

Note: A semicolon is optional when there is only one declaration. A semicolon is used to separate multiple declarations from one another. Also a rule can be written in a single line if needed as shown below:

```
p{ color:green }
```

CSS Selectors

CSS provides different types of selectors to apply style details to different sets of HTML elements in a web document.

Simple Selectors

A simple selector is a name of single HTML element like p, h1, b etc. A simple selector selects all the elements having the specified name and applies the style details to those elements. Consider the following example which demonstrate simple selectors:

```
p { color : red }
```

The above CSS rule selects all the paragraph elements and sets the colour of the text to red. We can combine several simple selectors with commas and specify style rules collectively. Consider the following example:

```
h2, h3, h4 { font-size: 16px }
```

The above CSS rule selects all h2, h3 and h4 elements and applies the font size of 16 pixels to the text.

We can also apply style rules to elements which follow a certain hierarchy. In the selector, the elements of the hierarchy are separated with white spaces as shown in the below example:

```
body p b { color: grey }
```

The above CSS rule selects all the bold elements inside the paragraph elements which are in the body element. This type of selectors are known as ***contextual selectors*** or ***descendant selectors***.

Class Selectors

When there is a need to select a subset of same elements irrespective of their location in the web document, we can use class selectors. To select an element using class selector, use the *class* attribute in the opening tag of the element as shown below:

```
<p class="info">Paragraph 1 text...</p>
<p>Paragraph 2 text...</p>
<p>Paragraph 3 text...</p>
<p class="info">Paragraph 4 text...</p>
```

In the above HTML code, we can apply styles to only paragraph1 and paragraph 4 by using class selector as shown below:

```
p.info { font-style : italic }
```

The above CSS rule selects the paragraph elements having the value *info* for their *class* attribute and displays the text in italics. In a class selector, element name and class value are separated with a dot (.).

Generic Selectors

When there is a need to select a subset of different elements irrespective of their location in the web document, we can use generic selectors. Unlike class selectors, there is no need to specify any element name before the dot as shown in the below example:

```
.info { font-style : italic }
```


The above CSS rule selects all the elements in the web document having the *class* attribute set to *info* and displays the text in italics.

id Selectors

Unlike a class selector which is used to select a group of elements, an id selector is used to select a unique element. To be selected by an id selector, the element's opening tag should contain the attribute *id* and its unique value as shown in the below example:

```
<p id="imp">Paragraph text...</p>
```

No other element in the web document should have the value *imp* for the *id* attribute. Now, we can apply styles for the paragraph element as shown below:

```
#imp { font-size: 12px; color: blue }
```

The above element selects the element with the *id* attribute set to *imp* and applies blue color to its text and sets its font size to 12 pixels.

Universal Selector

CSS provides a universal selector which selects all the elements in the web document. This selector is rarely used in style sheets. Below example demonstrates the universal selector:

```
* { margin : 0; padding : 0 }
```

The above CSS rule selects all the elements in a web document and sets their default margin and padding to zero.

Pseudo Classes

Unlike normal class selectors, pseudo class selectors selects elements on the occurrence of specific event or based on the position of an element. Two pseudo classes which are based on mouse events are *hover* and *focus*. Below example demonstrates the use of these pseudo classes:

```
<input type="text" size="25" />
<input type="password" size="25" />

input:hover {color:grey}
input:focus {color:green}
```

When using a pseudo class, the element name and the class name should be separated by a colon (:). The above CSS rule applies grey color to the input fields when the user hovers the mouse pointer over them and applies green color when the user clicks on them.

Font Properties

The primary content in any web document is text. To apply different styles to the font of the text, we can use font properties.

Font Families:

The shape and space between the characters depends upon the font family. A font family can be set using the CSS font property *font-family*. It is common to specify multiple font names separated by commas as a value to this property as shown below:

```
p { font-family : arial, helvetica, verdana }
```

Browser displays the above text in the paragraphs with arial font if it is found. Otherwise it displays the text with helvetica or else with verdana font. If none of the fonts are available in the client machine, browser chooses its own default font.

We can also specify generic font family names like serif (contains *Times New Roman* and *Georgia*), sans-serif (contains *arial* and *verdana*) and monospace (contains *courier new* and *lucida console*). Always specify a generic font family as the last choice as shown below:

```
p { font-family : arial, helvetica, serif }
```

When specifying a font family name having spaces, enclose the font name in single quotes or double quotes as shown below:

```
p { font-family : arial, 'Times New Roman' }
```

Font Style:

The *font-style* property can be used to display text in italics. Valid values for font-style property are *normal*, *italic* and *oblique*. Text in italics and oblique will have the same presentation in a browser. Oblique font contains natural letters created in italics. But italic forces normal characters to be displayed in italics. Below example demonstrates the font-style property:

```
p { font-style : italic }
```

Font Size:

The size of the font can be set using the *font-size* property as shown in the below example:

```
p { font-size : 20px }
```

The size can be specified in points (pt), percentage (%), pixels (px) or in em. One em is equal to 16 pixels. Percentage and em are recommended for font-size property as they can scale up or down as per the needs of the user.

Font Weight:

We can apply bold effect to the font using *font-weight* property. Valid values for this property are: *normal*, *bold*, *bolder*, *lighter* etc. Below example demonstrates applying bold effect to the text in a paragraph.

```
p { font-weight: bold }
```

Text Properties

Effects can also be applied to text written using a certain font. Properties that can be used to assign certain effects to the text are text properties.

Text Colour:

The colour of the text can be changed using the *color* property as shown below:

```
p { color: grey }
```

We can also assign HEX values or rgb color codes as value to the *color* property. Below example shows assigning red colour using HEX code:

```
p { color: #FF0000 }
```

Text Alignment:

Text in a paragraph or in any other element can be aligned (horizontally) using the *text-align* property. Valid values to this property are *left*, *right*, *center* and *justify*. Below example shows text in a paragraph element aligned to the center:

```
p { text-align: center }
```

Text Decoration:

Text can be underlined or strike out using the *text-decoration* property. Valid values for this property are *none, overline, underline* and *line-through*. Below example demonstrates removing the underline for hyperlinks using this property:

```
a { text-decoration: none }
```

Text Transformation:

To transform all the letters in a word into lowercase or uppercase we can use *text-transform* property. Valid values to this property are: *lowercase*, *uppercase* and *capitalize*. Below example demonstrates transforming the text in bold elements to uppercase:

```
b { text-transform: uppercase }
```


Text Indentation:

Text in the first line can be indented in a paragraph or other block element using the *text-indent* property as shown in the below example:

```
p { text-indent: 40px }
```

List Properties

In CSS we can apply style effects to order lists and unordered lists. In an unordered list, we can change the item marker (bullet by default) by using the *list-style-type* as shown below:

```
ul { list-style-type: circle }
```

Valid values that can be specified to the *list-style-type* property are: *disc, circle, square*. The default value for this property is *disc*. Similarly, we can change the default natural numbers (1,2,3,...) in an ordered list using the *list-style-type* property. Valid values for this property for an ordered list are: *lower-alpha, upper-alpha, lower-roman, upper-roman, decimal* etc. The value *decimal* is the default for ordered list. Below example applies *list-style-type* to an ordered list:

```
ol { list-style-type: upper-roman }
```

We can remove the list item marker by specifying *none* as a value to the *list-style-type* property.

To display an image in the place of a list item marker, we can use *list-style-image* property. It makes sense to use this property only with unordered lists as shown in the below example:

```
ul { list-style-image: url('arrow.jpg'); }
```

The above code might not work consistently across all the browsers. A cross browser solution is given below:

```
ul { list-style-type: none; padding: 0px; margin: 0px }
ul li { background-image: url('arrow.jpg'); background-repeat: no-repeat; background-position: 0px center; padding-left: 15px }
```

Background Properties

The properties that are used to set a background image or apply other effects to the background of elements are the background properties.

Background Colour:

The background colour of an element can be changed using the *background-color* property.

Background Image:

A background image can be set for an element using the *background-image* property.

Background Image Repeat:

A small background image can be tiled along x and y-axis or along x-axis or along y-axis using the *background-repeat* property. Valid values for this property are: *repeat, repeat-x, repeat-y* and *no-repeat*. Among these values, *repeat* is the default value.

Background Image Position:

The position of the background image with respect to the element can be set using *background-position* property. Valid values for this property are: *left top, left center, left bottom, etc.*

Background Image Attachment:

To specify whether the background image scrolls along with the page or is fixed in a certain location, we use *background-attachment* property. Valid values for this property are: *scroll, fixed,* and *local*. Among these values, *scroll* is the default.

Below example demonstrates all the background properties mentioned above:

```
body
{
        background-image: url('bg.jpg');
        background-repeat: no-repeat;
        background-position: center center;
        background-attachment: fixed;
}
```

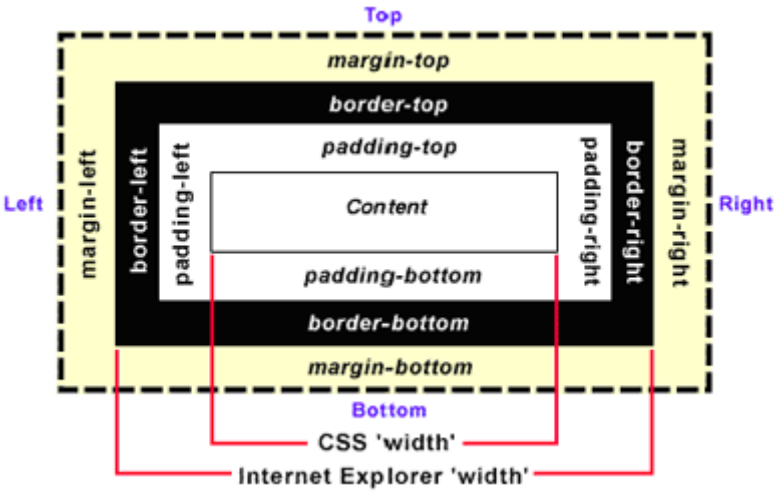
Styling Hyperlinks

Different pseudo classes can be used for styling the hyperlinks in a web document. The pseudo classes that can be used are: *link, visited, hover* and *active*. The pseudo class *link* represents all fresh hyperlinks, *visited* represents all visited hyperlinks, *hover* represents a hyperlink on which the mouse pointer is hovered and *active* represents a focused hyperlink. Below example demonstrates styling hyperlinks, the order of pseudo classes must be preserved:

```
/* unvisited link */
a:link {
   color: #FF0000;
}
/* visited link */
a:visited {
   color: #00FF00;
}
/* mouse over link */
a:hover {
   color: #FF00FF;
}
/* selected link */
a:active {
   color: #0000FF;
}
```

The Box Model

Every HTML element have an imaginary border around its content. The internal space between the content and the border of the element is known as *padding* and the external space between the border and another adjacent element is known as *margin*. This is known as the *Box Model* which is illustrated in the below figure:



Padding:

Padding of an element can be specified using the shorthand property *padding* as shown below:

p { padding: 10px; }

Above CSS rule specifies a padding of 10 pixels on all sides of the element. To specify padding only on individual sides we have the following properties: *padding-top, padding-bottom, padding-left, padding-right.* We can use the shorthand property *padding* also for specifying individual padding on all sides as shown below:

p { padding: 10px 20px 10px 20px; }

The order of padding in the above CSS rule is top, right, bottom and left. We can also combine top, bottom and left, right paddings as shown below:

p { padding: 10px 20px }

The above CSS rule specifies a padding of 10 pixels on top and bottom and 20 pixels to left and right of the content in the paragraph elements.

Margin:

Margin of an element can be specified using the shorthand property *margin* as shown below:

p { margin: 10px; }

Above CSS rule specifies a margin of 10 pixels on all sides of the element. To specify margin only on individual sides we have the following properties: *margin-top, margin-bottom, margin-left, margin-right.* We can use the shorthand property *margin* also for specifying individual margin on all sides as shown below:

p { margin: 10px 20px 10px 20px; }

The order of specifying a margin for the paragraph elements in the above CSS rule is top, right, bottom and left. We can also combine top, bottom and left, right margins as shown below:

p { margin: 10px 20px }

The above CSS rule specifies a margin of 10 pixels on top and bottom and 20 pixels to left and right of the content in the paragraph elements.

Border:

Every HTML element will be surrounded by a border. We can use *border-style* property to set the style of the border on all sides. Default value for this property is *solid*. Other valid values for this property are: *dotted, dashed, double* etc. We can also specify the border style on individual sides using the properties: *border-top-style, border-bottom-style, border-left-style* and *border-right-style*.

We can specify the width (thickness) of the border using the property *border-width*. Width of the border on individual sides can be specified by the properties: *border-top-width, border-bottom-width, border-left-width* and *border-right-width*.

We can specify the colour of the border using the property *border-color*. Colour of the border on individual sides can be specified using the properties: *border-top-color, border-bottom-color, border-left-color* and *border-right-color*.

Below example demonstrates different border properties for a table element:


```
table
{
        border-style: solid;
        border-width: 3px;
        border-color: grey;
}
```

We can also use the shorthand property *border* to achieve the above border effects:

table { border: solid 3px grey; }

Note: When there is no border, the space between two elements includes both padding and margin. The difference between these two can be seen when background image or background colour is applied. Background expands only up to the padding, not up to the margin.

JavaScript

Introduction

JavaScript primarily known as LiveScript was developed by Brendan Eich of Netscape Corporation. JavaScript is a general purpose programming language which supports multiple programming paradigms like object-orientation, imperative and functional.

Along with JavaScript examples for other client-side scripting languages are: ActionScript, VBScript, Python, Dart etc.

History:

JavaScript was developed and released to the public in 1995. Due to the wide spread adaption of JavaScript, Microsoft developed their own implementation of JavaScript known as JScript in 1996. JavaScript was standardized by ECMA (European Computer Manufacturers Association) and the first standardized version is released in 1997. ECMA's version of JavaScript is known as ECMAScript. Latest version of ECMAScript is 5.1 which was released in 2011.

Java Vs JavaScript:

Although JavaScript was named after Java, the popular programming language in 90's, it is quite different from Java. Java and JavaScript are similar regarding syntax and object-orientation concepts. Below are few popular difference between Java and JavaScript:

1. Java is a pure object-oriented language while JavaScript is not.
2. Objects in Java are static i.e., members of an object cannot be changed at runtime. Objects in JavaScript are dynamic.
3. Java is strongly typed whereas JavaScript is loosely typed.
4. Java is loaded from compiled byte code whereas JavaScript is loaded as human-readable source code.
5. Java supports block based scoping whereas JavaScript supports function based scoping.
6. Objects in Java are class based whereas objects in JavaScript are prototype based.

Uses of JavaScript:

JavaScript as a programming language has many uses. A non-exhaustive list of its uses is given below:

1. JavaScript is used for validating user input on client-side.
2. JavaScript is used for file handling and database connectivity on the server-side.
3. JavaScript can be used as an alternative to applets.
4. JavaScript is used to implement programming on client-side web documents.
5. JavaScript can be used to implement event driven programming.
6. Along with DOM, JavaScript can be used to develop DHTML (Dynamic HTML).

Including JavaScript

JavaScript is often written in another file which is saved with .js extension. But it can be also embedded with HTML code in a web document. JavaScript is embedded in a web document using the <script> tags as shown below:

```
<script type="text/javascript">
        ---JavaScript code here---
</script>
```

The <script> tags can be written in the *head* section and as well in the *body* section for multiple times. They will be interpreted by the JavaScript interpreter in the order in which they are written in the web document. The script which has to be executed once is written in the *body* section and other scripts are written in the *head* section.

Another way of specifying JavaScript is to write it in a separate file. Below example demonstrates external JavaScript:

```
script.js (JavaScript file)
document.write("Welcome to JavaScript");

hello.html (Web document)
<html>
        <head><title>First JavaScript</title></head>
        <body>
                <script type="text/javascript" src="script.js"></script>
        </body>
</html>
```

While specifying JavaScript in another file (like script.js), it should be remembered that there is a need to specify that file name as a value to the *src* attribute of <script> tag. The *type* attribute specifies the MIME type. The closing script tag </script> is mandatory even though there is no content for the script element.

JavaScript Fundamentals

Primitives

JavaScript provides five scalar primitive types: Number, Boolean, String, Undefined, Null and two compound primitive types Array and Object. Even though JavaScript supports object-orientation features, it still supports these scalar primitive data types simply because of performance reasons. Maintenance of primitive values is much faster than objects.

JavaScript also provides object versions known as *wrapper objects* for the primitive types Number, Boolean and String namely, *Number*, *Boolean* and *String*. All primitive values are stored on stack whereas objects are stored in heap.

Literals

A literal is a primitive constant value. In JavaScript all numeric values are of type Number. Internally all numerical values are stored as double-precision floating point values. This is why numeric values are often called as *numbers*.

In JavaScript integer values are a sequence of digits with no decimal points and floating point numbers contains digits and a decimal point or digits and an exponent or both. The exponent can be represented using *e* or *E*. Hexadecimal numbers can be represented by preceding a integer value with *0x* or *0X*.

A string literal is a sequence of characters enclosed in single quotes (') or double quotes ("). A null string can be denoted with a pair of single quotes (") or a pair of double quotes ("").

The only value of type Null is the reserved word *null*, which indicates no value and the only value of type Undefined is *undefined* which is not a reserved word like *null*. If a variable is not explicitly declared or is not assigned, it is said to be *null*. If a variable is explicitly declared and is not assigned, it is said to be *undefined*.

The only possible values for Boolean type are *true* and *false*.

Variables

A variable is named memory location to store data. JavaScript is a loosely typed (or dynamically typed) language i.e., there is no need to declare the data type of a variable. The type of a variable is dynamically decided at runtime based on the value assigned. A variable can be assigned any of the five primitive type values or it can refer an object.

A variable can be declared implicitly as shown below:

```
pi = 3.14;  //The data type of pi is Number
```

A variable can be declared explicitly as shown below:

```
var pi = 3.14;  //The data type of pi is Number
var name = "John";  //The data type of name is String
```


Comments

Every programming language supports comments through which a programmer can provide details like author name, creation date and purpose of the script. Apart from these details, comments allow programmers to write explanations or reviews about their script or elements in the script.

A single line comment in JavaScript starts with //. Everything followed by // in that line is treated as a comment.

A single line or multi-line comment starts with /* and ends with */. Everything in between these two markers is treated as a comment and will be ignored by the JavaScript interpreter.

Numeric Operators

As any typical programming language, JavaScript provides the numeric operators + (addition), - (subtraction), * (multiplication), / (division), % (modulo), ++ (increment) and -- (decrement).

If the increment appears before a variable, it is known as pre increment. Otherwise, if it appears after a variable, it is known as post increment. Similarly for decrement operator we have pre decrement and post decrement.

The *precedence rules* of a language specifies which operator should be evaluated when there are multiple operators in an expression which belong to different precedence levels. The *associativity rules* specify which operator should be evaluated when an expression contains multiple operations of the same precedence level. The precedence and associativity rules for numeric operators is shown below:

Operators	**Associativity**
++, --, unary -, unary +	Right-to-Left
*, /, %	Left-to-Right
+, -	Left-to-Right

String Concatenation Operator

Unlike C, strings in JavaScript are not stored or treated as arrays. Strings in JavaScript are unit scalar values. Two strings can be concatenated with each other using the + operator. For example, if a variable *first* holds the string value "Java" then the result of the following expression will be JavaScript:

first + "Script"

The typeof Operator

The *typeof* operator accepts a single operand either a value or a variable and returns its type as a string value. It returns "number", "string" and "boolean" for values or variables of the type Number, String and Boolean respectively. It returns "object" for values or variables of the type object or null. For a variable which is not assigned a value, the *typeof* operator returns "undefined".

The *typeof* operator can be used in situations where there is a need to be absolute about the type of a value of variable before proceeding further. The *typeof* operator either may include parentheses for the operand or may not. So, both *typeof variable* and *typeof(variable)* are same.

Assignment Statements

Like any other typical programming language, JavaScript supports assignment statements. An assignment statement contains the assignment operator or any one of the compound assignment operators as shown below:

x += 1; is same as:

x = x + 1;

Type Conversion

The mechanism of converting a value or variable from one type to another type is known as type conversion. There two types of type conversion: implicit type conversion and explicit type conversion.

Implicit Type Conversion:

Type conversion which is performed automatically by the JavaScript interpreter is known as implicit type conversion or *coercion*. Implicit type conversion may take place when the expected type is different from the actual type of the value given from the user or read from a file etc. For example, consider the following expression:

"John" + 123

It can be observed that in the above expression, + is a concatenation operator. So, even though the second operand is a number, it will be converted (implicit conversion) to a string and will be concatenated with "John". Now, consider the following example:

8 * "2"

The * operator is only applicable on numbers. So, the second operand will be converted from a string to a number and the resultant value will be 16.

Boolean *false* will be equal to 0 and *true* will be equal to 1 when converted to number. Both *null* and *undefined* will become *false* when converted to Boolean. A number 0 will be converted to *false* and any other number will be converted to *true*, when it is converted to a Boolean.

Explicit Type Conversion:

Sometimes there is a need for the programmer to force type conversion. Such conversion specified by the programmer is known as explicit type conversion or *type casting*. For example, a number can be casted to a string using the String constructor as shown below:

var str = String(5);

We can also cast a number to String using the toString( ) method which provides an advantage of specifying the *base* of the number. Consider the following expressions:

```
var x = 5;
var y = x.toString( );
var z = x.toString(2);  //2 specifies the base of the result. 5 in base 2 is 101
```

In the above expressions, y will have the string "5" and z will have the string "101".

A string can be casted to a number using the Number constructor as shown below:

```
var x = Number("10");
```

The above code works only when the supplied string parameter does not contain any other characters after the number. To convert a string which contains a number at any position, we can use parseInt( ) and parseFloat( ) methods.

Strings

A string is a collection of characters. Most of the times in scripts, there is a need to work with strings. JavaScript provides various properties and methods to work with String objects. Whenever a String property or method is used on a string value, it will be coerced to a String object.

One most frequently used property on String objects is *length*. The *length* property gives the number of characters in the given string. Consider the following example:

```
var str = "Hello World";
var len = str.length;
```

The value stored in the *len* variable will be 11 which the number of characters in the string.

Frequently used methods on String objects are given below:

Method	Description
charAt(pos)	Returns the character at specified position
indexOf(char)	Returns the position of the given character in the string
substring(startpos, endpos)	Returns the substring in between the specified positions
toLowerCase( )	Returns the string with all characters converted to lower case
toUpperCase( )	Returns the string with all characters converted to upper case

Math Object

The Math object provides a number of properties and methods to work with Number values. Among the methods there are *sin* and *cos* for trigonometric functions, *floor* and *round* for truncating and rounding the given numbers and *max* for returning the maximum of given numbers.

Number Object

The number object contains a collection of useful properties and methods to work with Numbers. The properties include: MAX_VALUE (represents largest number that is available), MIN_VALUE (represents smallest number that is available), NaN (represents Not a Number), POSITIVE_INFINITY (special value to represent infinity), NEGATIVE_INFINITY (special value to represent negative infinity) and PI (represents value of π.

To test whether the value in a variable is NaN or not, there is a method isNaN(var), which returns *true* if the value in the specified variable is NaN and *false* otherwise. To convert a Number to a String, use the method toString( ) as shown below:

```
var cost = 250;
var output = cost.toString( );
```

Date Object

At times you there will be a need to access the current date and time and also past and future date and times. JavaScript provides support for working with dates and time through the *Date* object.

Date objected is created using the *new* operator and one of the *Date's* constructors. Current date and time can be retrieved as shown below:

```
var today = new Date( );
```

Below are some of the frequently used methods available on a Date object:

Method	Description
toLocaleString	A string of the date information
getDate	The day of the month
getMonth	The month of the year, as a number in the range of 0 to 11
getDay	The day of a week, as a number in the range 0 to 6
getFullYear	The year
getTime	Number of milliseconds since Jan 1, 1970
getHours	Number of the hour, as a number in the range 0 to 23
getMinutes	Number of the minute, as a number in the range 0 to 59
getSeconds	Number of the second, as a number in the range 0 to 59
getMilliseconds	Number of milliseconds, as a number in the range of 0 to 999

Input and Output

JavaScript models the HTML/XHTML document as the *document* object and the browser window which displays the HTML/XHTML document as the *window* object. The *document* object contains a method named *write* to ***output*** text and data on to the document as shown below:

document.write("Welcome to JavaScript");

To display multiple lines of text using the *write* method, include the HTML tag
 as a part of the string parameter as shown below:

document.write("Welcome
to
JavaScript");

The output of the above code on the document will be as shown below:

Welcome
to
JavaScript

Although the *document* object has another method named *writeln* which appends a new line (\n), the output of both *write* and *writeln* will be same as the browser ignores the new line characters.

The *window* object is the default object in JavaScript. So, whenever a property or method on *window* object is to be referenced, there is no need to mention *window* object explicitly.

There are three methods available on the *window* object to perform input/output namely *alert* and *confirm* for output and *prompt* for input. The *alert* method is used to display a dialog box which shows the provided text or data as output along with a *OK* button. The *alert* method can be used as shown below:

alert("Sum of 10 and 20 is: " + (10 + 20));

The *alert* method allows the new line character (\n) for displaying text in multiple lines and does not support HTML tags like
 etc.

The *confirm* method displays a dialog box which displays the string provided as output along with two buttons labelled *OK* and *Cancel*. This method returns a Boolean value *true* when the user clicks *OK* button and *false* when user clicks *Cancel* button. The *confirm* method can be used as shown below:


```
var status = confirm("Do you want to proceed?");
```

The *prompt* method is used to accept string input from the user. This accepts two parameters. First parameter is a string that will be displayed on the dialog box and the second string is the default input string in case the user doesn't provide any input. The *prompt* method displays a string along with a textbox and two buttons labelled *OK* and *Cancel*. Prompt box can be created as shown below:

var a = prompt("Enter a value: ", "");

Control Statements

Statements that are used to control the flow of execution in a script are known as control statements. Control statements are used along with compound statements (a set of statements enclosed in braces). Local variables are not allowed in a *control construct* (control statement + single/compound statement). Even though a variable is declared within a control construct, it is treated as a global variable.

Control Expressions

The expressions upon which the flow of control depends are known as control expressions. These include primitive values, relational expressions and compound expressions. The result of evaluating a control expression is always a Boolean value either *true* or *false*.

A relational expression has two operands and one relational operator which may be one of the following:

Operator	Operation
= =	Equal to
! =	Not equal to
<	Less than
< =	Less than or equal to
>	Greater than
> =	Greater than or equal to
= = =	Is strictly equal to
! = =	Is strictly not equal to

The = = = and ! = = relational operators considers the type of the operands while evaluating the expression.

JavaScript also supports the logical operators && (AND), || (OR) and ! (NOT).

Operator precedence (highest to lowest) and associativity rules which are followed in expression evaluation are given below:

Operators	Associativity
++, --, unary -	Right
*, /, %	Left
+, -	Left
<, < =, >, > =	Left
= =, ! =	Left
= = =, ! = =	Left
&&	Left
||	Left
=, +=, -=, *=, /=, %=, &&=, ||=	Right

Selection Statements

Statements which are used to execute a set of statements based on a condition are known as selection statements. In JavaScript selection statements are: if, if-else and switch.

Syntax for an if statement is shown below:

```
if(expression)
{
        statement(s);
}
```

Syntax for an if-else statement is shown below:

```
if(expression)
{
        statement(s);
}
else
{
        statement(s);
}
```

Syntax of a *switch* statement is given below:

```
switch(expression)
{
        case label1:
                statement(s);
                break;
        case label2:
                statement(s);
                break;
        ...
        default:
                statement(s);
}
```


The *expression* and *labels* in a switch statement can be either numbers, strings or Boolean values.

Loop Statements

Statements which are used to execute a set of statements repeatedly based on a condition are known as loop statements or iteration statements. Loop statements supported by JavaScript are while, do-while, for and for-in.

Syntax of while loop is given below:

```
while(condition)
{
        statement(s);
}
```

Syntax for do-while loop is given below:

```
do
{
        statement(s);
}
while(condition);
```

In a do-while statement condition is evaluated after the body of the loop is executed. So, the difference between while and do-while is, unlike in while loop, the body of the loop is guaranteed to be executed at least once in a do-while loop.

Syntax of a for loop is given below:

```
for(initialization; condition; increment/decrement expression)
{
        statement(s);
}
```

JavaScript provides another loop statement to work with objects which is known as a for-in loop. It's syntax is given below:

```
for(identifier in object)
{
        statement(s);
}
```

Creation and Manipulation of Objects

An object is a real world entity that contains properties and behaviour. Properties are implemented as identifiers and behaviour is implemented using a set of methods. An object in JavaScript doesn't contain any predefined type. In JavaScript the *new* operator is used to create a blank object with no properties. A constructor is used to create and initialize properties in JavaScript.

Note: In Java *new* operator is used to create the object and its properties and a constructor is used to initialize the properties of the created object.

An object can be created as shown below:

```
var obj = new Object( );
```

The properties of an object can be accessed using the dot (.) operator. In JavaScript, the properties are not fixed as in Java. Number of properties can vary at runtime i.e., new properties can be added as and when needed. In JavaScript properties are not variables, they are just names which are used to access values and hence are never declared. Creating and accessing properties is shown below:

```
var person = new Object( );
person.name = "surya";
person.branch = "cse";
document.write(person.name);
```

Another way for creating the above object is shown below:

```
var  person = {name: "surya", branch: "cse" };
```

Nested objects can be created as shown below:

```
person.address = new Object( );
person.address.dno = "Door Number";
person.address.street = "Street Name";
```

The properties can be accessed with the dot operator or using the property name as an array subscript. For example let's consider printing the name of the person object created above:

```
document.write(person.name);
```

or

```
document.write(person["name"]);
```

To step through all the properties of an object we can use the for-in loop as shown below:

```
for(var  p  in  person)
{
        document.write("Property Name: ", p, "; Value: ", person[p], "<br />");
}
```


Arrays

An array is a collection of elements. Unlike in Java, arrays in JavaScript can contain elements of different types. A data element can be a primitive value or a reference to other objects or arrays.

An Array object can be created in two ways. First way is by using the *new* operator as shown below:

```
var array1 = new Array(10, 20, "hai");
```

```
var array2 = new Array(10);
```

In the first declaration, the size of array1 is 3 and the values are initialized to 10, 20 and "hai". In the second declaration, the size of array2 is 10 and the elements are not initialized.

Second way to create an Array object is by specifying an literal array in square brackets as shown below:

```
var array3 = [1, 2, "hai"];
```

In the above declaration, size of array3 is 3 and values are initialized to 1, 2 and "hai" respectively.

An array element can be accessed using the index or subscript which is included in square brackets. In JavaScript the index of every array starts with zero. An array element can be accessed as shown below:

```
document.write("Second array element is: ", array3[1]);
```

Arrays in JavaScript are dynamic and are always allocated memory from heap. The size of an array can be accessed or changed by using the *length* property of an Array object as shown below:

```
var array1 = new Array(10); //Length is 10
array1.length = 20; //Now length is 20
```

Array Methods

Several methods are available on an Array object to perform various manipulations on the array elements. Most frequently used methods are mentioned here:

The method *join* is used to convert the array elements into string and join them into a single string separated by the specified string. If no separator is given, all the elements are joined using commas. Below example demonstrates *join* method:


```
var names_array = ["Ken", "John", "Mary"];
var names_string = names_array.join("-");
```

Now *names_string* contains the string "Ken-John-Mary".

The method *reverse* is used to reverse the order of array elements and the method *sort* converts the array elements into strings and then sorts them in alphabetical order.

The method *concat* appends the specified elements to the end of the Array object as shown below:

```
var names = ["Ken", "John", "Mary"];
var names = names.concat("Mike", "James");
```

Now, *names* array contains, "Ken", "John", "Mary", "Mike" and "James". Length of *names* array will be 5.

The method *slice* is used to retrieve subset of elements in an Array object. This method accepts two parameters, starting index and ending index. All the elements from starting index excluding the ending index are retrieved. Below example demonstrates *slice* method:

```
var nums1 = [10, 20, 30, 40, 50];
var nums2 = nums1.slice(2,4);
```

Now *nums2* contains [30, 40]. If no ending index is specified, all the elements from starting index to last element in the array are returned.

The method *toString* is used to convert the array elements into strings and concatenate them using commas. This method behaves same as *join* method when applied on array of strings.

The methods *push* and *pop* behave as stack operations. The method *push* is used to append an element at the last position in an array and the method *pop* is used to return the last element of the array.

The methods *shift* and *unshift* are used to remove and add elements at the starting index of the array respectively.

Two-Dimensional Arrays

A two-dimensional array in JavaScript can be thought of as an array of arrays. It can be created using the *new* operator or with nested array literals as shown below:

```
var array1 = new Array(1, 2, 3);
var array2 = new Array(2, 3, 4);
var array3 = new Array(1, 3, 4);
var twod = new Array(array1, array2, array3);  //Two-dimensional array twod

var twod_array = [ [1, 2, 3], [2, 3, 4], [4, 5, 6] ]; //Two-dimensional array twod_array
```


Functions

A function is a block of code which can be executed again and again. Functions make the code modular which improves maintenance. Functions in JavaScript are similar to functions in C language. A function definition is as shown below:

```
function  functionName(param1, param2, ... , paramN)
{
        //Body of the function
        return; //This is optional
}
```

The function definition consists of function header and the body of the function (a compound statement). The function header consists of *function* keyword, function name followed by parameters (if any) enclosed in round brackets. The function body consists a set of statements and an optional *return* statement that are executed when the function is called. A function call is as shown below:

functionName(param1, param2, ... , paramN);

In JavaScript functions are objects. So, variables that reference them can be treated as object references. Example for function definition and function call is given below:

```
//Function Definition
function  add(x, y)
{
        return x+y;
}

//Function Call
addref = add;  //Here addref is a reference to the add(function) object
add(10, 20);
addref(20, 30);  //This is also valid
```

In JavaScript, a function definition must occur before a function call. So, it is wise to define a function in the *head* section of the page.

Local Variables

The scope of a variable denotes the range of statements over which the variable is visible. In JavaScript we have two kinds of variables: global variables and local variables.

Variables that are declared implicitly (without *var*) outside a function or inside a function definition have *global* scope. Variables that are declared explicitly (with *var*) outside (including compound statements) a function also have *global* scope. Variables that are declared explicitly inside a function definition have *local* scope.

When a global variable and a local variable have the same name, the local variable hides the global variable.

Parameters

The parameters available in the function call are known as *actual parameters* while the parameters in the function definition are known as *formal parameters*. When the formal parameters are more than the actual parameters, remaining formal parameters are set to *undefined*. When the formal parameters are less than the actual parameters, remaining actual parameters are truncated.

The actual parameters passed to a function can be accessed through the property array, *arguments*. So, even when the formal parameters are less than the actual parameters, all the actual parameters are directly accessible through *arguments* property.

We can use the arguments property as shown below:

arguments.length //Gives the number of actual parameters
arguments[n] //Access the n^{th} actual parameter

In JavaScript, parameters are passed by value. Even though object references are being passed, references itself are passed as a value.

Constructors

A constructor is a special function which is used to create and initialize the properties of an object. A constructor has the same name as the object. A constructor can be called by using the *new* keyword. A constructor can reference the properties of its object by using *this* keyword. A constructor can be defined as shown below:

```
function person(p_id, p_name)
{
        this.id = p_id;
        this.name = p_name;
}
```

Now, the above constructor *person* can be called by *new* keyword as shown below:

```
person1 = new person(101, "teja");
```

If the object should also contain a method, it is initialized in the same way as a property is initialized. Let's consider a method for displaying the details of the *person* object as shown below:


```
function  display_person( )
{
        document.write("Person id is: ", this.id, "<br />");
        document.write("Person name is: ", this.name);
}
```

Add a property named *display* to the constructor as shown below:

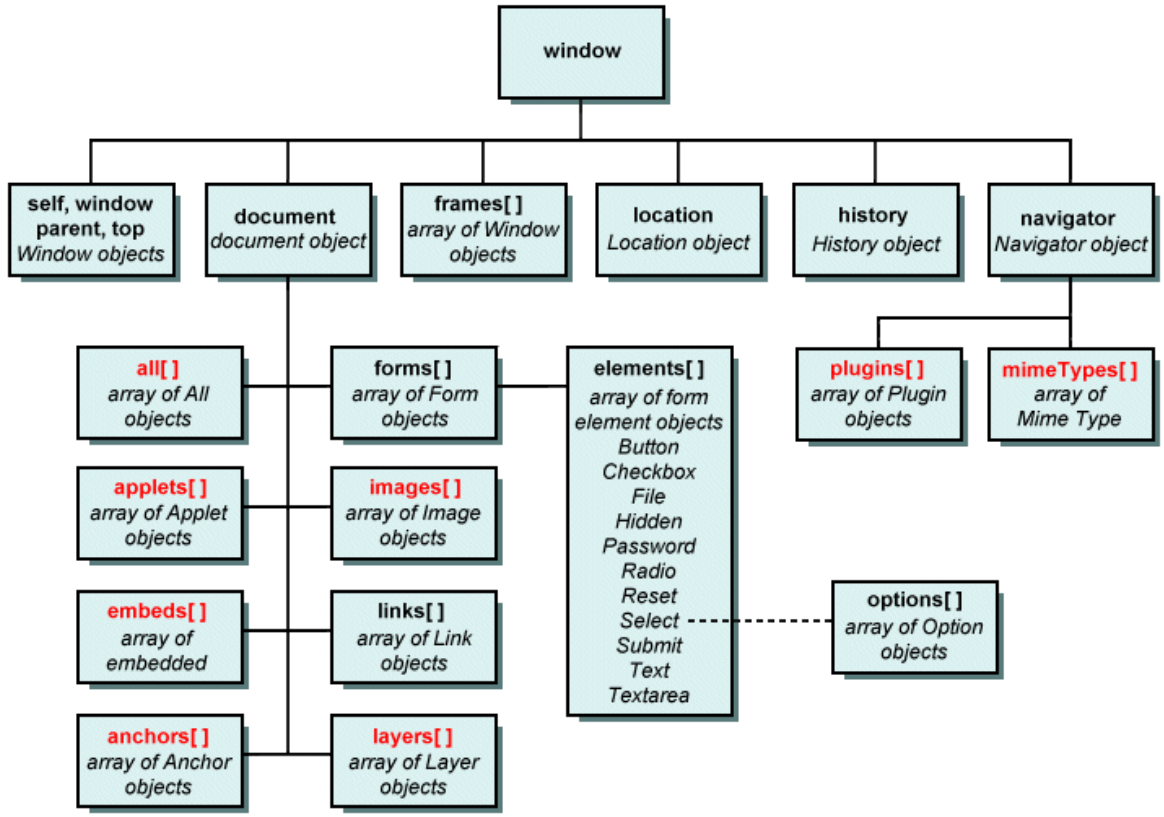
```
this.display = display_person;
```

Now, we can access the method to print the details of a *person* object as shown below:

```
person1.display( );
```

Document Object Model (DOM)

DOM is an Application Programming Interface (API) for application programs to interact with the web documents. Using DOM along with JavaScript, we can create documents, navigate through the document structure, add or delete elements and their content. Documents in the DOM have a tree like structure as shown below:



Accessing Elements in a Document

Different elements in a web document are treated as objects in JavaScript and each object has properties and methods. Using DOM, we can get the address of an HTML element in different ways.

First way is to use the *document* object's *forms* array property along with the *elements* array property. To understand this, let's consider the following HTML code:

```
<html>
        <head><title>Simple form</title></head>
        <body>
                <form action="">
                        Enter your name:
                        <input type = "text" />
                </form>
        </body>
</html>
```

To obtain the address of the textbox in the above HTML code, you can write the following code in your script:

```
var  name = document.forms[0].elements[0].value;
```

The disadvantage of this method is, it becomes difficult to obtain the address of elements when there are multiple forms in a document or if there are a large number of elements in a form or if new elements are added to a form.

The second way is by using the *name* attribute of the HTML elements. To demonstrate this, let's consider the following HTML code:

```
<html>
	<head><title>Simple form</title></head>
	<body>
		<form action="" name="frmmain">
			Enter your name:
			<input type = "text" name="txtname" />
		</form>
	</body>
</html>
```

To obtain the address of the textbox in the above HTML code, you can write the following code in your script:

```
var name = document.frmmain.txtname.value;
```

The problem with this method is, it doesn't work with a group of checkboxes or a group of radio buttons which will have the same value for their *name* attribute. Also XHTML 1.1 does not allow *name* attribute on form tag.

The third way and the most commonly used method is by using the *getElementById* method which was introduced in DOM 1. To demonstrate this let's consider the following HTML code:


```
<html>
	<head><title>Simple form</title></head>
	<body>
		<form action="">
			Enter your name:
			<input type = "text" id="txtname" />
		</form>
	</body>
</html>
```

To obtain the address of the textbox in the above HTML code, you can write the following code in your script:

```
var name = document.getElementById("txtname");
```

Since the value for *id* attribute can be different for checkboxes and radio buttons in a group, there will be no problems.

The *id* and *name* attributes can be used in combination for processing a group of checkboxes or radio buttons as shown below:

```
//HTML code
<form  id = "genderGroup">
        <input  type="radio"  name="gender"  value="male" />Male
        <input  type="radio"  name="gender"  value="female" />Female
</form>

//JavaScript code
var  count = 0;
var  dom = document.getElementById("genderGroup");
for(index = 0; index < dom.gender.length; index++)
        if(dom.gender[index].checked)
                count++;
```

Events and Event Handling

The programming which allows computations based on the activities in a browser or by the activities performed by the user on the elements in a document is known as *event-driven programming*.

An *event* is the specification (essentially an object created by the browser and JavaScript) of something significant has occurred. Examples of events are *click, submit, keyup* etc. In JavaScript all the event names are specified in lowercase. An *event handler* (essentially a function) is a set of statements (code) that handles the event.

Below table lists most commonly used events and their associated tag attributes:

Event	Tag Attribute
blur	onblur
change	onchange
click	onclick
dblclick	ondblclick
focus	onfocus
keydown	onkeydown
keypress	onkeypress
keyup	onkeyup
load	onload
mousedown	onmousedown
mouseup	onmouseup
mousemove	onmousemove
mouseover	onmouseover
mouseout	onmouseout
reset	onreset

select	onselect
submit	onsubmit
unload	onunload

Below table lists event attributes and their corresponding tags in HTML:

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

The process of linking an event handler with an event is known as *registration*. There are two ways to register an event handler in DOM 0 event model. First way is by assigning the event handler script to an event tag attribute as shown below:

```
<input  type="button"  value="Click Me"  onclick="func1( );" />
```

In the above code, the event handler *func1* is assigned to the event attribute *onclick* of a button.

The second way is by assigning the event handler to the event property of the element as shown below:

document.getElementById(ID_of_element).onclick = func1;

Form Validation

One of the best uses of client-side JavaScript is the form input validation. The input given by the user can be validated either on the client-side or on the server-side. By performing validation on the client-side using JavaScript, the advantages are less load on the server, saving network bandwidth and quicker response for the users.

Form input can be validated using different events, but the most common event used is *submit* i.e when the submit button is clicked. Corresponding attribute to be used is *onsubmit* of the *form* tag. Let's consider a simple form as shown below:

```
//HTML Code
<html>
	<head>
		<title>Login Form</title>
	</head>
	<body>
		<form  id="loginForm" action="next.html">
			Username: <input type="text" id="txtuser" /><br />
			Password: <input type="pass" id="txtpass" /><br />
			<input type="submit" value="Submit" />
			<input type="reset" value="Clear" />
		</form>
		<script type="text/javascript" src="script.js"></script>
	</body>
</html>

//JavaScript Code - script.js
function checkData( )
{
	var txtuser = document.getElementById("txtuser").value;
	var txtpass = document.getElementById("txtpass").value;
	if(txtuser == "")
	{
		alert("Username must not be blank!");
		return false;
	}
	if(txtpass == "")
	{
		alert("Password must not be blank!");
		return false;
	}
	return true;
}
```

document.getElementById("loginForm").onsubmit = checkData;

The above example can be written in another way as shown below:

```
//HTML Code
<html>
	<head>
		<title>Login Form</title>
		<script type="text/javascript" src="script.js"></script>
	</head>
	<body>
		<form  id="loginForm" action="next.html" onsubmit="checkData( );">
			Username: <input type="text" id="txtuser" /><br />
			Password: <input type="pass" id="txtpass" /><br />
			<input type="submit" value="Submit" />
			<input type="reset" value="Clear" />
		</form>
	</body>
</html>

//JavaScript Code - script.js
function checkData( )
{
	var txtuser = document.getElementById("txtuser").value;
	var txtpass = document.getElementById("txtpass").value;
	if(txtuser == "")
	{
		alert("Username must not be blank!");
		return false;
	}
	if(txtpass == "")
	{
		alert("Password must not be blank!");
		return false;
	}
	return true;
}
```


The general validation process is to check whether the user input matches the requirements. If it matches return *true* which will make the browser shift the control to the page mentioned in the *action* attribute of the *form* tag. Otherwise display appropriate error message and return *false* which will make the control stay in the current page.

Dynamic HTML

Dynamic HTML is not a new markup language. It is a collection of technologies (generally HTML, CSS and JavaScript) which changes the document content once it is loaded into a web browser.

Some examples for dynamism in a web document are given below:

- Changes to document content on user interactions or after a specified time interval or on specific browser events.
- Moving elements to new positions.
- Making elements disappear and reappear.
- Changing the colour of the background and foreground.
- Changing the font properties of elements.
- Changing the content of the elements.
- Overlapping elements in stack order.
- Allow dragging and dropping elements anywhere in the browser window.

As an example let's consider a web document which contains an image *image1* initially when the page is loaded. When the user clicks on the image, the image changes to *image2* which demonstrates dynamism and hence Dynamic HTML (DHTML).

```
//HTML Code
<html>
	<head>
		<title>DHTML Demo</title>
		<script type="text/javascript" src="script.js"></script>
	</head>
	<body>
		<img src="image1.jpg" id="image" />
	</body>
</html>

//JavaScript code - script.js
function change( )
{
	var img = document.getElementById("image");
	img.src = image2.jpg;
}
document.getElementById("image").onclick = change;
```


In Chapter 2 we will look at PHP and MYSQL

Chapter 2 - PHP & MYSQL

PHP

Introduction

PHP is a server-side scripting language. PHP processor is open source and can be downloaded without any cost. PHP is an alternative to other server-side scripting languages like CGI (Common Gateway Interface), Active Server Pages (ASP), Java Server Pages (JSP) and ColdFusion.

PHP was developed by Rasmus Lerdorf, a member of the Apache Group, in 1994. It was developed initially for the purpose of tracking the visitors to his personal web site. Most of the binaries were written in C. Lerdorf called the initial PHP as Personal Home Page. But later it was changed to Hypertext PreProcessor.

PHP code is generally embedded within a (X)HTML document. When a client requests a web document containing PHP code, the web server passes it initially to PHP processor. A web server recognizes that a document contains PHP code by the filename extension *.php*. The input to a PHP processor is a (X)HTML document along with PHP script and the output is a (X)HTML document which is sent to the client. A client has no clue of the PHP script in the web document.

PHP supports both procedural as well as object oriented programming. PHP is naturally used for form processing and for accessing databases on the server-side. PHP also supports the common mail protocols like POP3 and IMAP. It also supports distributed architectures like COM and CORBA.

Creating and Running a PHP Script

PHP scripts can be embedded in a (X)HTML document or written separately without any (X)HTML markup. In both cases, the file must be saved with the extension *.php*.

Creating PHP Script:
Any PHP script (code) must be enclosed within the PHP tags which are represented using <?php (opening tag) and ?> (closing tag). Let's consider a simple PHP script which prints "Hello World" on the web document:

```
<?php
        print "Hello World";
?>
```

Save the above file as hello.php.

Running a PHP Script:
To run a PHP script, a web server must be installed and started. Well known web servers are Apache Http Server and Microsoft's Internet Information Service (IIS).

After a web server is installed, place the PHP file *hello.php* in the web server's root directory and start the web server. Now, open a web browser like chrome, firefox or internet explorer and type the following URL in the address bar:

http://localhost/hello.php

or

http://localhost:80/hello.php

80 is the port at which the web server listens for incoming HTTP requests. The output of the PHP script is:

Hello World

General Syntactic Features

All the PHP code should be delimited with <?php and ?> tags. All variable names begin with a dollar ($) sign. Rules for variable names are as in other programming languages. PHP variable names are case-sensitive. Unlike variable names, reserved words and function names are case-insensitive. Below table lists the reserved words in PHP:

and	else	global	require	virtual
break	elseif	if	return	xor
case	extends	include	static	while
class	false	list	switch	
continue	for	new	this	
default	foreach	not	true	
do	function	or	var	

PHP allows comments to be specified in three ways. Single-line comments can be specified using # or //. A multi-line comment can be specified using /* and */. PHP statements are terminated with a semi-colon (;). Compound statements and control structures are delimited using braces.

Variables

A variable is a named location in memory to store data temporarily. PHP is dynamically typed language. So, there is no need for mentioning the data type of a variable. The type will be detected automatically based on the value assigned to the variable. A variable can be created as shown below:

$var1 = 10;

A variable which is not assigned a value contains NULL. In expressions containing numbers, NULL will be coerced to 0 and in case of strings, NULL will be coerced to an empty string. A variable can be printed as shown below:

```
print("Value of var1 is: $var1");
```

A variable can be checked if it contains a value other than NULL by using the function *IsSet*. This function returns *TRUE* if the variable contains a non-NULL value and *FALSE* if it contains a NULL value.

Constants

Constants are identifiers which are used to store values that cannot be changed once initialized. A constant doesn't begin with a dollar ($) sign. The convention for constants is, the name of a constant should always be written in uppercase. We use the function *define* to create constants. Below example demonstrates creating and using constants in PHP:

```
<?php
        define("PI", 3.142);
        print(PI);
        $area = PI*10*10;  //Let radius be 10
        print("<br />Area of circle is: $area");
?>
```

Data Types

PHP provides four scalar types namely *Boolean*, *integer*, *double* and *string* and two compound types namely *array* and *object* and two special types namely *resource* and *NULL*.

PHP has a single integer type, named *integer*. This type is same as *long* type in C. The size of an integer type is generally the size of word in the machine. In most of the machines that size will be 32 bits.

PHP's *double* type corresponds to the *double* type in C and its successors. Double literals can contain a decimal point, an exponent or both. An exponent is represented using E or e followed by a signed integer literal. Digits before and after the decimal point are optional. So, both .12 and 12. are valid double literals.

String is a collection of characters. There is no special type for characters in PHP. A character is considered as a string with length 1. String literals are represented with single quotes or double quotes. In a string literal enclosed in single quotes, escape sequences and variables are not recognized and no substitutions occurs. Such substitution is known as *interpolation*. In string literals enclosed in double quotes, escape sequence and variables are recognized and corresponding action is taken.

The only two possible values for a *Boolean* type are *TRUE* and *FALSE* both of which are case-insensitive. Integer value 0 is equal to Boolean FALSE and anything other than 0 is

equal to TRUE. An empty string and string "0" are equal to Boolean FALSE and remaining other strings are equal to TRUE. Only double value equal to Boolean FALSE is 0.0.

Operators

Operators are used in expressions to perform operations on operands. There are several operators supported by PHP which are categorized into following categories:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Increment/Decrement operators
- Logical operators
- String operators
- Array operators

Arithmetic Operators

PHP arithmetic operators are used along with numbers to perform operations like addition, subtraction, multiplication etc. Below is a list of arithmetic operators:

Operator	Name	Example	Description
+	Addition	$x + $y	Sum of x and y
-	Subtraction	$x - $y	Difference of x and y
*	Multiplication	$x * $y	Product of x and y
/	Division	$x / $y	Quotient of x divided by y
%	Modulus	$x % $y	Remainder of x divided by y
**	Exponentiation	$x ** $y	Result of x raised to the power of y

Assignment Operators

PHP assignment operators are used in assignment expressions to store the value of expression in to a variable. Below is a list of assignment operators:

Assignment	Same as	Description
x = y	x = y	Assigning value of y to x
x += y	x = x + y	Adding x and y and store the result in x
x -= y	x = x - y	Subtracting y from x and store the result in x
x *= y	x = x * y	Multiplying x and y and store the result in x
x /= y	x = x / y	Dividing x by y and store the quotient in x
x %= y	x = x % y	Dividing x by y and store the remainder in x

Increment/Decrement Operators

The increment/decrement operators are used to increment the value of variable by 1 or decrement the value of variable by 1. The increment operator is ++ and decrement operator is --.

Relational or Comparison Operators

PHP comparison operators are used to compare two values and are frequently seen in Boolean expressions. Below is a list of comparison operators:

Operator	Name	Example	Description
==	Equal	$x == $y	Returns true if x and y are equal
===	Identical	$x === $y	Returns true if x and y are equal and of same type
!=	Not equal	$x != $y	Returns true if x and y are not equal
!===	Not identical	$x !== $y	Returns true if x and y are not equal and of same type
<	Less than	$x < $y	Returns true if x is less than y
<=	Less than or equal to	$x <= $y	Returns true if x is less than or equal to y
>	Greater than	$x > $y	Returns true if x is greater than y
>=	Greater than or equal to	$x >= $y	Returns true if x is greater than or equal to y
<>	Not equal	$x <> $y	Returns true if x and y are not equal

Logical Operators

PHP logical operators are used find the Boolean value of multiple conditional expressions. Below is a list of logical operators:

Operator	Name	Example	Description
and	And	$x and $y	Returns true when both x and y are true
or	Or	$x or $y	Returns true when either x or y or both of them are true
xor	Xor	$x xor $y	Returns true when either x or y is true
&&	And	$x && $y	Returns true when both x and y are true
||	Or	$x || $y	Returns true when either x or y or both of them are true
!	Not	!$x	Returns true when x is false and vice versa

String Operators

PHP provides two operators which are used with strings only. They are listed below:

Operator	Name	Example	Description
.	Concatenation	$str1.$str2	str1and str2 are concatenated
.=	Concatenation Assignment	$str1.=$str2	str2 is appended to str1

Array Operators

Below is a list of operators which are used with arrays:

Operator	Name	Example	Description
==	Equality	$x==$y	Returns true if x and y have the same key-value pairs
===	Identity	$x===$y	Returns true if x and y have the same key-value

			pairs in same order and are of same type
!=	Inequality	$x!=$y	Returns true if x and y are not equal
!==	Non-Identity	$x!==$y	Returns true if x and y are not identical
<>	Inequality	$x<>$y	Returns true if x and y are not equal
+	Union	$x+$y	Returns union of x and y

String Operations

The only operators that can operate on strings are . and .= which are for concatenation and concatenation assignment respectively. Strings can be treated as arrays i.e., individual characters can be accessed using the index values. The index value begins with zero. A character at index *i* can be retrieved as $str[i]. An example is given below:

```
<?php
        $str1 = "master";
        print($str1[2]);
?>
```

The *print* statement in the above code prints 's'.

There are several predefined functions to manipulate strings. Some of them are listed below:

Function Name	Parameters	Description
strlen	One string	Returns number of characters in the string
strcmp	Two strings	Returns zero if both strings are equal, a -ve number if the first string occurs before second string or a +ve number if the first string occurs after the second string
strpos	Two strings	Returns position of second string in the first string or false if not found
substr	One string and one integer	Returns the substring from the specified string from the position specified as an integer. If a third integer value is specified, it represents the length of the substring to be retrieved
chop	One string	Returns the string with all white space characters removed from the end
trim	One string	Returns the string with all white space characters removed on both sides
ltrim	One string	Returns the string with all white space characters removed from the beginning
strtolower	One string	Returns the string with all the characters converted to lowercase
strtoupper	One string	Returns the string with all the characters converted to uppercase
strrev	One string	Returns the reverse of the given string
str_replace	Three strings	Returns the string in which a old substring is replaced by the new substring
str_word_count	One string	Returns the word count in the given string

Output

The output of a PHP processor is (X)HTML code. There are multiple ways to output information onto the web document. One way is to use *echo* statement with or without parentheses as shown below:

```
echo("Hello World");
```

Second way is by using *print* statement with or without parentheses as shown below:

```
print("This is PHP");
```

The difference between echo and print are: echo doesn't return anything while print returns 1. So print can be used in expressions. echo can accept multiple arguments while print accepts only one argument. Also echo is faster than print.

Third way is by using the *printf* function. The syntax of this function is given below:

```
printf(literal_string, param1, param2,....);
```

Below is an example which demonstrates echo, print and printf statements:

```
<?php
        $a = 10;
        $b = 20;
        $sum = $a + $b;
        echo("Sum of $a and $b is: $sum <br />");
        print("Sum of $a and $b is: $sum <br />");
        printf("Sum of %d and %d is: %d", $a, $b, $sum);
?>
```

Control Statements

Control statements are used to control the flow of execution in a script. There are three categories of control statements in PHP: selection statements, iteration / loop statements and jump statements.

Selection statements

The selection statements in PHP allows the PHP processor to select a set of statements based on the truth value of a condition or Boolean expression. Selection statements in PHP are if, if-else, elseif ladder and switch statement.

Syntax of *if* is given below:

```
if(condition / expression)
{
        statements(s);
```

```
}
```

Syntax of *if-else* is given below:

```
if(condition / expression)
{
        statements(s);
}
else
{
        statements(s);
}
```

Syntax of *elseif* ladder is given below:

```
if(condition / expression)
{
        statements(s);
}
elseif(condition / expression)
{
        statements(s);
}
elseif(condition / expression)
{
        statements(s);
}
else
{
        statements(s);
}
```

Syntax of *switch* statement is shown below:

```
switch(expression)
{
        case label1:
                statement(s);
                break;
        case label2:
                statement(s);
                break;
        case label3:
                statement(s);
                break;
        default:
                statement(s);
}
```

The labels for *case* statements can be either an integer, double or a string. The *default* block is optional. If the *break* statement is absent, the following cases will also execute until a *break* statement is found.

Iteration or Loop Statements

The iteration statements in PHP allows PHP processor to iterate over or repeat a set of statements for a finite or infinite times. Iteration statements supported by PHP are while, do-while, for and foreach.

Syntax of *while* loop is given below:

```
while(condition / expression)
{
        statements(s);
}
```

Syntax of *do-while* loop is given below:

```
do
{
        statement(s);
}
while(condition / expression);
```

Syntax of *for* loop is given below:

```
for(initialization; condition / expression; increment/decrement)
{
        statement(s);
}
```

The *foreach* loop is used to iterate over array elements and its syntax is given below:

```
//For normal arrays
foreach(array as variable_name)
{
        statement(s);
}

or

//For associative arrays
foreach(array as key => value)
{
        statement(s);
}
```

Jump Statements

The jump statements available in PHP are *break* and *continue*. The break statement is used to break the control from a loop, take it to the next statement after the loop and continue is used to skip the control from current line to the next iteration of the loop.

Arrays

Array is a collection of heterogeneous elements. There are two types of arrays in PHP. First type of array is a normal one that contains integer keys (indexes) which can be found in any typical programming languages. The second type of arrays is an *associative array*, where the keys are strings. Associative arrays are also known as hashes.

Array Creation

Arrays in PHP are dynamic. There is no need to specify the size of an array. A normal array can be created by using the integer index and the assignment operator as shown below:

```
$array1[0] = 10;
```

If no integer index is specified, the index will be set to 1 larger than the previous largest index value used. Consider the following example:

```
$array2[3] = 5;
$array2[ ] = 90;
```

In the above example, 90 will be stored at index location 4.

There is another way for creating an array, using the *array* construct, which is not a function. The data elements are passed to the *array* construct as shown in the below example:

```
$array3 = array(10, 15, 34, 56);
$array4 = array( );
```


In the above example *array4* is an empty array which can be used later to store elements.

A traditional array with irregular indexes can be created as shown below:

```
$array5 = array(3 => 15, 4 =>, 37, 5 => 23);
```

The above code creates an array with indexes 3, 4 and 5.

An associative array which contains named keys (indexes) can be created as shown below:

```
$ages = array("Ken" => 29, "John" => 30, "Steve" => 26, "Bob" => 28);
```

An array in PHP can be a mixture of both traditional and associative arrays.

Accessing Array Elements

Array elements can be accessed using the subscript (index or key) value which is enclosed in square brackets.

Consider a traditional array as shown below:

$array1 = array(10, 20, 30, 40);

Third element (30) in the above array can be accessed by writing $array1[2]. The index of any traditional array starts with 0.

Consider an associative array or hash as shown below:

$ages = array("Ken" => 29, "John" => 30, "Steve" => 26, "Bob" => 28);

We can access the age (30) of John by writing $ages['John'].

Iterating through Arrays

The *foreach* loop can be used to print / access the values in a traditional or an associative array.

Consider a traditional array as shown below:

$array1 = array(10, 20, 30, 40);

The *foreach* loop to print the values in the above array can be written as shown below:

```
foreach($array1 as $val)
        print("$val <br />");
```

Consider an associative array or hash as shown below:

$ages = array("Ken" => 29, "John" => 30, "Steve" => 26, "Bob" => 28);

The *foreach* loop to print the keys and values in the above array can be written as shown below:

```
foreach($ages as $name => $age)
        print("$name age is $age <br />");
```

Array Functions

The array elements can be manipulated or accessed in different ways. PHP has an extensive list of predefined functions that comes in handy while working with arrays. Some these functions are mentioned below:

Function Name	Parameters	Description
count	One array	Returns the number of elements in the array
unset	One array	Deletes the element from memory
array_keys	One array	Returns the keys (indexes) as an array
array_values	One array	Returns the values as an array
array_key_exists	Key, array	Returns *true* if the key is found, *false* otherwise
is_array	One array	Returns *true* if the argument is an array, *false* otherwise
in_array	Exp, array	Returns *true* if *exp* is in the array, *false* otherwise
explode	Delim, string	Returns an array of substrings based on the *delim*
implode	Delim, array	Returns a string joining the array elements with *delim*
sort	One array	Sorts the values in the array and renames the keys to integer values 0, 1, 2, ...
asort	One array	Sorts the values in the array preserving the keys
ksort	One array	Sorts the keys in the array preserving the values
rsort	One array	Reverse of sort (descending order)
rasort	One array	Reverse of asort (descending order)
rksort	One array	Reverse of ksort (descending order)

Functions

A function is a part of program which contains set of statements that can perform a desired task. A function can be defined with the *function* keyword followed by the function name and an optional set of parameters enclosed in parentheses. A function definition may not occur before a function call. It can be anywhere in the script. Although function definitions can be nested, it is not encouraged as they can make the script complex. Syntax for defining a function is given below:

```
function  function_name([param1, param2, ....])
{
        //Body of the function
        [return expression;]
}
```


The execution of the function terminates whenever a *return* statement is encountered or whenever the control reaches the end of the function's body.

Parameters

As in any typical programming language, parameters in a function definition are known as *formal parameters* and the parameters in a function call are known as *actual parameters*. The number of formal parameters and normal parameters may not be equal.

The default parameter passing mechanism is pass-by-value where a copy of the actual parameters are passed to actual parameters. Below example demonstrates pass-by-value:

```
function swap($x, $y)
{
        $temp = $x;
        $x = $y;
        $y = $temp;
}
$a = 10;
$b = 20;
sum($a, $b);
```

After the above code is executed the values of *a* and *b* remains 10 and 20 even after the function call.

Another way of passing parameters is *pass-by-reference* where the address of the variable is passed rather than a copy. In this mechanism changes on formal parameters are reflected on actual parameters. The address can be passed using & symbol before the variable as shown below:

```
function swap($x, $y)
{
        $temp = $x;
        $x = $y;
        $y = $temp;
}
$a = 10;
$b = 20;
sum(&$a, &$b);
```

or the above code may be also written as shown below:

```
function swap(&$x, &$y)
{
        $temp = $x;
        $x = $y;
        $y = $temp;
}
$a = 10;
$b = 20;
sum($a, $b);
```


Pass-by-reference can be used to return multiple values from a function.

Scope of Variables

Scope refers to the visibility of a variable in the PHP script. A variable defined inside a function will have local scope i.e., within the function. A variable outside the function can have the same name as a local variable in a function, where the local variable has higher precedence over the outer variable.

In some cases, a function might need access to a variable declared outside the function definition. In such cases, *global* declaration can be used before the variable name which allows the accessibility of the variable that is declared outside the function definition. Below example demonstrates local and global variables:

```
function sum_array($list)
{
        global $allsum; //Global variable
        $sum = 0; //Local variable
        foreach($list as $x)
                $sum += $x;
        $allsum += $sum;
        return $sum;
}
$allsum = 0;
$array1 = new array(10, 20, 30, 40);
$array2 = new array(1, 2, 3, 4);
$ans1 = sum_array($array1);
$ans2 = sum_array($array2);
print("Sum of array1 is: $ans1<br />");
print("Sum of all arrays is: $allsum");
```

In the above example, first print statement gives 100 and the second print statement gives 110.

Lifetime of Variables

The lifetime of a normal variable in a function is until the function execution completes. Sometimes, there might be a need to retain the value of a variable between function calls. In such cases, the variable must be declared with *static*. The lifetime of *static* variable is until the execution of script completes. Below example demonstrates the use of *static* variables:


```
function func1( )
{
        static $count = 0;
        $count++;
}
for($i = 1; $i <= 10; $i++)
        func1( );
print("func1 is called $count times.");
```

In the above example, the *print* statement prints 10 instead of 0.

Form Data Processing

One of the applications of PHP is processing the data provided by the users in (X)HTML forms. PHP provides two implicit arrays $_GET and $_POST which are global variables and are accessible anywhere in a PHP script.

The array $_GET is used when the attribute *method* of the form tag is set to *GET* and the array $_POST is used when the attribute *method* of the form tag is set to *POST*. Both arrays contain the form data stored as key-value pairs, where key contains the value of *name* attribute of the form element and value contains the value of the *value* attribute of the form element.

Below example demonstrates validation of user input in a simple (X)HTML form using PHP:

```
//HTML Code
<html>
	<head>
		<title>Login Form</title>
	</head>
	<body>
		<form  action="validate.php">
			Username: <input type="text" name="txtuser" /><br />
			Password: <input type="password" name="txtpass" /><br />
			<input type="submit" value="Submit" />
			<input type="reset" value="Clear" />
		</form>
	</body>
</html>
```

By default when the *method* attribute is not set, it will be implicitly set to GET. So, in PHP we have to use the array $_GET to retrieve the form data as shown below:

```
//PHP Code - validate.php
<?php
	$user = $_GET["txtuser"];  //txtuser is the name of textfield in the HTML form
	$pass = $_GET["txtpass"];  //txtpass is the name of password field in the HTML form
	if($user == "")
		print("Username cannot be empty!");
	elseif($pass == "")
		print("Password cannot be empty!");
	else
		print("Login success! <a href='home.html'>Click Here</a> to proceed");
?>
```

Database Access using PHP

Relational databases are the primary choice for data storage in the Web. A DBMS provides a database along with a set of tools to manage the data in the database. One example for DBMS is MYSQL. More than half of the websites in the Internet are created using PHP and the data store as MYSQL.

The steps required to access data from a table in a database provided by MYSQL can be summarized as follows:

- Establish or open a connection to the MYSQL server.
- Select a database.
- Execute the query against the database.
- Process the result returned by the server.
- Close the connection

To work with MYSQL databases, PHP provides in-built support in the form of predefined functions. To establish a connection with the MYSQL server, use the function *mysql_connect* which accepts three optional parameters. First parameter is the server name, second parameter is the MYSQL server's user name and the third parameter is the password for MYSQL server. If the connection to MYSQL fails, the function returns *false.* This function is generally used in conjunction with *die* function which can be used to print errors using the function *mysql_error* and terminate the execution of the script.

After opening a connection to the database, a database must be selected to execute the SQL queries. A database can be selected by using the function *mysql_select_db* which accepts a single string parameter, the name of the database.

After selecting a database, the next step is to specify the query which is generally stored as a string in a variable. This variable will be passed as a parameter to the function *mysql_query* which executes the query against the database.

The result of execution of the query can be stored in a variable. The result can be parsed row by row as an array by using the function *mysql_fetch_array*.

Below is a list of functions provides by PHP to work with MYSQL databases:

Function	**Description**
mysqli_connect(localhost, username, password)	Established a connection with the MYSQL server. Returns *false* if the connection fails
mysqli_select_db(link, "DB_Name")	Selects the specified database
mysqli_query(link, "SQL_Query")	Executes the specified query against the database and returns the result
mysqli_fetch_array($result)	Returns an array of the next row
mysql_num_rows($result)	Returns the number of rows in the result
mysql_num_fields($result)	Returns the number of columns in the result row
mysql_error( )	Returns a error message

Below example demonstrates user validation against the details stored in a database using PHP and MYSQL:

```
//HTML Code
<html>
	<head>
		<title>Login Form</title>
	</head>
	<body>
		<form action="getdb.php" method="get">
			<label>Username: </label>
			<input type="text" name="user" /><br />
			<label>Password: </label>
			<input type="password" name="pass" /><br />
			<input type="submit" value="Submit" />
			<input type="reset" value="Clear" />
		</form>
	</body>
</html>


//PHP Code - getdb.php
<?php
	$utext = $_REQUEST["user"];
	$ptext = $_REQUEST["pass"];
	$flag = false;

	$hostname = "localhost";
	$username = "root";
	$password = "123456";

	$con = mysqli_connect($hostname, $username, $password) or die(mysql_error());
	mysqli_select_db($con, "myapp") or die(mysql_error());
	$result = mysqli_query($con, "select * from users") or die(mysql_error());

	while($x = mysqli_fetch_array($result))
	{
		if($utext == $x["uname"] && $ptext == $x["pwd"])
			$flag = true;
	}

	if($flag)
		echo "Valid user!";
	else
		echo "Invalid username or password!";
?>
```

In *Chapter 3* we will look at XML.

Chapter 3 - XML

XML

Introduction

SGML is a meta-markup language which is a language for defining other markup languages. SGML was the basis for the development of HTML in 1990. In 1996 W3C started work on developing another meta-markup language called XML (eXtensible Markup Language).

The motivation for developing XML was due to the deficiencies in HTML. One deficiency is, HTML is a markup language for describing the general form layout of information in a web document. It doesn't describe the meaning of information in the web document. For example, if a web document displays students information from different branches, there is no way to identify all the students that belong to CSE using the tags and attributes in HTML.

Another deficiency in HTML is, it enforces few restrictions on the arrangement and order of tags in a document. For example, an opening tag can occur within the content of another element, but its corresponding closing tag can appear after the end of the element in which it is nested as shown below:

This is a paragraph

One solution to the deficiencies in HTML is to develop a markup language with its own set of tags and attributes for each set of users with a common need using SGML. As SGML is quite large and creating a parser that understands SGML is complex, this solution is not feasible.

Another solution to the problems in HTML is to develop a simplified version of SGML that allows the users to describe the data in the document. That simplified version of SGML is XML which is a meta-markup language that was developed to describe the meaning of data in a document. XML doesn't include any predefined tags. It specifies rules for creating user-defined tags.

XML is not a replacement to HTML. Both of them have different goals. HTML is a markup language for describing the form and layout of the information in a document. XML is a meta-markup language which specifies rules for creating user-defined tags (markup) and attributes for describing the meaning of the information in a document.

XML provides a simple and universal way to store any textual data. Data stored in XML documents can be distributed electronically and can be processed by any application. It is easy to write applications that can process XML documents as data in a XML document is stored in a standard way. That is why XML is a universal data interchange language.

Unlike (X)HTML, XML doesn't describe the basic presentation details of the information in a document. So, a browser renders the content of an XML document as plain text. To specify presentation details we have to use CSS or XSLT. Applications processing a XML document should analyze the document to obtain information like tags, attributes and data strings. This task of parsing and analyzing is done by XML processor.

Syntax of XML

The syntax of XML can be thought of at two distinct levels. First, there is the general low-level syntax that specifies rules to be followed by all XML documents. Second, there is the high-level syntax that is distinct for each XML document which can be specified by using either DTD (Document Type Definition) or XML Schema. They specify rules about which tags are allowed, what attributes are allowed with certain tags and in which order the tags are allowed to be written or laid out in the document.

Like (X)HTML, an XML *element* is the combination of opening tag, the content and the closing tag. An XML document can contain several kinds of statements. Most common of these are the data elements of the document. XML documents may also include markup declarations and processing instructions.

All XML documents begin with an XML declaration, which is like a processing instruction but is not one. The XML declaration identifies the document as being XML and provides the version number of XML standard and the encoding scheme being used if any as shown below:

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

Comments in XML are same as in (X)HTML as shown below:

<!-- This is a comment in XML -->

XML names are used for creating tags and attributes. An XML name must start with a letter or an underscore and can include digits, hyphens and periods (dots). XML names are case-sensitive i.e., student and Student are different in XML.

Every XML document must contain a root element, whose opening tag must be the first line of XML code. All other XML elements are nested inside the root element. XML tags are same as in (X)HTML, tag-name surrounded by angular brackets as shown below:

<tag-name>

Every XML element that can have content should have a closing tag as shown below:

</tag-name>

Elements that do not include any content can be closed as shown below:

<tag-name />

XML tags can have attributes, which can be specified as name-value assignments. All the attribute values must be enclosed in double quotes as shown below:
<tag-name attr1="value" attr2="value">content</tag-name>

An XML document that strictly follows all the above mentioned rules are said to be ***well formed***.

When designing a XML document, the designer is often faced with the choice between adding a new attribute to an element or defining a nested element. The situation in which an attribute must be always preferred over a nested element is when identifying numbers or names of elements, exactly as the *id* and *name* attributes are used in (X)HTML. Finally, attributes should be used if there is no substructure or if it is really just information about that element and nested elements should be used if the data can be structured. Below is an example of a simple *well formed* XML document:

```
<?xml version = "1.0" encoding = "utf-8" ?>
<students>
        <student id="101">
                <name>Ken</name>
                <branch>CSE</branch>
                <age>21</age>
        </student>
        <student id="201">
                <name>John</name>
                <branch>ECE</branch>
                <age>22</age>
        </student>
        <student id="301">
                <name>James</name>
                <branch>IT</branch>
                <age>21</age>
        </student>
</students>
```


XML Document Structure

An XML document often uses two supplementary files. One file specifies the syntactic rules and the other file specifies the presentation details about how the content of the document is displayed.

An XML document contains one or more entities that are logically related collections of information, ranging in size from a single character to a book chapter. One of these entities, called the *document entity*, is always physically in the file that represents the document. A document entity might contain references to entities in other documents.

Many documents include information that cannot be represented as text, such as images. Such information units are generally stored as binary data and must be specified separately. Such entities are called *binary entities*.

Entity names can be of any length. They must begin with a letter, a dash or a colon. After the first character, a name can have letters, digits, periods, dashes, underscores or colons. A reference to an entity is its name with a prepended ampersand and an appended semicolon. For example, if *sun_image* is the name of an entity, *&sun_image;* is a reference to it.

When several predefined entities must appear near each other in a XML document, their references clutter the content and make it difficult to read. In such cases, a character data section can be used. The content of a character data section is not parsed by the XML parser, so it cannot include any tags. A character data section is represented as shown below:

<![CDATA[*content*]]>

An example of using character data section is given below:

<![CDATA[The last word of the line is >>> here <<<]]>

As the content of a character data section is not parsed by the XML parser, any entity references that are included are not expanded. For example, the content of the line:

<![CDATA[The form of a tag is <tag name>]]>

is as follows:

The form of a tag is <tag name>

Document Type Definitions (DTDs)

A Document Type Definition (DTD) is a set of structural rules called *declarations*, which should be followed by the tags, attributes and entities in a XML document. A document can be tested against the DTD to determine whether it confirms to the rules that the DTD describes.

A DTD can be embedded in the XML document in which case it is called as *internal DTD* or the DTD can be specified in a separate file which can be linked to several XML documents. In such case, it is known as *external DTD*.

Syntactically, a DTD is a sequence of declarations, each of which has the form of a markup declaration as shown below:

<!keyword ... >

The *keyword* can be any one of the following four keywords:

1. ELEMENT - which defines a tag
2. ATTLIST - which defines the attributes of a tag
3. ENTITY - which defines an entity
4. NOTATION - which defines data type notations

Declaring Elements

Each element declaration in a DTD specifies the structure of one category of elements. The declaration provides the element name along with the specification of the structure of that element. An XML document can be thought of as a tree. An element is an internal node or a leaf node in the tree. The form of an element declaration for elements that contain other elements is as shown below:

<!ELEMENT element_name (list of names of child elements)>

For example, the declaration of a *student* element can be created as shown below:

<!ELEMENT student (name, regdno, branch, section)>

Multiple occurrences of the child elements can be specified using the child element specification modifiers which are given below:

Modifier	Meaning
+	One or more occurrences
*	Zero or more occurrences
?	Zero or one occurrence

For example, consider the modified declaration of the above *student* element:

<!ELEMENT student (name, regdno, branch, section?, email*)>

In the above example declaration, *section* element can occur zero or one time and *email* element can occur zero or many times.

The leaf nodes of a DTD specify the data types of the content of their parent nodes, which are elements. Generally the content of leaf node is PCDATA, for *parsable character data*. It is a string of any printable characters except < and &. Two other content types that can be specified are EMPTY and ANY. The EMPTY type specifies that the element has no content and ANY type specifies that the element might contain any content.

For example, the leaf element declaration is as shown below:

<!ELEMENT element_name (#PCDATA)>

Declaring Attributes

The attributes of an element are declared separately from the element declaration. The declaration of an attribute is as shown below:

<!ATTLIST element_name attribute_name attribute_type [default_value]>

If more than one attribute is declared for a given element, such declarations can be combined as shown below:

```
<!ATTLIST  element_name
                attribute_name_1  attribute_type  default_value_1
                attribute_name_2  attribute_type  default_value_2
                ---
                attribute_name_n  attribute_type  default_value_n
>
```

There are ten different attribute types. Among them, most frequently used type if CDATA, which specifies character data (any string characters except < and &).

The default value of an attribute can be an actual value or a requirement for the value of the attribute in the XML document. The possible default values for an attribute are given below:

Value	**Meaning**
A value	The quoted value, which is used if none is specified in an element
#FIXED value	The quoted value, which every element will have and which cannot be changed
#REQUIRED	No default value is given. Every instance of the element must specify a value
#IMPLIED	No default value. The value may or may not be specified in an element

Declaring Entities

Entities can be defined so that they can be referenced anywhere in the content of an XML document, in which case they are called *general entities*. All predefined entities are general entities. Entities can also be defined so that they can be referenced only in DTDs. Such entities are called *parameter entities*.

An entity declaration is as shown below:

```
<!ENTITY  [%]  entity_name  "entity_value">
```

The optional percentage sign (%) when present in the entity declaration denotes a parameter entity rather than a general entity.

When a document includes a large number of references to the abbreviation HyperText Markup Language, it can be defined as an entity as shown below:

```
<!ENTITY  html  "HyperText Markup Language">
```

Any XML document that includes the DTD containing the above declaration can specify the complete name with just the reference &html;

When an entity is longer than a few words, its text is defined outside the DTD. In such cases, the entity is called an *external text entity*. The declaration of an external entity is shown below:

```
<!ENTITY  entity_name  SYSTEM  "file_location">
```

The keyword SYSTEM specifies that the definition of the entity is in a different file, which is specified as the string following SYSTEM.

A Sample DTD

A Document Type Definition is saved with the extension .dtd and a normal XML file is saved with the extension .xml

Below is an example DTD which contains the specification for storing the details of students:

```
//students.dtd - DTD file
<?xml version="1.0" encoding="utf-8" ?>
<!ELEMENT students (student+)>
<!ELEMENT student (name, branch, section, regdno)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT branch (#PCDATA)>
<!ELEMENT section (#PCDATA)>
<!ELEMENT regdno (#PCDATA)>

<!ATTLIST student id CDATA #REQUIRED>
```

The XML code that conforms to the above DTD is given below:

```
//student.xml - XML file
<?xml version="1.0" encoding="utf-8"?>
<students>
        <student id="1">
                <name>K.Ramesh</name>
                <branch>CSE</branch>
                <section>A</section>
                <regdno>12PA1A0501</regdno>
        </student>
</students>
```


Internal and External DTDs

A DTD can be placed within the XML file or in a separate file. If the DTD is placed within the XML document, then it is called as internal DTD. An internal DTD is specified as shown below as the second line in the XML document:

```
<!DOCTYPE  root-element  [  ---DTD text--- ]>
```

Below is an example for internal DTD:

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

<!DOCTYPE students[
```

```
<!ELEMENT students (student+)>
<!ELEMENT student (name, branch, section, regdno)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT branch (#PCDATA)>
<!ELEMENT section (#PCDATA)>
<!ELEMENT regdno (#PCDATA)>

<!ATTLIST student id CDATA #REQUIRED>
]>

<students>
        <student id="1">
                <name>K.Ramesh</name>
                <branch>CSE</branch>
                <section>A</section>
                <regdno>12PA1A0501</regdno>
        </student>
</students>
```

If the DTD is written separately in another file, then it is called as external DTD. An external DTD is linked with an XML document as shown below:

```
<!DOCTYPE  root-element  SYSTEM  "filename.dtd">
```

Below is an example for external DTD:

```
//students.dtd - DTD file
<?xml version="1.0" encoding="utf-8" ?>
<!ELEMENT students (student+)>
<!ELEMENT student (name, branch, section, regdno)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT branch (#PCDATA)>
<!ELEMENT section (#PCDATA)>
<!ELEMENT regdno (#PCDATA)>
<!ATTLIST student id CDATA #REQUIRED>

//student.xml - XML file
<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE  students  SYSTEM  "students.dtd">
<students>
        <student id="1">
                <name>K.Ramesh</name>
                <branch>CSE</branch>
                <section>A</section>
                <regdno>12PA1A0501</regdno>
        </student>
</students>
```

An XML document which contains a DTD and is validated by a validating XML parser is known as a ***valid*** XML document.

XML Schema

Like DTD, a XML Schema also specifies the structure of the tags and attributes in a XML document. Why XML Schema when there is already DTD? There are several disadvantages of using DTD for specifying the structure of a XML document. First disadvantage is, DTDs syntax is different from that of XMLs syntax. We have to learn new syntax to work with DTDs. Second disadvantage is DTD supports less data types (ten types) for controlling the values of tags and attributes. In order to overcome these disadvantages, one of the alternatives proposed by W3C is XML Schema.

Differences between DTD and XML Schema are listed below:

- XML schemas are written in XML while DTD are derived from SGML syntax.
- XML schemas define data types for elements and attributes while DTD doesn't support data types.
- XML schemas allow support for namespaces while DTD does not.
- XML schemas define number and order of child elements, while DTD does not.
- XML schemas can be manipulated on your own with XML DOM but it is not possible in case of DTD.
- For using XML schema there is no need to learn new syntax but working with DTD requires learning new syntax.
- XML schema provides secure data communication i.e sender can describe the data in a way that receiver will understand, but in case of DTD data can be misunderstood by the receiver.
- XML schemas are extensible while DTD is not extensible.

Defining a Schema

Schemas are written using a collection of names from a namespace that is, in effect, a schema of schemas. The name of this namespace is *http://www.w3.org/2001/XMLSchema*. Some of the names available in this namespace are *element*, *schema*, *sequence*, and *string*.

Every schema has its root element as *schema*. The *schema* element specifies the namespace from which we can use the predefined names to write our schema. This namespace is specified using the attribute *xmlns*. This attribute also often specifies a prefix that should be used by the elements in the XML document. The namespace is specified as shown below:

xmlns:xsd = "http://www.w3.org/2001/XMLSchema"

A schema defines a namespace in the same sense as a DTD defines a tag set. The name of the namespace must be specified with the *targetNamespace* attribute of the *schema* element as shown below:

targetNamespace = "http://www.startertutorials.com/studentSchema"

If all the elements and attributes that are nested inside top-level elements have to be included in the target namespace, set *elementFormDefault* attribute on *schema* element to *qualified* as shown below:

elementFormDefault = "qualified"

The default namespace, which is the source of the unprefixed names in the schema, is given with another xmlns specification as shown below:

xmlns = "http://www.startertutorials.com/studentSchema"

An example for the complete *schema* opening tag is given below:

```
<xsd:schema
        xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
        targetNamespace = "http://www.startertutorials.com/studentSchema"
        xmlns = "http://www.startertutorials.com/studentSchema"
        elementFormDefault = "qualified">
```

Defining a Schema Instance

A XML document which adheres to the schema is known as a schema instance. An instance of a schema must include specifications of the namespaces it uses. These are given as attribute assignments in the tag for its root element.

First attribute, *xmlns* defines the default namespace to be the one defined in its schema as shown below:

<students xmlns = "http://www.startertutorials.com/studentSchema">

Second attribute, specifies the standard namespace for instances, which is *XMLSchema-instance* as shown below:

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

Third attribute, *schemaLocation* specifies the location of the default namespace as shown below:

xsi: schemaLocation = "http://www.startertutorials.com/studentSchema students.xsd"

An example for the complete declaration of the opening tag *students* is given below:

```
<students
        xmlns = "http://www.startertutorials.com/studentSchema"
        xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation = "http://www.startertutorials.com/studentSchema students.xsd">
```

Data Types in XML Schema

The user-defined types in XML schema can be divided into two types: *simple* and *complex*. A simple data type is one whose content is restricted to strings. A simple type cannot have attributes and nested elements. A complex type can have attributes and include other data types as elements.

XML schema defines 44 data types, 19 of which are primitive and 25 are derived types. The primitive data types include *string, Boolean, float, time* etc. The predefined derived types include *byte, long, decimal, unsignedInt, positiveInteger* etc.

User-defined types are defined by specifying restrictions on existing types. These constraints are given in terms of the *facets* of the base type. For example, the integer primitive data type has eight possible facets: *totalDigits, maxInclusive, maxExclusive, minInclusive, minExclusive, pattern, enumeration, and whitespace*.

Both simple and complex types can be *named* or *anonymous*. If anonymous, a type cannot be used outside the element in which it is declared. Data declarations in a XML Schema can be either *local* or *global*. A local declaration is one that appears inside an element that is a child of the *schema* element. A global declaration is one that appears as a child of the *schema* element. Global elements are visible in the whole schema in which they are declared.

Simple Types

An element can be created in XML Schema using the *element* tag as shown below:

```
<xsd:element  name="students"  type="xsd:string" />
```

The *name* attribute specifies the name of the element and *type* attribute specifies the data type of the element. A *default* value can be included as shown in the below example:

```
<xsd:element  name="class-strength"  type="xsd:decimal"  default="0" />
```


Elements with a constant value can be created as shown below:

```
<xsd:element  name="class-strength"  type="xsd:decimal"  fixed="60" />
```

Attributes declarations must be placed at the end. Attributes can be created only if the element is a complex type. An attribute can be created as shown below:

```
<xsd:attribute  name="id"  type="xsd:decimal"  use="required" />
```

User-defined types can be either simple types or complex types. A simple user-defined type can be created using the *simpleType* element. The *base type* for the user-defined type is specified using the *restriction* element and various *facets*. A facet is specified as an element along with *value* attribute.

An example of simple type, *firstName* of a person which can contain only 10 characters is as shown below:

```
<xsd:simpleType  name="firstName">
        <xsd:restriction  base="xsd:string">
                <xsd:maxLength  value="10" />
        </xsd:restriction>
</xsd:simpleType>
```

Another example of a simple type, *phoneNumber* which contains 10 digits only is shown below:

```
<xsd:simpleType  name="phoneNumber">
        <xsd:restriction  base="xsd:decimal">
                <xsd:precision  value="7" />
        </xsd:restriction>
</xsd:simpleType>
```

Complex Types

User-defined types which contains nested elements or attributes are known as complex types. A complex user-defined type can be created using the *complexType* tag. The elements in a complex type can be ordered or unordered. An ordered group of elements is specified by the *sequence* tag as shown below:

```
<xsd:complexType>
        <xsd:sequence>
                <xsd:element  name="name"  type="xsd:string" />
                <xsd:element  name="branch" type="xsd:string" />
                <xsd:element  name="section" type="xsd:string" />
                <xsd:element  name="age"  type="xsd:decimal" />
        </xsd:sequence>
</xsd:complexType>
```


An unordered group of elements is specified by the *all* tag and the no. of occurrences of an element can be specified by using *minOccurs* and *maxOccurs* attributes as shown below:

```
<xsd:complexType>
        <xsd:all>
                <xsd:element  name="name"  type="xsd:string" />
                <xsd:element  name="branch" type="xsd:string" />
                <xsd:element  name="section" type="xsd:string" minOccurs="1" />
                <xsd:element  name="age"  type="xsd:decimal" />
                <xsd:element  name="email" type="xsd:string"  maxOccurs="unbounded" />
        </xsd:all>
</xsd:complexType>
```

An example for valid XML document and its schema is given below:

```
//students.xsd - Schema file
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema
        xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
        targetNamespace = "http://www.startertutorials.com/studentSchema"
        xmlns = "http://www.startertutorials.com/studentSchema"
        elementFormDefault = "qualified">
<xsd:element name="students">
<xsd:complexType>
    <xsd:sequence>
        <xsd:element name="student" maxOccurs="unbounded">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="name" type="xsd:string" />
                    <xsd:element name="branch" type="xsd:string" />
                    <xsd:element name="age" type="xsd:decimal" />
                </xsd:sequence>
                <xsd:attribute name="id" type="xsd:decimal" use="required" />
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>


//student.xml - XML document
<?xml version="1.0" encoding="utf-8"?>
<students
        xmlns = "http://www.startertutorials.com/studentSchema"
        xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation = "http://www.startertutorials.com/studentSchema
        students.xsd">
    <student id="1">
        <name>K.Ramesh</name>
        <branch>CSE</branch>
        <age>21</age>
    </student>
    <student id="2">
        <name>M.Suresh</name>
        <branch>CSE</branch>
        <age>21</age>
    </student>
</students>
```

Styling XML Documents

Styling information can be specified for an XML in one of the two ways. First is by using CSS (Cascading Style Sheets) and the second is by using XSLT (eXtensible Stylesheet Language Transformations) which was developed by W3C. Although using CSS is effective, XSLT is more powerful.

Cascading Style Sheets

CSS can be specified in a separate file and linked with an XML document by using the *xml-stylesheet* processing instruction as shown below:

```
<?xml-stylesheet  type="text/css"  href="style.css" ?>
```

As an example consider the following XML document:

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type="text/css" href="style.css" ?>
<students>
        <student id="1">
                <name>K.Ramesh</name>
                <branch>CSE</branch>
                <age>21</age>
        </student>
        <student id="2">
                <name>M.Suresh</name>
                <branch>CSE</branch>
                <age>21</age>
        </student>
</students>
```

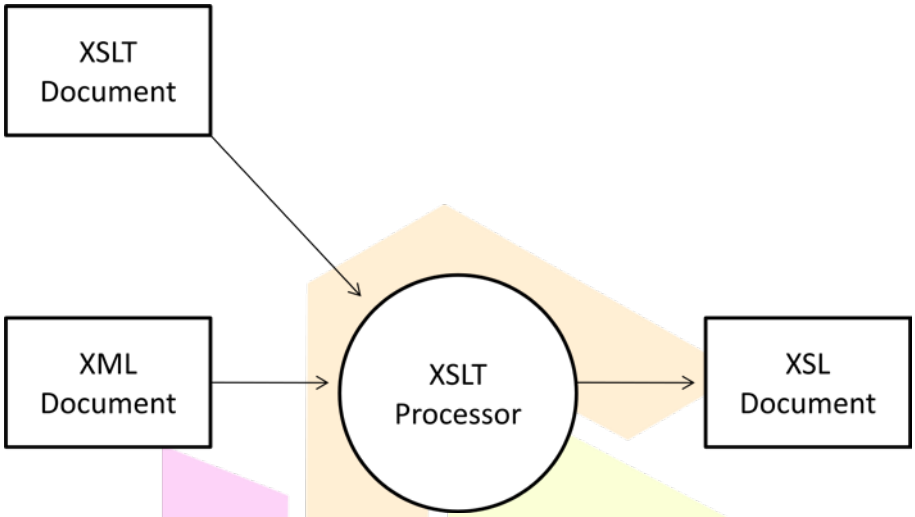
Below is the styling information specified in *style.css* file:

```
student {display:block}
name {color:magenta; font-style:italic
```


XSLT Style Sheets

The eXtensible Stylesheet Language (XSL) is a family of recommendations for defining the presentation and transformations of XML documents. It consists of three related standards. First is eXtensible Stylesheet Language Transformations (XSLT) which is used to transform XML documents into different formats including HTML, plain text or XSL-FO. Second is XML Path Language (XPath) is a language for expressions used to identify parts of XML documents. Third is XSL Formatting Objects (XSL-FO) which is used to generate high-quality printable documents in formats such as PDF and PostScript.

XSLT is a simple declarative programming language like Prolog. XSLT processor takes both an XML document and an XSLT document as input. The XSLT document is the program to be executed and the XML document is the input data to the program. Parts of the XML document are selected using XPath expressions, possibly modified and merged with parts of the XSLT document to form a new document, which is sometimes called an XSL document. The output document can be stored for future use, or it may be immediately displayed by an application (often a browser). This whole process can be illustrated as shown below:



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

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

An XSLT style sheet is an XML document whose root element is the special purpose element *stylesheet*. If the style sheet includes XHTML elements, then the *stylesheet* tag will be as shown below:

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


A XSLT style sheet document must include at least one *template* element. The template opening tag includes a *match* attribute to specify a XPath expression to select a node in the XML document. The content of a template element specifies what should be placed in the output document for the matched node.

The *apply-templates* element applies appropriate templates to the descendant nodes of the current node. In many cases, the content of an XML element should be copied to the output document. This is done by using the *value-of* element, which uses a *select* attribute to specify the element whose contents are to be copied.

For applying the same style for multiple occurrences of the same element, we can use the *for-each* element, which uses a *select* attribute to specify an element in the XML document.

Following example presents a XML document and the corresponding XSLT style sheet:

```
//student.xml - XML document
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type="text/xsl" href="student.xsl" ?>
<students>
	<student id="1">
		<name>K.Ramesh</name>
		<branch>CSE</branch>
		<age>21</age>
	</student>
	<student id="2">
		<name>M.Suresh</name>
		<branch>CSE</branch>
		<age>21</age>
	</student>
</students>
```

```
//student.xsl - XSLT Style Sheet
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
				xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
				xmlns = "http://www.w3.org/1999/xhtml">
	<xsl:template match="students">
		<html>
		<body>
		<h1>Student Details</h1>
			<table border="1" cellpadding="10">
			<tr>
				<th>Name</th><th>Branch</th><th>Age</th>
			</tr>
			<xsl:for-each select="student">
				<tr>
					<td><xsl:value-of select="name" /></td>
					<td><xsl:value-of select="branch" /></td>
					<td><xsl:value-of select="age" /></td>
				</tr>
			</xsl:for-each>
			</table>
		</body>
		</html>
	</xsl:template>
</xsl:stylesheet>
```

XML Processors

XML processors are needed for the following reasons:

- The processor must check the basic syntax of the document for well-formedness.
- The processor must replace all occurrences of an entity with its definition.
- The processor must copy the default values for attributes in a XML document.
- The processor must check for the validity of the XML document if either a DTD or XML Schema is included.

Although an XML document exhibits a regular and elegant structure, that structure does not provide applications with convenient access to document's data. This need led to the development of two standard APIs for XML processors: SAX (Simple API for XML) and DOM (Document Object Model).

SAX Approach

The SAX standard, released in May 1998, was developed by an XML user group, XML-DEV. SAX has been widely accepted as a de facto standard and is widely supported by XML processors.

The SAX approach to processing is known as event processing. The processor scans the document from beginning to end sequentially. Every time a syntactic structure like opening tag, attributes, text or a closing tag is recognized, the processor signals an event to the application by calling an event handler for the particular structure that was found. The interfaces that describe the event handlers form the SAX API.

Below is an example Java program which reads an XML document using SAX API:

```
//file.xml - XML document
<?xml version="1.0"?>
<company>
        <staff>
                <firstname>yong</firstname>
                <lastname>mook kim</lastname>
                <nickname>mkyong</nickname>
                <salary>100000</salary>
        </staff>
        <staff>
                <firstname>low</firstname>
                <lastname>yin fong</lastname>
                <nickname>fong fong</nickname>
                <salary>200000</salary>
        </staff>
</company>
```

```
//ReadXMLFile.java - Java File
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class ReadXMLFile {
  public static void main(String argv[]) {
   try {
        SAXParserFactory factory = SAXParserFactory.newInstance();
        SAXParser saxParser = factory.newSAXParser();
        DefaultHandler handler = new DefaultHandler() {
        boolean bfname = false;
        boolean blname = false;
        boolean bnname = false;
        boolean bsalary = false;
        public void startElement(String uri, String localName,String qName,
           Attributes attributes) throws SAXException {
                System.out.println("Start Element :" + qName);
                if (qName.equalsIgnoreCase("FIRSTNAME")) {
                        bfname = true;
                }
                if (qName.equalsIgnoreCase("LASTNAME")) {
                        blname = true;
                }
                if (qName.equalsIgnoreCase("NICKNAME")) {
                        bnname = true;
                }
                if (qName.equalsIgnoreCase("SALARY")) {
                        bsalary = true;
                }
        }

        public void endElement(String uri, String localName,
                String qName) throws SAXException {
                System.out.println("End Element :" + qName);
        }

        public void characters(char ch[], int start, int length) throws SAXException {
                if (bfname) {
                        System.out.println("First Name : " + new String(ch, start, length));
                        bfname = false;
                }
                if (blname) {
                        System.out.println("Last Name : " + new String(ch, start, length));
                        blname = false;
                }
                if (bnname) {
                        System.out.println("Nick Name : " + new String(ch, start, length));
```

```
                    bnname = false;
                }
                if (bsalary) {
                    System.out.println("Salary : " + new String(ch, start, length));
                    bsalary = false;
                }
            }
        };

            saxParser.parse("c:\\file.xml", handler);

        }
            catch (Exception e)
            {
          e.printStackTrace();
        }
      }
}
```

Output:

```
 Start Element :company
Start Element :staff
Start Element :firstname
First Name : yong
End Element :firstname
Start Element :lastname
Last Name : mook kim
End Element :lastname
Start Element :nickname
Nick Name : mkyong
End Element :nickname
Start Element :salary
Salary : 100000
End Element :salary
End Element :staff
Start Element :staff
Start Element :firstname
First Name : low
End Element :firstname
Start Element :lastname
Last Name : yin fong
End Element :lastname
Start Element :nickname
Nick Name : fong fong
End Element :nickname
Start Element :salary
Salary : 200000
End Element :salary
End Element :staff
End Element :company
```

DOM Approach

An alternative to SAX approach is DOM. In a XML processor, the parser builds the DOM tree for an XML document. The nodes of the tree are represented as objects that can be accessed and manipulated by the application.

The advantages of DOM over SAX are:

- DOM is better for accessing a part of an XML document more than once.
- DOM is better for rearranging (sorting) the elements in a XML document.
- DOM is best for random access over SAX's sequential access.
- DOM can detect invalid nodes later in the document without any further processing.

The disadvantages of DOM over SAX are:

- DOM structure (tree) is stored entirely in the memory, so large XML documents require more memory.
- Large documents cannot be parsed using DOM.
- DOM is slower when compared to SAX.

Below is an example Java program which reads an XML document using DOM API:

```
//file.xml - XML document
<?xml version="1.0"?>
<company>
        <staff id="1001">
                <firstname>yong</firstname>
                <lastname>mook kim</lastname>
                <nickname>mkyong</nickname>
                <salary>100000</salary>
        </staff>
        <staff id="2001">
                <firstname>low</firstname>
                <lastname>yin fong</lastname>
                <nickname>fong fong</nickname>
                <salary>200000</salary>
        </staff>
</company>
```

```
//ReadXMLFile.java - Java File
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import org.w3c.dom.Element;
import java.io.File;

public class ReadXMLFile {
  public static void main(String argv[]) {
    try {
        File fXmlFile = new File("/Users/mkyong/staff.xml");
        DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
        DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
        Document doc = dBuilder.parse(fXmlFile);
        doc.getDocumentElement().normalize();
        System.out.println("Root element :" + doc.getDocumentElement().getNodeName());
        NodeList nList = doc.getElementsByTagName("staff");
        System.out.println("----------------------------");
        for (int temp = 0; temp < nList.getLength(); temp++) {
            Node nNode = nList.item(temp);
            System.out.println("\nCurrent Element :" + nNode.getNodeName());
            if (nNode.getNodeType() == Node.ELEMENT_NODE) {
                Element eElement = (Element) nNode;
                System.out.println("Staff id : " + eElement.getAttribute("id"));
                System.out.println("First Name : " +
eElement.getElementsByTagName("firstname").item(0).getTextContent());
                System.out.println("Last Name : " +
eElement.getElementsByTagName("lastname").item(0).getTextContent());
                System.out.println("Nick Name : " +
eElement.getElementsByTagName("nickname").item(0).getTextContent());
                System.out.println("Salary : " +
eElement.getElementsByTagName("salary").item(0).getTextContent());
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
  }
}
```

Output:

```
Root element :company
----------------------------

Current Element :staff
Staff id : 1001
First Name : yong
```

Last Name : mook kim
Nick Name : mkyong
Salary : 100000

Current Element :staff
Staff id : 2001
First Name : low
Last Name : yin fong
Nick Name : fong fong
Salary : 200000

In *Chapter 4* we will look at JavaBeans.

Chapter 4 - JavaBeans

JavaBeans

Introduction

A JavaBean is a platform independent reusable software component. JavaBeans provides a standard format for creating Java classes. Once a bean is created, it can be used over and over again in many applications as per their requirements. JavaBeans can be used by IDEs and other Java APIs to create new applications. The information of a bean is automatically discovered and then manipulated without explicitly coding them again. A JavaBean is a Java class which follows the requirements given below:

- The class must have a public no-argument constructor
- The class must provide public accessor and mutator methods to read and write properties of a bean.
- The class should implement *Serializable* or *Externalizable* interface available in *java.io* package

A bean may have a Graphical User Interface (GUI) like a button provided by the IDEs. Some beans doesn't have a user interface and are used to perform computations. Example of a bean that doesn't have a user interface is stock analyzer.

Advantages of JavaBeans

Following are the advantages of JavaBeans:

- A bean obtains all the benefits of Java's "write once, run anywhere" paradigm.
- The properties, methods and events of a bean that are exposed to an application builder tool can be controlled.
- A bean may be used to operate correctly in different locales, which makes it useful in global markets.
- Auxiliary software can be provided to help a person configure a bean.
- The configuration settings of a bean can be saved in persistent storage and restored at a later time.

Following Java code is an example for simple user bean:

```
import  java.io.*;
public  class  UserBean  implements Serializable
{
        private  String  name;
        private  String  pass;
        public  User( )  {  }
        public  String  getName( )
```

```
        {
                return  name;
        }
        public  String  getPass( )
        {
                return  pass;
        }
        public  void  setName(String  n)
        {
                name = n;
        }
        public void  setPass(String  p)
        {
                pass = p;
        }
}
```

The above user bean contains two properties: *name*, *pass* and two accessor methods: *getName( ), getPass( )* and two mutator methods: *setName( ), setPass( ).*

Introspection

Before going to know about introspection, we should know about bean builder. A bean builder is a readymade tool which provides an environment to build and test beans. Examples of bean builder tools are BeanBox, NetBeans, JBuilder etc. A builder tool allows the developer to modify the bean without writing any code.

Features provided by a bean builder are:

- Introspection
- Properties
- Events
- Customization
- Persistence

Introspection is the mechanism that a bean builder uses to find the properties, methods, events and other necessary info available in a bean. Bean builder uses Java's reflection API and design patterns for introspection.

Design patterns are set of rules or conventions that Java follows for naming classes and methods of a bean. Design patterns allows the builder tools to provide a way to customize the beans.

Following are some of the design patterns that should be followed by a bean class:

- For a XXX property on a bean, getXXX( ) will be the reader (accessor) method and setXXX( ) will be the writer (mutator) method. For boolean properties, the reader method will be isXXX( ).

- A bean info class implementing the *BeanInfo* interface should be named as class name followed by the string *BeanInfo*. For example, if the bean class name is *Student*, the bean info class name should be *StudentBeanInfo*.
- The name of a customizer class which implements the *Customizer* interface must have the string *Customizer* after the class name. For example, if the bean class name is *Student*, the customizer class name will be *StudentCustomizer*.

Properties of a Bean

Properties of a bean represent its state and control the behaviour of the bean. There different types of bean properties:

- Simple properties
- Indexed properties
- Bound properties
- Constrained properties

Simple Properties

A simple or basic property is one which contains a single value. The value can be a primitive type value or an object reference. Each simple property should have reader and writer methods to access and manipulate the value of the property. Consider the following example which demonstrates simple properties:

```
import java.io.*;
public class UserBean implements Serializable
{
        private String name;
        private String regdno;
        public UserBean( ) {}
        public String getName( )
        {
                return name;
        }
        public String getRegdno( )
        {
                return regdno;
        }
        public void setName(String n)
        {
                name = n;
        }
        public void setRegdno(String r)
        {
                regdno = r;
        }
}
```

In the above example, *name* and *regdno* are examples for simple properties.

Indexed Properties

An indexed property is one which contains multiple values like an array. Each indexed property must have reader and writer methods to store and retrieve all the values and as well as individual values of the property. Consider the following example which demonstrates indexed properties:

```
import java.io.*;
public class UserBean implements Serializable
{
        private int marks[ ] = new int[6];
        public UserBean( ) {}
        public int[ ] getMarks( )
        {
                return marks;
        }
        public int getMarks(int index)
        {
                return marks[index];
        }
        public void setMarks(int[ ] m)
        {
                marks = m;
        }
        public void setMarks(int m, int index)
        {
                marks[index] = m;
        }
}
```

In the above example, *marks* is an example for indexed property.

Bound Properties

A property which is bound to a listener is known as a bound property. The listener is automatically notified when the bound property changes. *PropertyChangeListener* interface and *PropertyChangeEvent* class of *java.beans* package are used for implementing bound properties. The bean class should provide the methods *addPropertyChangeListener( )* and *removePropertyChangeListener( )* for managing bean's listeners. The *java.beans* package provides a utility class *PropertyChangeSupport* for keeping track of the listeners of the bound property. As soon as the bound property changes, a *PropertyChangeEvent* object is sent to all the listeners by the method *firePropertyChange( ).*

Below example demonstrates bound property:

```
import java.beans.*;
public class UserBean
{
	private String name;
	private PropertyChangeSupport pcs = new PropertyChangeSupport(this);
	public String getName( )
	{
		return name;
	}
	public void setName(String n)
	{
		String oldName = name;
		name = n;
		pcs.firePropertyChange("name", oldName, n);
	}
	public void addPropertyChangeListener(PropertyChangeListener listener)
	{
		pcs.addPropertyChangeListener(listener);
	}
	public void removePropertyChangeListener(PropertyChangeListener listener)
	{
		pcs.removePropertyChangeListener(listener);
	}
}
```

Constrained Properties

A constrained property is a special type of bound property. While changing the value of a constrained property, all the listeners of that property are notified about the change. If any one of the listeners veto's (rejects) the change, a *PropertyVetoException* is thrown. The *java.beans* package provides *VetoableChangeListener* interface and *VetoableChangeSupport* class to work with constrained properties.

Below example demonstrates a constrained property:

```
import java.beans.*;
public class UserBean
{
	private String name;
	private PropertyChangeSupport pcs = new PropertyChangeSupport(this);
	private VetoableChangeSupport vcs = new VetoableChangeSupport(this);
	public String getName( )
	{
		return name;
	}
	public void setName(String n)
	{
		String oldName = name;
		vcs.fireVetoableChange("name", oldName, n);
```

```
            name = n;
            pcs.firePropertyChange("name", oldName, n);
        }
        public void addPropertyChangeListener(PropertyChangeListener listener)
        {
            pcs.addPropertyChangeListener(listener);
        }
        public void removePropertyChangeListener(PropertyChangeListener listener)
        {
            pcs.removePropertyChangeListener(listener);
        }
        public void addVetoableChangeListener(VetoableChangeListener listener)
        {
            pcs.addVetoableChangeListener(listener);
        }
        public void removeVetoableChangeListener(VetoableChangeListener listener)
        {
            pcs.removeVetoableChangeListener(listener);
        }
}
```

Bean Development Kit (BDK)