Message.js

```
function msg()
{
 alert("Hello Javatpoint");
}
```

Let's include the JavaScript file into html page. It calls the JavaScript function on button click.

index.html

```
<html>
        <head>
        <script type="text/javascript" src="message.js"></script>
        </head>
        <body>
        <p>Welcome to JavaScript</p>
        <form>
        <input type="button" value="click" onclick="msg()"/>
        </form>
        </body>
</html>
```

You can place an external script reference in <head> or <body> as you like.

The script will behave as if it was located exactly where the <script> tag is located. External scripts cannot contain <script> tags.

External JavaScript Advantages

Placing scripts in external files has some advantages:

- *It separates HTML and code*
- *It makes HTML and JavaScript easier to read and maintain*
- *Cached JavaScript files can speed up page loads*

To add several script files to one page - use several script tags:

Example

```
<script src="myScript1.js"></script>
<script src="myScript2.js"></script>
```

JavaScript Introduction

JavaScript is the programming language of HTML and the web. JavaScript is one of the **3 languages** all web developers **must** learn:

1. **HTML** to define the content of web pages

2. **CSS** to specify the layout of web pages

3. **JavaScript** to program the behaviour of web pages

JavaScript is a dynamic computer programming language. It is lightweight and most commonly used as a part of web pages, whose implementations allow client-side script to interact with the user and make dynamic pages. It is an interpreted programming language with object-oriented capabilities.

JavaScript was first known as **LiveScript,** but Netscape changed its name to JavaScript, possibly because of the excitement being generated by Java. JavaScript made its first appearance in Netscape 2.0 in 1995 with the name **LiveScript**.

Advantages of JavaScript

- **Less server interaction** – You can validate user input before sending the page off to the server. This saves server traffic, which means less load on your server.
- **Immediate feedback to the visitors** – They don't have to wait for a page reload to see if they have forgotten to enter something.
- **Increased interactivity** – You can create interfaces that react when the user hovers over them with a mouse or activates them via the keyboard.
- **Richer interfaces** – You can use JavaScript to include such items as drag-and-drop components and sliders to give a Rich Interface to your site visitors.

Limitations of JavaScript

- Client-side JavaScript does not allow the reading or writing of files. This has been kept for security reason.
- JavaScript cannot be used for networking applications because there is no such support available.
- JavaScript doesn't have any multithreading or multiprocessor capabilities.

JavaScript – Syntax

In HTML, JavaScript code must be inserted between <script> and </script> tags. A simple **syntax** of your JavaScript will appear as follows:

```
<script language=”javascript” type=”text/javascript”>
    JavaScript code
</script>
```

The script tag takes two important attributes –

- **Language** – This attribute specifies what scripting language you are using. Typically, its value will be javascript. Although recent versions of HTML (and XHTML, its successor) have phased out the use of this attribute.
- **Type** – This attribute is what is now recommended to indicate the scripting language in use and its value should be set to "text/javascript".

The type attribute is not required. JavaScript is the default scripting language in HTML.

3 Places to put JavaScript code

1. Between the body tag of html
2. Between the head tag of html
3. In .js file (external javaScript)

1) JavaScript Example : code between the body tag

```
<html>
        <body>
                <script type="text/javascript">
                        alert("Hello JavaScript");
                </script>
        </body>
</html>
```

2) JavaScript Example : code between the head tag

```
<html>
        <head>
                <script type="text/javascript">
                        alert("Hello JavaScript");
                </script>
        </head>
        <body>
                <p> Welcome to JavaScript </p>
        </body>
</html>
```

3) External JavaScript File

We can create external JavaScript file and embed it in many html page.It provides **code re usability** *because single JavaScript file can be used in several html pages.An external JavaScript file must be saved by .js extension. It is recommended to embed all JavaScript files into a single file. It increases the speed of the webpage.*

Let's create an external JavaScript file that prints Hello Javatpoint in a alert dialog box.

This function can be used to write text, HTML, or both. Take a look at the following code.

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

This code will produce the following result:

```
Hello World!
```

Whitespace and Line Breaks

JavaScript ignores spaces, tabs, and newlines that appear in JavaScript programs. You can use spaces, tabs, and newlines freely in your program and you are free to format and indent your programs in a neat and consistent way that makes the code easy to read and understand.

Semicolons are Optional

Simple statements in JavaScript are generally followed by a semicolon character, just as they are in C, C++, and Java. JavaScript, however, allows you to omit this semicolon if each of your statements are placed on a separate line. For example, the following code could be written without semicolons.

```
<script language="javascript" type="text/javascript">
<!--
  var1 = 10
  var2 = 20
//-->
</script>
```

But when formatted in a single line as follows, you must use semicolons:

```
<script language="javascript" type="text/javascript">
<!--
  var1 = 10; var2 = 20;
//-->
</script>
```

Note: It is a good programming practice to use semicolons.

Case Sensitivity

JavaScript is a case-sensitive language. This means that the language keywords, variables, function names, and any other identifiers must always be typed with a consistent capitalization of letters.

So the identifiers **Time** and **TIME** will convey different meanings in JavaScript.

NOTE: Care should be taken while writing variable and function names in JavaScript.

Comments in JavaScript

JavaScript supports both C-style and C++-style comments. Thus:

- Any text between a // and the end of a line is treated as a comment and is ignored by JavaScript.
- Any text between the characters /* and */ is treated as a comment. This may span multiple lines.
- JavaScript also recognizes the HTML comment opening sequence <!--. JavaScript treats this as a single-line comment, just as it does the // comment.
- The HTML comment closing sequence --> is not recognized by JavaScript so it should be written as //-->.

5. VARIABLES

JavaScript Datatypes

One of the most fundamental characteristics of a programming language is the set of data types it supports. These are the type of values that can be represented and manipulated in a programming language.

JavaScript allows you to work with three primitive data types:

- **Numbers**, e.g., 123, 120.50 etc.
- **Strings** of text, e.g. "This text string" etc.
- **Boolean**, e.g. true or false.

JavaScript also defines two trivial data types, **null** and **undefined**, each of which defines only a single value. In addition to these primitive data types, JavaScript supports a composite data type known as **object**. We will cover objects in detail in a separate chapter.

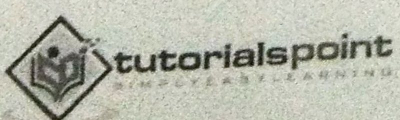
Note: Java does not make a distinction between integer values and floating-point values. All numbers in JavaScript are represented as floating-point values. JavaScript represents numbers using the 64-bit floating-point format defined by the IEEE 754 standard.

JavaScript Variables

Like many other programming languages, JavaScript has variables. Variables can be thought of as named containers. You can place data into these containers and then refer to the data simply by naming the container.

Before you use a variable in a JavaScript program, you must declare it. Variables are declared with the **var** keyword as follows.

```
<script type="text/javascript">
<!--
var money;
var name;
//-->
</script>
```

You can also declare multiple variables with the same **var** keyword as follows:

```
<script type="text/javascript">
<!--
var money, name;
//-->
</script>
```

Storing a value in a variable is called **variable initialization**. You can do variable initialization at the time of variable creation or at a later point in time when you need that variable.

For instance, you might create a variable named **money** and assign the value 2000.50 to it later. For another variable, you can assign a value at the time of initialization as follows.

```
<script type="text/javascript">
<!--
var name = "Ali";
var money;
money = 2000.50;
//-->
</script>
```

Note: Use the **var** keyword only for declaration or initialization, once for the life of any variable name in a document. You should not re-declare same variable twice.

JavaScript is **untyped** language. This means that a JavaScript variable can hold a value of any data type. Unlike many other languages, you don't have to tell JavaScript during variable declaration what type of value the variable will hold. The value type of a variable can change during the execution of a program and JavaScript takes care of it automatically.

JavaScript Variable Scope

The scope of a variable is the region of your program in which it is defined. JavaScript variables have only two scopes.

- **Global Variables:** A global variable has global scope which means it can be defined anywhere in your JavaScript code.
- **Local Variables:** A local variable will be visible only within a function where it is defined. Function parameters are always local to that function.

Within the body of a function, a local variable takes precedence over a global variable with the same name. If you declare a local variable or function parameter with the same name as a global variable, you effectively hide the global variable. Take a look into the following example.

```
<script type="text/javascript">
<!--
var myVar = "global"; // Declare a global variable
function checkscope( ) {
   var myVar = "local";  // Declare a local variable
   document.write(myVar);
}
//-->
</script>
```

It will produce the following result:

```
Local
```

JavaScript Variable Names

While naming your variables in JavaScript, keep the following rules in mind.

- You should not use any of the JavaScript reserved keywords as a variable name. These keywords are mentioned in the next section. For example, **break** or **boolean** variable names are not valid.
- JavaScript variable names should not start with a numeral (0-9). They must begin with a letter or an underscore character. For example, **123test** is an invalid variable name but **_123test** is a valid one.
- JavaScript variable names are case-sensitive. For example, **Name** and **name** are two different variables.

JavaScript Data Types

JavaScript variables can hold many **data types**: numbers, strings, objects and more:

```
var length = 16;                               // Number
var lastName = "Johnson";                      // String
var x = {firstName:"John", lastName:"Doe"};    // Object
```

The Concept of Data Types

In programming, data types is an important concept. To be able to operate on variables, it is important to know something about the type.
Without data types, a computer cannot safely solve this:

```
var x = 16 + "Volvo";
```

Does it make any sense to add "Volvo" to sixteen? Will it produce an error or will it produce a result?
JavaScript will treat the example above as:

```
var x = "16" + "Volvo";
```

When adding a number and a string, JavaScript will treat the number as a string.

Example

```
var x = 16 + "Volvo";
```

Example

```
var x = "Volvo" + 16;
```

JavaScript evaluates expressions from left to right. Different sequences can produce different results:

JavaScript:

```
var x = 16 + 4 + "Volvo";
```

Result:

```
20Volvo
```

JavaScript:

```
var x = "Volvo" + 16 + 4;
```

Result:

```
Volvo164
```

In the first example, JavaScript treats 16 and 4 as numbers, until it reaches "Volvo".
In the second example, since the first operand is a string, all operands are treated as strings.

JavaScript Types are Dynamic.

JavaScript has dynamic types. This means that the same variable can be used to hold different data types:

Example

```
var x;              // Now x is undefined
x = 5;              // Now x is a Number
x = "John";         // Now x is a String
```

JavaScript Strings

A string (or a text string) is a series of characters like "John Doe". Strings are written with quotes. You can use single or double quotes:

Example

```
var carName = "Volvo XC60";   // Using double quotes
var carName = 'Volvo XC60';   // Using single quotes
```

You can use quotes inside a string, as long as they don't match the quotes surrounding the string:

Example

```
var answer = "It's alright";             // Single quote inside
double quotes
var answer = "He is called 'Johnny'";    // Single quotes inside
double quotes
var answer = 'He is called "Johnny"';    // Double quotes inside
single quotes
```

JavaScript Numbers

JavaScript has only one type of numbers. Numbers can be written with, or without decimals:

Example

```
var x1 = 34.00;      // Written with decimals
var x2 = 34;         // Written without decimals
```

Extra large or extra small numbers can be written with scientific (exponential) notation:

Example

```
var y = 123e5;      // 12300000
var z = 123e-5;     // 0.00123
```

You will learn more about numbers later in this tutorial.

JavaScript Booleans

Booleans can only have two values: true or false.

Example

```
var x = 5;
var y = 5;
var z = 6;
(x == y)       // Returns true
(x == z)       // Returns false
```

Booleans are often used in conditional testing.

JavaScript Arrays

JavaScript arrays are written with square brackets. Array items are separated by commas. The following code declares (creates) an array called cars, containing three items (car names):

Example

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

Array indexes are zero-based, which means the first item is [0], second is [1], and so on.

JavaScript Objects

JavaScript objects are written with curly braces. Object properties are written as name:value pairs, separated by commas.

Composite DT: Instead of holding value they hold the address of another memory location

Eg Arrays, functions, Date etc. are all object DT

Example

```
var person = {firstName:"John", lastName:"Doe", age:50,
eyeColor:"blue"};
```

The object (person) in the example above has 4 properties: firstName, lastName, age, and eyeColor.

The typeof Operator

You can use the JavaScript **typeof** operator to find the type of a JavaScript variable.The **typeof** operator returns the type of a variable or an expression:

Example

```
typeof ""                  // Returns "string"
typeof "John"              // Returns "string"
typeof "John Doe"          // Returns "string"
```

Example

```
typeof 0                   // Returns "number"
typeof 314                 // Returns "number"
typeof 3.14                // Returns "number"
typeof (3)                 // Returns "number"
typeof (3 + 4)             // Returns "number"
```

Undefined

In JavaScript, a variable without a value, has the value **undefined**. The typeof is also **undefined**.

Example

```
var car;                   // Value is undefined, type is undefined
```

Any variable can be emptied, by setting the value to **undefined**. The type will also be **undefined**.

Example

```
car = undefined;           // Value is undefined, type is undefined
```

Empty Values

An empty value has nothing to do with undefined. An empty string has both a legal value and a type.

Example

```
var car = "";              // The value is "", the typeof is
"string"
```

Null

In JavaScript null is "nothing". It is supposed to be something that doesn't exist. Unfortunately, in JavaScript, the data type of null is an object.
You can consider it a bug in JavaScript that typeof null is an object. It should be null.
You can empty an object by setting it to null:

Example

```
var person = {firstName:"John", lastName:"Doe", age:50,
eyeColor:"blue"};
person = null;             // Now value is null, but type is still an
object
```

You can also empty an object by setting it to undefined:

Example

```
var person = {firstName:"John", lastName:"Doe", age:50,
eyeColor:"blue"};
person = undefined;   // Now both value and type is undefined
```

Difference Between Undefined and Null

Undefined and null are equal in value but different in type:

```
typeof undefined           // undefined
typeof null                // object

null === undefined         // false
null == undefined          // true
```

Primitive Data

A primitive data value is a single simple data value with no additional properties and methods.

The **typeof** operator can return one of these primitive types:

- string
- number
- boolean
- undefined

Example

```
typeof "John"              // Returns "string"
typeof 3.14                // Returns "number"
typeof true                // Returns "boolean"
typeof false               // Returns "boolean"
typeof x                   // Returns "undefined" (if x has no
value)
```

Complex Data

The **typeof** operator can return one of two complex types:

- function
- object

The typeof operator returns object for both objects, arrays, and null. The typeof operator does not return object for functions.

Example

```
typeof {name:'John', age:34} // Returns "object"
typeof [1,2,3,4]             // Returns "object" (not "array", see
note below)
typeof null                  // Returns "object"
typeof function myFunc(){}   // Returns "function"
```

The typeof operator returns "object" for arrays because in JavaScript arrays are objects.

JavaScript arrays

JavaScript arrays are used to store multiple values in a single variable.

Example

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

What is an Array?

An array is a special variable, which can hold more than one value at a time.
If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:

```
var car1 = "Saab";
var car2 = "Volvo";
var car3 = "BMW";
```

However, what if you want to loop through the cars and find a specific one?
And what if you had not 3 cars, but 300?
The solution is an array!
An array can hold many values under a single name, and you can access the values by referring to an index number.

Creating an Array

Using an array literal is the easiest way to create a JavaScript Array.
Syntax:

```
var array_name = [item1, item2, ...];
```

Example

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

Spaces and line breaks are not important. A declaration can span multiple lines:

Example

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

Using the JavaScript Keyword new

The following example also creates an Array, and assigns values to it:

Example

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

The two examples above do exactly the same. There is no need to use new Array().
For simplicity, readability and execution speed, use the first one (the array literal method).

Access the Elements of an Array

You refer to an array element by referring to the **index number**.
This statement accesses the value of the first element in cars:

```
var name = cars[0];
```

This statement modifies the first element in cars:

```
cars[0] = "Opel";
```

Example

```
var cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars[0];
```

[0] is the first element in an array. [1] is the second. Array indexes start with 0.

Access the Full Array

With JavaScript, the full array can be accessed by referring to the array name:

Example

```
var cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars;
```

Arrays are Objects

Arrays are a special type of objects. The **typeof** operator in JavaScript returns "object" for arrays. But, JavaScript arrays are best described as arrays. Arrays use **numbers** to access its "elements". In this example, **person[0]** returns John:

Array:

```
var person = ["John", "Doe", 46];
```

Objects use **names** to access its "members". In this example, **person.firstName** returns John:

Object:

```
var person = {firstName:"John", lastName:"Doe", age:46};
```

Array Elements Can Be Objects

JavaScript variables can be objects. Arrays are special kinds of objects. Because of this, you can have variables of different types in the same Array. You can have objects in an Array. You can have functions in an Array. You can have arrays in an Array:

```
myArray[0] = Date.now;
myArray[1] = myFunction;
myArray[2] = myCars;
```

Array Properties and Methods

The real strength of JavaScript arrays are the built-in array properties and methods:

Examples

```
var x = cars.length;   // The length property returns the number of
elements
var y = cars.sort();   // The sort() method sorts arrays
```

cars.reverse();

The length Property

The **length** property of an array returns the length of an array (the number of array elements).

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.length;                          // the length of fruits is 4
```

The length property is always one more than the highest array index.

Looping Array Elements

The best way to loop through an array, is using a "for" loop:

Example

```
var fruits, text, fLen, i;

fruits = ["Banana", "Orange", "Apple", "Mango"];
fLen = fruits.length;
text = "<ul>";
for (i = 0; i < fLen; i++) {
    text += "<li>" + fruits[i] + "</li>";
}
```

Adding Array Elements

The easiest way to add a new element to an array is using the push method:

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.push("Lemon");      // adds a new element (Lemon) to fruits
```

New element can also be added to an array using the length property:

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits[fruits.length] = "Lemon";     // adds a new element (Lemon) to
fruits
```

WARNING !
Adding elements with high indexes can create undefined "holes" in an array:

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits[6] = "Lemon";        // adds a new element (Lemon) to fruits
```

Associative Arrays

Many programming languages support arrays with named indexes.
Arrays with named indexes are called associative arrays (or hashes).
JavaScript does **not** support arrays with named indexes.
In JavaScript, **arrays** always use **numbered indexes**.

Example

```
var person = [];
person[0] = "John";
person[1] = "Doe";
person[2] = 46;
```

```
var x = person.length;          // person.length will return 3
var y = person[0];              // person[0] will return "John"
```

WARNING !!
If you use named indexes, JavaScript will redefine the array to a standard object.
After that, some array methods and properties will produce **incorrect results**.

Example:

```
var person = [];
person["firstName"] = "John";
person["lastName"] = "Doe";
person["age"] = 46;
var x = person.length;          // person.length will return 0
var y = person[0];              // person[0] will return undefined
```

The Difference Between Arrays and Objects

In JavaScript, **arrays** use **numbered indexes**.
In JavaScript, **objects** use **named indexes**.
Arrays are a special kind of objects, with numbered indexes.

When to Use Arrays. When to use Objects.

- JavaScript does not support associative arrays.
- You should use **objects** when you want the element names to be **strings (text)**.
- You should use **arrays** when you want the element names to be **numbers**.

3 types of array

1. literal var x = [1,2,3,4];
2. Regular var x = new Array();
   x[0] = 1;
   x[1] = 2;
   x[2] = 3;
3. Condensed var x = new Array(1,2,3);

JavaScript Reserved Words

A list of all the reserved words in JavaScript are given in the following table. ... Script variables, functions, methods, loop labels, or

JavaScript Array Methods

Converting Arrays to Strings

1. The JavaScript method **toString()** converts an array to a string of (comma separated) array values.

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.toString();
```

Result

```
Banana,Orange,Apple,Mango
```

2. The **join()** method also joins all array elements into a string.

It behaves just like toString(), but in addition you can specify the separator:

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.join(" * ");
```

Result

```
Banana * Orange * Apple * Mango
```

3. Popping and Pushing

When you work with arrays, it is easy to remove elements and add new elements.

This is what popping and pushing is:

Popping items **out** of an array, or pushing items **into** an array.

Popping

The **pop()** method removes the last element from an array:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.pop();
```

The pop() method returns the value that was "popped out":

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
var x = fruits.pop();
```

The **push()** method adds a new element to an array (at the end):

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.push("Kiwi");

var x = fruits.push("Pineapple");    //  the value of x is 5
```

Shifting Elements

Shifting is equivalent to popping, working on the first element instead of the last.

The **shift()** method removes the first array element and "shifts" all other elements to a lower index.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.shift();

document.getElementById("demo2").innerHTML = fruits.shift();   // return Banana
```

The **unshift()** method adds a new element to an array (at the beginning), and "unshifts" older elements:

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.unshift("Lemon"); //Adds a new element "Lemon" to fruits & return 5
```

Merging (Concatenating) Arrays

The **concat()** method creates a new array by merging (concatenating) existing arrays:

Example (Merging Two Arrays)

```
var myGirls = ["Cecilie", "Lone"];
var myBoys = ["Emil", "Tobias", "Linus"];
var myChildren = myGirls.concat(myBoys);
```

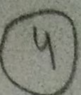
Other:

1) Sorting   fruits.sort();

2) Reverse   fruits.reverse();

3) email.indexOf(@);

4) email.lastIndexOf(.);

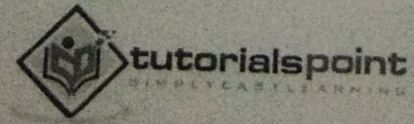


JavaScript Reserved Words

A list of all the reserved words in JavaScript are given in the following table. They cannot be used as JavaScript variables, functions, methods, loop labels, or any object names.

abstract	else	Instanceof	switch
boolean	enum	int	synchronized
break	export	interface	this
byte	extends	long	throw
case	false	native	throws
catch	final	new	transient
char	finally	null	true
class	float	package	try
const	for	private	typeof
continue	function	protected	var
debugger	goto	public	void
default	if	return	volatile
delete	implements	short	while
do	import	static	with
double	in	super	

Operators ⇒



6. OPERATORS

What is an Operator?

Let us take a simple expression **4 + 5 is equal to 9**. Here 4 and 5 are called **operands** and '+' is called the **operator**. JavaScript supports the following types of operators.

- Arithmetic Operators
- Comparison Operators
- Logical (or Relational) Operators
- Assignment Operators
- Conditional (or ternary) Operators

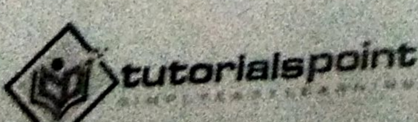
Let's have a look at all the operators one by one.

Arithmetic Operators

JavaScript supports the following arithmetic operators:

Assume variable A holds 10 and variable B holds 20, then:

S. No.	Operator and Description
1	**+ (Addition)** Adds two operands **Ex:** A + B will give 30
2	**- (Subtraction)** Subtracts the second operand from the first **Ex:** A - B will give -10
3	*** (Multiplication)** Multiply both operands **Ex:** A * B will give 200
4	**/ (Division)**



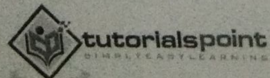
	Divide the numerator by the denominator **Ex:** B / A will give 2
5	**% (Modulus)** Outputs the remainder of an integer division **Ex:** B % A will give 0
6	**++ (Increment)** Increases an integer value by one **Ex:** A++ will give 11
7	**-- (Decrement)** Decreases an integer value by one **Ex:** A-- will give 9

Note: Addition operator (**+**) works for Numeric as well as Strings. e.g. "a" + 10 will give "a10".

Example

The following code shows how to use arithmetic operators in JavaScript.

```
<html>
<body>

<script type="text/javascript">
<!--
var a = 33;
var b = 10;
var c = "Test";
var linebreak = "<br />";

document.write("a + b = ");
result = a + b;
document.write(result);
document.write(linebreak);
```

```
document.write("a - b = ");
result = a - b;
document.write(result);
document.write(linebreak);

document.write("a / b = ");
result = a / b;
document.write(result);
document.write(linebreak);

document.write("a % b = ");
result = a % b;
document.write(result);
document.write(linebreak);

document.write("a + b + c = ");
result = a + b + c;
document.write(result);
document.write(linebreak);

a = a++;
document.write("a++ = ");
result = a++;
document.write(result);
document.write(linebreak);

b = b--;
document.write("b-- = ");
result = b--;
document.write(result);
document.write(linebreak);
```

```
//-->
</script>


<p>Set the variables to different values and then try...</p>
</body>
</html>
```

Output

```
a + b = 43
a - b = 23
a / b = 3.3
a % b = 3
a + b + c = 43Test
a++ = 33
b-- = 10

Set the variables to different values and then try...
```

Comparison Operators

JavaScript supports the following comparison operators:

Assume variable A holds 10 and variable B holds 20, then:

S.No	Operator and Description
1	**== (Equal)** Checks if the value of two operands are equal or not, if yes, then the condition becomes true. **Ex:** (A == B) is not true.
2	**!= (Not Equal)** Checks if the value of two operands are equal or not, if the values are not equal, then the condition becomes true. **Ex:** (A != B) is true.
3	**> (Greater than)** Checks if the value of the left operand is greater than the value of

	the right operand, if yes, then the condition becomes true. **Ex:** (A > B) is not true.
4	**< (Less than)** Checks if the value of the left operand is less than the value of the right operand, if yes, then the condition becomes true. **Ex:** (A < B) is true.
5	**>= (Greater than or Equal to)** Checks if the value of the left operand is greater than or equal to the value of the right operand, if yes, then the condition becomes true. **Ex:** (A >= B) is not true.
6	**<= (Less than or Equal to)** Checks if the value of the left operand is less than or equal to the value of the right operand, if yes, then the condition becomes true. **Ex:** (A <= B) is true.

Example

The following code shows how to use comparison operators in JavaScript.

```
<html>
<body>

<script type="text/javascript">
<!--
var a = 10;
var b = 20;
var linebreak = "<br />";

document.write("(a == b) => ");
result = (a == b);
document.write(result);
document.write(linebreak);
```

```
document.write("(a < b) => ");
result = (a < b);
document.write(result);
document.write(linebreak);

document.write("(a > b) => ");
result = (a > b);
document.write(result);
document.write(linebreak);

document.write("(a != b) => ");
result = (a != b);
document.write(result);
document.write(linebreak);

document.write("(a >= b) => ");
result = (a >= b);
document.write(result);
document.write(linebreak);

document.write("(a <= b) => ");
result = (a <= b);
document.write(result);
document.write(linebreak);


//-->
</script>

<p>Set the variables to different values and different operators and
then try...</p>
</body>
</html>
```

Output

```
(a == b) => false
(a < b) => true
(a > b) => false
(a != b) => true
(a >= b) => false
(a <= b) => true

Set the variables to different values and different operators and then
try...
```

Logical Operators

JavaScript supports the following logical operators:

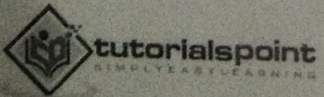
Assume variable A holds 10 and variable B holds 20, then:

S.No	Operator and Description
1	**&& (Logical AND)** If both the operands are non-zero, then the condition becomes true. **Ex:** (A && B) is true.
2	**|| (Logical OR)** If any of the two operands are non-zero, then the condition becomes true. **Ex:** (A || B) is true.
3	**! (Logical NOT)** Reverses the logical state of its operand. If a condition is true, then the Logical NOT operator will make it false. **Ex:** ! (A && B) is false.

Example

Try the following code to learn how to implement Logical Operators in JavaScript.

```
<html>
<body>

<script type="text/javascript">
<!--
var a = true;
var b = false;
var linebreak = "<br />";

document.write("(a &&  b) => ");
result = (a && b);
document.write(result);
document.write(linebreak);

document.write("(a || b) => ");
result = (a || b);
document.write(result);
document.write(linebreak);

document.write("!(a && b) => ");
result = (!(a && b));
document.write(result);
document.write(linebreak);

//-->
</script>



<p>Set the variables to different values and different operators and
then try...</p>
```

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

Output

```
(a && b) => false
(a || b) => true
!(a && b) => true

Set the variables to different values and different operators and then
try...
```

Bitwise Operators

JavaScript supports the following bitwise operators:

Assume variable A holds 2 and variable B holds 3, then:

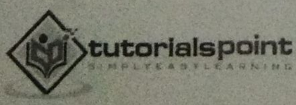
S.No	Operator and Description
1	**& (Bitwise AND)** It performs a Boolean AND operation on each bit of its integer arguments. **Ex:** (A & B) is 2.
2	**| (BitWise OR)** It performs a Boolean OR operation on each bit of its integer arguments. **Ex:** (A | B) is 3.
3	**^ (Bitwise XOR)** It performs a Boolean exclusive OR operation on each bit of its integer arguments. Exclusive OR means that either operand one is true or operand two is true, but not both. **Ex:** (A ^ B) is 1.
4	**~ (Bitwise Not)** It is a unary operator and operates by reversing all the bits in the operand.

	Ex: (~B) is -4.
5	**<< (Left Shift)** It moves all the bits in its first operand to the left by the number of places specified in the second operand. New bits are filled with zeros. Shifting a value left by one position is equivalent to multiplying it by 2, shifting two positions is equivalent to multiplying by 4, and so on. **Ex:** (A << 1) is 4.
6	**>> (Right Shift)** Binary Right Shift Operator. The left operand's value is moved right by the number of bits specified by the right operand. **Ex:** (A >> 1) is 1.
7	**>>> (Right shift with Zero)** This operator is just like the >> operator, except that the bits shifted in on the left are always zero. **Ex:** (A >>> 1) is 1.

Example

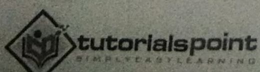
Try the following code to implement Bitwise operator in JavaScript.

```
<html>
<body>

<script type="text/javascript">
<!--
var a = 2;  // Bit presentation 10
var b = 3;  // Bit presentation 11
var linebreak = "<br />";

document.write("(a &  b) => ");
result = (a & b);
document.write(result);
document.write(linebreak);
```

```
document.write("(a | b) => ");
result = (a | b);
document.write(result);
document.write(linebreak);

document.write("(a ^ b) => ");
result = (a ^ b);
document.write(result);
document.write(linebreak);

document.write("(~b) => ");
result = (~b);
document.write(result);
document.write(linebreak);

document.write("(a << b) => ");
result = (a << b);
document.write(result);
document.write(linebreak);

document.write("(a >> b) => ");
result = (a >> b);
document.write(result);
document.write(linebreak);

//-->
</script>

<p>Set the variables to different values and different operators and
then try...</p>
</body>
</html>
```

Output

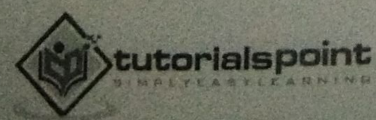


```
(a & b) => 2
(a | b) => 3
(a ^ b) => 1
(~b) => -4
(a << b) => 16
(a >> b) => 0

Set the variables to different values and different operators and then
try...
```

Assignment Operators

JavaScript supports the following assignment operators:

S.No	Operator and Description
1	**= (Simple Assignment)** Assigns values from the right side operand to the left side operand **Ex:** C = A + B will assign the value of A + B into C
2	**+= (Add and Assignment)** It adds the right operand to the left operand and assigns the result to the left operand. **Ex:** C += A is equivalent to C = C + A
3	**-= (Subtract and Assignment)** It subtracts the right operand from the left operand and assigns the result to the left operand. **Ex:** C -= A is equivalent to C = C - A
4	***= (Multiply and Assignment)** It multiplies the right operand with the left operand and assigns the result to the left operand. **Ex:** C *= A is equivalent to C = C * A
5	**/= (Divide and Assignment)** It divides the left operand with the right operand and assigns the result to the left operand.



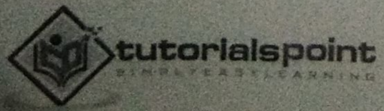
	Ex: C /= A is equivalent to C = C / A
6	**%= (Modules and Assignment)** It takes modulus using two operands and assigns the result to the left operand. **Ex:** C %= A is equivalent to C = C % A

Note: Same logic applies to Bitwise operators, so they will become <<=, >>=, >>=, &=, |= and ^=.

Example

Try the following code to implement assignment operator in JavaScript.

```
<html>
<body>

<script type="text/javascript">
<!--
var a = 33;
var b = 10;
var linebreak = "<br />";

document.write("Value of a => (a = b) => ");
result = (a = b);
document.write(result);
document.write(linebreak);

document.write("Value of a => (a += b) => ");
result = (a += b);
document.write(result);
document.write(linebreak);

document.write("Value of a => (a -= b) => ");
result = (a -= b);
document.write(result);
```

```
document.write(linebreak);

document.write("Value of a => (a *=  b) => ");
result = (a *= b);
document.write(result);
document.write(linebreak);

document.write("Value of a => (a /= b) => ");
result = (a /= b);
document.write(result);
document.write(linebreak);

document.write("Value of a => (a %= b) => ");
result = (a %= b);
document.write(result);
document.write(linebreak);

//-->
</script>

<p>Set the variables to different values and different operators and
then try...</p>
</body>
</html>
```

Output

```
Value of a => (a = b) => 10
Value of a => (a += b) => 20
Value of a => (a -= b) => 10
Value of a => (a *= b) => 100
Value of a => (a /= b) => 10
Value of a => (a %= b) => 0
```

```
Set the variables to different values and different operators and then
try...
```

Miscellaneous Operators

We will discuss two operators here that are quite useful in JavaScript: the **conditional operator (? :)** and the **typeof operator**.

Conditional Operator (? :)

The conditional operator first evaluates an expression for a true or false value and then executes one of the two given statements depending upon the result of the evaluation.

S.No	Operator and Description
1	**? : (Conditional)** If Condition is true? Then value X : Otherwise value Y

Example

Try the following code to understand how the Conditional Operator works in JavaScript.

```
<html>
<body>

<script type="text/javascript">
<!--
var a = 10;
var b = 20;
var linebreak = "<br />";

document.write ("((a > b) ? 100 : 200) => ");
result = (a > b) ? 100 : 200;
document.write(result);
document.write(linebreak);

document.write ("((a < b) ? 100 : 200) => ");
```

```
result = (a < b) ? 100 : 200;
document.write(result);
document.write(linebreak);


//-->
</script>

<p>Set the variables to different values and different operators and
then try...</p>
</body>
</html>
```

Output

```
((a > b) ? 100 : 200) => 200
((a < b) ? 100 : 200) => 100

Set the variables to different values and different operators and then
try...
```

typeof Operator

The **typeof** operator is a unary operator that is placed before its single operand, which can be of any type. Its value is a string indicating the data type of the operand.

The *typeof* operator evaluates to "number", "string", or "boolean" if its operand is a number, string, or boolean value and returns true or false based on the evaluation.

Here is a list of the return values for the **typeof** Operator.

Type	String Returned by typeof
Number	"number"
String	"string"
Boolean	"boolean"
Object	"object"

Function	"function"
Undefined	"undefined"
Null	"object"

Example

The following code shows how to implement **typeof** operator.

```
<html>
<body>

<script type="text/javascript">
<!--
var a = 10;
var b = "String";
var linebreak = "<br />";

result = (typeof b == "string" ? "B is String" : "B is Numeric");
document.write("Result => ");
document.write(result);
document.write(linebreak);

result = (typeof a == "string" ? "A is String" : "A is Numeric");
document.write("Result => ");
document.write(result);
document.write(linebreak);

//-->
</script>

<p>Set the variables to different values and different operators and
then try...</p>
</body>
</html>
```

Output

```
Result => B is String
Result => A is Numeric


Set the variables to different values and different operators and then
try...
```

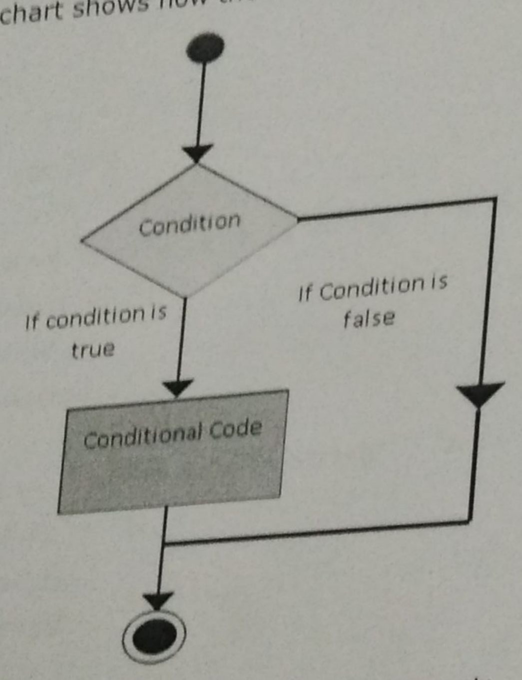
7. IF-ELSE

While writing a program, there may be a situation when you need to adopt one out of a given set of paths. In such cases, you need to use conditional statements that allow your program to make correct decisions and perform right actions.

JavaScript supports conditional statements which are used to perform different actions based on different conditions. Here we will explain the **if..else** statement.

Flow Chart of if-else

The following flow chart shows how the if-else statement works.



JavaScript supports the following forms of **if..else** statement:

- if statement
- if...else statement
- if...else if... statement

Syntax:

if (expression)
{
statement if expression is true
}

8. switch-case
9. while loop
10. do while loop
11. for loop
12. loop control
→ break
→ continue

13. FUNCTIONS

A function is a group of reusable code which can be called anywhere in your program. This eliminates the need of writing the same code again and again. It helps programmers in writing modular codes. Functions allow a programmer to divide a big program into a number of small and manageable functions.

Like any other advanced programming language, JavaScript also supports all the features necessary to write modular code using functions. You must have seen functions like **alert()** and **write()** in the earlier chapters. We were using these functions again and again, but they had been written in core JavaScript only once.

JavaScript allows us to write our own functions as well. This section explains how to write your own functions in JavaScript.

Function Definition

Before we use a function, we need to define it. The most common way to define a function in JavaScript is by using the **function** keyword, followed by a unique function name, a list of parameters (that might be empty), and a statement block surrounded by curly braces.

Syntax

The basic syntax is shown here.

```
<script type="text/javascript">
<!--
function functionname(parameter-list)
{
  statements
}
//-->
</script>
```

Example

Try the following example. It defines a function called sayHello that takes no parameters:

```
<script type="text/javascript">
<!--
function sayHello()
{
   alert("Hello there");
}
//-->
</script>
```

Calling a Function

To invoke a function somewhere later in the script, you would simply need to write the name of that function as shown in the following code.

```
<html>
<head>
<script type="text/javascript">
function sayHello()
{
  document.write ("Hello there!");
}
</script>
</head>

<body>
<p>Click the following button to call the function</p>
<form>
<input type="button" onclick="sayHello()" value="Say Hello">
</form>

<p>Use different text in write method and then try...</p>
</body>
</html>
```

called function

→ calling function

Output

```
Click the following button to call the function

Say Hello
```

Function Parameters

Till now, we have seen functions without parameters. But there is a facility to pass different parameters while calling a function. These passed parameters can be captured inside the function and any manipulation can be done over those parameters. A function can take multiple parameters separated by comma.

Example

Try the following example. We have modified our **sayHello** function here. Now it takes two parameters.

```
<html>
<head>
<script type="text/javascript">
function sayHello(name, age)
{
   document.write (name + " is " + age + " years old.");
}
</script>
</head>

<body>
<p>Click the following button to call the function</p>
<form>
<input type="button" onclick="sayHello('Zara', 7)" value="Say Hello">
</form>

<p>Use different parameters inside the function and then try...</p>
</body>
</html>
```

Output

```
Click the following button to call the function

[Say Hello]

Use different parameters inside the function and then try...
```

The return Statement

A JavaScript function can have an optional **return** statement. This is required if you want to return a value from a function. This statement should be the last statement in a function.

For example, you can pass two numbers in a function and then you can expect the function to return their multiplication in your calling program.

Example

Try the following example. It defines a function that takes two parameters and concatenates them before returning the resultant in the calling program.

```
<html>
<head>
<script type="text/javascript">
function concatenate(first, last)
{
   var full;

   full = first + last;
   return  full;
}
function secondFunction()
{
   var result;
   result = concatenate('Zara', 'Ali');
   document.write (result );
}
</script>
</head>
```

```
<body>
<p>Click the following button to call the function</p>
<form>
<input type="button" onclick="secondFunction()" value="Call Function">
</form>

<p>Use different parameters inside the function and then try...</p>
</body>
</html>
```

Output

```
Click the following button to call the function
[Call Function]

Use different parameters inside the function and then try...
```

There is a lot to learn about JavaScript functions, however we have covered the most important concepts in this tutorial.

Nested Functions

Prior to JavaScript 1.2, function definition was allowed only in top level global code, but JavaScript 1.2 allows function definitions to be nested within other functions as well. Still there is a restriction that function definitions may not appear within loops or conditionals. These restrictions on function definitions apply only to function declarations with the function statement.

As we'll discuss later in the next chapter, function literals (another feature introduced in JavaScript 1.2) may appear within any JavaScript expression, which means that they can appear within **if** and other statements.

Example

Try the following example to learn how to implement nested functions.

```
<html>
```

```
<head>
<script type="text/javascript">
<!--
function hypotenuse(a, b) {
   function square(x) { return x*x; }

   return Math.sqrt(square(a) + square(b));
}
function secondFunction(){
   var result;
   result = hypotenuse(1,2);
   document.write ( result );
}
//-->
</script>
</head>

<body>
<p>Click the following button to call the function</p>

<form>
<input type="button" onclick="secondFunction()" value="Call Function">
</form>

<p>Use different parameters inside the function and then try...</p>
</body>
</html>
```

Output

Click the following button to call the function

Call Function

```
Use different parameters inside the function and then try...
```

Function () Constructor

The *function* statement is not the only way to define a new function; you can define your function dynamically using **Function()** constructor along with the **new** operator.

Note: Constructor is a terminology from Object Oriented Programming. You may not feel comfortable for the first time, which is OK.

Syntax

Following is the syntax to create a function using **Function()** constructor along with the **new** operator.

```
<script type="text/javascript">
<!--
var variablename = new Function(Arg1, Arg2..., "Function Body");
//-->
</script>
```

The **Function()** constructor expects any number of string arguments. The last argument is the body of the function – it can contain arbitrary JavaScript statements, separated from each other by semicolons.

Notice that the **Function()** constructor is not passed any argument that specifies a name for the function it creates. The **unnamed** functions created with the **Function()** constructor are called **anonymous** functions.

Example

Try the following example.

```
<html>
<head>
<script type="text/javascript">
<!--
var func = new Function("x", "y", "return x*y;");

function secondFunction(){
   var result;
   result = func(10,20);
```

```
    document.write ( result );
}
//-->
</script>
</head>

<body>
<p>Click the following button to call the function</p>

<form>
<input type="button" onclick="secondFunction()" value="Call Function">
</form>

<p>Use different parameters inside the function and then try...</p>
</body>
</html>
```

Output

Click the following button to call the function

Call Function

Use different parameters inside the function and then try...

Function Literals

JavaScript 1.2 introduces the concept of **function literals** which is another new way of defining functions. A function literal is an expression that defines an unnamed function.

Syntax

The syntax for a **function literal** is much like a function statement, except that it is used as an expression rather than a statement and no function name is required.

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

```
var variablename = function(Argument List){
                    Function Body
  };
//-->
</script>
```

Syntactically, you can specify a function name while creating a literal function as follows.

```
<script type="text/javascript">
<!--
var variablename = function FunctionName(Argument List){
                    Function Body
        };
//-->
</script>
```

But this name does not have any significance, so it is not worthwhile.

Example

Try the following example. It shows the usage of function literals.

```
<html>
<head>
<script type="text/javascript">
<!--
var func = function(x,y){ return x*y };

function secondFunction(){
    var result;
    result = func(10,20);
    document.write ( result );
}
//-->
</script>
</head>
<body>
```

```
<p>Click the following button to call the function</p>
<form>
<input type="button" onclick="secondFunction()" value="Call Function">
</form>
<p>Use different parameters inside the function and then try...</p>
</body>
</html>
```

Output

Click the following button to call the function

Call Function

Use different parameters inside the function and then try...

30. FORM VALIDATION

Form validation normally used to occur at the server, after the client had entered all the necessary data and then pressed the Submit button. If the data entered by a client was incorrect or was simply missing, the server would have to send all the data back to the client and request that the form be resubmitted with correct information. This was really a lengthy process which used to put a lot of burden on the server.

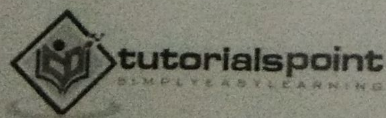
JavaScript provides a way to validate form's data on the client's computer before sending it to the web server. Form validation generally performs two functions.

- **Basic Validation** - First of all, the form must be checked to make sure all the mandatory fields are filled in. It would require just a loop through each field in the form and check for data.
- **Data Format Validation** - Secondly, the data that is entered must be checked for correct form and value. Your code must include appropriate logic to test correctness of data.

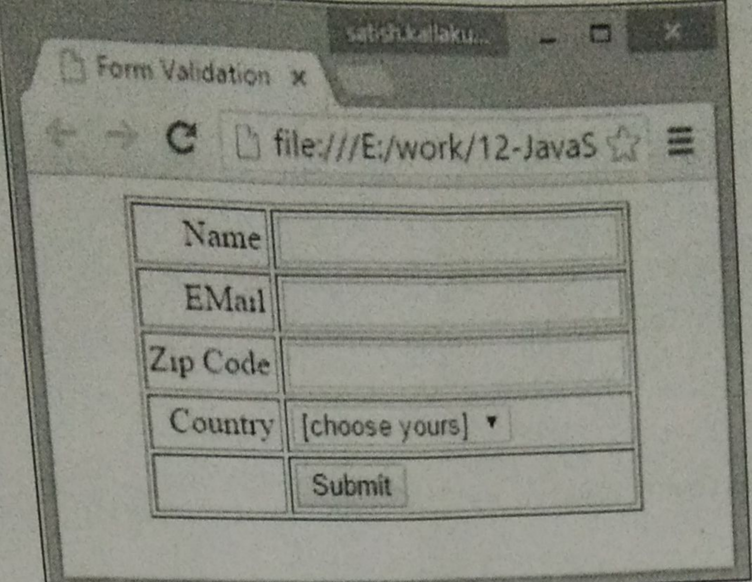
Example

We will take an example to understand the process of validation. Here is a simple form in html format.

```
<html>
<head>
<title>Form Validation</title>
<script type="text/javascript">
<!--
// Form validation code will come here.
//-->
</script>
</head>
<body>
<form action="/cgi-bin/test.cgi" name="myForm"
         onsubmit="return(validate());">
<table cellspacing="2" cellpadding="2" border="1">
<tr>
  <td align="right">Name</td>
```

```
   <td><input type="text" name="Name" /></td>
  </tr>
  <tr>
    <td align="right">EMail</td>
    <td><input type="text" name="EMail" /></td>
  </tr>
  <tr>
    <td align="right">Zip Code</td>
    <td><input type="text" name="Zip" /></td>
  </tr>
  <tr>
  <td align="right">Country</td>
  <td>
  <select name="Country">
    <option value="-1" selected>[choose yours]</option>
    <option value="1">USA</option>
    <option value="2">UK</option>
    <option value="3">INDIA</option>
  </select>
  </td>
  </tr>
  <tr>
    <td align="right"></td>
    <td><input type="submit" value="Submit" /></td>
  </tr>
  </table>
  </form>
  </body>
  </html>
```



Basic Form Validation

First let us see how to do a basic form validation. In the above form, we are calling **validate()** to validate data when **onsubmit** event is occurring. The following code shows the implementation of this validate() function.

```
<script type="text/javascript">
<!--
// Form validation code will come here.
function validate()
{

   if( document.myForm.Name.value == "" )
   {
     alert( "Please provide your name!" );
     document.myForm.Name.focus() ;
     return false;
   }
   if( document.myForm.EMail.value == "" )
   {
     alert( "Please provide your Email!" );
     document.myForm.EMail.focus() ;
     return false;
```

```
    }
    if( document.myForm.Zip.value == "" ||
            isNaN( document.myForm.Zip.value ) ||
            document.myForm.Zip.value.length != 5 )
    {
      alert( "Please provide a zip in the format #####." );
      document.myForm.Zip.focus() ;
      return false;
    }
    if( document.myForm.Country.value == "-1" )
    {
      alert( "Please provide your country!" );
      return false;
    }
    return( true );
}
//-->
</script>
```

Data Format Validation

Now we will see how we can validate our entered form data before submitting it to the web server.

The following example shows how to validate an entered email address. An email address must contain at least a '@' sign and a dot (.). Also, the '@' must not be the first character of the email address, and the last dot must at least be one character after the '@' sign.

Example

Try the following code for email validation.

```
<script type="text/javascript">
<!--
function validateEmail()
{
```

```
var emailID = document.myForm.EMail.value;
atpos = emailID.indexOf("@");
dotpos = emailID.lastIndexOf(".");
if (atpos < 1 || ( dotpos - atpos < 2 ))
{
    alert("Please enter correct email ID")
    document.myForm.EMail.focus() ;
    return false;
}
return( true );
}
//-->
</script>
```

@gmail.com