4Introduction

- ⇨ JavaScript was invented by Brendan Eich at Netscape (with Navigator 2.0), and has appeared in all browsers since 1996.
- ⇨ Javascript is from Netscape
- ⇨ javaScript initially it was called as LiveScript
- ⇨ JavaScript is a lightweight, interpreted programming language that allows you to build interactivity into otherwise static HTML pages.
- ⇨ JavaScript is used in billions of Web pages to add functionality, validate forms, communicate with the server, and much more.
- ⇨ JavaScript is the most popular scripting language on the internet, and works in all major browsers, such as Internet Explorer, Firefox, Chrome, Opera, and Safari.

What You Should Already Know

Before you continue you should have a basic understanding of the following:

- HTML and CSS

What is JavaScript?

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

Are Java and JavaScript the same?

NO!

Java and JavaScript are two completely different languages in both concept and design!

What Can JavaScript do?

- **JavaScript gives HTML designers a programming tool** - HTML authors are normally not programmers, but JavaScript is a scripting language with a very simple syntax! Almost anyone can put small "snippets" of code into their HTML pages
- **JavaScript can react to events** - A JavaScript can be set to execute when something happens, like when a page has finished loading or when a user clicks on an HTML element
- **JavaScript can read and write HTML elements** - A JavaScript can read and change the content of an HTML element
- **JavaScript can be used to validate data** - A JavaScript can be used to validate form data before it is submitted to a server. This saves the server from extra processing

- **JavaScript can be used to detect the visitor's browser** - A JavaScript can be used to detect the visitor's browser, and - depending on the browser - load another page specifically designed for that browser
- **JavaScript can be used to create cookies** - A JavaScript can be used to store and retrieve information on the visitor's computer

Limitations with JavaScript:

We can not treat JavaScript as a full fledged programming language. It lacks the following important features:

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

JavaScript Development Tools:

One of JavaScript's strengths is that expensive development tools are not usually required. You can start with a simple text editor such as Notepad.

Since it is an interpreted language inside the context of a web browser, you don't even need to buy a compiler.

To make our life simpler, various vendors have come up with very nice JavaScript editing tools. Few of them are listed here:

- **Microsoft FrontPage:** Microsoft has developed a popular HTML editor called FrontPage. FrontPage also provides web developers with a number of JavaScript tools to assist in the creation of an interactive web site.
- **Macromedia Dreamweaver MX:** Macromedia Dreamweaver MX is a very popular HTML and JavaScript editor in the professional web development crowd. It provides several handy prebuilt JavaScript components, integrates well with databases, and conforms to new standards such as XHTML and XML.
- **Macromedia HomeSite 5:** This provided a well-liked HTML and JavaScript editor, which will manage their personal web site just fine.

Where JavaScript is Today ?

The ECMAScript is the standard for scripting languages

The JavaScript 2.0 adheres to ECMAScript Edition 4.

Today, Netscape's JavaScript and Microsoft's JScript conform to the ECMAScript standard

JavaScript Syntax

A JavaScript consists of JavaScript statements that are placed within the <script>... </script> HTML tags in a web page.

You can place the <script> tag containing your JavaScript anywhere within you web page but it is preferred way to keep it within the <head> tags.

The <script> tag alert the browser program to begin interpreting all the text between these tags as a script. So simple syntax of your JavaScript will be as follows

```
<script ...>
  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"*.

So your JavaScript segment will look like:

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

Your First JavaScript Script:

Let us write our class example to print out "Hello World".

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

Next, we call a function *document.write* which writes a string into our HTML document. This function can be used to write text, HTML, or both. So above code will display following result:

```
Hello World!
```

Whitespace and Line Breaks:

JavaScript ignores spaces, tabs, and newlines that appear in JavaScript programs.

Because you can use spaces, tabs, and newlines freely in your program so 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 your statements are each 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, the semicolons are required:

```
<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 language keywords, variables, function names, and any other identifiers must always be typed with a consistent capitalization of letters.

So identifiers *Time*, *TIme* and *TIME* will have different meanings in JavaScript.

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

Some Browsers do Not Support JavaScript

Browsers that do not support JavaScript, will display JavaScript as page content.

To prevent them from doing this, and as a part of the JavaScript standard, the HTML comment tag should be used to "hide" the JavaScript.

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

```
<html>
<body>
<script type="text/javascript">
<!--
document.getElementById("demo").innerHTML=Date();
//-->
</script>
```

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

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

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 //-->.

Example:

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

// This is a comment. It is similar to comments in C++

/*
 * This is a multiline comment in JavaScript
 * It is very similar to comments in C Programming
 */
//-->
</script>
```

Enabling JavaScript in Browsers

All the modern browsers come with built-in support for JavaScript. Many times you may need to enable or disable this support manually.

JavaScript in Internet Explorer:

Here are simple steps to turn on or turn off JavaScript in your Internet Explorer:

1. Follow **Tools-> Internet Options** from the menu
2. Select **Security** tab from the dialog box
3. Click the **Custom Level** button
4. Scroll down till you find **Scripting option**
5. Select *Enable* radio button under **Active scripting**
6. Finally click OK and come out

To disable JavaScript support in your Internet Explorer, you need to select *Disable* radio button under **Active scripting**.

JavaScript in Firefox:

Here are simple steps to turn on or turn off JavaScript in your Firefox:

1. Follow **Tools-> Options**

   from the menu

2. Select **Content** option from the dialog box
3. Select *Enable JavaScript* checkbox
4. Finally click OK and come out

To disable JavaScript support in your Firefox, you should not select *Enable JavaScript* checkbox.

JavaScript in Opera:

Here are simple steps to turn on or turn off JavaScript in your Opera:

1. Follow **Tools-> Preferences**

   from the menu

2. Select **Advanced** option from the dialog box
3. Select **Content** from the listed items
4. Select *Enable JavaScript* checkbox
5. Finally click OK and come out

To disable JavaScript support in your Opera, you should not select *Enable JavaScript* checkbox.

Warning for Non-JavaScript Browsers:

If you have to do something important using JavaScript then you can display a warning message to the user using <noscript> tags.

You can add a *noscript* block immediately after the script block as follows:

```
<html>
<body>

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

<noscript>
  Sorry...JavaScript is needed to go ahead.
</noscript>
</body>
</html>
```

Now, if user's browser does not support JavaScript or JavaScript is not enabled then message from </noscript> will be displayed on the screen.

JavaScript Placement in HTML File

There is a flexibility given to include JavaScript code anywhere in an HTML document. But there are following most preferred ways to include JavaScript in your HTML file.

- Script in <head>...</head> section.
- Script in <body>...</body> section.
- Script in <body>...</body> and <head>...</head> sections.
- Script in and external file and then include in <head>...</head> section.

In the following section we will see how we can put JavaScript in different ways:

JavaScript in <head>...</head> section:

If you want to have a script run on some event, such as when a user clicks somewhere, then you will place that script in the head as follows:

```
<html>
<head>
<script type="text/javascript">
<!--
function sayHello() {
   alert("Hello World")
}
//-->
</script>
</head>
<body>
<input type="button" onclick="sayHello()" value="Say Hello" />
</body>
</html>
```

JavaScript in <body>...</body> section:

If you need a script to run as the page loads so that the script generates content in the page, the script goes in the <body> portion of the document. In this case you would not have any function defined using JavaScript:

```
<html>
<head>
</head>
<body>
<script type="text/javascript">
<!--
document.write("Hello World")
//-->
</script>
<p>This is web page body </p>
</body>
</html>
```

JavaScript in <body> and <head> sections:

You can put your JavaScript code in <head> and <body> section altogether as follows:

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

```
<script type="text/javascript">
<!--
function sayHello() {
   alert("Hello World")
}
//-->
</script>
</head>
<body>
<script type="text/javascript">
<!--
document.write("Hello World")
//-->
</script>
<input type="button" onclick="sayHello()" value="Say Hello" />
</body>
</html>
```

JavaScript in External File :

As you begin to work more extensively with JavaScript, you will likely find that there are cases where you are reusing identical JavaScript code on multiple pages of a site.

You are not restricted to be maintaining identical code in multiple HTML files. The *script* tag provides a mechanism to allow you to store JavaScript in an external file and then include it into your HTML files.

Here is an example to show how you can include an external JavaScript file in your HTML code using *script* tag and its *src* attribute:

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

To use JavaScript from an external file source, you need to write your all JavaScript source code in a simple text file with extension ".js" and then include that file as shown above.

For example, you can keep following content in filename.js file and then you can use *sayHello* function in your HTML file after including filename.js file:

```
function sayHello() {
   alert("Hello World")
}
```

JavaScript Variables and DataTypes

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 eg. 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 see an object 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 later point in time when you need that variable as follows:

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 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 variable will have only two scopes.

- **Global Variables:** A global variable has global scope which means it is defined everywhere 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. Following example explains it:

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

JavaScript Variable Names:

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

- You should not use any of the JavaScript reserved keyword as 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 the 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 Reserved Words:

The following are reserved words in JavaScript. 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	

JavaScript Operators

What is an operator?

Simple answer can be given using expression *4 + 5 is equal to 9*. Here 4 and 5 are called operands and + is called operator. JavaScript language supports following type of operators.

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

Lets have a look on all operators one by one.

The Arithmatic Operators:

There are following arithmatic operators supported by JavaScript language:

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

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiply both operands	A * B will give 200
/	Divide numerator by denumerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

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

The Comparison Operators:

There are following comparison operators supported by JavaScript language

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

Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.

>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

The Logical Operators:

There are following logical operators supported by JavaScript language

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

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non zero then then condition becomes true.	(A && B) is true.
||	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is false.

The Bitwise Operators:

There are following bitwise operators supported by JavaScript language

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

Operator	Description	Example
&	Called Bitwise AND operator. It performs a Boolean AND operation on each bit of its integer arguments.	(A & B) is 2 .
|	Called Bitwise OR Operator. It performs a Boolean OR operation on each bit of its integer arguments.	(A | B) is 3.
^	Called Bitwise XOR Operator. 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.	(A ^ B) is 1.
~	Called Bitwise NOT Operator. It is a is a unary operator and operates by reversing all bits in the operand.	(~B) is -4 .
<<	Called Bitwise Shift Left Operator. It moves all 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 by 2, shifting two positions is equivalent to multiplying by 4, etc.	(A << 1) is 4.
>>	Called Bitwise Shift Right with Sign Operator. It moves all bits in its first operand to the right by the number of places specified in the second operand. The bits filled in on the left depend on the sign bit of the original operand, in order to preserve the sign of the result. If the first operand is positive, the result has zeros placed in the high bits; if the first operand is negative, the result has ones placed in the high bits. Shifting a value right one place is equivalent to dividing by 2 (discarding the remainder), shifting right two places is equivalent to integer division by 4, and so on.	(A >> 1) is 1.

>>>	Called Bitwise Shift Right with Zero Operator. This operator is just like the >> operator, except that the bits shifted in on the left are always zero,	(A >>> 1) is 1.

The Assignment Operators:

There are following assignment operators supported by JavaScript language:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assigne value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A

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

Miscellaneous Operator

The Conditional Operator (? :)

There is an oprator called conditional operator. This first evaluates an expression for a true or false value and then execute one of the two given statements depending upon the result of the evaluation. The conditioanl operator has this syntax:

Operator	Description	Example
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y

The *typeof* Operator

The *typeof* 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 the list of 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"

JavaScript if...else Statements

While writing a program, there may be a situation when you need to adopt one path out of the given two paths. So you need to make use of 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 **if..else** statement.

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

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

if statement:

The **if** statement is the fundamental control statement that allows JavaScript to make decisions and execute statements conditionally.

Syntax:

```
if (expression){
   Statement(s) to be executed if expression is true
}
```

Here JavaScript *expression* is evaluated. If the resulting value is *true*, given *statement(s)* are executed. If *expression* is *false* then no statement would be not executed. Most of the times you will use comparison operators while making decisions.

Example:

```
<script type="text/javascript">
<!--
var age = 20;
if( age > 18 ){
   document.write("<b>Qualifies for driving</b>");
}
//-->
</script>
```

This will produce following result:

```
Qualifies for driving
```

if...else statement:

The **if...else** statement is the next form of control statement that allows JavaScript to execute statements in more controlled way.

Syntax:

```
if (expression){
   Statement(s) to be executed if expression is true
}else{
   Statement(s) to be executed if expression is false
}
```

Here JavaScript *expression* is evaluated. If the resulting value is *true*, given *statement(s)* in the *if* block, are executed. If *expression* is *false* then given *statement(s)* in the *else* block, are executed.

Example:

```
<script type="text/javascript">
<!--
var age = 15;
if( age > 18 ){
   document.write("<b>Qualifies for driving</b>");
}else{
   document.write("<b>Does not qualify for driving</b>");
}
//-->
</script>
```

This will produce following result:

```
Does not qualify for driving
```

if...else if... statement:

The **if...else if...** statement is the one level advance form of control statement that allows JavaScript to make correct decision out of several conditions.

Syntax:

```
if (expression 1){
   Statement(s) to be executed if expression 1 is true
}else if (expression 2){
   Statement(s) to be executed if expression 2 is true
}else if (expression 3){
   Statement(s) to be executed if expression 3 is true
}else{
   Statement(s) to be executed if no expression is true
}
```

There is nothing special about this code. It is just a series of *if* statements, where each *if* is part of the *else* clause of the previous statement. Statement(s) are executed based on the true condition, if non of the condition is true then *else* block is executed.

Example:

```
<script type="text/javascript">
<!--
var book = "maths";
if( book == "history" ){
   document.write("<b>History Book</b>");
}else if( book == "maths" ){
   document.write("<b>Maths Book</b>");
}else if( book == "economics" ){
   document.write("<b>Economics Book</b>");
}else{
  document.write("<b>Unknown Book</b>");
}
//-->
</script>
```

This will produce following result:

```
Maths Book
```

JavaScript Switch Case

You can use multiple *if...else if* statements, as in the previous chapter, to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Starting with JavaScript 1.2, you can use a **switch** statement which handles exactly this situation, and it does so more efficiently than repeated *if...else if* statements.

Syntax:

The basic syntax of the **switch** statement is to give an expression to evaluate and several different statements to execute based on the value of the expression. The interpreter checks each **case** against the value of the expression until a match is found. If nothing matches, a **default** condition will be used.

```
switch (expression)
{
  case condition 1: statement(s)
                    break;
  case condition 2: statement(s)
                    break;
   ...
  case condition n: statement(s)
                    break;
  default: statement(s)
}
```

The **break** statements indicate to the interpreter the end of that particular case. If they were omitted, the interpreter would continue executing each statement in each of the following cases.

We will explain **break** statement in *Loop Control* chapter.

Example:

Following example illustrates a basic while loop:

```
<script type="text/javascript">
<!--
var grade='A';
document.write("Entering switch block<br />");
switch (grade)
{
  case 'A': document.write("Good job<br />");
            break;
  case 'B': document.write("Pretty good<br />");
            break;
  case 'C': document.write("Passed<br />");
            break;
  case 'D': document.write("Not so good<br />");
            break;
  case 'F': document.write("Failed<br />");
            break;
  default:  document.write("Unknown grade<br />")
}
document.write("Exiting switch block");
//-->
</script>
```

Example:

Consider a case if you do not use **break** statement:

```
<script type="text/javascript">
<!--
var grade='A';
document.write("Entering switch block<br />");
switch (grade)
{
  case 'A': document.write("Good job<br />");
  case 'B': document.write("Pretty good<br />");
  case 'C': document.write("Passed<br />");
  case 'D': document.write("Not so good<br />");
  case 'F': document.write("Failed<br />");
  default:  document.write("Unknown grade<br />")
}
document.write("Exiting switch block");
//-->
</script>
```

JavaScript while Loops

While writing a program, there may be a situation when you need to perform some action over and over again. In such situation you would need to write loop statements to reduce the number of lines.

JavaScript supports all the necessary loops to help you on all steps of programming.

The *while* Loop

The most basic loop in JavaScript is the **while** loop which would be discussed in this tutorial.

Syntax:

```
while (expression){
   Statement(s) to be executed if expression is true
}
```

The purpose of a **while** loop is to execute a statement or code block repeatedly as long as *expression* is true. Once expression becomes *false*, the loop will be exited.

Example:

Following example illustrates a basic while loop:

```
<script type="text/javascript">
<!--
var count = 0;
document.write("Starting Loop" + "<br />");
while (count < 10){
  document.write("Current Count : " + count + "<br />");
  count++;
}
document.write("Loop stopped!");
//-->
</script>
```

The *do...while* Loop:

The **do...while** loop is similar to the **while** loop except that the condition check happens at the end of the loop. This means that the loop will always be executed at least once, even if the condition is *false*.

Syntax:

```
do{
   Statement(s) to be executed;
} while (expression);
```

Note the semicolon used at the end of the **do...while** loop.

Example:

Let us write above example in terms of **do...while** loop.

```
<script type="text/javascript">
<!--
var count = 0;
document.write("Starting Loop" + "<br />");
do{
  document.write("Current Count : " + count + "<br />");
  count++;
}while (count < 0);
document.write("Loop stopped!");
//-->
</script>
```

JavaScript *for* Loops

We have seen different variants of **while** loop. This chapter will explain another popular loop called **for** loop.

The *for* Loop

The **for** loop is the most compact form of looping and includes the following three important parts:

- The loop initialization where we initialize our counter to a starting value. The initialization statement is executed before the loop begins.
- The test statement which will test if the given condition is true or not. If condition is true then code given inside the loop will be executed otherwise loop will come out.
- The iteration statement where you can increase or decrease your counter.

You can put all the three parts in a single line separated by a semicolon.

Syntax:

```
for (initialization; test condition; iteration statement){
     Statement(s) to be executed if test condition is true
}
```

Example:

Following example illustrates a basic for loop:

```
<script type="text/javascript">
<!--
var count;
document.write("Starting Loop" + "<br />");
for(count = 0; count < 10; count++){
  document.write("Current Count : " + count );
  document.write("<br />");
}
document.write("Loop stopped!");
//-->
</script>
```

JavaScript *for...in* loop

There is one more loop supported by JavaScript. It is called **for...in** loop. This loop is used to loop through an object's properties.

Because we have not discussed Objects yet, so you may not feel comfortable with this loop. But once you will have understanding on JavaScript objects then you will find this loop very useful.

Syntax:

```
for (variablename in object){
  statement or block to execute
}
```

In each iteration one property from *object* is assigned to *variablename* and this loop continues till all the properties of the object are exhausted.

Example:

Here is the following example that prints out the properties of a Web browser's **Navigator** object:

```
<script type="text/javascript">
<!--
var aProperty;
document.write("Navigator Object Properties<br /> ");
for (aProperty in navigator)
{
  document.write(aProperty);
  document.write("<br />");
}
document.write("Exiting from the loop!");
//-->
</script>
```

JavaScript Loop Control

JavaScript provides you full control to handle your loops and switch statement. There may be a situation when you need to come out of a loop without reaching at its bottom. There may also be a situation when you want to skip a part of your code block and want to start next iteration of the look.

To handle all such situations, JavaScript provides **break** and **continue** statements. These statements are used to immediately come out of any loop or to start the next iteration of any loop respectively.

The *break* Statement:

The **break** statement, which was briefly introduced with the *switch* statement, is used to exit a loop early, breaking out of the enclosing curly braces.

Example:

This example illustrates the use of a **break** statement with a while loop. Notice how the loop breaks out early once x reaches 5 and reaches to *document.write(..)* statement just below to closing curly brace:

```
<script type="text/javascript">
<!--
var x = 1;
document.write("Entering the loop<br /> ");
while (x < 20)
{
  if (x == 5){
     break;  // breaks out of loop completely
  }
  x = x + 1;
  document.write( x + "<br />");
}
```

```
document.write("Exiting the loop!<br /> ");
//-->
</script>
```

The *continue* Statement:

The **continue** statement tells the interpreter to immediately start the next iteration of the loop and skip remaining code block.

When a **continue** statement is encountered, program flow will move to the loop check expression immediately and if condition remain true then it start next iteration otherwise control comes out of the loop.

Example:

This example illustrates the use of a **continue** statement with a while loop. Notice how the **continue** statement is used to skip printing when the index held in variable x reaches 5:

```
<script type="text/javascript">
<!--
var x = 1;
document.write("Entering the loop<br /> ");
while (x < 10)
{
  x = x + 1;
  if (x == 5){
     continue;  // skill rest of the loop body
  }
  document.write( x + "<br />");
}
document.write("Exiting the loop!<br /> ");
//-->
</script>
```

Using Labels to Control the Flow:

Starting from JavaScript 1.2, a label can be used with **break** and **continue** to control the flow more precisely.

A **label** is simply an identifier followed by a colon that is applied to a statement or block of code. We will see two different examples to understand label with break and continue.

Note: Line breaks are not allowed between the *continue* or **break** statement and its label name. Also, there should not be any other statement in between a label name and associated loop.

Example 1:

```
<script type="text/javascript">
<!--
document.write("Entering the loop!<br /> ");
outerloop:    // This is the label name
for (var i = 0; i < 5; i++)
{
```

```
  document.write("Outerloop: " + i + "<br />");
  innerloop:
  for (var j = 0; j < 5; j++)
  {
     if (j >  3 ) break ;          // Quit the innermost loop
     if (i == 2) break innerloop; // Do the same thing
     if (i == 4) break outerloop; // Quit the outer loop
     document.write("Innerloop: " + j + "  <br />");
   }
}
document.write("Exiting the loop!<br /> ");
//-->
</script>
```

Example 2:

```
<script type="text/javascript">
<!--
document.write("Entering the loop!<br /> ");
outerloop:    // This is the label name
for (var i = 0; i < 3; i++)
{
   document.write("Outerloop: " + i + "<br />");
   for (var j = 0; j < 5; j++)
   {
      if (j == 3){
         continue outerloop;
      }
      document.write("Innerloop: " + j + "<br />");
   }
}
document.write("Exiting the loop!<br /> ");
//-->
</script>
```

JavaScript - Dialog 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 is a group of reusable code which can be called anywhere in your programme. This eliminates the need of writing same code again and again. This will help programmers to write modular code. You can divide your big programme in a number of small and manageable functions.

Like any other advance 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 previous chapters. We are using these function again and again but they have been written in core JavaScript only once.

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

Function Definition:

Before we use a function we need to define that function. 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. The basic syntax is shown here:

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

Example:

A simple function that takes no parameters called sayHello is defined here:

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

Calling a Function:

To invoke a function somewhere later in the script, you would simple need to write the name of that function as follows:

```
<script type="text/javascript">
<!--
sayHello();
//-->
</script>
```

Function Parameters:

Till now we have seen function without a 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:

Let us do a bit modification in our *sayHello* function. This time it will take two parameters:

```
<script type="text/javascript">
<!--
function sayHello(name, age)
{
   alert( name + " is " + age + " years old.");
}
//-->
</script>
```

Note: We are using + operator to concatenate string and number all together. JavaScript does not mind in adding numbers into strings.

Now we can call this function as follows:

```
<script type="text/javascript">
<!--
sayHello('Zara', 7 );
//-->
</script>
```

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 from the function to return their multiplication in your calling program.

Example:

This function takes two parameters and concatenates them and return resultant in the calling program:

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

```
   full = first + last;
   return  full;
}
//-->
</script>
```

Now we can call this function as follows:

```
<script type="text/javascript">
<!--
   var result;
   result = concatenate('Zara', 'Ali');
   alert(result );
//-->
</script>
```

JavaScript Special Characters

In JavaScript you can add special characters to a text string by using the backslash sign.

Insert Special Characters

The backslash (\) is used to insert apostrophes, new lines, quotes, and other special characters into a text string.

Look at the following JavaScript code:

```
var txt="We are the so-called "Vikings" from the north.";
document.write(txt);
```

In JavaScript, a string is started and stopped with either single or double quotes. This means that the string above will be chopped to: We are the so-called

To solve this problem, you must place a backslash (\) before each double quote in "Viking". This turns each double quote into a string literal:

```
var txt="We are the so-called \"Vikings\" from the north.";
document.write(txt);
```

JavaScript will now output the proper text string: We are the so-called "Vikings" from the north.

The table below lists other special characters that can be added to a text string with the backslash sign:

Code	Outputs
\'	single quote

\"	double quote
\\	Backslash
\n	new line
\r	carriage return
\t	Tab
\b	Backspace
\f	form feed

JavaScript Events

What is an Event ?

JavaScript's interaction with HTML is handled through events that occur when the user or browser manipulates a page.

When the page loads, that is an event. When the user clicks a button, that click, too, is an event. Another example of events are like pressing any key, closing window, resizing window etc.

Developers can use these events to execute JavaScript coded responses, which cause buttons to close windows, messages to be displayed to users, data to be validated, and virtually any other type of response imaginable to occur.

Events are a part of the Document Object Model (DOM) Level 3 and every HTML element have a certain set of events which can trigger JavaScript Code.

onclick Event Type:

This is the most frequently used event type which occurs when a user clicks mouse left button. You can put your validation, warning etc against this event type.

Example:

```
<html>
<head>
<script type="text/javascript">
<!--
function sayHello() {
   alert("Hello World")
}
//-->
</script>
</head>
<body>
<input type="button" onclick="sayHello()" value="Say Hello" />
```

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

onsubmit event type:

Another most important event type is onsubmit. This event occurs when you try to submit a form. So you can put your form validation against this event type.

Here is simple example showing its usage. Here we are calling a validate() function before submitting a form data to the webserver. If validate() function returns true the form will be submitted otherwise it will not submit the data.

Example:

```
<html>
<head>
<script type="text/javascript">
<!--
function validation() {
   all validation goes here
   .........
   return either true or false
}
//-->
</script>
</head>
<body>
<form method="POST" action="t.cgi" onsubmit="return validate()">
.......
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

onmouseover and onmouseout:

These two event types will help you to create nice effects with images or even with text as well. The onmouseover event occurs when you bring your mouse over any element and the onmouseout occurs when you take your mouse out from that element.

Example:

Following example shows how a division reacts when we bring our mouse in that division:

```
<html>
<head>
<script type="text/javascript">
<!--
function over() {
   alert("Mouse Over");
}
function out() {
   alert("Mouse Out");
}
```

```
//-->
</script>
</head>
<body>
<div onmouseover="over()" onmouseout="out()">
<h2> This is inside the division </h2>
</div>
</body>
</html>
```

You can change different images using these two event types or you can create help baloon to help your users.

HTML 4 Standard Events

The standard HTML 4 events are listed here for your reference. Here script indicates a Javascript function to be executed agains that event.

Event	Value	Description
onchange	script	Script runs when the element changes
onsubmit	script	Script runs when the form is submitted
onreset	script	Script runs when the form is reset
onselect	script	Script runs when the element is selected
onblur	script	Script runs when the element loses focus
onfocus	script	Script runs when the element gets focus
onkeydown	script	Script runs when key is pressed
onkeypress	script	Script runs when key is pressed and released
onkeyup	script	Script runs when key is released
onclick	script	Script runs when a mouse click
ondblclick	script	Script runs when a mouse double-click
onmousedown	script	Script runs when mouse button is pressed
onmousemove	script	Script runs when mouse pointer moves

onmouseout	script	Script runs when mouse pointer moves out of an element
onmouseover	script	Script runs when mouse pointer moves over an element
onmouseup	script	Script runs when mouse button is released

onblur Event

```
<html>
<head>
<script type="text/javascript">
function upperCase()
{
var x=document.getElementById("fname").value
document.getElementById("fname").value=x.toUpperCase()
}
</script>
</head>
<body>

Enter your name: <input type="text" id="fname" onblur="upperCase()">

</body>
</html>
```

onchange Event

```
<html>
<head>
<script type="text/javascript">
function upperCase(x)
{
var y=document.getElementById(x).value
document.getElementById(x).value=y.toUpperCase()
}
</script>
</head>
<body>

Enter your name:
<input type="text" id="fname"
onchange="upperCase(this.id)">

</body>
</html>
```

onclick Event

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

```
Field1: <input type="text" id="field1" value="Hello World!">
<br />
Field2: <input type="text" id="field2">
<br /><br />
Click the button below to copy the content of Field1 to Field2.
<br />
<button onclick="document.getElementById('field2').value=
document.getElementById('field1').value">Copy Text</button>

</body>
</html>
```

ondblclick Event

```
<html>
<body>

Field1: <input type="text" id="field1" value="Hello World!">
<br />
Field2: <input type="text" id="field2">
<br /><br />
Click the button below to copy the content of Field1 to Field2.
<br />
<button ondblclick="document.getElementById('field2').value=
document.getElementById('field1').value">Copy Text</button>

</body>
</html>
```

onerror Event

```
<img src="image.gif"
onerror="alert('The image could not be loaded.')">
```

onfocus Event

```
<html>
<head>
<script type="text/javascript">
function setStyle(x)
{
document.getElementById(x).style.background="yellow"
}
</script>
</head>
<body>

First name: <input type="text"
onfocus="setStyle(this.id)" id="fname">
<br />
Last name: <input type="text"
onfocus="setStyle(this.id)" id="lname">

</body>
</html>
```

onkeydown Event

```
<html>
<body>
<script type="text/javascript">
function noNumbers(e)
{
var keynum
var keychar
var numcheck

if(window.event) // IE
{
keynum = e.keyCode
}
else if(e.which) // Netscape/Firefox/Opera
{
keynum = e.which
}
keychar = String.fromCharCode(keynum)
numcheck = /\d/
return !numcheck.test(keychar)
}
</script>

<form>
<input type="text" onkeydown="return noNumbers(event)" />
</form>

</body>
</html>
```

onkeypress Event

```
<html>
<body>
<script type="text/javascript">
function noNumbers(e)
{
var keynum
var keychar
var numcheck

if(window.event) // IE
{
keynum = e.keyCode
}
else if(e.which) // Netscape/Firefox/Opera
{
keynum = e.which
}
keychar = String.fromCharCode(keynum)
numcheck = /\d/
return !numcheck.test(keychar)
}
</script>

<form>
```

```
<input type="text" onkeypress="return noNumbers(event)" />
</form>
</body>
</html>
```

onKeyUp Event

```
<html>
<head>
<script type="text/javascript">
function upperCase(x)
{
var y=document.getElementById(x).value
document.getElementById(x).value=y.toUpperCase()
}
</script>
</head>
<body>

Enter your name: <input type="text"
id="fname" onkeyup="upperCase(this.id)">

</body> </html>
```

onload Event

```
<html>
<head>
<script type="text/javascript">
function load()
{
alert("Page is loaded");
}
</script>
</head>

<body onload="load()">
<h1>Hello World!</h1>
</body>
</html>
```

onmousedown Event

```
<html>
<head>
<script type="text/javascript">
function whichElement(e)
{
var targ
if (!e) var e = window.event
if (e.target) targ = e.target
else if (e.srcElement) targ = e.srcElement
if (targ.nodeType == 3) // defeat Safari bug
targ = targ.parentNode
```

```
var tname
tname=targ.tagName
alert("You clicked on a " + tname + " element.")
}
</script>
</head>

<body onmousedown="whichElement(event)">

<h2>This is a header</h2>
<p>This is a paragraph</p>
<img border="0" src="ball16.gif" alt="Ball">

</body>
</html>
```

onmousemove Event

```
<img src="image_w3default.gif" alt="W3Schools"
onmousemove="alert('Visit W3Schools!')" />
```

onmouseout Event

```
<html>
<head>
<script type="text/javascript">
function mouseOver()
{
document.getElementById("b1").src="b_blue.gif"
}
function mouseOut()
{
document.getElementById("b1").src="b_pink.gif"
}
</script>
</head>

<body>
<a href="http://www.w3schools.com" target="_blank" onmouseover="mouseOver()"
onmouseout="mouseOut()">
<img border="0" alt="Visit W3Schools!" src="b_pink.gif" id="b1" /></a>
</body>
</html>
```

onmouseover Event

```
<html>
<head>
<script type="text/javascript">
function mouseOver()
{
document.getElementById("b1").src ="b_blue.gif"
}
function mouseOut()
{
document.getElementById("b1").src ="b_pink.gif"
```

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

<body>
<a href="http://www.w3schools.com" target="_blank" onmouseover="mouseOver()"
onmouseout="mouseOut()">
<img border="0" alt="Visit W3Schools!" src="b_pink.gif" id="b1" /></a>
</body>
</html>
```

onmouseup Event

```
<html>
<head>
<script type="text/javascript">
function whichElement(e)
{
var targ
if (!e) var e = window.event
if (e.target) targ = e.target
else if (e.srcElement) targ = e.srcElement
if (targ.nodeType == 3) // defeat Safari bug
targ = targ.parentNode
var tname
tname=targ.tagName
alert("You clicked on a " + tname + " element.")
}
</script>
</head>

<body onmouseup="whichElement(event)">
<h2>This is a header</h2>
<p>This is a paragraph</p>
<img border="0" src="ball16.gif" alt="Ball">
</body>
</html>
```

onresize Event

```
<body onresize="alert('You have changed the size of the window')">
</body>
```

onselect Event

```
<form>
Select text: <input type="text" value="Hello world!"
onselect="alert('You have selected some of the text.')">
</form>
```

onunload Event

```
<body onunload="alert('The onunload event was triggered')">
</body>
```

Javascript - Page Printing

Many times you would like to give a button at your webpage to print out the content of that web page via an actual printer.

JavaScript helps you to implement this functionality using **print** function of *window* object.

The JavaScript print function **window.print()** will print the current web page when executed. You can call this function directly using *onclick* event as follows:

```
<head>
<script type="text/javascript">
<!--
//-->
</script>
</head>
<body>
<form>
<input type="button" value="Print" onclick="window.print()" />
</form>
</body>
```

How to print a page:

If someone is providing none of the above facilities then you can use browser's standard toolbar to get web pages printed out. Follow the link as follows:

File --> Print --> Click OK button.

JavaScript - Page Redirection

When you click a URL to reach to a page X but internally you are directed to another page Y that simply happens because of page re-direction.

```
<head>
<script type="text/javascript">
  function fun1(){
        window.location="http://www.newlocation.com";
  }
</script>
</head>
```

JavaScript and Cookies

What are Cookies ?

Web Browser and Server use HTTP protocol to communicate and HTTP is a stateless protocol. But for a commercial website it is required to maintain session information among different pages. For example one user registration ends after completing many pages. But how to maintain user's session information across all the web pages.

In many situations, using cookies is the most efficient method of remembering and tracking preferences, purchases, commissions, and other information required for better visitor experience or site statistics.

How It Works ?

Your server sends some data to the visitor's browser in the form of a cookie. The browser may accept the cookie. If it does, it is stored as a plain text record on the visitor's hard drive. Now, when the visitor arrives at another page on your site, the browser sends the same cookie to the server for retrieval. Once retrieved, your server knows/remembers what was stored earlier.

Cookies are a plain text data record of 5 variable-length fields:

- **Expires :** The date the cookie will expire. If this is blank, the cookie will expire when the visitor quits the browser.
- **Domain :** The domain name of your site.
- **Path :** The path to the directory or web page that set the cookie. This may be blank if you want to retrieve the cookie from any directory or page.
- **Secure :** If this field contains the word "secure" then the cookie may only be retrieved with a secure server. If this field is blank, no such restriction exists.
- **Name=Value :** Cookies are set and retrieved in the form of key and value pairs.

Cookies were originally designed for CGI programming and cookies' data is automatically transmitted between the web browser and web server, so CGI scripts on the server can read and write cookie values that are stored on the client.

JavaScript can also manipulate cookies using the *cookie* property of the *Document* object. JavaScript can read, create, modify, and delete the cookie or cookies that apply to the current web page.

Storing Cookies:

The simplest way to create a cookie is to assign a string value to the *document.cookie* object, which looks like this:

Syntax:

```
document.cookie = "key1=value1;key2=value2;expires=date";
```

Here *expires* attribute is option. If you provide this attribute with a valid date or time then cookie will expire at the given date or time and after that cookies' value will not be accessible.

Note: Cookie values may not include semicolons, commas, or whitespace. For this reason, you may want to use the JavaScript *escape()* function to encode the *value* before storing it in the cookie. If you do this, you will also have to use the corresponding *unescape()* function when you read the cookie value.

Example:

Following is the example to set a customer name in *input* cookie.

```
<html>
<head>
<script type="text/javascript">
<!--
function WriteCookie()
{
   if( document.myform.customer.value == "" ){
      alert("Enter some value!");
      return;
   }

   cookievalue= escape(document.myform.customer.value) + ";";
   document.cookie="name=" + cookievalue;
   alert("Setting Cookies : " + "name=" + cookievalue );
}
//-->
</script>
</head>
<body>
<form name="myform" action="">
Enter name: <input type="text" name="customer"/>
<input type="button" value="Set Cookie" onclick="WriteCookie();"/>
</form>
</body>
</html>
```

Now your machine has a cookie called *name*. You can set multiple cookies using multiple *key=value* pairs separated by comma.

You will learn how to read this cookie in next section.

Reading Cookies:

Reading a cookie is just as simple as writing one, because the value of the *document.cookie* object is the cookie. So you can use this string whenever you want to access the cookie.

The *document.cookie* string will keep a list of *name=value* pairs separated by semicolons, where *name* is the *name* of a cookie and value is its string value.

You can use strings' *split()* function to break the string into key and values as follows:

Example:

Following is the example to get the cookies set in previous section.

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

```
function ReadCookie()
{
   var allcookies = document.cookie;
   alert("All Cookies : " + allcookies );

   // Get all the cookies pairs in an array
   cookiearray  = allcookies.split(';');

   // Now take key value pair out of this array
   for(var i=0; i<cookiearray.length; i++){
      name = cookiearray[i].split('=')[0];
      value = cookiearray[i].split('=')[1];
      alert("Key is : " + name + " and Value is : " + value);
   }
}
//-->
</script>
</head>
<body>
<form name="myform" action="">
<input type="button" value="Get Cookie" onclick="ReadCookie()"/>
</form>
</body>
</html>
```

Note: Here *length* is a method of *Array* class which returns the length of an array.

Note: There may be some other cookies already set on your machine. So above code will show you all the cookies set at your machine.

Setting the Cookies Expiration Date:

You can extend the life of a cookie beyond the current browser session by setting an expiration date and saving the expiration date within the cookie. This can be done by setting the *expires* attribute to a date and time.

Example:

The following example illustrates how to set cookie expiration date after 1 Month :

```
<html>
<head>
<script type="text/javascript">
<!--
function WriteCookie()
{
   var now = new Date();
   now.setMonth( now.getMonth() + 1 );
   cookievalue = escape(document.myform.customer.value) + ";"
   document.cookie="name=" + cookievalue;
   document.cookie = "expires=" + now.getGMTString() + ";"
   alert("Setting Cookies : " + "name=" + cookievalue );
}
//-->
</script>
</head>
```

```
<body>
<form name="formname" action="">
Enter name: <input type="text" name="customer"/>
<input type="button" value="Set Cookie" onclick="WriteCookie()"/>
</form>
</body>
</html>
```

Deleting a Cookie:

Sometimes you will want to delete a cookie so that subsequent attempts to read the cookie return nothing. To do this, you just need to set the expiration date to a time in the past.

Example:

The following example illustrates how to delete cookie by setting expiration date one Month in past :

```
<html>
<head>
<script type="text/javascript">
<!--
function WriteCookie()
{
   var now = new Date();
   now.setMonth( now.getMonth() - 1 );
   cookievalue = escape(document.myform.customer.value) + ";"
   document.cookie="name=" + cookievalue;
   document.cookie = "expires=" + now.getGMTString() + ";"
   alert("Setting Cookies : " + "name=" + cookievalue );
}
//-->
</script>
</head>
<body>
<form name="formname" action="">
Enter name: <input type="text" name="customer"/>
<input type="button" value="Set Cookie" onclick="WriteCookie()"/>
</form>
</body>
</html>
```

Note: Instead of setting date, you can see new time using *setTime()* function.

Javascript Objects Overview

JavaScript is an Object Oriented Programming (OOP) language. A programming language can be called object-oriented if it provides four basic capabilities to developers:

- **Encapsulation .** the capability to store related information, whether data or methods, together in an object

- **Aggregation .** the capability to store one object inside of another object
- **Inheritance .** the capability of a class to rely upon another class (or number of classes) for some of its properties and methods
- **Polymorphism .** the capability to write one function or method that works in a variety of different ways

Objects are composed of attributes. If an attribute contains a function, it is considered to be a method of the object otherwise, the attribute is considered a property.

Object Properties:

Object properties can be any of the three primitive data types, or any of the abstract data types, such as another object. Object properties are usually variables that are used internally in the object's methods, but can also be globally visible variables that are used throughout the page.

The syntax for adding a property to an object is:

```
objectName.objectProperty = propertyValue;
```

Example:

Following is a simple example to show how to get a document title using "title" property of document object:

```
var str = document.title;
```

Object Methods:

The methods are functions that let the object do something or let something be done to it. There is little difference between a function and a method, except that a function is a standalone unit of statements and a method is attached to an object and can be referenced by the **this** keyword.

Methods are useful for everything from displaying the contents of the object to the screen to performing complex mathematical operations on a group of local properties and parameters.

Example:

Following is a simple example to show how to use **write()** method of document object to write any content on the document:

```
document.write("This is test");
```

User-Defined Objects:

All user-defined objects and built-in objects are descendants of an object called Object.

The *new* Operator:

The *new* operator is used to create an instance of an object. To create an object, the *new* operator is followed by the constructor method.

In the following example, the constructor methods are Object(), Array(), and Date(). These constructors are built-in JavaScript functions.

```
var employee = new Object();
var books = new Array("C++", "Perl", "Java");
var day = new Date("August 15, 1947");
```

The *Object()* Constructor:

A constructor is a function that creates and initializes an object. JavaScript provides a special constructor function called *Object()* to build the object. The return value of the Object() constructor is assigned to a variable.

The variable contains a reference to the new object. The properties assigned to the object are not variables and are not defined with the var keyword.

Example 1:

This example demonstrates how to create an object:

```
<html>
<head>
<title>User-defined objects</title>
<script type="text/javascript">
var book = new Object();    // Create the object
    book.subject = "Perl"; // Assign properties to the object
    book.author  = "Mohtashim";
</script>
</head>
<body>
<script type="text/javascript">
   document.write("Book name is : " + book.subject + "<br>");
   document.write("Book author is : " + book.author + "<br>");
</script>
</body>
</html>
```

Example 2:

This example demonstrates how to create an object with a User-Defined Function. Here *this* keyword is used to refer to the object that has been passed to a function:

```
<html>
<head>
<title>User-defined objects</title>
<script type="text/javascript">
function book(title, author){
    this.title = title;
    this.author  = author;
}
</script>
</head>
<body>
<script type="text/javascript">
   var myBook = new book("Perl", "Mohtashim");
   document.write("Book title is : " + myBook.title + "<br>");
   document.write("Book author is : " + myBook.author + "<br>");
</script>
</body>
</html>
```

Defining Methods for an Object:

The previous examples demonstrate how the constructor creates the object and assigns properties. But we need to complete the definition of an object by assigning methods to it.

Example:

Here is a simple example to show how to add a function along with an object:

```
<html>
<head>
<title>User-defined objects</title>
<script type="text/javascript">

// Define a function which will work as a method
function addPrice(amount){
    this.price = amount;
}

function book(title, author){
    this.title = title;
    this.author  = author;
    this.addPrice = addPrice; // Assign that method as property.
}

</script>
</head>
```

```
<body>
<script type="text/javascript">
   var myBook = new book("Perl", "Mohtashim");
   myBook.addPrice(100);
   document.write("Book title is : " + myBook.title + "<br>");
   document.write("Book author is : " + myBook.author + "<br>");
   document.write("Book price is : " + myBook.price + "<br>");
</script>
</body>
</html>
```

The *with* Keyword:

The *with* keyword is used as a kind of shorthand for referencing an object's properties or methods.

The object specified as an argument to with becomes the default object for the duration of the block that follows. The properties and methods for the object can be used without naming the object.

Syntax:

```
with (object){
    properties used without the object name and dot
}
```

Example:

```
<html>
<head>
<title>User-defined objects</title>
<script type="text/javascript">

// Define a function which will work as a method
function addPrice(amount){
    with(this){
       price = amount;
    }
}
function book(title, author){
    this.title = title;
    this.author  = author;
    this.price = 0;
    this.addPrice = addPrice; // Assign that method as property.
}
</script>
</head>
<body>
<script type="text/javascript">
   var myBook = new book("Perl", "Mohtashim");
   myBook.addPrice(100);
   document.write("Book title is : " + myBook.title + "<br>");
   document.write("Book author is : " + myBook.author + "<br>");
   document.write("Book price is : " + myBook.price + "<br>");
```

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

JavaScript Native Objects:

JavaScript has several built-in or native objects. These objects are accessible anywhere in your program and will work the same way in any browser running in any operating system.

Here is the list of all important JavaScript Native Objects:

- JavaScript Number Object
- JavaScript Boolean Object
- JavaScript String Object
- JavaScript Array Object
- JavaScript Date Object
- JavaScript Math Object
- JavaScript RegExp Object

Javascript - The String Object

The **String** object let's you work with a series of characters and wraps Javascript's string primitive data type with a number of helper methods.

Because Javascript automatically converts between string primitives and String objects, you can call any of the helper methods of the String object on a string primitive.

Syntax:

Creating a **String** object:

```
var val = new String(string);
```

The *string* parameter is series of characters that has been properly encoded.

String Properties:

Here is a list of each property and their description.

Property	Description

constructor	Returns a reference to the String function that created the object.
length	Returns the length of the string.
prototype	The prototype property allows you to add properties and methods to an object.

String Methods

Here is a list of each method and its description.

Method	Description
charAt()	Returns the character at the specified index.
charCodeAt()	Returns a number indicating the Unicode value of the character at the given index.
concat()	Combines the text of two strings and returns a new string.
indexOf()	Returns the index within the calling String object of the first occurrence of the specified value, or -1 if not found.
lastIndexOf()	Returns the index within the calling String object of the last occurrence of the specified value, or -1 if not found.
localeCompare()	Returns a number indicating whether a reference string comes before or after or is the same as the given string in sort order.
match()	Used to match a regular expression against a string.
replace()	Used to find a match between a regular expression and a string, and to replace the matched substring with a new substring.
search()	Executes the search for a match between a regular expression and a specified string.
slice()	Extracts a section of a string and returns a new string.
split()	Splits a String object into an array of strings by separating the string into substrings.

substr()	Returns the characters in a string beginning at the specified location through the specified number of characters.
substring()	Returns the characters in a string between two indexes into the string.
toLocaleLowerCase()	The characters within a string are converted to lower case while respecting the current locale.
toLocaleUpperCase()	The characters within a string are converted to upper case while respecting the current locale.
toLowerCase()	Returns the calling string value converted to lower case.
toString()	Returns a string representing the specified object.
toUpperCase()	Returns the calling string value converted to uppercase.
valueOf()	Returns the primitive value of the specified object.

String HTML wrappers

Here is a list of each method which returns a copy of the string wrapped inside the appropriate HTML tag.

Method	Description
anchor()	Creates an HTML anchor that is used as a hypertext target.
big()	Creates a string to be displayed in a big font as if it were in a <big> tag.
blink()	Creates a string to blink as if it were in a <blink> tag.
bold()	Creates a string to be displayed as bold as if it were in a <b> tag.
fixed()	Causes a string to be displayed in fixed-pitch font as if it were in a <tt> tag
fontcolor()	Causes a string to be displayed in the specified color as if it were in a <font color="color"> tag.
fontsize()	Causes a string to be displayed in the specified font size as if it were in a <font size="size"> tag.

italics()	Causes a string to be italic, as if it were in an <i> tag.
link()	Creates an HTML hypertext link that requests another URL.
small()	Causes a string to be displayed in a small font, as if it were in a <small> tag.
strike()	Causes a string to be displayed as struck-out text, as if it were in a <strike> tag.
sub()	Causes a string to be displayed as a subscript, as if it were in a <sub> tag
sup()	Causes a string to be displayed as a superscript, as if it were in a <sup> tag

Javascript - The Arrays Object

The **Array** object let's you store multiple values in a single variable.

Syntax:

Creating a **Array** object:

```
var fruits = new Array( "apple", "orange", "mango" );
```

The *Array* parameter is a list of strings or integers. When you specify a single numeric parameter with the Array constructor, you specify the initial length of the array. The maximum length allowed for an array is 4,294,967,295.

You can create array by simply assigning values as follows:

```
var fruits = [ "apple", "orange", "mango" ];
```

You will use ordinal numbers to access and to set values inside an array as follows:

- fruits[0] is the first element
- fruits[1] is the second element
- fruits[2] is the third element

Array Properties:

Here is a list of each property and their description.

Property	Description
constructor	Returns a reference to the array function that created the object.
index	The property represents the zero-based index of the match in the string
input	This property is only present in arrays created by regular expression matches.
length	Reflects the number of elements in an array.
prototype	The prototype property allows you to add properties and methods to an object.

Array Methods

Here is a list of each method and its description.

Method	Description
concat()	Returns a new array comprised of this array joined with other array(s) and/or value(s).
every()	Returns true if every element in this array satisfies the provided testing function.
filter()	Creates a new array with all of the elements of this array for which the provided filtering function returns true.
forEach()	Calls a function for each element in the array.
indexOf()	Returns the first (least) index of an element within the array equal to the specified value, or -1 if none is found.
join()	Joins all elements of an array into a string.
lastIndexOf()	Returns the last (greatest) index of an element within the array equal to the specified value, or -1 if none is found.

map()	Creates a new array with the results of calling a provided function on every element in this array.
pop()	Removes the last element from an array and returns that element.
push()	Adds one or more elements to the end of an array and returns the new length of the array.
reduce()	Apply a function simultaneously against two values of the array (from left-to-right) as to reduce it to a single value.
reduceRight()	Apply a function simultaneously against two values of the array (from right-to-left) as to reduce it to a single value.
reverse()	Reverses the order of the elements of an array -- the first becomes the last, and the last becomes the first.
shift()	Removes the first element from an array and returns that element.
slice()	Extracts a section of an array and returns a new array.
some()	Returns true if at least one element in this array satisfies the provided testing function.
toSource()	Represents the source code of an object
sort()	Sorts the elements of an array.
splice()	Adds and/or removes elements from an array.
toString()	Returns a string representing the array and its elements.
unshift()	Adds one or more elements to the front of an array and returns the new length of the array.

JavaScript - The Date Object

The Date object is a datatype built into the JavaScript language. Date objects are created with the **new Date( )** as shown below.

Once a Date object is created, a number of methods allow you to operate on it. Most methods simply allow you to get and set the year, month, day, hour, minute, second, and millisecond fields of the object, using either local time or UTC (universal, or GMT) time.

The ECMAScript standard requires the Date object to be able to represent any date and time, to millisecond precision, within 100 million days before or after 1/1/1970. This is a range of plus or minus 273,785 years, so the JavaScript is able to represent date and time till year 275755.

Syntax:

Here are different variant of Date() constructor:

```
new Date( )
new Date(milliseconds)
new Date(datestring)
new Date(year,month,date[,hour,minute,second,millisecond ])
```

Note: Paramters in the brackets are always optional

Here is the description of the parameters:

- **No Argument:** With no arguments, the Date( ) constructor creates a Date object set to the current date and time.
- **milliseconds:** When one numeric argument is passed, it is taken as the internal numeric representation of the date in milliseconds, as returned by the getTime( ) method. For example, passing the argument 5000 creates a date that represents five seconds past midnight on 1/1/70.
- **datestring:**When one string argument is passed, it is a string representation of a date, in the format accepted by the Date.parse( ) method.
- **7 agruments:** To use the last form of constructor given above, Here is the description of each argument:
    1. **year:** Integer value representing the year. For compatibility (in order to avoid the Y2K problem), you should always specify the year in full; use 1998, rather than 98.
    2. **month:** Integer value representing the month, beginning with 0 for January to 11 for December.
    3. **date:** Integer value representing the day of the month.
    4. **hour:** Integer value representing the hour of the day (24-hour scale).
    5. **minute:** Integer value representing the minute segment of a time reading.
    6. **second:** Integer value representing the second segment of a time reading.
    7. **millisecond:** Integer value representing the millisecond segment of a time reading.

Date Properties:

Here is a list of each property and their description.

Property	Description

constructor	Specifies the function that creates an object's prototype.
prototype	The prototype property allows you to add properties and methods to an object.

Date Methods:

Here is a list of each method and its description.

Method	Description
Date()	Returns today's date and time
getDate()	Returns the day of the month for the specified date according to local time.
getDay()	Returns the day of the week for the specified date according to local time.
getFullYear()	Returns the year of the specified date according to local time.
getHours()	Returns the hour in the specified date according to local time.
getMilliseconds()	Returns the milliseconds in the specified date according to local time.
getMinutes()	Returns the minutes in the specified date according to local time.
getMonth()	Returns the month in the specified date according to local time.
getSeconds()	Returns the seconds in the specified date according to local time.
getTime()	Returns the numeric value of the specified date as the number of milliseconds since January 1, 1970, 00:00:00 UTC.
getTimezoneOffset()	Returns the time-zone offset in minutes for the current locale.
getUTCDate()	Returns the day (date) of the month in the specified date according to universal time.
getUTCDay()	Returns the day of the week in the specified date according to universal time.

getUTCFullYear()	Returns the year in the specified date according to universal time.
getUTCHours()	Returns the hours in the specified date according to universal time.
getUTCMilliseconds()	Returns the milliseconds in the specified date according to universal time.
getUTCMinutes()	Returns the minutes in the specified date according to universal time.
getUTCMonth()	Returns the month in the specified date according to universal time.
getUTCSeconds()	Returns the seconds in the specified date according to universal time.
getYear()	**Deprecated** - Returns the year in the specified date according to local time. Use getFullYear instead.
setDate()	Sets the day of the month for a specified date according to local time.
setFullYear()	Sets the full year for a specified date according to local time.
setHours()	Sets the hours for a specified date according to local time.
setMilliseconds()	Sets the milliseconds for a specified date according to local time.
setMinutes()	Sets the minutes for a specified date according to local time.
setMonth()	Sets the month for a specified date according to local time.
setSeconds()	Sets the seconds for a specified date according to local time.
setTime()	Sets the Date object to the time represented by a number of milliseconds since January 1, 1970, 00:00:00 UTC.
setUTCDate()	Sets the day of the month for a specified date according to universal time.
setUTCFullYear()	Sets the full year for a specified date according to universal time.
setUTCHours()	Sets the hour for a specified date according to universal time.

setUTCMilliseconds()	Sets the milliseconds for a specified date according to universal time.
setUTCMinutes()	Sets the minutes for a specified date according to universal time.
setUTCMonth()	Sets the month for a specified date according to universal time.
setUTCSeconds()	Sets the seconds for a specified date according to universal time.
setYear()	**Deprecated** - Sets the year for a specified date according to local time. Use setFullYear instead.
toDateString()	Returns the "date" portion of the Date as a human-readable string.
toGMTString()	**Deprecated** - Converts a date to a string, using the Internet GMT conventions. Use toUTCString instead.
toLocaleDateString()	Returns the "date" portion of the Date as a string, using the current locale's conventions.
toLocaleFormat()	Converts a date to a string, using a format string.
toLocaleString()	Converts a date to a string, using the current locale's conventions.
toLocaleTimeString()	Returns the "time" portion of the Date as a string, using the current locale's conventions.
toSource()	Returns a string representing the source for an equivalent Date object; you can use this value to create a new object.
toString()	Returns a string representing the specified Date object.
toTimeString()	Returns the "time" portion of the Date as a human-readable string.
toUTCString()	Converts a date to a string, using the universal time convention.
valueOf()	Returns the primitive value of a Date object.

Date Static Methods:

In addition to the many instance methods listed previously, the Date object also defines two static methods. These methods are invoked through the Date( ) constructor itself:

Method	Description
Date.parse( )	Parses a string representation of a date and time and returns the internal millisecond representation of that date.
Date.UTC( )	Returns the millisecond representation of the specified UTC date and time.

Javascript - The Math Object

The **math** object provides you properties and methods for mathematical constants and functions.

Unlike the other global objects, *Math* is not a constructor. All properties and methods of Math are static and can be called by using *Math* as an object without creating it.

Thus, you refer to the constant pi as **Math.PI** and you call the *sine* function as **Math.sin(x)**, where x is the method's argument.

Syntax:

Here is the simple syntax to call properties and methods of Math.

```
var pi_val = Math.PI;
var sine_val = Math.sin(30);
```

Math Properties:

Here is a list of each property and their description.

Property	Description
E	Euler's constant and the base of natural logarithms, approximately 2.718.
LN2	Natural logarithm of 2, approximately 0.693.

LN10	Natural logarithm of 10, approximately 2.302.
LOG2E	Base 2 logarithm of E, approximately 1.442.
LOG10E	Base 10 logarithm of E, approximately 0.434.
PI	Ratio of the circumference of a circle to its diameter, approximately 3.14159.
SQRT1_2	Square root of 1/2; equivalently, 1 over the square root of 2, approximately 0.707.
SQRT2	Square root of 2, approximately 1.414.

Math Methods

Here is a list of each method and its description.

Method	Description
abs()	Returns the absolute value of a number.
acos()	Returns the arccosine (in radians) of a number.
asin()	Returns the arcsine (in radians) of a number.
atan()	Returns the arctangent (in radians) of a number.
atan2()	Returns the arctangent of the quotient of its arguments.
ceil()	Returns the smallest integer greater than or equal to a number.
cos()	Returns the cosine of a number.
exp()	Returns E^N, where N is the argument, and E is Euler's constant, the base of the natural logarithm.
floor()	Returns the largest integer less than or equal to a number.

log()	Returns the natural logarithm (base E) of a number.
max()	Returns the largest of zero or more numbers.
min()	Returns the smallest of zero or more numbers.
pow()	Returns base to the exponent power, that is, base exponent.
random()	Returns a pseudo-random number between 0 and 1.
round()	Returns the value of a number rounded to the nearest integer.
sin()	Returns the sine of a number.
sqrt()	Returns the square root of a number.
tan()	Returns the tangent of a number.
toSource()	Returns the string "Math".

JavaScript - Document Object Model or DOM

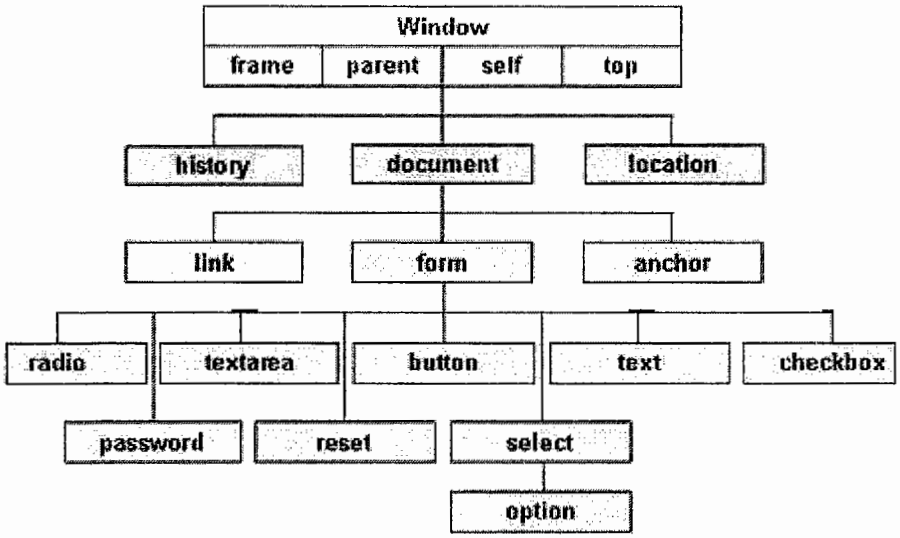
Every web page resides inside a browser window which can be considered as an object.

A Document object represents the HTML document that is displayed in that window. The Document object has various properties that refer to other objects which allow access to and modification of document content.

The way that document content is accessed and modified is called the **Document Object Model**, or **DOM**. The Objects are organized in a hierarchy. This hierarchical structure applies to the organization of objects in a Web document.

- **Window object:** Top of the hierarchy. It is the outmost element of the object hierarchy.
- **Document object:** Each HTML document that gets loaded into a window becomes a document object. The document contains the content of the page.
- **Form object:** Everything enclosed in the <form>...</form> tags sets the form object.
- **Form control elements:** The form object contains all the elements defined for that object such as text fields, buttons, radio buttons, and checkboxes.

Here is a simple hierarchy of few important objects:



JavaScript Browser Detection

The Navigator object contains information about the visitor's browser.

Browser Detection

Almost everything in this tutorial works on all JavaScript-enabled browsers. However, there are some things that just don't work on certain browsers - especially on older browsers.

Sometimes it can be useful to detect the visitor's browser, and then serve the appropriate information.

The Navigator object contains information about the visitor's browser name, version, and more.

Note: There is no public standard that applies to the navigator object, but all major browsers support it.

The Navigator Object

The Navigator object contains all information about the visitor's browser:

Example

```
<html>
<body>
<div id="example"></div>

<script type="text/javascript">

txt = "<p>Browser CodeName: " + navigator.appCodeName + "</p>";
txt+= "<p>Browser Name: " + navigator.appName + "</p>";
txt+= "<p>Browser Version: " + navigator.appVersion + "</p>";
txt+= "<p>Cookies Enabled: " + navigator.cookieEnabled + "</p>";
txt+= "<p>Platform: " + navigator.platform + "</p>";
```

```
txt+= "<p>User-agent header: " + navigator.userAgent + "</p>";

document.getElementById("example").innerHTML=txt;

</script>

</body>
</html>
```

```
<html>
<head>
<title>Browser Detection Example</title>
</head>
<body>
<script type="text/javascript">
<!--
var userAgent   = navigator.userAgent;
var opera       = (userAgent.indexOf('Opera') != -1);
var ie          = (userAgent.indexOf('MSIE') != -1);
var gecko       = (userAgent.indexOf('Gecko') != -1);
var netscape    = (userAgent.indexOf('Mozilla') != -1);
var version     = navigator.appVersion;

if (opera){
  document.write("Opera based browser");
  // Keep your opera specific URL here.
}else if (gecko){
  document.write("Mozilla based browser");
  // Keep your gecko specific URL here.
}else if (ie){
  document.write("IE based browser");
  // Keep your IE specific URL here.
}else if (netscape){
  document.write("Netscape based browser");
  // Keep your Netscape specific URL here.
}else{
  document.write("Unknown browser");
}
// You can include version to along with any above condition.
document.write("<br /> Browser version info : " + version );
//-->
</script>
</body>
</html>
```

JavaScript - Errors & Exceptions Handling

There are three types of errors in programming: (a) Syntax Errors and (b) Runtime Errors (c) Logical Errors:

Syntax errors:

Syntax errors, also called parsing errors, occur at compile time for traditional programming languages and at interpret time for JavaScript.

For example, the following line causes a syntax error because it is missing a closing parenthesis:

```
<script type="text/javascript">
<!--
window.print(;
//-->
</script>
```

When a syntax error occurs in JavaScript, only the code contained within the same thread as the syntax error is affected and code in other threads gets executed assuming nothing in them depends on the code containing the error.

Runtime errors:

Runtime errors, also called exceptions, occur during execution (after compilation/interpretation).

For example, the following line causes a run time error because here syntax is correct but at run time it is trying to call a non existed method:

```
<script type="text/javascript">
<!--
window.printme();
//-->
</script>
```

Exceptions also affect the thread in which they occur, allowing other JavaScript threads to continue normal execution.

Logical errors:

Logic errors can be the most difficult type of errors to track down. These errors are not the result of a syntax or runtime error. Instead, they occur when you make a mistake in the logic that drives your script and you do not get the result you expected.

You can not catch those errors, because it depends on your business requirement what type of logic you want to put in your program.

The *try...catch...finally* Statement:

The latest versions of JavaScript added exception handling capabilities. JavaScript implements the **try...catch...finally** construct as well as the **throw** operator to handle exceptions.

You can *catch* programmer-generated and *runtime* exceptions, but you cannot *catch* JavaScript syntax errors.

Here is the **try...catch...finally** block syntax:

```
<script type="text/javascript">
<!--
try {
    // Code to run
    [break;]
} catch ( e ) {
    // Code to run if an exception occurs
    [break;]
}[ finally {
    // Code that is always executed regardless of
    // an exception occurring
}]
//-->
</script>
```

The **try** block must be followed by either exactly one **catch** block or one **finally** block (or one of both). When an exception occurs in the **try** block, the exception is placed in **e** and the **catch** block is executed. The optional **finally** block executes unconditionally after try/catch.

Examples:

Here is one example where we are trying to call a non existing function this is causing an exception raise. Let us see how it behaves without with **try...catch**:

```
<html>
<head>
<script type="text/javascript">
<!--
function myFunc()
{
   var a = 100;

   alert("Value of variable a is : " + a );

}
//-->
</script>
</head>
<body>
<p>Click the following to see the result:</p>
<form>
<input type="button" value="Click Me" onclick="myFunc();" />
```

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

Now let us try to catch this exception using **try...catch** and display a user friendly message. You can also suppress this message, if you want to hide this error from a user.

```
<html>
<head>
<script type="text/javascript">
<!--
function myFunc()
{
   var a = 100;

   try {
      alert("Value of variable a is : " + a );
   } catch ( e ) {
      alert("Error: " + e.description );
   }
}
//-->
</script>
</head>
<body>
<p>Click the following to see the result:</p>
<form>
<input type="button" value="Click Me" onclick="myFunc();" />
</form>
</body>
</html>
```

You can use **finally** block which will always execute unconditionally after try/catch. Here is an example:

```
<html>
<head>
<script type="text/javascript">
<!--
function myFunc()
{
   var a = 100;

   try {
      alert("Value of variable a is : " + a );
   }catch ( e ) {
      alert("Error: " + e.description );
   }finally {
      alert("Finally block will always execute!" );
   }
}
//-->
</script>
</head>
```

```
<body>
<p>Click the following to see the result:</p>
<form>
<input type="button" value="Click Me" onclick="myFunc();" />
</form>
</body>
</html>
```

The *throw* Statement:

You can use **throw** statement to raise your built-in exceptions or your customized exceptions. Later these exceptions can be captured and you can take an appropriate action.

Following is the example showing usage of **throw** statement.

```
<html>
<head>
<script type="text/javascript">
<!--
function myFunc()
{
   var a = 100;
   var b = 0;

   try{
      if ( b == 0 ){
         throw{ "Divide by zero error." );
      }else{
         var c = a / b;
      }
   }catch ( e ) {
      alert("Error: " + e );
   }
}
//-->
</script>
</head>
<body>
<p>Click the following to see the result:</p>
<form>
<input type="button" value="Click Me" onclick="myFunc();" />
</form>
</body>
</html>
```

You can raise an exception in one function using a string, integer, Boolean or an object and then you can capture that exception either in the same function as we did above, or in other function using **try...catch** block.

The *onerror()* Method

The **onerror** event handler was the first feature to facilitate error handling for JavaScript. The **error** event is fired on the window object whenever an exception occurs on the page. Example:

```
<html>
<head>
<script type="text/javascript">
<!--
window.onerror = function () {
   alert("An error occurred.");
}
//-->
</script>
</head>
<body>
<p>Click the following to see the result:</p>
<form>
<input type="button" value="Click Me" onclick="myFunc();" />
</form>
</body>
</html>
```

The **onerror** event handler provides three pieces of information to identify the exact nature of the error:

- **Error message** . The same message that the browser would display for the given error
- **URL** . The file in which the error occurred
- **Line number** . The line number in the given URL that caused the error

Here is the example to show how to extract this information

```
<html>
<head>
<script type="text/javascript">
<!--
window.onerror = function (msg, url, line) {
   alert("Message : " + msg );
   alert("url : " + url );
   alert("Line number : " + line );
}
//-->
</script>
</head>
<body>
<p>Click the following to see the result:</p>
<form>
<input type="button" value="Click Me" onclick="myFunc();" />
</form>
</body>
</html>
```

You can display extracted information in whatever way you think it is better.

You can use **onerror** method to show an error message in case there is any problem in loading an image as follows:

```
<img src="myimage.gif"
    onerror="alert('An error occurred loading the image.')" />
```

You can use **onerror** with many HTML tags to display appropriate messages in case of errors.

JavaScript - Form Validation

Form validation used to occur at the server, after the client had entered all necessary data and then pressed the Submit button. If some of the data that had been entered by the client had been in the wrong form 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 and over burdening 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 data was entered into each form field that required it. This would need just 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. This would need to put more logic to test correctness of data.

We will take an example to understand the process of validation. Here is the simple form to proceed :

```
<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 we will show how to do a basic form validation. In the above form we are calling **validate()** function to validate data when **onsubmit** event is occurring. Following is 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() ;
```