SHRIMATI INDIRA GANDHI COLLEGE
(Nationally Accredited with ‘A’ Grade(3rd Cycle) by NAAC)
An ISO 9001:2015 Certified Institution
Tiruchirappalli – 620 002

DEPARTMENTS OF COMPUTER SCIENCE, IT & APPLICATIONS

WEB TECHNOLOGIES - STUDY MATERIAL (UNIT –II)

M.Sc COMPUTER SCIENCE (2021-2022)

AUTHOR:

Dr.M.GOMATHY
Asst. Professor
Department of Computer Science
Shrimati Indira Gandhi College
Trichirappalli-02

UNIT –II –JAVA SCRIPT

Client Side Programming : JavaScript

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

What can a JavaScript do?

- **JavaScript can put dynamic text into an HTML page**
- **JavaScript can react to events -** A JavaScript can be set to execute when something happens, like when a page has finished loading or when a user clicks on an HTML element
- **JavaScript can read and write HTML elements -** A JavaScript can read and change the content of an HTML element
- **JavaScript can be used to validate data -** A JavaScript can be used to validate form data before it is submitted to a server. This saves the server from extra processing
- **JavaScript can be used to create cookies** - A JavaScript can be used to store and retrieve information on the visitor's computer

The Real Name is ECMAScript. JavaScript is an implementation of the ECMAScript language standard.ECMAScript is developed and maintained by the ECMA organization. ECMA-262 is the official JavaScript standard. The language was invented by Brendan Eich at Netscape (with Navigator 2.0), and has appeared in all Netscape and Microsoft browsers since 1996. The development of ECMA-262 started in 1996, and the first edition of was adopted by the ECMA General Assembly in June 1997. The standard was approved as an international ISO (ISO/IEC 16262) standard in 1998. The development of the standard is still in progress.

Structure of Java Script

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

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

So, the <script type="text/javascript"> and </script> tells where the JavaScript starts and ends. It is possible to have script code inside the body tag also as shown below. If it is placed inside the body tag, the script will be executed when the content of HTML document is displayed.

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

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

Scripts in <head>

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

Example

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

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

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

Scripts in <head> and <body>

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

Example

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

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

</html>
```

Using an External JavaScript

JavaScript can also be placed in external files. External JavaScript files often contains code to be used on several different web pages. External JavaScript files have the file extension .js. External script cannot contain the <script></script> tags. To use an external script, point to the .js file in the "src" attribute of the <script> tag:

Example

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

JavaScript Variables

JavaScript variables are used to hold values or expressions. A variable can have a short name, like x, or a more descriptive name, like carname. Rules for JavaScript variable names:

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

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

Declaring (Creating) JavaScript Variables

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

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

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

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

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

Assigning Values to Undeclared JavaScript Variables

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

```
x=5;
carname="Volvo";
```

have the same effect as:

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

Redeclaring JavaScript Variables

If you redeclare a JavaScript variable, it will not lose its original value.

```
var x=5;
var x;
```

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

The Lifetime of JavaScript Variables

If you declare a variable within a function, the variable can only be accessed within that function. When you exit the function, the variable is destroyed. These variables are called local variables. You can have local variables with the same name in different functions, because each is recognized only by the function in which it is declared.

If you declare a variable outside a function, all the functions on your page can access it. The lifetime of these variables starts when they are declared, and ends when the page is closed.

JavaScript Arithmetic Operators

Arithmetic operators are used to perform arithmetic between variables and/or values. Given that **y=5**, the table below explains the arithmetic operators:

Operator	**Description**	**Example**	**Result**
+	Addition	x=y+2	x=7
-	Subtraction	x=y-2	x=3
*	Multiplication	x=y*2	x=10
/	Division	x=y/2	x=2.5
%	Modulus (division remainder)	x=y%2	x=1
++	Increment	x=++y	x=6
--	Decrement	x=--y	x=4

JavaScript Assignment Operators

Assignment operators are used to assign values to JavaScript variables. Given that **x=10** and **y=5**, the table below explains the assignment operators:

Operator	**Example**	**Same As**	**Result**
=	x=y		x=5
+=	x+=y	x=x+y	x=15
-=	x-=y	x=x-y	x=5
*=	x*=y	x=x*y	x=50
/=	x/=y	x=x/y	x=2
%=	x%=y	x=x%y	x=0

Comparison Operators Comparison operators are used in logical statements to determine equality or difference between variables or values. Given that x=5, the table below explains the comparison operators:

Operator	Description	Example
==	is equal to	x==8 is false
===	is exactly equal to (value and type)	x===5 is true x==="5" is false
!=	is not equal	x!=8 is true
>	is greater than	x>8 is false
<	is less than	x<8 is true
>=	is greater than or equal to	x>=8 is false
<=	is less than or equal to	x<=8 is true

Logical Operators

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

Operator	Description	Example
&&	and	(x < 10 && y > 1) is true
||	or	(x==5 || y==5) is false
!	not	!(x==y) is true

Conditional Operator

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

Syntax

```
variablename=(condition)?value1:value2
```

Example

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

Conditional Statements

Very often when you write code, you want to perform different actions for different decisions. You can use conditional statements in your code to do this.

In JavaScript we have the following conditional statements:

- **if statement** - use this statement to execute some code only if a specified condition is true
- **if...else statement** - use this statement to execute some code if the condition is true and another code if the condition is false
- **if...else if....else statement** - use this statement to select one of many blocks of code to be executed

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

If Statement

Use the if statement to execute some code only if a specified condition is true.

Syntax

```
if (condition)
  {
  code to be executed if condition is true
  }
```

Note that if is written in lowercase letters. Using uppercase letters (IF) will generate a JavaScript error!

Example

```
<script type="text/javascript">
//Write a "Good morning" greeting if
//the time is less than 10

var d=new Date();
var time=d.getHours();

if (time<10)
  {
  document.write("<b>Good morning</b>");
  }
</script>
```

If...else Statement

Use the if....else statement to execute some code if a condition is true and another code if the condition is not true.

Syntax

```
if (condition)
  {
  code to be executed if condition is true
  }
else
  {
  code to be executed if condition is not true
  }
```

Example

```
<script type="text/javascript">
//If the time is less than 10, you will get a "Good morning" greeting.
//Otherwise you will get a "Good day" greeting.

var d = new Date();
var time = d.getHours();

if (time < 10)
  {
  document.write("Good morning!");
  }
else
  {
  document.write("Good day!");
  }
</script>
```

If...else if...else Statement

Use the if....else if...else statement to select one of several blocks of code to be executed.

Syntax

```
if (condition1)
  {
  code to be executed if condition1 is true
  }
else if (condition2)
  {
  code to be executed if condition2 is true
  }
else
  {
  code to be executed if condition1 and condition2 are not true
  }
```

Example

```
script type="text/javascript">
var d = new Date()
var time = d.getHours()
if (time<10)
  {
  document.write("<b>Good morning</b>");
  }
else if (time>10 && time<16)
  {
  document.write("<b>Good day</b>");
  }
else
  {
  document.write("<b>Hello World!</b>");
  }
</script>
```

The JavaScript Switch Statement

Use the switch statement to select one of many blocks of code to be executed.

Syntax

```
switch(n)
{
case 1:
  execute code block 1
  break;
case 2:
  execute code block 2
  break;
default:
  code to be executed if n is different from case 1 and 2
}
```

JavaScript Popup Boxes

JavaScript has three kind of popup boxes: Alert box, Confirm box, and Prompt box.

Alert Box

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

Syntax

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

Example

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

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

Confirm Box

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

Syntax

confirm("*sometext*");

Example

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

Prompt Box

A prompt box is often used if you want the user to input a value before entering a page. When a prompt box pops up, the user will have to click either "OK" or "Cancel" to proceed after entering an input value. If the user clicks "OK" the box returns the input value. If the user clicks "Cancel" the box returns null.

Syntax

prompt("*sometext*","*defaultvalue*");

Example

```
<html>
<head>
<script type="text/javascript">
function show_prompt()
{
var name=prompt("Please enter your name","Harry Potter");
if (name!=null && name!="")
  {
  document.write("Hello " + name + "! How are you today?");
  }
}
</script>
</head>
<body>

<input type="button" onclick="show_prompt()" value="Show prompt box" />

</body>
</html>
```

JavaScript Functions

A function will be executed by an event or by a call to the function.

JavaScript Functions

To keep the browser from executing a script when the page loads, you can put your script into a function. A function contains code that will be executed by an event or by a call to the function. You may call a function from anywhere within a page (or even from other pages if the function is embedded in an external .js file). Functions can be defined both in the <head> and in the <body> section of a document. However, to assure that a function is read/loaded by the browser before it is called, it could be wise to put functions in the <head> section.

How to Define a Function

Syntax

function *functionname*(*var1,var2,...,varX*)
{
some code
}

The parameters var1, var2, etc. are variables or values passed into the function. The { and the } defines the start and end of the function. The word *function* must be written in lowercase letters, otherwise a JavaScript error occurs! Also note that you must call a function with the exact same capitals as in the function name

JavaScript Function Example

Example

```
<html>
<head>
<script type="text/javascript">
function displaymessage()
{
alert("Hello World!");
}
</script>
</head>

<body>
<form>
<input type="button" value="Click me!" onclick="displaymessage()" />
</form>
</body>
</html>
```

The return Statement

The return statement is used to specify the value that is returned from the function. So, functions that are going to return a value must use the return statement. The example below returns the product of two numbers (a and b):

Example

```
<html>
<head>
<script type="text/javascript">
function product(a,b)
{
return a*b;
}
</script>
</head>

<body>
<script type="text/javascript">
document.write(product(4,3));
</script>

</body>
</html>
```

JavaScript Loops

Often when you write code, you want the same block of code to run over and over again in a row. Instead of adding several almost equal lines in a script we can use loops to perform a task like this.

In JavaScript, there are two different kind of loops:

- **for** - loops through a block of code a specified number of times
- **while** - loops through a block of code while a specified condition is true

The for Loop

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

Syntax

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

Example

The example below defines a loop that starts with i=0. The loop will continue to run as long as **i** is less than, or equal to 5. **i** will increase by 1 each time the loop runs.

Note: The increment parameter could also be negative, and the <= could be any comparing statement.

Example

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

JavaScript While Loop

Loops execute a block of code a specified number of times, or while a specified condition is true.

The while Loop

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

Syntax

```
while (variable<=endvalue)
  {
  code to be executed
  }
```

Note: The <= could be any comparing operator.

Example

The example below defines a loop that starts with i=0. The loop will continue to run as long as **i** is less than, or equal to 5. **i** will increase by 1 each time the loop runs:

Example

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

The do...while Loop

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

Syntax

```
do
  {
  code to be executed
  }
while (variable<=endvalue);
```

Example

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

Example

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

The break Statement

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

Example

```
<html>
<body>
<script type="text/javascript">
var i=0;
for (i=0;i<=10;i++)
  {
  if (i==3)
    {
    break;
    }
  document.write("The number is " + i);
  document.write("<br />");
  }
</script>
</body>
</html>
```

The continue Statement

The continue statement will break the current loop and continue with the next value.

Example

```
<html>
<body>
<script type="text/javascript">
var i=0
for (i=0;i<=10;i++)
  {
  if (i==3)
    {
    continue;
    }
  document.write("The number is " + i);
  document.write("<br />");
  }
</script>
</body>
</html>
```

JavaScript For...In Statement

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

Syntax

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

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

Note: The variable argument can be a named variable, an array element, or a property of an object.

Example

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

Example

```
<html>
<body>

<script type="text/javascript">
var x;
var mycars = new Array();
mycars[0] = "Saab";
mycars[1] = "Volvo";
mycars[2] = "BMW";

for (x in mycars)
  {
  document.write(mycars[x] + "<br />");
  }
</script>

</body>
</html>
```

JavaScript Events

Events are actions that can be detected by JavaScript.

Events

By using JavaScript, we have the ability to create dynamic web pages. Events are actions that can be detected by JavaScript. Every element on a web page has certain events which can trigger a JavaScript. For example, we can use the onClick event of a button element to indicate that a function will run when a user clicks on the button. We define the events in the HTML tags.

Examples of events:

- A mouse click
- A web page or an image loading
- Mousing over a hot spot on the web page
- Selecting an input field in an HTML form
- Submitting an HTML form
- A keystroke

Events are normally used in combination with functions, and the function will not be executed before the event occurs.

Event Association

Events are associated with HTML tags. The definitions of the events described below are as follows:

Event handler	Applies to:	Triggered when:
onAbort	Image	The loading of the image is cancelled.
onBlur	Button, Checkbox, Password, Radio, Reset, Select, Submit, Text, TextArea, Window	The object in question loses focus (e.g. by clicking outside it or pressing the TAB key).
onChange	Select, Text, TextArea	The data in the form element is changed by the user.
onClick	Button, Checkbox, Link, Radio, Reset, Submit	The object is clicked on.
onDblClick	Document, Link	The object is double-clicked on.
onError	Image	A JavaScript error occurs.
onFocus	Button, Checkbox, Password, Radio, Reset, Select, Submit, Text, TextArea	The object in question gains focus (e.g. by clicking on it or pressing the TAB key).
onKeyDown	Image, Link, TextArea	The user presses a key.
onKeyPress	Image, Link, TextArea	The user presses or holds down a key.
onKeyUp	Image, Link, TextArea	The user releases a key.
onLoad	Image, Window	The whole page has finished loading.
onMouseDown	Button, Link	The user presses a mouse button.
onMouseMove	None	The user moves the mouse.
onMouseOut	Image, Link	The user moves the mouse away from the object.
onMouseOver	Image, Link	The user moves the mouse over the object.
onMouseUp	Button, Link	The user releases a mouse button.
onMove	Window	The user moves the browser window or frame.
onReset	Form	The user clicks the form's Reset button.
onResize	Window	The user resizes the browser window

		or frame.
onSelect	Text, Textarea	The user selects text within the field.
onSubmit	Form	The user clicks the form's Submit button.
onUnload	Window	The user leaves the page.

JavaScript Objects

Object oriented Programming in an important aspect of JavaScript. It is possible to use built-in objects available in JavaScript. It is also possible for a JavaScript programmer to define his own objects and variable types. In this JavaScript tutorial, you will learn how to make use of built-in objects available in JavaScript.

Built-in objects in JavaScript:

Some of the built-in objects available in JavaScript are:

- Date
- Math
- String, Number, Boolean
- RegExp
- window (Global Obejct)

JavaScript String Object

Of the above objects, the most widely used one is the String object. Objects are nothing but special kind of data. Each object has Properties and Methods associated with it. *property* is the value that is tagged to the object. For example let us consider one of the properties associated with the most popularly used String object - the *length* property. *Length* property of the string object returns the length of the string that is in other words the number of characters present in the string.

General syntax of using the *length* property of the string object is as below:

variablename.length

Here *variablename* is the name of the variable to which the string is assigned and *length* is the keyword.

For example consider the JavaScript below:

```
<html>
  <body>
    <script type="text/javascript">
      var exf="Welcome"
      document.write(exf.length)

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

The output of the above is

```
7
```

Method of an Object:

Method of an object refers to the actions than can be performed on the object. For example in String Object there are several methods available in JavaScript.

Example to understand how method can be used in an Object.

In the example below, we have used *toUpperCase* method of String object.

```
<html>
  <body>
    <script type="text/javascript">
      var exf="Welcome"
      document.write(exf.toUpperCase())

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

The output of the above script is

```
WELCOME
```

In the above script since the method *toUpperCase* is applied to the string object *exf* which has value initialized as *Welcome* all letters get converted as upper case and hence the output is as above.

Purpose of String Object in JavaScript:

The main purpose of String Object in JavaScript is for storing text. General method of using String Object is to declare a variable and assign a string, in other words a text to the variable.

var exf="Welcome"

assigns the text *Welcome* to the variable *exf* defined.

String Object Methods

Method	**Description**
charAt()	Returns the character at the specified index
charCodeAt()	Returns the Unicode of the character at the specified index
concat()	Joins two or more strings, and returns a copy of the joined strings
indexOf()	Returns the position of the first found occurrence of a specified value in a string
lastIndexOf()	Returns the position of the last found occurrence of a specified value in a string
match()	Searches for a match between a regular expression and a string, and returns the matches
replace()	Searches for a match between a substring (or regular expression) and a string, and replaces the matched substring with a new substring
search()	Searches for a match between a regular expression and a string, and returns the position of the match
slice()	Extracts a part of a string and returns a new string
split()	Splits a string into an array of substrings
substr()	Extracts the characters from a string, beginning at a specified start position, and through the specified number of character
substring()	Extracts the characters from a string, between two specified indices
toLowerCase()	Converts a string to lowercase letters
toUpperCase()	Converts a string to uppercase letters
valueOf()	Returns the primitive value of a String object

JavaScript Date Object

Usage of Date Object:

Date object of Java Script is used to work with date and times. General syntax for defining Date object in Java Script is as follows:

var variablename=new Date( )

In the above *new* is a keyword which creates an instance of object and Date() defines *variablename* as Date Object.

For example:

var exf=new Date( )

In the above example, variable *exf* is defined as Date object which has current date and time as its initial value.

Methods of Date Object:

Some of the methods available with Date object are:

setSeconds()- Sets the seconds for a specified date according to local time.
setMinutes() - Sets the minutes for a specified date according to local time.
setHours() - Sets the hours for a specified date according to local time.
setDate() - Sets the day of the month for a specified date according to local time.
setMonth() - Sets the month for a specified date according to local time.
setYear() - Sets the year (deprecated) for a specified date according to local time.
setFullYear() - Sets the full year for a specified date according to local time.
toString() - Returns a string representing the specified Date object.
getSeconds() - Returns seconds in the specified date according to local time.
getMinutes() - Returns minutes in the specified date according to local time.
getHours() - Returns hour in the specified date according to local time.
getDay() - Returns day of the week for a specified date according to local time
getDate() - Returns day of the month for a specified date according to local time.
getMonth() - Returns month in the specified date according to local time.
getYear() - Returns year (deprecated) in the specified date according to local time.
getFullYear() - Returns year of the specified date according to local time.

Example for usage of Date Object methods mentioned above:

var exf=new Date()

exf.setFullYear(2020,0,20)

As we have seen *setFullYear()* is used for Setting the full year for a specified date according to local time. In the above example the Date object *exf* is set to the specific date and year *20th January 2020*

Example for using methods of Date Object

```
<html>
 <body>
   <script type="text/javascript">
     var exforsys=new Date();
     var currentDay=exforsys.getDate();
```

```
      var currentMonth=exforsys.getMonth() + 1;
      var currentYear=exforsys.getFullYear();
      document.write(currentMonth + "/" + currentDay +
      "/" + currentYear);

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

Output of the above program is:

```
11/15/2006
```

JavaScript Math Object

Usage of Math Object:

JavaScript *Math* object is used to perform mathematical tasks. But unlike the *String* and the *Date* object which requires defining the object, *Math* object need not be defined. *Math* object in JavaScript has two main attributes:

- Properties
- Methods

Properties of Math Object:

The JavaScript has eight mathematical values and this can be accessed by using the *Math* Object. The eight mathematical values are:

- E
- PI
- square root of 2 denoted as SQRT2
- square root of 1/2 denoted as SQRT1_2
- natural log of 2 denoted as LN2
- natural log of 10 denoted as LN10
- base-2 log of E denoted as LOG2E
- base-10 log of E denoted as LOG10E

The way of accessing these values in JavaScript is by using the word *Math* before these values namely as

- Math.E
- Math.LOG10E *and so on*

Methods of Math Object:

There are numerous methods available in JavaScript for *Math* Object. Some of them are mentioned below namely:

- abs(x) - Returns absolute value of x.
- acos(x) - Returns arc cosine of x in radians.
- asin(x) - Returns arc sine of x in radians.
- atan(x) - Returns arc tan of x in radians.
- atan2(y, x) - Counterclockwise angle between x axis and point (x,y).
- ceil(x) - Returns the smallest integer greater than or equal to x. (round up).
- cos(x) - Returns cosine of x, where x is in radians.
- exp(x) - Returns ex
- floor(x) - Returns the largest integer less than or equal to x. (round down)
- log(x) - Returns the natural logarithm (base E) of x.
- max(a, b) - Returns the larger of a and b.
- min(a, b) - Returns the lesser of a and b.
- pow(x, y) - Returns xy
- random() - Returns a pseudorandom number between 0 and 1.
- round(x) - Rounds x up or down to the nearest integer. It rounds .5 up.
- sin(x) - Returns the Sin of x, where x is in radians.
- sqrt(x) - Returns the square root of x.
- tan(x) - Returns the Tan of x, where x is in radians.

Example for *Math* Object methods mentioned above:

```
<html>
  <body>
    <script type="text/javascript">
       document.write(Math.round(5.8))

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

The output of the above program is

6

This is because the *round()* method rounds the number given in argument namely here 5.8 to the nearest integer. It rounds .5 up which gives 6.

Another example for using *Math* Object in JavaScript.

```
<html>
  <body>
    <script type="text/javascript">
       document.write(Math.max(8,9) + "<br />")
       document.write(Math.max(-5,3) + "<br />")
       document.write(Math.max(-2,-7) + "<br />")

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

Output of the above program is

```
9
3
-2
```

The above example uses the *max()* method of the *Math* object which returns the largest of the two numbers given in arguments of the max method.

JavaScript Boolean Object

The Boolean object is used to convert a non-Boolean value to a Boolean value (true or false).

Boolean Object Methods

Method	**Description**
toString()	Converts a Boolean value to a string, and returns the result
valueOf()	Returns the primitive value of a Boolean object

Number Object

The Number object is an object wrapper for primitive numeric values.

Number objects are created with new Number().

Syntax

var num = new Number(value);

Number Object Methods

Method	Description
toExponential(x)	Converts a number into an exponential notation
toFixed(x)	Formats a number with x numbers of digits after the decimal point
toPrecision(x)	Formats a number to x length
toString()	Converts a Number object to a string
valueOf()	Returns the primitive value of a Number object

String Object

The String object is used to manipulate a stored piece of text.

String objects are created with new String().

Syntax

var txt = new String(string);

or more simply:

var txt = string;

Window Object

The window object represents an open window in a browser.

If a document contain frames (<frame> or <iframe> tags), the browser creates one window object for the HTML document, and one additional window object for each frame.

Window Object Methods

Method	Description
alert()	Displays an alert box with a message and an OK button
blur()	Removes focus from the current window
clearInterval()	Clears a timer set with setInterval()
clearTimeout()	Clears a timer set with setTimeout()
close()	Closes the current window
confirm()	Displays a dialog box with a message and an OK and a Cancel button
createPopup()	Creates a pop-up window
focus()	Sets focus to the current window
moveBy()	Moves a window relative to its current position
moveTo()	Moves a window to the specified position
open()	Opens a new browser window

print()	Prints the content of the current window
prompt()	Displays a dialog box that prompts the visitor for input
resizeBy()	Resizes the window by the specified pixels
resizeTo()	Resizes the window to the specified width and height
scroll()	
scrollBy()	Scrolls the content by the specified number of pixels
scrollTo()	Scrolls the content to the specified coordinates
setInterval()	Calls a function or evaluates an expression at specified intervals (in milliseconds)
setTimeout()	Calls a function or evaluates an expression after a specified number of milliseconds

JavaScript RegExp Object
Regular expressions are used to do sophisticated pattern matching, which can often be helpful in form validation. For example, a regular expression can be used to check whether an email address entered into a form field is syntactically correct. JavaScript supports Perl-compatible regular expressions.

There are two ways to create a regular expression in JavaScript:

1. Using literal syntax

```
var reExample = /pattern/;
```

2. Using the RegExp() constructor

```
var reExample = new
RegExp("pattern");
```

Assuming you know the regular expression pattern you are going to use, there is no real difference between the two; however, if you don't know the pattern ahead of time (e.g, you're retrieving it from a form), it can be easier to use the RegExp() constructor.

JavaScript's Regular Expression Methods

The regular expression method in JavaScript has two main methods for testing strings: test() and exec().
The exec() Method

The exec() method takes one argument, a string, and checks whether that string contains one or more matches of the pattern specified by the regular expression. If one or more matches is found, the method returns a result array with the starting points of the matches. If no match is found, the method returns null.

The test() Method

The test() method also takes one argument, a string, and checks whether that string contains a match of the pattern specified by the regular expression. It returns true if it does contain a match and false if it does not. This method is very useful in form validation scripts. The code sample below shows how it can be used for checking a social security number. Don't worry about the syntax of the regular expression itself. We'll cover that shortly.

Code Sample: RegularExpressions for validating social security number

```
<html>
<head>
<script type="text/javascript">
var exp = /^[0-9]{3}[\- ]?[0-9]{2}[\- ]?[0-9]{4}$/;

function f1(ssn)
{
   if (exp.test(ssn))  {   alert("VALID  SSN");  }
  else  {   alert("INVALID SSN");  }
}
</script>
</head>
<body>
 <form name="f1">
  <input type="text" name="t1" />
  <input type="button" value="Check"    onclick="f1(this.f1.t1.value);" />
 </form>
</body>
</html>
```

Code Explanation

Let's examine the code more closely:

1. First, a variable containing a regular expression object for a social security number is declared.

```
var exp = /^[0-9]{3}[\- ]?[0-9]{2}[\- ]?[0-9]{4}$/;
```

2. Next, a function called f1() is created. This function takes one argument: ssn, which is a string. The function then tests to see if the string matches the regular expression pattern by passing it to the regular expression object's test() method. If it does match, the function alerts "VALID SSN". Otherwise, it alerts "INVALID SSN".

```
function f1(ssn)
{
 if (exp.test(ssn)) {     alert("VALID SSN");    }
 else {     alert("INVALID SSN");    }
}
```

3. A form in the body of the page provides a text field for inserting a social security number and a button that passes the user-entered social security number to the f1() function.

```
<form >
 <input type="text" name="t1" />
 <input type="button" value="Check"    onclick="checkSsn(this.form.ssn.value);" />
</form>
```

Flags

Flags appearing after the end slash modify how a regular expression works.

* The i flag makes a regular expression case insensitive. For example, /aeiou/i matches all lowercase and uppercase vowels.
* The g flag specifies a global match, meaning that all matches of the specified pattern should be returned.

Regular Expression Syntax

A regular expression is a pattern that specifies a list of characters.

Start and End :^ $

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

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

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

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

Number of Occurrences : ? + * {}

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

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

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

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

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

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

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

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

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

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

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

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

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

Common Characters: . \d \D \w \W \s \S

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

* The pattern fo.d can be found in "food", "foad", "fo9d", and "fo*d".

Backslash-d (\d) represents any digit. It is the equivalent of [0-9].

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

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

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

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

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

Backslash-W (\W) represents any character except a word character.

* The pattern fo\Wd can be found in "fo*d", "fo@d" and "fo.d", but not in "food".

Backslash-s (\s) represents any whitespace character (e.g, space, tab, newline, etc.).

* The pattern fo\sd can be found in "fo d", but not in "food".

Backslash-S (\S) represents any character except a whitespace character.

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

Grouping: []

Square brackets ([]) are used to group options.

* The pattern f[aeiou]d can be found in "fad" and "fed", but not in "food", "faed" or "fd".
* The pattern f[aeiou]{2}d can be found in "faed" and "feod", but not in "fod", "fed" or "fd".

Negation : ^

When used after the first character of the regular expression, the caret (^) is used for negation.

* The pattern f[^aeiou]d can be found in "fqd" and "f4d", but not in "fad" or "fed".

Subpatterns: ()

Parentheses () are used to capture subpatterns.

* The pattern f(oo)?d can be found in "food" and "fd", but not in "fod".

Alternatives: |

The pipe (|) is used to create optional patterns.

* The pattern foo$|^bar can be found in "foo" and "bar", but not "foobar".

Escape Character : \

The backslash (\ ) is used to escape special characters.

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

A more practical example has to do matching the delimiter in social security numbers. Examine the following regular expression.

```
^\d{3}([\- ]?)\d{2}([\- ]?)\d{4}$
```

Within the caret (^) and dollar sign ($), which are used to specify the beginning and end of the pattern, there are three sequences of digits, optionally separated by a hyphen or a space. This pattern will be matched in all of following strings (and more).

* 123-45-6789
* 123 45 6789
* 123456789
* 123-45 6789
* 123 45-6789
* 123-456789

The last three strings are not ideal, but they do match the pattern. Back references can be used to make sure that the second delimiter matches the first delimiter. The regular expression would look like this.

```
^\d{3}([\- ]?)\d{2}\1\d{4}$
```

The \1 refers back to the first subpattern. Only the first three strings listed above match this regular expression.

<u>Form Validation with Regular Expressions</u>

Regular expressions make it easy to create powerful form validation functions. Take a look at the following example.
Code Sample: Login.html

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

var RE_EMAIL = /^(\w+[\-\.])*\w+@(\w+\.)+[A-Za-z]+$/;
var RE_PASSWORD = /^[A-Za-z\d]{6,8}$/;

function validate()
{
 var email = form.Email.value;
 var password = form.Password.value;
 var errors = [];
 if (!RE_EMAIL.test(email))  {   alert( "You must enter a valid email address.");  }
 if (!RE_PASSWORD.test(password))  { alert( "You must enter a valid password.");  }
 }

</script>
</head>
```

```
<body>
 <form name="form">
 Email: <input type="text" name="Email" />
 Password: <input type="password" name="Password" />
 *Password must be between 6 and 10 characters and can only contain letters and digits.

 <input type="submit" value="Submit" onclick="Validate();"/>
 <input type="reset" value="Reset Form" />
 </p>
</form>
</body>
</html>
```

Code Explanation

This code starts by defining regular expressions for an email address and a password. Let's break each one down.

```
var RE_EMAIL = /^(\w+\.)*\w+@(\w+\.)+[A-Za-z]+$/;
```

1. The caret (^) says to start at the beginning. This prevents the user from entering invalid characters at the beginning of the email address.
2. (\w+[\-\.])* allows for a sequence of word characters followed by a dot or a dash. The * indicates that the pattern can be repeated zero or more times. Successful patterns include "ndunn.", "ndunn-", "nat.s.", and "nat-s-".
3. \w+ allows for one or more word characters.
4. @ allows for a single @ symbol.
5. (\w+\.)+ allows for a sequence of word characters followed by a dot. The + indicates that the pattern can be repeated one or more times. This is the domain name without the last portion (e.g, without the "com" or "gov").
6. [A-Za-z]+ allows for one or more letters. This is the "com" or "gov" portion of the email address.
7. The dollar sign ($) says to end here. This prevents the user from entering invalid characters at the end of the email address.

```
var RE_PASSWORD = /^[A-Za-z\d]{6,8}$/;
```

1. The caret (^) says to start at the beginning. This prevents the user from entering invalid characters at the beginning of the password.
2. [A-Za-z\d]{6,8} allows for a six- to eight-character sequence of letters and digits.
3. The dollar sign ($) says to end here. This prevents the user from entering invalid characters at the end of the password.

Exercises:

1. Construct a reg exp to validate a text field which should be used to accept only a string composed by 3 letters, one space, 6 numbers, a "-" and a number such as MJHJ 123456-6

Ans: /^[A-Za-z]{4}\s\d{6}\-\d{1}$/

2. Write regular expressions to check for:
    1. Proper Name
        - o starts with capital letter
        - o followed by one or more letters or apostophes
        - o may be multiple words (e.g, "New York City")
    2. Initial
        - o zero or one capital letters
    3. State
        - o two capital letters
    4. Postal Code
        - o five digits (e.g, "02138")
        - o possibly followed by a dash and four digits (e.g, "-1234")
    5. Username
        - o between 6 and 15 letters or digits
3. Add validation to check the following fields:
    1. first name
    2. middle initial
    3. last name
    4. city
    5. state
    6. zip
    7. username
3. Test your solution in a browser.

Document Object

Each HTML document loaded into a browser window becomes a Document object. The Document object provides access to all HTML elements in a page, from within a script.

Document Object Methods

Method	**Description**
close()	Closes the output stream previously opened with document.open()
getElementById()	Accesses the first element with the specified id
getElementsByName()	Accesses all elements with a specified name
getElementsByTagName()	Accesses all elements with a specified tagname
open()	Opens an output stream to collect the output from document.write() or document.writeln()
write()	Writes HTML expressions or JavaScript code to a document

writeln()	Same as write(), but adds a newline character after each statement

Arrays

It describes the JavaScript array object including parameters, properties, and methods.
Parameters

* arrayLength
* elementN - Array element list of values

Properties

* index
* input
* length - The quantity of elements in the object.
* prototype - For creating more properties.

Methods

* chop() - Used to truncate the last character of a all strings that are part of an array. This method is not defined so it must be written and included in your code.

```
var exclamations = new Array("Look out!", "Duck!" )
exclamations.chop()
```

Causes the values of exclamations to become:

```
Look out
Duck
```

* concat()
* grep(searchstring) - Takes an array and returns those array element strings that contain matching strings. This method is not defined so it must be written and included in your code.

```
words = new Array("limit","lines","finish","complete","In","Out")
inwords = words.grep("in")
```

The array, inwords, will be:

```
lines, finish
```

* join(delimiter) - Puts all elements in the array into a string, separating each element with the specified delimiter.

```
words = new Array("limit","lines","finish","complete","In","Out")
```

```
var jwords = words.join(";")
```

The value of the string jwords is:

```
limit;lines;finish;complete;In;Out
```

* pop() - Pops the last string off the array and returns it. This method is not defined so it must be written and included in your code.

```
words = new Array("limit","lines","finish","complete","In","Out")
var lastword = words.pop()
```

The value of the string lastword is:

```
Out
```

* push(strings) - Strings are placed at the end of the array. This method is not defined so it must be written and included in your code.

```
words = new Array("limit","lines","finish")
words.push("complete","In","Out")
```

The array, words, will be:

```
limit, lines, finish, complete, In, Out
```

* reverse() - Puts array elements in reverse order.

```
words = new Array("limit","lines","finish","complete","In","Out")
words.reverse()
```

The array, words, will be:

```
Out, In, complete, finish, lines, limit
```

* shift() - Decreases array element size by one by shifting the first element off the array and returning it. This method is not defined so it must be written and included in your code.

```
words = new Array("limit","lines","finish","complete","In","Out")
word = words.shift()
```

The array, words, will be:

```
In, complete, finish, lines, limit
```

The string word will be:

Out

* sort() - Sorts the array elements in dictionary order or using a compare function passed to the method.

```
words = new Array("limit","lines","finish","complete","In","Out")
word = words.sort()
```

The value of words becomes:

```
In,Out,complete,finish,limit,lines
```

* splice() - It is used to take elements out of an array and replace them with those specified. In the below example the element starting at element 3 is removed, two of them are removed and replaced with the specified strings. The value returned are those values that are replaced. This method is not defined so it must be written and included in your code.

```
words = new Array("limit","lines","finish","complete","In","Out")
words1 = words.splice(3, 2, "done", "On")
```

The value of words becomes:

```
limit, lines, finish, done, On, Out
```

The value of words1 is set to:

```
complete, In
```

* split(deliimiter) - Splits a string using the delimiter and returns an array.

```
words = new String("limit;lines;finish;complete;In;Out")
var swords = words.split(";")
```

The values in the array swords is:

```
limit, lines, finish, complete, In, Out
```

* unshift() - Places elementa at the start of an array

```
words = new Array("finish","complete","In","Out")
word = words.shift("limit","lines")
```

The array, words, will be:

```
limit, lines,finish, complete, In, Out
```

Form validation

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

<u>Validate text field to accept e-mail id</u>

```
<html>
<head>
<script>
function f1()
{
   var email=myForm.t1.value;
   if(email=="") alert("Enter E-mail ID");              // if no input
   if(email.indexOf("@")==-1) alert("invalid Id");  // if input is: raj.com
   if(!(email.indexOf("@")==email.lastIndexOf("@"))) alert("invalid Id");   // if input is:raj@kumar@com
   if(email.indexOf(".", 0)==-1) alert("Invalid Id");   // if input is:raj@yahoocom
   if(!(email.indexOf("@")<email.lastIndexOf("."))) alert("Invalid Id"); // if input is: raj.kumar@yahoocom
}
</script>
</head>
<body>
<form name="myForm">
Email ID:<input type="text" name="t1" />
<input type="button" value="click" onclick=f1() />
</form>
</body>
</html>
```

Output of the program

The above program will accept the input in any one of the following valid form

raj@yahoo.com   raj.kumar@yahoo.com   raj.k@yahoo.co.in

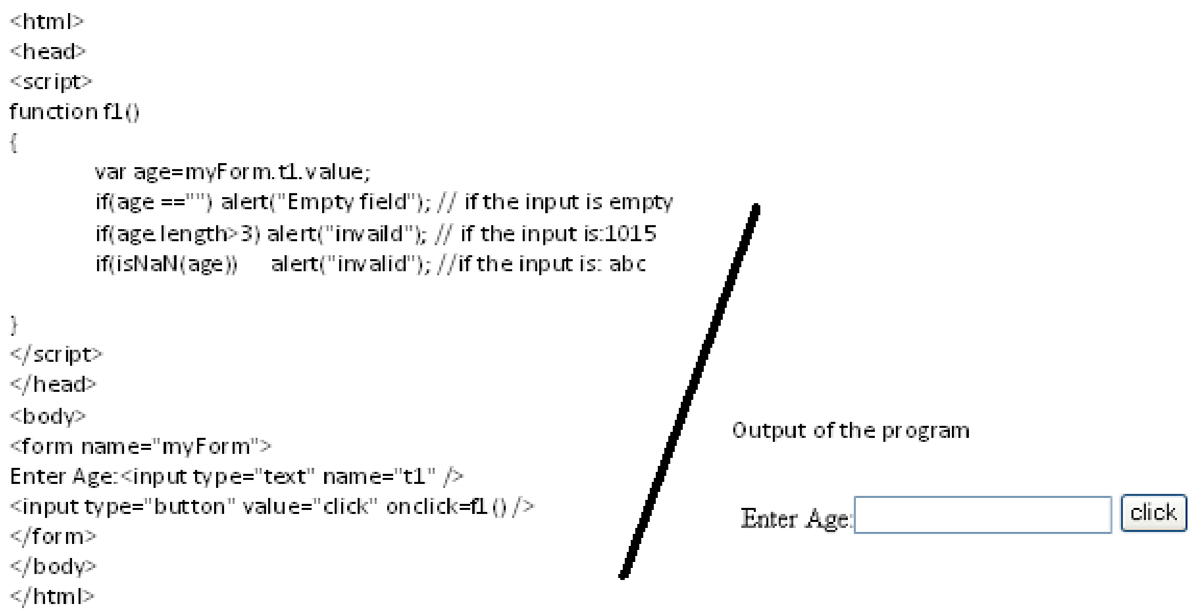
<u>validate text field to accept name</u>

```
<html>
<head>
<script>
function f1()
{
          var name=myForm.t1.value;
          if(name=="") alert("Enter name");     // if the input is empty
          for(var i=0; i<name.length; i++)
          {
              if ( (! (name.charAt(i)>='a' && name.charAt(i) <='z' ) | |
                     (name.charAt(i)>='A' && name.charAt(i) <='Z' ) ))       // if input is: raj321 or 321raj
                   {
                            alert("not a valid name");
                            break;
                   }
          }

}
</script>
</head>
<body>
<form name="myForm">
Enter Name:<input type="text" name="t1" />
<input type="button" value="click" onclick=f1() />
</form>
</body>
</html>
```

Output of the program

The above program will accept the input only in the following valid form
Rajkumar

<u>Validate text field to accept an age</u>

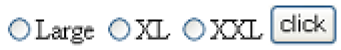


```
<html>
<head>
<script>
function f1()
{
        var age=myForm.t1.value;
        if(age =="") alert("Empty field"); // if the input is empty
        if(age.length>3) alert("invaild"); // if the input is:1015
        if(isNaN(age))    alert("invalid"); //if the input is: abc

}
</script>
</head>
<body>
<form name="myForm">
Enter Age:<input type="text" name="t1" />
<input type="button" value="click" onclick=f1() />
</form>
</body>
</html>
```

The above program will accept the input only in any one of the following valid form
25   5   101

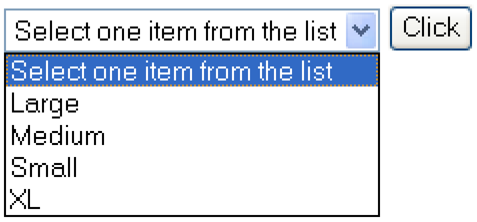
<u>Validate a checkbox</u>

```
<html>
<head>
<script>
function f1()
{

          if( !myForm.c1[0].checked && !myForm.c1[1].checked &&
                              !myForm.c1[2].checked )

          {
           alert("Select any one size");
          }
}
</script>
</head>
<body>
<form name="myForm">
<input type="radio" name="c1" />Large
<input type="radio" name="c1" />XL
<input type="radio" name="c1" />XXL
<input type="button" value="click" onclick=f1() />
</form>
</body>
</html>
```



Validate form selection

```
<html>
<head>
<script>
function f1()
{
          var sel=myForm.s1.selectedIndex;
          if(sel==0) alert("Select one item");


}
</script>
</head>
<body>
<form name="myForm">
<select name="s1">
<OPTION SELECTED>Select one item from the list
<OPTION VALUE="one">Large
<OPTION VALUE="two">Medium
<OPTION VALUE="three">Small
<OPTION VALUE="four">XL
</option>
</select>
<input type="button" value="Click" onclick=f1() />
</form>
</body>
</html>
```



Host Objects

JavaScript supports three types of objects: native, host, and user-defined. Native objects are objects supplied by the JavaScript language. String, Boolean, Math, and Number are examples of native objects.

Host objects are JavaScript objects that provide special access to the host environment. They are provided by the browser for the purpose of interaction with the loaded document. In a browser environment,

1. window
2. document

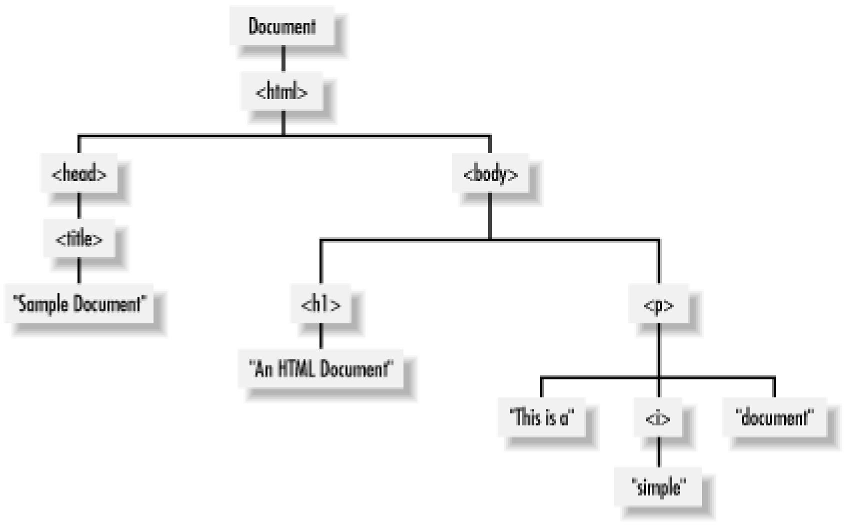
objects are host objects. Several other browser host objects are informal, *de facto* standards. They are: alert, prompt, confirm.

DOM

- The Document Object Model (DOM) is an API that allows programs to interact with HTML (or XML) documents
- The primary function of the Document Object Model is to view, access, and change the structure of an HTML document separate from the content contained within it.
- The DOM will provide you with methods and properties to retrieve, modify, update, and delete parts of the document you are working on. The properties of the Document Object Model are used to describe the web page or document and the methods of the Document Object Model are used for working with parts of the web page.
- In DOM, HTML document is represented in tree like structure. It constructs a hierarchical tree structure for a HTML document to traverse and to manipulate the document.
- For example,

```
<html>
  <head>
    <title>Sample Document</title>
  </head>
  <body>
    <h1>An HTML Document</h1>
    <p>This is a <i>simple</i> document.
  </body>
</html>
```

The DOM representation of this document is as follows:



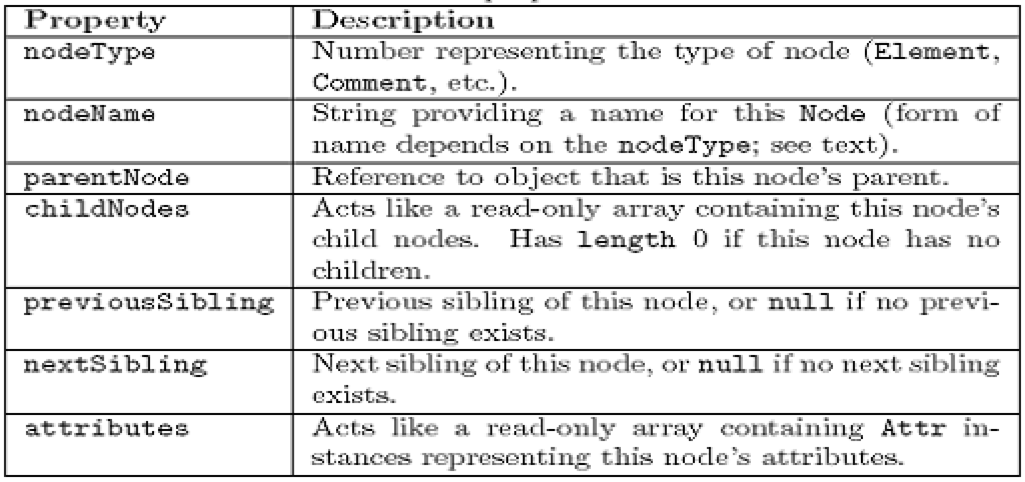
The node directly above a node is the *parent* of that node. The nodes one level directly below another node are the *children* of that node. Nodes at the same level, and with the same parent, are *siblings*. The set of nodes any number of levels below another node are the *descendants* of that node.

Types of nodes

- There are many types of nodes in the DOM document tree that specifies what kind of node it is. Every Object in the DOM document tree has properties and methods defined by the Node host object.

The following table lists the non method properties of Node object.

TABLE 5.2: Non-method properties of `Node` instances.



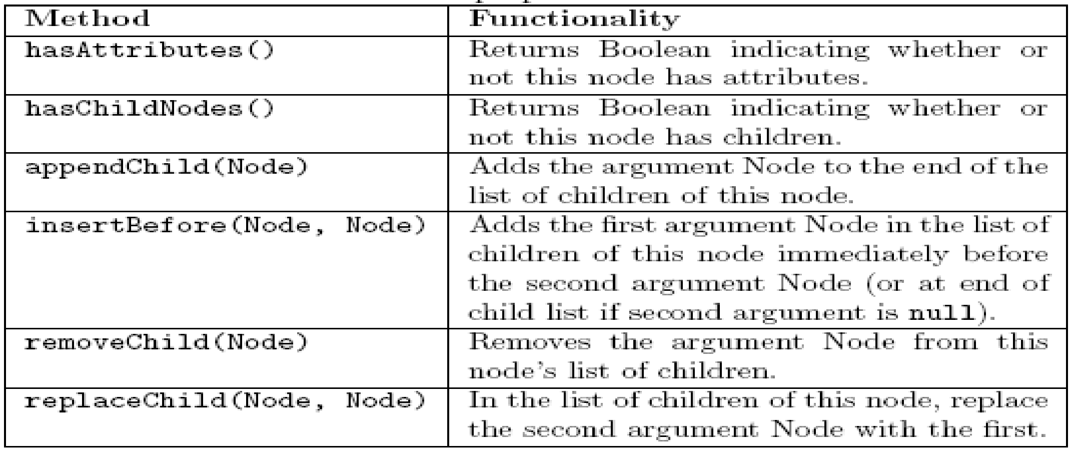
Property	Description
`nodeType`	Number representing the type of node (`Element`, `Comment`, etc.).
`nodeName`	String providing a name for this `Node` (form of name depends on the `nodeType`; see text).
`parentNode`	Reference to object that is this node's parent.
`childNodes`	Acts like a read-only array containing this node's child nodes. Has `length` 0 if this node has no children.
`previousSibling`	Previous sibling of this node, or `null` if no previous sibling exists.
`nextSibling`	Next sibling of this node, or `null` if no next sibling exists.
`attributes`	Acts like a read-only array containing `Attr` instances representing this node's attributes.

The following table lists the node types commonly encountered in HTML documents and the nodeType value for each one.

Node Type	nodeType constant	nodeType value
Element	Node.ELEMENT_NODE	1
Text	Node.TEXT_NODE	3
Document	Node.DOCUMENT_NODE	9
Comment	Node.COMMENT_NODE	8
DocumentFragment	Node.DOCUMENT_FRAGMENT_NODE	11
Attr	Node.ATTRIBUTE_NODE	2

The following table lists the method properties of Node object.

TABLE 5.4: Method properties of `Node` instances.



Method	Functionality
`hasAttributes()`	Returns Boolean indicating whether or not this node has attributes.
`hasChildNodes()`	Returns Boolean indicating whether or not this node has children.
`appendChild(Node)`	Adds the argument Node to the end of the list of children of this node.
`insertBefore(Node, Node)`	Adds the first argument Node in the list of children of this node immediately before the second argument Node (or at end of child list if second argument is `null`).
`removeChild(Node)`	Removes the argument Node from this node's list of children.
`replaceChild(Node, Node)`	In the list of children of this node, replace the second argument Node with the first.

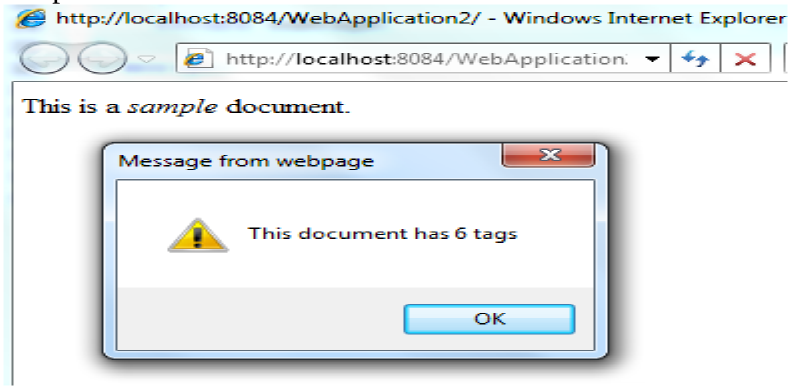
<u>Traversing a Document: Counting the number of Tags</u>

The DOM represents an HTML document as a tree of Node objects. With any tree structure, one of the most common things to do is traverse the tree, examining each node of the tree in turn. The following program shows one way to do this.

```
<html>
<head>
<script>
function countTags(n)
{                       // n is a Node
   var numtags = 0;              // Initialize the tag counter
   if (n.nodeType == 1 )  // Check if n is an Element
      numtags++;              // Increment the counter if so
   var children = n.childNodes;      // Now get all children of n
   for(var i=0; i < children.length; i++)
   {    // Loop through the children
      numtags += countTags(children[i]);   // Recurse on each one
   }
```

```
    return numtags;             // Return the total number of tags
}
</script>
</head>
<body onload="alert('This document has ' + countTags(document) + ' tags')">
This is a <i>sample</i> document.
</body>
</html>
```

Output



```
This is a sample document.

Message from webpage
This document has 6 tags
OK
```

http://localhost:8084/WebApplication2/ - Windows Internet Explorer
http://localhost:8084/WebApplication.

<u>Finding Specific Elements in a Document</u>

The ability to traverse all nodes in a document tree gives us the power to find specific nodes. When programming with the DOM API, it is quite common to need a particular node within the document or a list of nodes of a specific type within the document.

You can use getElementById() and getElementsByTagName( ) methods of Document Object to obtain a list of any type of HTML element. For example, to find all the tables within a document, you'd do this:

```
var tables = document.getElementsByTagName("table");
```

This code finds <table> tags and returns elements in the order in which they appear in the document.

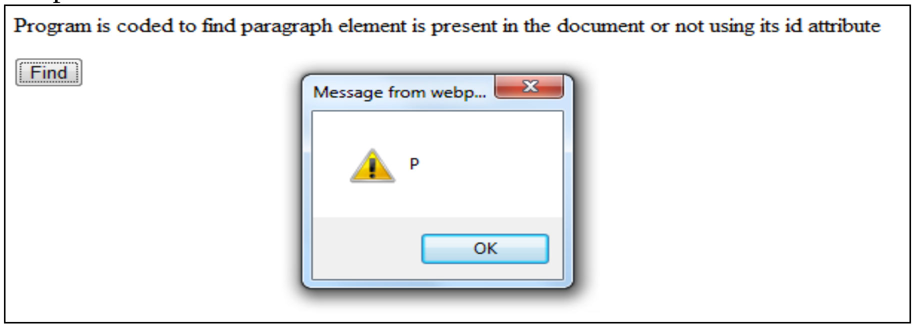
getElementById( ) to find a specific element whereas getElementsByName( ) returns an array of elements rather than a single element.

The following program illustrates this.

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

```
            function f1()
            {
               alert(document.getElementById("p1").nodeName);
            }
         </script>
         </head>
      <body>
         <p id="p1"> Program is coded to find paragraph element is present in the document
or not using
            its id attribute</p>
         <input type="button" value="Find" onclick="f1();" />

      </body>
</html>
```

Output



Program is coded to find paragraph element is present in the document or not using its id attribute

In the above program the method getElementById() finds the specific element and nodeName is used property return the specific element name.

Modifying a Document: Reversing the nodes of a document

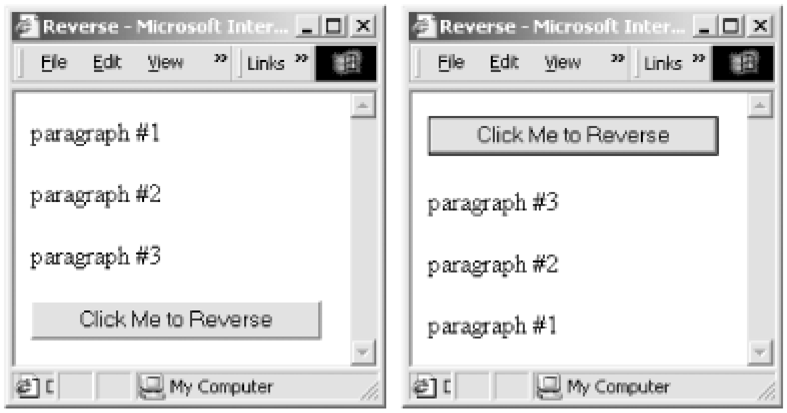
DOM API lies in the features that allow you to use JavaScript to dynamically modify documents. The following examples demonstrate the basic techniques of modifying documents and illustrate some of the possibilities.

The following example includes a JavaScript function named reverse( ), a sample document, and an HTML button that, when pressed, calls the reverse( ) function, passing it the node that represents the <body> element of the document. The reverse( ) function loops backward through the children of the supplied node and uses the removeChild( ) and appendChild( ) methods of the Node object to reverse the order of those children.

```
<html>
   <head><title>Reverse</title>
      <script>
         function reverse(n) {          // Reverse the order of the children of Node n
            var kids = n.childNodes;    // Get the list of children
            var numkids = kids.length;  // Figure out how many children there are
```

```
                for(var i = numkids-1; i >= 0; i--) {  // Loop backward through the children
                    var c = n.removeChild(kids[i]);    // Remove a child
                    n.appendChild(c);                  // Put it back at its new position
                }
            }
        </script>
    </head>
    <body>
        <pre>
            paragraph #1
            paragraph #2
            paragraph #3
        </pre>
        <form>
            <input type="button" value="Click Me to reverse"
onclick="reverse(document.body);"/>
        </form>
    </body>
</html>
```

when the user clicks the button, the order of the paragraphs and of the button are reversed.

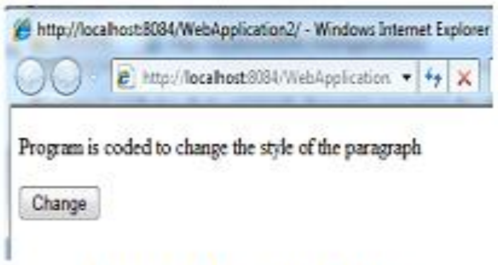


<u>Changing element Style</u>

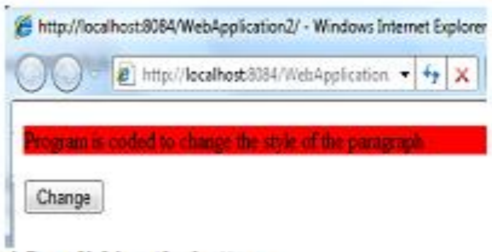
The following program illustrates how to change the element style using DOM properties and methods.

```
<html>
    <head>
        <script type="text/javascript">
            function f1()
            {
              document.getElementById("p1").style.backgroundColor="red";
            }
        </script>
        </head>
    <body>
        <p id="p1"> Program is coded to change the style of the paragraph</p>
        <input type="button" value="Change" onclick="f1();" />
    </body>
</html>
```

The method getElementById() gets the paragraph element in the document and the property style is used to change the background color of the paragraph to “red” as shown below.


http://localhost:8084/WebApplication2/ - Windows Internet Explorer
http://localhost:8084/WebApplication
Change
Program is coded to change the style of the paragraph

Before clicking the button


http://localhost:8084/WebApplication2/ - Windows Internet Explorer
http://localhost:8084/WebApplication
Change
Program is coded to change the style of the paragraph

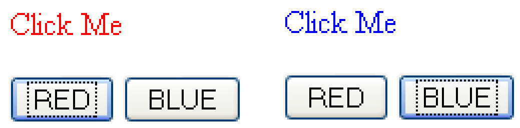
After clicking the button

<u>Changing element style</u>

```
<html>
<head>
<script type="text/javascript">
function f1()
{
            var o=document.getElementById("p1");
            o.style.color="red";
}
function f2()
{
            var o=document.getElementById("p1");
            o.style.color="blue";
}
</script>
</head>
<body>
<p id="p1">
            Click Me
</p>
<form>
<input type=button id="b1" value="RED" onclick=f1() />
```

```
<input type=button id="b2" value="BLUE" onclick=f2() />
</form>
</body>
</html>
```

<u>Output</u>

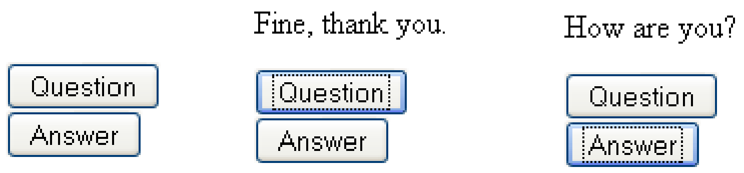


<u>Changing HTML Content</u>

This page shows a example of how to change a HTML page's content

```
<html>
<head>
<script type="text/javascript">
function f1()
{

        document.getElementById("p1").childNodes[0].nodeValue="Fine, thank you.";
}
function f2()
{
        document.getElementById("p1").childNodes[0].nodeValue="How are you?";
}
</script>
</head>
<body id="p1">
<pre>
<input type="button" id="b1" value="Question" onclick=f1() />
<input type="button" id="b2" value="Answer" onclick=f2() />
</pre>
</body>
</html>
```

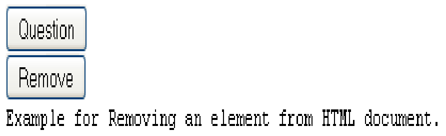
After pressing the Question button, it adds the content, How are you?" to the HTML document and after pressing the Answer button, it replaces the content "How are you?" with "Fine, thank you"



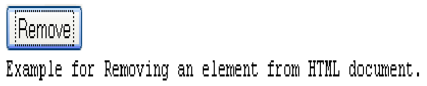
Removing Element from HTML documents

```
<html>
<head>
<script type="text/javascript">
function f1()
{
	var node=document.getElementById("p1");
	node.removeChild(node.childNodes[0]);
}
</script>
</head>
<body >
<pre id="p1"><input type="button" id="b1" value="Question" />
<input type="button" id="b2" value="Remove" onclick=f1() />
Example for Removing an element from HTML document.
</pre>
</body>
</html>
```

After pressing the “Remove” button, the element “Question” is removed from the document.



Example for Removing an element from HTML document.



Example for Removing an element from HTML document.

Server-side Programming: Servlet

The combination of

- HTML
- JavaScript
- DOM

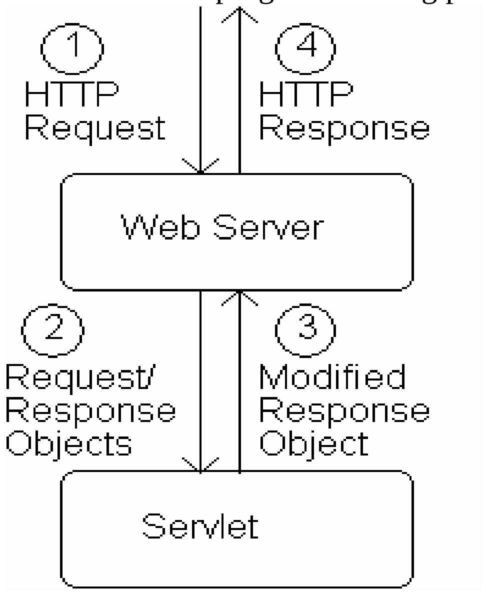
is sometimes referred to as Dynamic HTML (DHTML). Web pages that include scripting are often called dynamic pages. Similarly, web server response can be static or dynamic

- Static: **HTML document is retrieved** from the file system by the server and the same returned to the client.
- Dynamic: In server, a **HTML document is generated** by a program in response to an HTTP request

Java servlets are one technology for producing dynamic server responses. Servlet is a class instantiated by the server to produce a dynamic response.

Servlet Overview

The following figure illustrates the servlet program working principle.



1. When server starts it instantiates servlets
2. Server receives HTTP request, determines need for dynamic response
3. Server selects the appropriate servlet to generate the response, creates request/response objects, and passes them to a method on the servlet instance
4. Servlet adds information to response object via method calls

5. Server generates HTTP response based on information stored in response object

Types of Servlet

- Generic Servlet
- HttpServlet

Servlets vs. Java Applications

- Servlets do not have a main() method
- Entry point to servlet code is via call to a method doGet() /doPost()
- Servlet interaction with end user is indirect via request/response object APIs
- Primary servlet output is typically HTML

Running Servlets

1. Compile servlet (make sure that JWSDP libraries are on path)
2. Copy .class file to **shared/classes** directory
3. (Re)start the Tomcat web server
4. If the class is named ServletHello, browse to http://localhost:8080/servlet/ServletHello

What are Servlets?

Java Servlets are programs that run on a Web or Application server and act as a middle layer between a request coming from a Web browser or other HTTP client and databases or applications on the HTTP server.

Using Servlets, you can collect input from users through web page forms, present records from a database or another source, and create web pages dynamically.

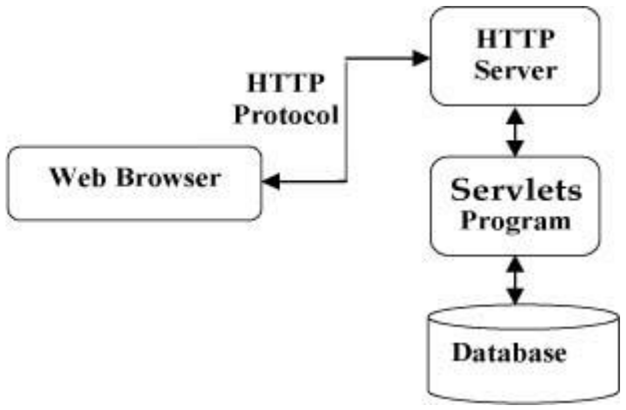
Java Servlets often serve the same purpose as programs implemented using the Common Gateway Interface (CGI). But Servlets offer several advantages in comparison with the CGI.

- Performance is significantly better.
- Servlets execute within the address space of a Web server. It is not necessary to create a separate process to handle each client request.
- Servlets are platform-independent because they are written in Java.
- Java security manager on the server enforces a set of restrictions to protect the resources on a server machine. So servlets are trusted.

- The full functionality of the Java class libraries is available to a servlet. It can communicate with applets, databases, or other software via the sockets and RMI mechanisms that you have seen already.

Servlets Architecture:

Following diagram shows the position of Servelts in a Web Application.



Servlets Tasks:

Servlets perform the following major tasks:

1. Read the explicit data sent by the clients (browsers). This includes an HTML form on a Web page or it could also come from an applet or a custom HTTP client program.
2. Read the implicit HTTP request data sent by the clients (browsers). This includes cookies, media types and compression schemes the browser understands, and so forth.
3. Process the data and generate the results. This process may require talking to a database, executing an RMI or CORBA call, invoking a Web service, or computing the response directly.
4. Send the explicit data (i.e., the document) to the clients (browsers). This document can be sent in a variety of formats, including text (HTML or XML), binary (GIF images), Excel, etc.
5. Send the implicit HTTP response to the clients (browsers). This includes telling the browsers or other clients what type of document is being returned (e.g., HTML), setting cookies and caching parameters, and other such tasks.

Servlets Packages:

Java Servlets are Java classes run by a web server that has an interpreter that supports the Java Servlet specification.

Servlets can be created using the **javax.servlet** and **javax.servlet.http** packages, which are a standard part of the Java's enterprise edition, an expanded version of the Java class library that supports large-scale development projects.

These classes implement the Java Servlet and JSP specifications. At the time of writing this tutorial, the versions are Java Servlet 2.5 and JSP 2.1.

Java servlets have been created and compiled just like any other Java class. After you install the servlet packages and add them to your computer's Classpath, you can compile servlets with the JDK's Java compiler or any other current compiler.

Servlets - Life Cycle

A servlet life cycle can be defined as the entire process from its creation till the destruction. The following are the paths followed by a servlet

- The servlet is initialized by calling the **init ()** method.
- The servlet calls **service()** method to process a client's request.
- The servlet is terminated by calling the **destroy()** method.
- Finally, servlet is garbage collected by the garbage collector of the JVM.

Now let us discuss the life cycle methods in details.

The init() method :

The init method is designed to be called only once. It is called when the servlet is first created, and not called again for each user request. So, it is used for one-time initializations, just as with the init method of applets.

The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started.

When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate. The init() method simply creates or loads some data that will be used throughout the life of the servlet.

The init method definition looks like this:

```
public void init() throws ServletException {
  // Initialization code...
}
```

The service() method :

The service() method is the main method to perform the actual task. The servlet container (i.e. web server) calls the service() method to handle requests coming from the client(browsers) and to write the formatted response back to the client.

Each time the server receives a request for a servlet, the server spawns a new thread and calls service. The service() method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc. methods as appropriate.

Here is the signature of this method:

```
public void service(ServletRequest request,
                  ServletResponse response)
     throws ServletException, IOException{
}
```

The service () method is called by the container and service method invokes doGe, doPost, doPut, doDelete, etc. methods as appropriate. So you have nothing to do with service() method but you override either doGet() or doPost() depending on what type of request you receive from the client.

The doGet() and doPost() are most frequently used methods with in each service request. Here are the signature of these two methods.

The doGet() Method

A GET request results from a normal request for a URL or from an HTML form that has no METHOD specified and it should be handled by doGet() method.

```
public void doGet(HttpServletRequest request,
                HttpServletResponse response)
   throws ServletException, IOException {
   // Servlet code
}
```

The doPost() Method

A POST request results from an HTML form that specifically lists POST as the METHOD and it should be handled by doPost() method.

```
public void doPost(HttpServletRequest request,   HttpServletResponse response)
   throws ServletException, IOException {
   // Servlet code
}
```

The destroy() method :

The destroy() method is called only once at the end of the life cycle of a servlet. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.

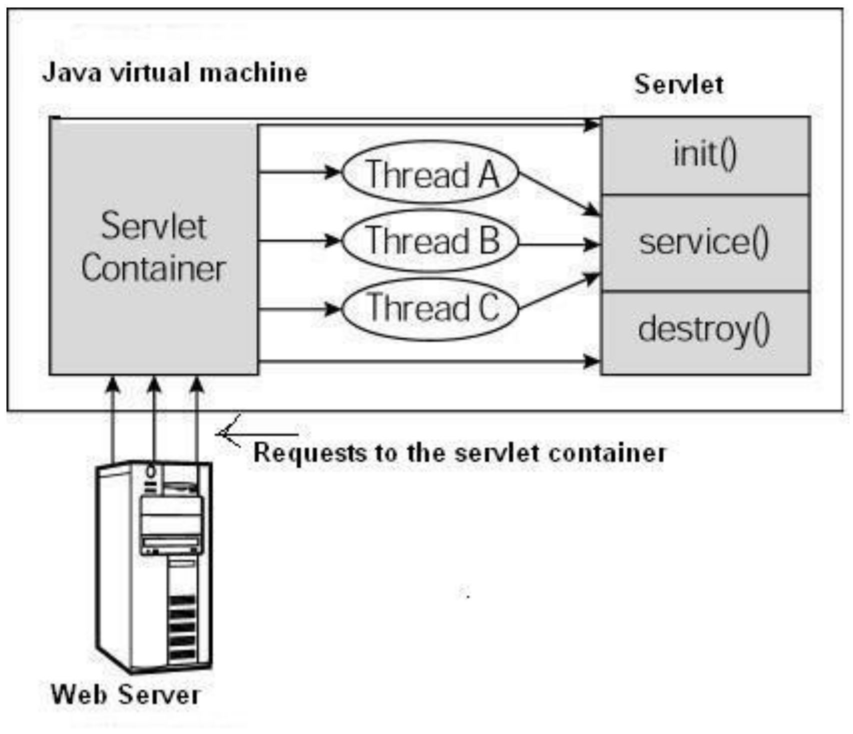
After the destroy() method is called, the servlet object is marked for garbage collection. The destroy method definition looks like this:

```
  public void destroy() {
    // Finalization code...
  }
```

Architecture Diagram:

The following figure depicts a typical servlet life-cycle scenario.

- First the HTTP requests coming to the server are delegated to the servlet container.
- The servlet container loads the servlet before invoking the service() method.
- Then the servlet container handles multiple requests by spawning multiple threads, each thread executing the service() method of a single instance of the servlet.



Structure of a servlet program

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class NewServlet extends HttpServlet
{
public void doGet(HttpServletRequest request, HttpServletResponse response)
          throws ServletException, IOException
{
      response.setContentType("text/html");              // content type of the response
      PrintWriter out = response.getWriter();            // used to create a response as a Html
doc
      try {

          out.println("<html>");
          ---------------------
          ---------------------
          out.println("</html>");

      }catch(Exception e){}
      }
   }
}
```

Servlets - Examples

Servlets are Java classes which service HTTP requests and implement the **javax.servlet.Servlet** interface. Web application developers typically write servlets that extend javax.servlet.http.HttpServlet, an abstract class that implements the Servlet interface and is specially designed to handle HTTP requests.

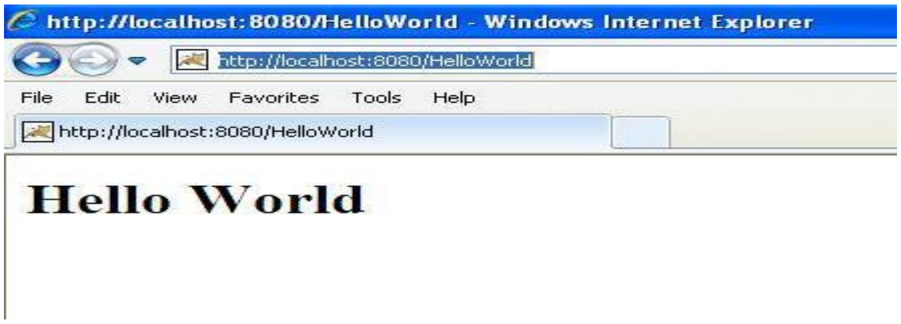
Sample Code for Hello World:

Following is the sample source code structure of a servlet example to write Hello World:

```
// Import required java libraries
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Extend HttpServlet class
public class HelloWorld extends HttpServlet {

  private String message;

  public void init() throws ServletException
  {
      // Do required initialization
      message = "Hello World";
  }

  public void doGet(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException
  {
      // Set response content type
      response.setContentType("text/html");

      // Actual logic goes here.
      PrintWriter out = response.getWriter();
      out.println("<html><body><b>" + message +
“</b></body></html>");
      out.close();
  }

  public void destroy()   {   }
}
```

Output:

finally type **http://localhost:8080/HelloWorld** in browser's address box. If everything goes fine, you would get following result:


http://localhost:8080/HelloWorld - Windows Internet Explorer
http://localhost:8080/HelloWorld
File Edit View Favorites Tools Help
http://localhost:8080/HelloWorld

Hello World

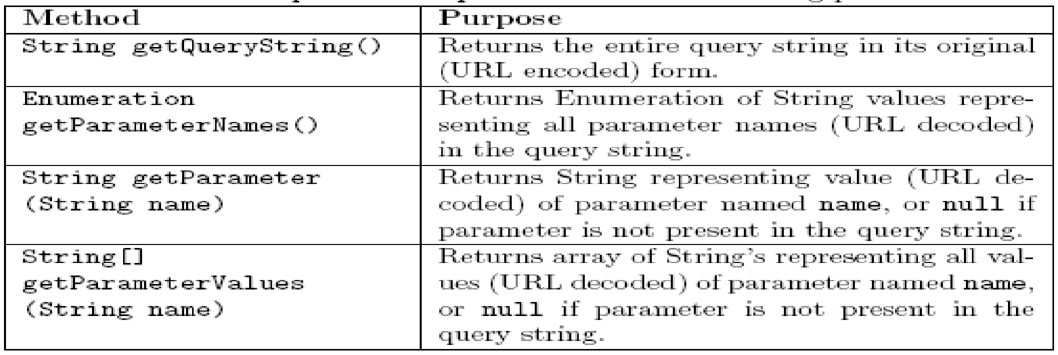
Parameter data and Query Strings

Servlet has methods to access data contained in HTTP Request (URL) sent to the server from the browser. The Query String portion of the HTTP request is so called parameter data. For example,

http://www.example.com/servlet/PrintThis?name=Raj&color=Red

where the portion after the ? is called a query string. Here it is “name=Raj&color=Red”, in which name and color are parameter names and “Raj” and “Red” are parameter values. Printthis is a servlet filename and servelt is a directory. Multiple parameters are separated by &. All parameter values are strings by default. Parameter names and values can be any 8-bit characters.
The following methods are used to process these parameter data in sevlets.

TABLE 6.1: Some `HttpServletRequest` methods for accessing parameter data.



Method	Purpose
`String getQueryString()`	Returns the entire query string in its original (URL encoded) form.
`Enumeration getParameterNames()`	Returns Enumeration of String values representing all parameter names (URL decoded) in the query string.
`String getParameter (String name)`	Returns String representing value (URL decoded) of parameter named `name`, or `null` if parameter is not present in the query string.
`String[] getParameterValues (String name)`	Returns array of String's representing all values (URL decoded) of parameter named `name`, or `null` if parameter is not present in the query string.

The following program explains how to process these parameter names and values as well as path of the resource using servlet.

Example program

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class NewServlet extends HttpServlet
{
public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
{
      response.setContentType("text/html”);
      PrintWriter out = response.getWriter();
      try {

        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet NewServlet</title>");
```

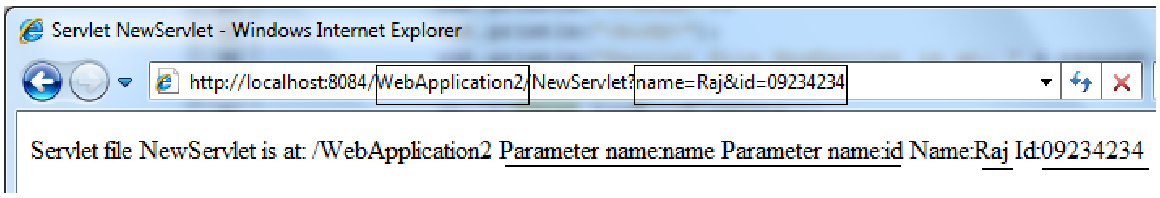
```
            out.println("</head>");
            out.println("<body>");
            out.println("Servlet file NewServlet is at: " + request.getContextPath());
            Enumeration para1=request.getParameterNames();
            while(para1.hasMoreElements())
            {
               out.println("Parameter name:"+para1.nextElement());
            }
            String name = request.getParameter("name");
            String id = request.getParameter("id");
            out.println("Name:" + name);
            out.println("Id:" + id);
            out.println("</body>");
            out.println("</html>");

        }catch(Exception e){}
        }
    }
}
```

The method getContextPath() of HttpServeltRequest object is used to get the location of the resource

The method getParameter() is used to get the value of the parameter. The method getParameterNames() is used to return the paranameter names as well. It returns enumeration. The following code in the above program is used to retrieve the parameter names from the enumeration.

```
            Enumeration para1=request.getParameterNames();
            while(para1.hasMoreElements())
            {
               out.println("Parameter name:"+para1.nextElement());
            }
```

Output:

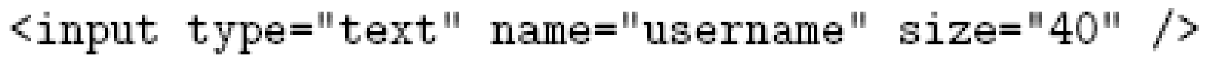

Servlet NewServlet - Windows Internet Explorer

http://localhost:8084/WebApplication2/NewServlet?name=Raj&id=09234234

```
Servlet file NewServlet is at: /WebApplication2 Parameter name:name Parameter name:id Name:Raj Id:09234234
```

<u>Forms and Parameter data:(Passing values from HTML document to Servlet)</u>

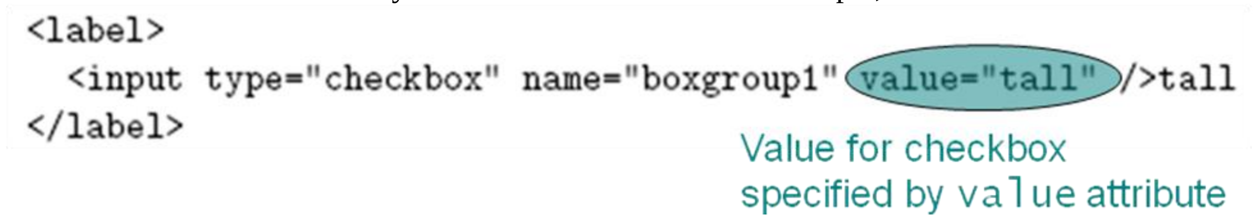
A form automatically generates a query string when submitted. The parameter name specified by value of name attributes of form controls. For example,



```
<input type="text" name="username" size="40" />
```

where username is the parameter name.

Parameter value can be the value of value attribute of any form control or it may be the value received from the user by the control at run time. For example,



```
<label>
  <input type="checkbox" name="boxgroup1" value="tall" />tall
</label>
```

The following program explains how to send the data to server from a web page and the same how to receive it from the server.

<u>Html for creating a web page</u>

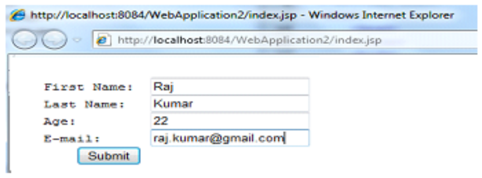
```
<html>
   <head>
   </head>
   <body>
      <pre>
      <form action="NewServlet" method="post">
         First Name:  <input type="text" name="t1" />
         Last Name:   <input type="text" name="t2" />
         Age:         <input type="text" name="t3" />
         E-mail:      <input type="text" name="t4" />
            <input type="submit" value="Submit"  />
      <form>
 </pre>
 </body>
</html>
```

<u>Servlet for processing the data coming from this web page</u>

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class NewServlet extends HttpServlet {

    public void goPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
      response.setContentType("text/html");
```

```
        PrintWriter out = response.getWriter();
        try {

            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet NewServlet</title>");
            out.println("</head>");
            out.println("<body>");

            String s1 = request.getParameter("t1");
            String s2 = request.getParameter("t2");
            String s3 = request.getParameter("t3");
            String s4 = request.getParameter("t4");

            out.println("First Name:" + s1);
            out.println("Last Name:" + s2);
            out.println("Age:" + s3);
            out.println("E-mail:" + s4);

            out.println("</body>");
            out.println("</html>");

        } catch(Exception e) {}
    }
}
```

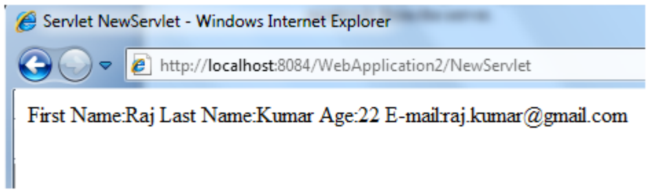
Output



http://localhost:8084/WebApplication2/index.jsp - Windows Internet Explorer

http://localhost:8084/WebApplication2/index.jsp

First Name:	Raj
Last Name:	Kumar
Age:	22
E-mail:	raj.kumar@gmail.com



Servlet NewServlet - Windows Internet Explorer

http://localhost:8084/WebApplication2/NewServlet

```
First Name:Raj Last Name:Kumar Age:22 E-mail:raj.kumar@gmail.com
```

GET vs. POST method for forms:

GET:

- It is used to process the query string which is part of URL
- If the length of query string is limited it may be used.
- It is recommended when parameter data is not stored but used only to request information.

POST:

- It is used to process the query string as well as to store the data on server.

- If the Query string is sent as body of HTTP request, the post method will be used to retrieve.
- If the length of query string is unlimited, it can be used
- It is recommended if parameter data is intended to cause the server to update stored data
- Most browsers will warn you if they are about to resubmit POST data to avoid duplicate updates

Important note:

For the HTTP Session, Cookies, URL rewriting use our class notes. The soft copy for these topics will be given later.