What is JavaScript?

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

JavaScript was first known as **LiveScript,** but Netscape changed its name to JavaScript, possibly because of the excitement being generated by Java. JavaScript made its first appearance in Netscape 2.0 in 1995 with the name **LiveScript**. The general-purpose core of the language has been embedded in Netscape, Internet Explorer, and other web browsers.

Client-side JavaScript

Client-side JavaScript is the most common form of the language. The script should be included in or referenced by an HTML document for the code to be interpreted by the browser.

It means that a web page need not be a static HTML, but can include programs that interact with the user, control the browser, and dynamically create HTML content.

The JavaScript client-side mechanism provides many advantages over traditional CGI server-side scripts. For example, you might use JavaScript to check if the user has entered a valid e-mail address in a form field.

The JavaScript code is executed when the user submits the form, and only if all the entries are valid, they would be submitted to the Web Server.

JavaScript can be used to trap user-initiated events such as button clicks, link navigation, and other actions that the user initiates explicitly or implicitly

Limitations of JavaScript

We cannot 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 cannot be used for networking applications because there is no such support available.
- JavaScript doesn't have any multithreading or multiprocessor capabilities.

Features of JavaScript?

- Full integration with HTML/CSS
- Simple things can be done simply
- Supported by all browsers and enabled by default

- JavaScript is a lightweight, interpreted programming language.
- Designed for creating network-centric applications.
- Open and cross-platform

Also, remember that JavaScript is alive, under constant development. New features are coming, the modern ECMAScript standard brings nice features, and new JavaScript engines work better and faster.

Trends in JavaScript. HTML5.

Besides the modern ECMAScript specification, which enhances the language itself, the browsers-makers are adopting features from HTML 5. That's a related standard, or more precisely a pack of standards, containing many features which people have been wanting for ages.

Just a few:

- Reading/writing files on visitor's disk (with proper security to keep it safe).
- A database embedded into the browser, which allows to store data on client side.
- Multithreading (can use multiple CPUs).
- Video playback.
- Drawing 2d and 3d, with hardware acceleration, just like in modern games.

The trend is: JavaScript is enhancing its abilities. It is becoming more and more powerful, trying to reach desktop apps.

Modern browsers improve their engines to achieve higher JavaScript execution speed. They also fix bugs and try to follow the standard.

The trend is: JavaScript is becoming faster and more stable.

It is also very important that new standards HTML5 and ECMAScript 5/6 are mostly compatible with older standards. That means they don't break existing apps.

Alternative technologies

Abilities of JavaScript in certain areas are limited. That's why alternative technologies are used.

The point is: all of them play really well with JavaScript. Sometimes, a task can't be solved by JavaScript, but there are possibilities to use JavaScript + Java or JavaScript + Flash or drop in ActiveX.

Java

As you already know, JavaScript is not Java. In fact, they don't have much in common besides a name. Java is a different language that allows to write applets and embed them into HTML-page.

A Java applet is a program for a browser just like an executable file. A programmer writes it in Java, then compiles and puts a link on it into HTML. A browser then opens a page, finds the reference to an applet, downloads and executes it (if Java is enabled).

An important difference between a Java applet and JavaScript is their abilities.

- Java can do *everything*, just like an installed executable. For security, an unsafe action requires visitor's confirmation.
- Java development is easier: IDEs are cool.
- Java takes more time to load and is heavy to start running.
- Java needs to be installed and enabled.
- Java is not integrated with HTML page, it runs in a separate container within the page.

Adobe Flash

Adobe Flash initially appeared as a cross-browser platform and language for multimedia, for making web alive with animation, audio and video. But there are other interesting features in Flash.

A *flash movie* is a compiled program, written in ActionScript, usually bundles with images and other resources.

- Great stuff for networking (sockets, UDP for P2P)
- Support for complex multimedia: images, audio, video is much more advanced compared to HTML5. Camera and microphone are here too.
- Comfortable IDE for Flash, no browser incompatibilities.
- Flash has to be installed and enabled.
- Flash is not integrated with HTML page, it runs in a separate container within the page.
- Security limitations on Flash are almost as strict as on JavaScript.

As of now, there is a high pressure on Flash monopoly in many areas of it's use. For example, HTML5 provides means for browser to play video, draw, etc. Browsers which implement HTML5 stuff don't require Flash to do such things. And most browsers really go forward in making HTML5 features work.

But **both Java and Flash functions can call JavaScript and vice versa**, so usually a site uses mostly JavaScript, but also Java/Flash in places where JavaScript can't cope.

ActiveX, browser plugins/extensions

ActiveX is a great, but IE-only thing. It allows to write a program in C language which integrates with the page *if the visitor allows*.

- Integrated with HTML/CSS
- Written in C, hence very fast and featured.
- Can do *everything* if the visitor allows it to install.
- Internet Explorer only. Chrome has partial support that has to be enabled.
- Development of ActiveX is difficult.

Programs on Windows provide interfaces which can be used by ActiveX. So, a page can call Microsoft Word, or load a document into Excel, etc.

Other technologies: Silverlight, XUL, VBscript

These technologies are much less widespread.

- **XUL** is a language for interfaces, useful if you only write for Mozilla browsers or making extensions to Firefox. Also used for desktop applications.
- **Silverlight** is an Adobe Flash competitor from Microsoft based on .NET. It runs best on Windows, the cross-platform support improves gradually time. Mostly used for Windows-based applications and intranet.
- **VBscript** is an outdated attempt of Microsoft to do a JavaScript-like language based on Visual Basic. It is not being developed, VBScripts lack many abilities of JavaScript and hence are almost not used in modern web programming.

Summary

- JavaScript is unique because it is a wide-spread and it's integration with HTML/CSS is best.
- JavaScript has the bright and more-or-less compatible future.
- But a good JavaScript programmer should keep other technologies in mind too. For example, Flash, Java have their own unique features. They are able to call JavaScript functions and vice versa.
- So there are tasks which can be solved using a combinations of JavaScript + Flash, JavaScript + Java. Examples: selecting uploading multiple files at once (Flash), using camera and microphone (Flash), doing complex multimedia and graphics, including calculations (Flash, Java) and much more. You'll meet them on your way.

JavaScript – Syntax

JavaScript can be implemented using JavaScript statements that are placed within the **<script>... </script>** HTML tags in a web page.

You can place the **<script>** tags, containing your JavaScript, anywhere within you web page, but it is normally recommended that you should keep it within the **<head>** tags.

The <script> tag alerts the browser program to start interpreting all the text between these tags as a script. A simple syntax of your JavaScript will appear 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".

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

Your First JavaScript Script

Let us take a sample example to print out "Hello World". We added an optional HTML comment that surrounds our JavaScript code. This is to save our code from a browser that does not support JavaScript. The comment ends with a "//-->". Here "//" signifies a comment in JavaScript, so we add that to prevent a browser from reading the end of the HTML comment as a piece of JavaScript code. 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. Take a look at the following code.

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

Semicolons are Optional

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

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

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

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

Case Sensitivity

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

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

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

Comments in JavaScript

JavaScript supports both C-style and C++-style comments, Thus −

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

Example

The following example shows how to use comments in JavaScript.

```
<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>
```

JavaScript - Placement in HTML File

There is a flexibility given to include JavaScript code anywhere in an HTML document. However the most preferred ways to include JavaScript in an HTML file are as follows −

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

JavaScript in External File

As you begin to work more extensively with JavaScript, you will be likely to 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>
```

Variable

A variable is a named storage location that holds a value, e.g., sum, average, name and message.

A variable name (aka identifier) must begins with a letter (a-z, A-Z), underscore '_', or dollar sign '$'. Subsequent characters could contain digits (0-9). Take note that jQuery uses $, $(), which are valid identifiers. Hyphen '-' and space are not allowed, hence, font-size, roman new, are NOT valid identifiers.

- JavaScript is case sensitive. A ROSE is not a rose and is not a Rose.
- A variable is declared using keyword var.
- You can assign (and re-assign) a value to a variable using the assignment '=' operator.

JavaScript Variables

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

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

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

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

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

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

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

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

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

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

JavaScript Variable Scope

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

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

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

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

Literals / constant

A literal is a fixed value, e.g., 5566, 3.14, "Hello", true, that can be assigned to a variable, or form part of an expression.

Data Types

JavaScript is object-oriented. But, It supports both *primitive types* and *objects*.

JavaScript supports these primitive types:

- **string**: a sequence of characters. Strings literals are enclosed in a pair of single quotes or double quotes (e.g., `"Hello"`, `'world'`).
- **number**: takes both integer (e.g., `5566`) or floating-point (e.g., `3.14159265`).
- **boolean**: takes boolean literal of either `true` or `false` (in lowercase).

- **undefined**: takes a special literal value called `undefined`. Take note that `undefined` is both a type and a literal value.

JavaScript also supports these object types and value (we shall discuss object later):

- **object**: for general objects.
- **function**: for function objects.
- **null**: A special literal value for unallocated `object`.

Unlike most of the general programming languages (such as Java/C/C++/C#) which are strongly type, JavaScript is loosely type (similar to most of the scripting languages such as UNIX Shell Script, Perl, Python).

You do not have to explicitly declare the type of a variable (such as int and float) during declaration. The type is decided when a value is assigned to that variable. If a number is assigned, the variable takes on the number type and can perform numeric operations such as addition and subtraction. If a string is assigned, the variable takes on the string type and can perform string operations such as string concatenation

Operator typeof

You can use the operator `typeof` to check the type of a variable.

The undefined Type and undefined Literal Value

An undeclared variable (via var keyword) takes on a special type called undefined. You cannot refer to its value.

When a variable is declared (via var keyword) without assigning an initial value, it takes on the type undefined and holds a special value called undefined (uninitialized is probably more precise?!), As soon as a value is assigned, the variable takes on the type of that value.

The act of putting a value into a variable sets its type. You can change the type of a variable by re-assigning a value of another type. Types are converted automatically as needed during execution (known as dynamically-typed).

Constants

You can create a read-only, named *constant* with the keyword `const` (in place of `var`). For example,

```
const SIZE = 9;
```

The number Type, Literals & Operations

A variable of type `number` holds a number, either an integer or a floating-point number.

Integer literals can be expressed in:

- **Decimal**: begins with a digit `1` to `9` (not `0`), e.g., `123` or `-456`,
- **Octal**: begins with a digit `0`, e.g., `0123` or `-0456`,
- **Hexadecimal**: begins with `0x` (or `0X`), e.g., `0xA1B2` or `-0XA1B2`.
- **Binary**: begins with `0b` (or `0B`), e.g., `0b10011100` or `-0B11001100`. [Binary may not be supported in some browsers.]

Floating-point literals can be expressed in the usual form (e.g., `3.1416`) or scientific notation, e.g., `-1.23e4`, `4.56E-7`.

JavaScript also provides some special `number` literals:

- **infinity**: e.g., `1/0`
- **-infinity**: e.g., `-1/0`
- **NaN (Not-a-Number):** e.g., `0/0`, or converting the string `'Hello'` to a number.

Arithmetic Operations

Arithmetic operations, as tabulated below, can be applied to numbers. The following results are obtained assuming that `x=5`, `y=2` before the operation.

Operator	Description	Example (x=5, y=2)	Result
+	Addition	`z = x + y;`	`z is 7`
-	Subtraction (or Unary Negation)	`z = x - y;`	`z is 3`
*****	Multiplication	`z = x * y;`	`z is 10`
/	Division	`z = x / y;`	`z is 2.5`
%	Modulus (Division Remainder)	`z = x % y;`	`z is 1`
++	Unary Pre- or Post-Increment	`y = x++; z = ++x;` `Same as: y = x; x = x+1; x = x+1; z = x;`	`y is 5; z is 7; x is 7`
--	Unary Pre- or Post-Decrement	`y = --x; z = x--;` `Same as: x = x-1; y = x; z = x; x = x-1;`	`y is 4; z is 4; x is 3`

In JavaScript, arithmetic operations are always performed in double-precision floating-point. That is, `1/2` gives `0.5` (instead of 0 in Java/C/C++). You may use the built-in function `parseInt()` to truncate a floating-point value to an integer, e.g., `parseInt(55.66)` and `parseInt("55.66")` gives `55`.

You may also use the built-in mathematical functions such as `Math.round()`, `Math.floor()`, `Math.ceil()` for converting a floating-point number to an integer.

Arithmetic cum Assignment Operators

These are short-hand operators to combine two operations.

Operator	Description	Example	Result
+=	Addition cum Assignment	`x += y;`	`Same as: x = x + y;`
-=	Subtraction cum Assignment	`x -= y;`	`Same as: x = x - y;`
***=**	Multiplication cum Assignment	`x *= y;`	`Same as: x = x * y;`
/=	Division cum Assignment	`x /= y;`	`Same as: x = x / y;`
%=	Modulus cum Assignment		

The Number Operations

- `parseInt(`*aString*`)`: Parse the *aString* until the first non-digit, and return the number; or `NaN`.
- `parseFloat(`*aString*`)`:
- `Math.round(`*aNumber*`), Math.floor(`*aNumber*`), Math.ceil(`*aNumber*`)`:
- `Math.random()`: Generate a random number between 0 (inclusive) and 1 (exclusive).
- `isNaN(`*aString*`)`: return true if the *aString* is not a number. For example,

```
document.write(isNaN('123'));      // false

document.write(isNaN('1.23'));     // false

document.write(isNaN('123abc'));   // true
```

It is interesting to note that JavaScript does not have counterpart functions like `isNumber()`, `isNumeric()`.

- `Number(`*aString*`)`: Return the number represented by *aString*, or `NaN`. Take that that this function name begins with uppercase, because this is a type casting operation.
- `.toFixed(`*deciamlPlaces*`)`: Return this number/string to the given number of decimal places. For example,

```
var n = 1.2345;

document.write(n.toFixed(2));  // 1.23

document.write(n);             // 1.2345 - No change!
```

The string Type, Literals & Operations

A string is a sequence of characters enclosed within a pair of single quotes or double quotes (e.g., "Hello", 'world', "5566", '3.1416').

Unlike Java/C/C++, but like HTML/CSS's attributes, you can use either single quotes or double quotes for string. This is handy as you can use single quotes if the string contains double quotes (e.g., '<div id="header"></div>'), without using the escape sequences (e.g., "<div id=\"header\"></div>").

JavaScript is dynamically-type, and performs type conversion automatically. When a string value is used in arithmetic operations (such as subtraction or multiplication), JavaScript runtime automatically converts the string to a number if it represents a valid number; or a special number called NaN (not-a-number) otherwise

The '+' Operator: Addition or Concatenation?

If both the operands to a `'+'` operator are `numbers`, it performs the usual numeric **addition**. However, if one (or both) of the operand is a `string`, the `'+'` operator is *overloaded* to perform string **concatenation**. The other operand will be converted to a `string`, if necessary

String's Properties

`.length`: E.g., aString.length returns the length of the string.

String's Operations

.toUpperCase()

returns the uppercase string

.toLowerCase()

Returns the lowercase string

.charAt(index)

returns the character at the index position. Index begins from 0. Negative index can be used, which counts from the end of the string

.substring(beginIndex, endIndex)

returns the substring from beginIndex (inclusive) to endIndex (exclusive)

.substr(beginIndex, length)

returns the substring from beginIndex of length.

.indexOf(searchString, fromIndex)

Return the beginning index of the first occurrence of searchstring, starting from an optional fromIndex (default of 0); or -1 if not found.

.lastIndexOf(searchString, fromIndex)

Return the beginning index of the last occurrence of substring, starting from an optional fromIndex (default of string.length); or -1 if not found.

.slice(beginIndex, endIndex)

Return the substring from beginIndex (inclusive) to endIndex (exclusive).

.split(delimiter)

returns an array by splitting the string using delimiter.

JavaScript – Operators

What is an operator?

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

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

Arithmetic Operators

JavaScript supports the following arithmetic operators –

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

Sr.No	Operator and Description
1	**+ (Addition)** Adds two operands **Ex:** A + B will give 30
2	**- (Subtraction)** Subtracts the second operand from the first **Ex:** A - B will give -10

3	*** (Multiplication)** Multiply both operands **Ex:** A * B will give 200
4	**/ (Division)** Divide the numerator by the denominator **Ex:** B / A will give 2
5	**% (Modulus)** Outputs the remainder of an integer division **Ex:** B % A will give 0
6	**++ (Increment)** Increases an integer value by one **Ex:** A++ will give 11
7	**-- (Decrement)** Decreases an integer value by one **Ex:** A-- will give 9

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

Example

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

```
<html>
   <body>

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

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

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

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

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

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

            b = --b;
            document.write("--b = ");
            result = --b;
            document.write(result);
            document.write(linebreak);
         //-->
      </script>

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

Comparison Operators

JavaScript supports the following comparison operators –

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

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

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

Example

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

```
<html>
   <body>

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

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

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

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

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

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

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

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

Logical Operators

JavaScript supports the following logical operators −

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

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

Example

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

```
<html>
   <body>
   
      <script type="text/javascript">
         <!--
            var a = true;
            var b = false;
            var linebreak = "<br />";
            
            document.write("(a && b) => ");
            result = (a && b);
            document.write(result);
            document.write(linebreak);
            
            document.write("(a || b) => ");
            result = (a || b);
            document.write(result);
            document.write(linebreak);
            
            document.write("!(a && b) => ");
            result = (!(a && b));
            document.write(result);
            document.write(linebreak);
         //-->
      </script>
      
      <p>Set the variables to different values and different operators and then 
try...</p>
```

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

Assignment Operators

JavaScript supports the following assignment operators –

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

Example

Try the following code to implement assignment operator in JavaScript.

```
<html>
   <body>
   
      <script type="text/javascript">
         <!--
            var a = 33;
            var b = 10;
            var linebreak = "<br />";
         
            document.write("Value of a => (a = b) => ");
            result = (a = b);
            document.write(result);
            document.write(linebreak);
         
            document.write("Value of a => (a += b) => ");
            result = (a += b);
            document.write(result);
            document.write(linebreak);

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

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

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

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

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

Conditional Operator (? :)

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

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

Example

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

```
<html>
   <body>

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

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

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

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

typeof Operator

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

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

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

Type	String Returned by typeof
Number	"number"
String	"string"
Boolean	"boolean"
Object	"object"
Function	"function"
Undefined	"undefined"
Null	"object"

```
<html>
   <body>

      <script type="text/javascript">
         <!--
            var a = 10;
            var b = "String";
            var linebreak = "<br />";
            result = (typeof b == "string" ? "B is String" : "B is Numeric");
            document.write("Result => ");
            document.write(result);
            document.write(linebreak);

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

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

Flow Control - Decision

JavaScript provides these flow control construct. The syntax is the same as Java/C/C++.

if...else Statement

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

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

JavaScript supports the 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.

```
if (expression){

   Statement(s) to be executed if expression is true

}
```

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

Example

Try the following example to understand how the if statement works.

```
<html>
   <body>

      <script type="text/javascript">
         <!--
            var age = 20;

            if( age > 18 ){
               document.write("<b>Qualifies for driving</b>");
            }
         //-->
      </script>

      <p>Set the variable to different value and then try...</p>
   </body>
</html>
```

if...else statement:

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

```
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, the given statement(s) in the 'if' block, are executed. If the expression is false, then the given statement(s) in the else block are executed.

```
<html>
   <body>

      <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>

      <p>Set the variable to different value and then try...</p>
   </body>
</html>
```

if...else if... statement

The **if...else if...** statement is an advanced form of **if…else** that allows JavaScript to make a correct decision out of several conditions.

```
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 a part of the **else** clause of the previous statement. Statement(s) are executed based on the true condition, if none of the conditions is true, then the **else** block is executed.

```
<html>
   <body>

      <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>

      <p>Set the variable to different value and then try...</p>
   </body>
<html>
```

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.

The objective of a **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 the end of a particular case. If they were omitted, the interpreter would continue executing each statement in each of the following cases.

```
<html>
   <body>

      <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>

       <p>Set the variable to different value and then try...</p>
    </body>
 </html>
```

Looping Statements

While writing a program, you may encounter a situation where you need to perform an action repeatedly. In such situations, you would need to write loop statements to reduce the number of lines.

The while Loop

The most basic loop in JavaScript is the **while** loop which would be discussed in this chapter. The purpose of a **while** loop is to execute a statement or code block repeatedly as long as an **expression** is true. Once the expression becomes **false,** the loop terminates.

Syntax

The syntax of **while loop** in JavaScript is as follows –

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

Example

```
<html>
   <body>

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

```
            <!--
               var count = 0;
               document.write("Starting Loop ");
            
               while (count < 10){
                  document.write("Current Count : " + count + "<br />");
                  count++;
               }
            
               document.write("Loop stopped!");
            //-->
         </script>
         
         <p>Set the variable to different value and then try...</p>
      </body>
   </html>
```

The do...while Loop

The **do...while** loop is like 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

The syntax for **do-while** loop in JavaScript is as follows −

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

Note − Don't miss the semicolon used at the end of the do...while loop.

Example

```
   <html>
      <body>
      
         <script type="text/javascript">
            <!--
               var count = 0;
               
               document.write("Starting Loop" + "<br />");
               do{
                  document.write("Current Count : " + count + "<br />");
                  count++;
               }
               
               while (count < 5);
               document.write ("Loop stopped!");
            //-->
         </script>
```

```
            <p>Set the variable to different value and then try...</p>
        </body>
    </html>
```

For Loop

- The '**for**' loop is the most compact form of looping. It 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 a given condition is true or not. If the condition is true, then the code given inside the loop will be executed, otherwise the control will come out of the loop.
- The **iteration statement** where you can increase or decrease your counter.

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

Syntax

The syntax of for loop is JavaScript is as follows –

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

Example

```
    <html>
        <body>

            <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>

            <p>Set the variable to different value and then try...</p>
        </body>
    </html>
```

for...in loop

The **for...in** loop is used to loop through an object's properties. As we have not discussed Objects yet, you may not feel comfortable with this loop. But once you understand how objects behave in JavaScript, 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.

```
<html>
   <body>

      <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>

      <p>Set the variable to different object and then try...</p>
   </body>
</html>
```

The Array Object

The **Array** object lets you store multiple values in a single variable. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

An array is an indexed collection. An array can be used to store a list of items (elements) under a single name with an integral index. You can reference individual element via the integral index in the form of an ArrayName[Index]. Furthermore, you can conveniently process all the elements of an array collectively via a loop.

Syntax

Use the following syntax to create an **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's length

The length of the array is maintained in a variable called length, which can be accessed via anArrayName.length. In fact, the property .length returns the last integral index plus 1, as JavaScript's array index is 0-based. Nonetheless, you are allow to manipulate the .length

Dynamic Array

Unlike Java/C/C++, the JavaScript array is dynamically allocated. You can add more elements to an array. You can also remove the content of an element using keyword delete

Array Methods

join([separator])

joins the elements of an array together into a single string, separated by the separator (defaulted to ','). For example,

```
var fruits = ["apple", "orange", "banana"];
document.write(fruits.join());     // apple,orange,banana (string)
document.write(fruits.join("|"));  // apple|orange|banana (string)
```

aString.split([separator, limit]):

Reverse of `join()`. Take a string and split into an array based on the *separator*. For example,

```
var str = 'apple, orange, banana';
var fruits = str.split();
document.write(fruits);   // ["apple, orange, banana"]  (3 items)

str = 'apple|*|orange|*|banana';
fruits = str.split();
document.write(fruits);   // ["apple|*|orange|*|banana"]  (one item)

fruits = str.split('|*|');
document.write(fruits);   // ["apple", "orange", "banana"]  (3 items)

fruits = str.split('*');
document.write(fruits);   // ["apple|", "|orange|", "|banana"]  (3 items)
```

.concat(value1, value2, ..., valueN)

returns a new array composing of this array and the given arrays or values. For example,

```
var fruits = ['apple', 'orange'];
moreFruits = fruits.concat('banana', 'watermelon');
document.write(moreFruits);  // ["apple", "orange", "banana", "watermelon"]
document.write(fruits);      // ["apple", "orange"]  (No change!)
```

.reverse()

reverses the order of elements in the array, the first becomes last. For example,

```
var a = ["1", "2", "3", "4", "5"];
a.reverse();
document.write(a);   // ["5", "4", "3", "2", "1"]
```

.sort()

sorts the elements in the array. For example,

```
var a = ["8", "10", "a", "b"];
a.sort();
document.write(a);   // ["10", "8", "a", "b"]
   // Strings are sorted based on ASCII (Unicode) order.
   // Hence, "10" is before "8", as '1' is before '8'
```

```
a = [8, 20, 5, 100];
a.sort();
document.write(a);   // [100, 20, 5, 8]
   // numbers are also sorted based on ASCII (Unicode) order!!!
```

Take note take, by default, number are also sorted based on ASCII (Unicode) order. To sort numbers numerically, you can supply a callback comparison function. The function shall take 2 arguments, say `a` and `b`, and return a negative number if `a < b;` a positive number if `a > b`; and 0 if `a == b. For example,`

```
var a = [8, 20, 5, 100];
a.sort( function(a, b) { return a - b; } );
document.write(a);   // [5, 8, 20, 100]
```

.slice(beginIndex, endIndex)

extracts and returns a section of an array from *`beginIndex`* (inclusive) to *`endIndex`* (exclusive). For example,

```
var a = ["a", "b", "c", "d", "e"];
document.write(a.slice(1, 4));  // ["b", "c", "d"]
                             // include start index but exclude end index
document.write(a);              // ["a", "b", "c", "d", "e"]  (No Change)
```

.splice(startIndex, countToRemove, insertItem1, insertItem2, ...)

removes elements from an array, and insert elements at its place. For example,

```
var a = ["1", "2", "3", "4", "5"];
a.splice(2, 2, "a", "b", "c", "d");
document.write(a);   // ["1", "2", "a", "b", "c", "d", "5"]
   // 2 elements starting at index 2 (i.e., indexes 2 and 3) were removed
   //   and the given elements inserted at its place (index 2).
   // Renumber the indexes.
document.write(a.length);   // 7
```

.indexOf(searchItem[, startIndex]) and .lastIndexOf(searchItem[, startIndex])

search for the index of the item forward or backward. It returns -1 if item cannot be found. For example,

```
var a = ['a', 'b', 'c', 'a', 'b', 'c'];
var searchItem = 'b';
var idx = a.indexOf(searchItem)
document.write(idx);  // 1
```

```
// Search again from the last found index
idx = a.indexOf(searchItem, idx + 1)
document.write(idx);  // 4

idx = a.indexOf(searchItem, idx + 1)
document.write(idx);  // -1 (not found)


// Search backwards
document.write(a.lastIndexOf('a'));    // 3
document.write(a.lastIndexOf('a', 2)); // 0
```

.push()
adds one or more elements to the end of an array and returns the resultant length of the array.

.pop()
removes and return the last element from an array.

.shift()
removes and returns the first element from an array.

.unshift()
adds one or more elements to the front of an array and returns the resultant length of the array.

Iterating an Array

.forEach(callback)
takes a function with an argument which iterates through all the items in the array.

```
var fruits = ['apple', 'orange', 'banana'];
fruits.forEach( function(item) {
   document.write('processing item: ' + item);
});
      // processing item: apple
      // processing item: orange
      // processing item: banana
```

NOTE: Using `for-index-in` or `for-item-of` or simple `for` loop is probably more conventional.

.map(callback)

return a new array, which contains all the return value from executing `callback` on each item. For example,

```
var fruits = ['apple', 'orange', 'banana'];
var results = fruits.map( function(item) {
   document.write('processing item: ' + item);
   return item.toUpperCase();
});
document.write(results);
      // processing item: apple
      // processing item: orange
      // processing item: banana
      // ["APPLE", "ORANGE", "BANANA"]
```

.filter(callback)

return a new array, containing the items for which `callback` returned `true`. For example,

```
var fruits = ['apple', 'orange', 'banana'];
var result = fruits.filter( function(item) {
   document.write('processing item: ' + item);
   return /.*e$/.test(item);   // ends with 'e'?
});
document.write(result);
      // processing item: apple
      // processing item: orange
      // processing item: banana
      // ["apple", "orange"]
```

Associative Array

An associative array is an array of key-value pair. Instead of using numbers 0, 1, 2,... as keys as in the regular array, you can use anything (such as string) as key in an associative array. Associative arrays are used extensively in JavaScript and jQuery.

JavaScript does not support native associative array (it does not support native array too). In JavaScript, associative arrays (and arrays) are implemented as objects

You can create an associative array via the Object Initializer.

```
// Create an associative array of key-value pairs
var aStudent = {
   name:      'peter',  // string
   id:        8888,     // number
   isMarried: false     // boolean
};

// Add a new property via "index" operator, like an array
aStudent['age'] = 24;

// Use for..in loop to access all items
for (key in aStudent) {
   document.write(key + ": " + aStudent[key]);
}
   // name: peter
   // id: 8888
   // isMarried: false
   // age: 24

// An associative array is actually an object
document.write(typeof aStudent);  // object

// You can also access a property via "dot" operator, like an object
aStudent.height = 190;
document.write(aStudent.height);   // 190
```

Functions

A function is a "subprogram" that can be called by code external (or internal in the case of recursion) to the function. Like the program itself, a function is composed of a sequence of statements called the function body. Values can be passed to a function, and the function will return a value.

In JavaScript, functions are first-class objects, because they can have properties and methods just like any other object. What distinguishes them from other objects is that functions can be called. In brief, they are `Function` objects.

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

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

To return a value other than the default, a function must have a **`return`** statement that specifies the value to return. A function without a return statement will return a default value. In the case of a constructor called with the `new` keyword, the default value is the value of its `this` parameter. For all other functions, the default return value is undefined.

The parameters of a function call are the function's arguments. Arguments are passed to functions by value. If the function changes the value of an argument, this change is not reflected globally or in the calling function. However, object references are values, too, and they are special: if the function changes the referred object's properties, that change is visible outside the function, as shown in the following example:

JavaScript allows us to write our own functions as well.

Function Definition

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

Syntax

The basic syntax is shown here.

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

Example

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

Calling a Function

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

```
<html>
   <head>

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

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

      <form>
         <input type="button" onclick="sayHello()" value="Say Hello">
      </form>

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

Function Parameters

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

Example

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

```
<html>
```

```
      <head>
      
         <script type="text/javascript">
            function sayHello(name, age)
            {
               document.write (name + " is " + age + " years old.");
            }
         </script>
         
      </head>
      <body>
         <p>Click the following button to call the function</p>
         
         <form>
            <input type="button" onclick="sayHello('Zara', 7)" value="Say Hello">
         </form>
         
         <p>Use different parameters inside the function and then try...</p>
      </body>
   </html>
```

The return Statement

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

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

Example

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

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

      </head>

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

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

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

     </body>
   </html>
```

JavaScript Form Validation

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

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

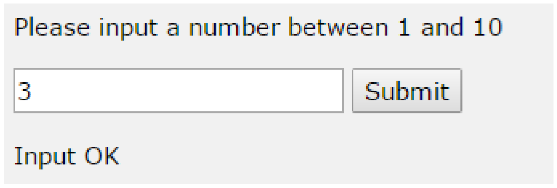
- **Basic Validation** − First of all, the form must be checked to make sure all the mandatory fields are filled in. It would require just a loop through each field in the form and check for data.
- **Data Format Validation** − Secondly, the data that is entered must be checked for correct form and value.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

HTML form validation can be done by JavaScript. If a form field (fname) is empty, this function alerts a message, and returns false, to prevent the form from being submitted:

```
function validateForm() {
    var x = document.forms["myForm"]["fname"].value;
    if (x == "") {
        alert("Name must be filled out");
        return false;
    }
}
<form name="myForm" action="/action_page_post.php" onsubmit="return validateForm()" method="post">
Name: <input type="text" name="fname">
```

```
<input type="submit" value="Submit">
</form>
```

Validate Numeric Input


Please input a number between 1 and 10

3 Submit

Input OK

```
<html>
<body>

<p>Please input a number between 1 and 10:</p>

<input id="numb">
<button type="button" onclick="myFunction()">Submit</button>
<p id="demo"></p>
<script>
function myFunction() {
    var x, text;

    // Get the value of the input field with id="numb"
    x = document.getElementById("numb").value;

    // If x is Not a Number or less than one or greater than 10
    if (isNaN(x) || x < 1 || x > 10) {
        text = "Input not valid";
    } else {
        text = "Input OK";
    }
    document.getElementById("demo").innerHTML = text;
}
</script>

</body>
</html>
```

Basic Form Validation

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

Html

```
<html>

   <head>
      <title>Form Validation</title>

      <script type="text/javascript">
         <!--
            // Form validation code will come here.
         //-->
      </script>

   </head>

   <body>
      <form action=" " 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>
```

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

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

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

         if( document.myForm.Zip.value == "" ||
             isNaN( document.myForm.Zip.value ) ||
             document.myForm.Zip.value.length != 5 )
         {
            alert( "Please provide a zip in the format #####." );
            document.myForm.Zip.focus() ;
            return false;
         }

         if( document.myForm.Country.value == "-1" )
         {
            alert( "Please provide your country!" );
            return false;
         }
         return( true );
      }
   //-->
</script>
```

Data Format Validation

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

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

```
<script type="text/javascript">
   <!--
      function validateEmail()
      {
         var emailID = document.myForm.EMail.value;
         atpos = emailID.indexOf("@");
         dotpos = emailID.lastIndexOf(".");

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

Restricting input to alphanumeric characters

In the following example, the single input box, inputfield,

- Must not be empty
- contain only alphanumeric characters and spaces

Only if both tests are passed can the form be submitted (when the script returns a value of true).

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

  function checkForm(form)
  {
    // validation fails if the input is blank
    if(form.inputfield.value == "") {
      alert("Error: Input is empty!");
      form.inputfield.focus();
      return false;
    }

    // regular expression to match only alphanumeric characters and spaces
    var re = /^[\w ]+$/;

    // validation fails if the input doesn't match our regular expression
```

```
        if(!re.test(form.inputfield.value)) {
          alert("Error: Input contains invalid characters!");
          form.inputfield.focus();
          return false;
        }

        // validation was successful
        return true;
      }

    </script>
```

The pre-defined class \w represents any alphanumeric character or the '_' underscore.

The regular expression **^[\w]+$** will fail if the input is empty as it requires at least one character (because we used + instead of *). The first test in the example is therefore only necessary in order to provide a different error message when the input is blank.

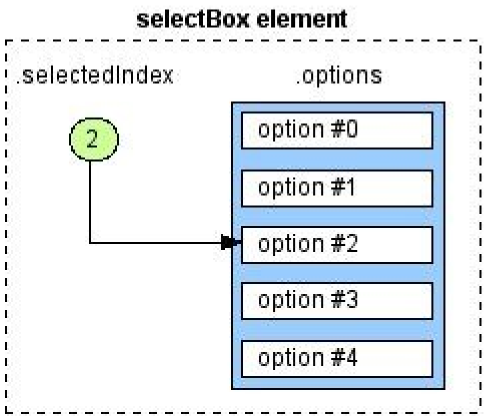
The purpose of a form validation script is to return a boolean value (true or false) to the onsubmit event handler. A value of true means that form will be submitted while a false value will block the form from being submitting.
The focus() command is used to set the focus to the problem element.

Select/Combo/Drop-down boxes

The value of a SELECT input element is accessed using:

```
var selectBox = form.fieldname;
var selectedValue = selectBox.options[selectBox.selectedIndex].value
var selectedText = selectBox.options[selectBox.selectedIndex].text
```

where fieldname is the SELECT element, which has an array of options and a value selectedIndex that tells you which option has been selected. The illustration below shows this relationship:



Note that the 'I' in selectedIndex needs to be capitalised - JavaScript functions and variables are **always** case-sensitive.

If you define a value for the OPTION elements in your SELECT list, then .value will return that, while .text will return the text that is visible in the browser. Here's an example of what this refers to:

```
  <option value="value">text</option>
```

Checkboxes

These really are simple:

```
form.checkboxfield.checked
```

will return a boolean value (true or false) indicating whether the checkbox is in a 'checked' state.

```
  function checkForm(form)
  {
    if(form.checkboxfield.checked) {
      alert("The checkbox IS checked");
    } else {
      alert("The checkbox IS NOT checked");
    }
    return false;
  }
```

You don't need to test using form.checkboxfield.checked == true as the value is already boolean.

Radio buttons

Radio buttons are implemented as if they were an array of checkboxes. To find out which value (if any) has been selected, you need to loop through the array until you find which one has been selected:

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

function checkRadio(field)
{
  for(var i=0; i < field.length; i++) {
    if(field[i].checked) return field[i].value;
  }
  return false;
}

</script>
```

The form handler function is then the following:

```
  function checkForm(form)
  {
    if(radioValue = checkRadio(form.radiofield)) {
      alert("You selected " + radioValue);
```

```
            return true;
          } else {
            alert("Error: No value was selected!");
            return false;
          }
        }
```

There is a bug in the above function in that it doesn't handle the case when there is only a single radio button. In that case field.length is undefined and the function will always return false. Below is a patched version:

```
function checkRadio(field)
{
  if((typeof field.length == "undefined") && (field.type == "radio")) {
    if(field.checked) return field.value;
  } else {
    for(var i=0; i < field.length; i++) {
      if(field[i].checked) return field[i].value;
    }
  }
  return false;
}
```

In the case of a single radio button we have nothing to loop through so just return either it's value, if the radio button is checked, or *false*

Checkbox arrays

If you're working with arrays of checkboxes to submit data to a server-side script then you might already have some grey hairs from trying to figure out how to validate the input using JavaScript.

The problem is that, to have the data submitted in a 'nice' format to the server, the name attributes of all the checkboxes in the array are often set to the same value: a name ending with []. This makes it difficult to address them directly using JavaScript.

In this example, the checkboxes are defined as:

```
<input type="checkbox" name="pref[]" value="value"> label
```

When you submit the form you will be notified through an alert message how many items you checked, and what they were. This is calculated using a new function:

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

// Original JavaScript code by Chirp Internet: www.chirp.com.au
// Please acknowledge use of this code by including this header.
```

```
function checkArray(form, arrayName)
{
  var retval = new Array();
  for(var i=0; i < form.elements.length; i++) {
    var el = form.elements[i];
    if(el.type == "checkbox" && el.name == arrayName && el.checked) {
      retval.push(el.value);
    }
  }
  return retval;
}

</script>
```

The form handler that calls this function and displays the alert is as follows:

```
  function checkForm(form)
  {
    var itemsChecked = checkArray(form, "pref[]");
    alert("You selected " + itemsChecked.length + " items");
    if(itemsChecked.length > 0) {
      alert("The items selected were:\n\t" + itemsChecked);
    }
    return false;
  }
```

The checkArray function returns an array containing all the selected checkbox values.

Normally you would modify this so that you could submit or not submit the form based on the number of items selected. For example "at least two" or "no more than five"

Combining Form Elements in Conditions

In more complicated forms you will want to set conditions on the form that combine multiple elements. For example, a text input that only needs to have a value if a checkbox is checked:

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

  function checkForm(form)
  {
    ...

    if(form.checkboxname.checked && (form.textinputname.value == "")) {
      alert("Error: error message");
      form.textinputname.focus();
      return false;
    }

    ...
    return true;
  }

</script>
```

or conditions that vary according to a radio button selection:

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

 function checkRadio(field)
  {
    for(var i=0; i < field.length; i++) {
      if(field[i].checked) return field[i].value;
    }
    return false;
  }

  function checkForm(form)
  {
    ...

    var radioValue = checkRadio(radiofield);
    switch(radioValue)
    {
      case "Red":
        <conditions to apply if 'Red' is selected>
        break;

      case "Blue":
        <conditions to apply if 'Blue' is selected>
        break;

      default:
        <conditions to apply in all other cases>
    }

    ...
    return true;
    }

</script>
```

Using simple logical operators and the functions supplied above you can do all sorts of client-side form validation.

JavaScript: A simple modal feedback form with no plugins

1. Setting up the HTML

Below you will see code for a basic HTML form. The form is contained in a DIV identified as `#modal_window` which will be hidden by default when the page loads.

Above this, there is a button that will appear on the page, #modal_open, and the form is contained by `#modal_wrapper` (any other containing element can also be used).

```
<p><button id="modal_open">Open Email Form</button></p>

<div id="modal_wrapper">
<div id="modal_window">

<div style="text-align: right;">
       <a id="modal_close" href="#">close <b>X</b></a></div>

<p>Complete the form below to send an email:</p>

<form id="modal_feedback" method="POST" action="#" accept-charset="UTF-8">
<p><label>Your Name<strong>*</strong><br>
<input type="text" autofocus required size="48" name="name" value=""></label></p>
<p><label>Email Address<strong>*</strong><br>
<input type="email" required title="Please enter a valid email address" size="48" name="email"
value=""></label></p>
<p><label>Subject<br>
<input type="text" size="48" name="subject" value=""></label></p>
<p><label>Enquiry<strong>*</strong><br>
<textarea required name="message" cols="48" rows="8"></textarea></label></p>
<p><input type="submit" name="feedbackForm" value="Send Message"></p>
</form>

</div> <!-- #modal_window -->
</div> <!-- #modal_wrapper -->
```

What we want to achieve is for when 'Open Email Form' is clicked, an overlay appears over the website and the modal window is displayed at the centre of the screen.

The autofocus attribute on the first input field works in Safari, Chrome and Opera in setting focus on that field when the modal window is displayed. It doesn't appear to work in Internet Explorer or Firefox.

The HTML5 required attribute is recognised in IE10 and higher, and many browsers will now display the associated title text as a tooltip if the field is left empty

2. Required CSS styles

We've used a minimum of CSS here, so the form is rather plain. Feel free to jazz it up yourself later. The CSS styles should be added to your existing style sheets or included in the document HEAD.

```
<style type="text/css">

  #modal_wrapper.overlay::before {
    content: " ";
    width: 100%;
    height: 100%;
    position: fixed;
    z-index: 100;
    top: 0;
    left: 0;
    background: #000;
    background: rgba(0,0,0,0.7);
  }

  #modal_window {
    display: none;
    z-index: 200;
    position: fixed;
    left: 50%;
    top: 50%;
    width: 360px;
    overflow: auto;
    padding: 10px 20px;
    background: #fff;
    border: 5px solid #999;
    border-radius: 10px;
    box-shadow: 0 0 10px rgba(0,0,0,0.5);
  }
```

```
      #modal_wrapper.overlay #modal_window {
        display: block;
      }

    </style>
```

In desktop browsers position: fixed will keep the popup centred on the screen as the page scrolls or the browser window is resized.

For this example we've set the width of the modal window to 360 pixels. This can be changed to suit your requirements. If you don't set either a width or max-width than any long segments of text will force the modal window wider.

3. JavaScript Form Validation

You should recognize the code here as a basic form validation script. It displays an alert message and prevents the form from being submitted if any of the mentioned fields are left blank.

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

      // Original JavaScript code by Chirp Internet: www.chirp.com.au
      // Please acknowledge use of this code by including this header.

      var checkForm = function(e)
      {
        var form = (e.target) ? e.target : e.srcElement;
        if(form.name.value == "") {
          alert("Please enter your Name");
          form.name.focus();
          e.preventDefault ? e.preventDefault() : e.returnValue = false;
          return;
        }
        if(form.email.value == "") {
          alert("Please enter a valid Email address");
          form.email.focus();
          e.preventDefault ? e.preventDefault() : e.returnValue = false;
          return;
        }
        if(form.message.value == "") {
          alert("Please enter your comment or question in the Message box");
          form.message.focus();
          e.preventDefault ? e.preventDefault() : e.returnValue = false;
          return;
        }
```

```
  };
</script>
```

The checkForm() function will be assigned to the form submit handler later in the code.

4. Opening and closing the modal window

Now we come to the juicy part of the code. We want our modal window, containing the feedback form, to be displayed when the trigger is clicked, and to be hidden when any of a number of different events take place - when the 'close' link is clicked, the 'Esc' key is pressed, or when a mouse click occurs outside the modal window.

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

  // Original JavaScript code by Chirp Internet: www.chirp.com.au
  // Please acknowledge use of this code by including this header.

  var modal_init = function() {

    var modalWrapper = document.getElementById("modal_wrapper");
    var modalWindow  = document.getElementById("modal_window");

    var openModal = function(e)
    {
      modalWrapper.className = "overlay";
      var overflow = modalWindow.offsetHeight - document.documentElement.clientHeight;
      if(overflow > 0) {
        modalWindow.style.maxHeight = (parseInt(window.getComputedStyle(modalWindow).height) - 
overflow) + "px";
      }
      modalWindow.style.marginTop = (-modalWindow.offsetHeight)/2 + "px";
      modalWindow.style.marginLeft = (-modalWindow.offsetWidth)/2 + "px";
      e.preventDefault ? e.preventDefault() : e.returnValue = false;
    };

    var closeModal = function(e)
    {
      modalWrapper.className = "";
      e.preventDefault ? e.preventDefault() : e.returnValue = false;
    };

    var clickHandler = function(e) {
      if(!e.target) e.target = e.srcElement;
      if(e.target.tagName == "DIV") {
        if(e.target.id != "modal_window") closeModal(e);
      }
    };

    var keyHandler = function(e) {
      if(e.keyCode == 27) closeModal(e);
    };

    if(document.addEventListener) {
      document.getElementById("modal_open").addEventListener("click", openModal, false);
      document.getElementById("modal_close").addEventListener("click", closeModal, false);
      document.addEventListener("click", clickHandler, false);
      document.addEventListener("keydown", keyHandler, false);
    } else {
      document.getElementById("modal_open").attachEvent("onclick", openModal);
      document.getElementById("modal_close").attachEvent("onclick", closeModal);
      document.attachEvent("onclick", clickHandler);
      document.attachEvent("onkeydown", keyHandler);
    }

  };

</script>
```

The openModal() function assigns a class 'overlay' to the wrapper DIV which triggers the CSS to display the overlay and modal window. It also centres the modal window to the screen using the calculated width and height.

To hide the modal window the closeModal function simply removes the 'overlay' class, hiding the extra elements. If the form is re-opened any information already entered will still be there.

The rest of the code is for tracking key presses to detect and 'Esc' (key code 27) or a click on the background.

5. Initialization

The final step is to add an 'onload' handler to the page. This is used firstly to assign the form validation script to our form, and secondly to initialise the modal window code and events:

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

  if(document.addEventListener) {
    document.getElementById("modal_feedback").addEventListener("submit", checkForm, false);
    document.addEventListener("DOMContentLoaded", modal_init, false);
  } else {
    document.getElementById("modal_feedback").attachEvent("onsubmit", checkForm);
    window.attachEvent("onload", modal_init);
  }

</script>
```

6. Final Markup

The JavaScript code and CSS styles can of course be moved to external files. The CSS should be included in the HEAD and the JavaScript at the end of the BODY - as is standard:

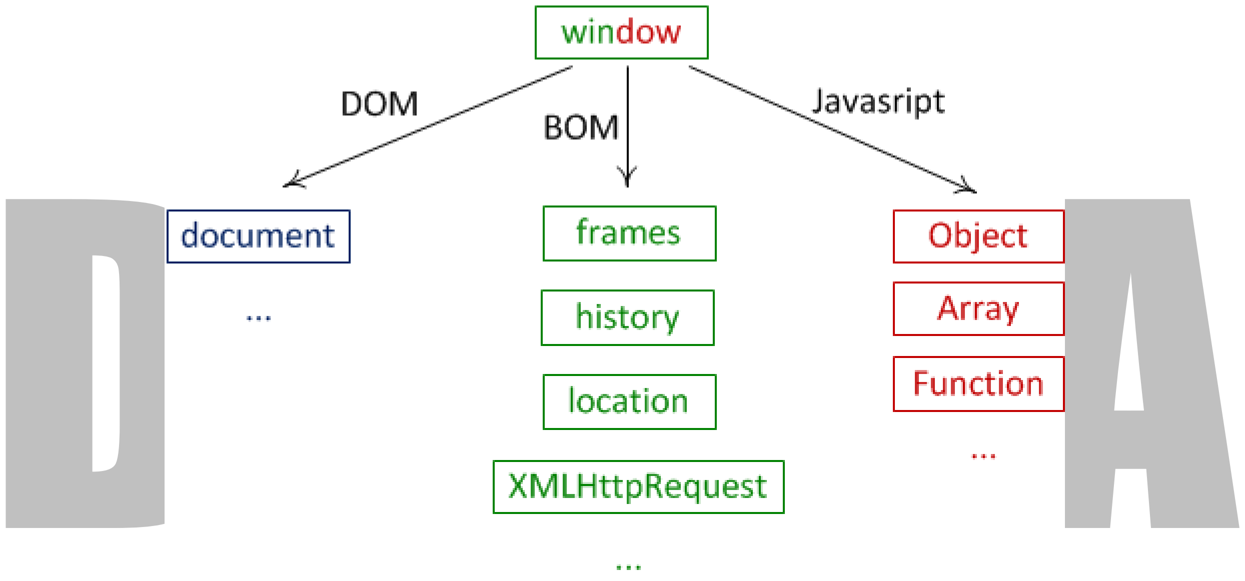
```
<!DOCTYPE html>
<html>

<head>
  <title>...</title>
  ...
  <link rel="stylesheet" type="text/css" href="/feedback-modal-window.css">
</head>

<body>

... html code ...

<script type="text/javascript" src="/feedback-modal-window.js"></script>
</body>
</html>
```

Document Object Model (DOM) API for JavaScript

Document Object Model (DOM), is a standard API that allows programmers to access and manipulate the contents of an HTML/XHTML/XML document dynamically inside their program. It models an HTML/XHTML/XML document as an object-oriented, tree-like structure, known as a DOM-tree, consisting of nodes resembling the tags (elements) and contents. DOM also provides an interface for event handling, allowing you to respond to user's or browser's action.

The global structure

The browser provides access to a large hierarchy of objects for developers to manipulate. You can see a part of it below:



On the top is the window, also called global object. All other objects form 3 groups.

Document Object Model (DOM)

document and related objects allow to access contents of the page, modify elements etc. Most interaction with HTML is handled here.

There is a pack of standards for DOM, developed by W3C. You can find it at W3C DOM page. There are three levels of DOM, each level expands on the previous. Modern browsers generally support pre-W3C features from the browser dark-ages, called DOM 0.

Browser Object Model (BOM)

BOM is a pack of objects that allow to control the browser, e.g change current URL, access frames, do background requests to server with XMLHttpRequest etc. Functions like alert,confirm,prompt also belong BOM, they are provided by the browser.

Many BOM features are standartized in HTML5, but not all.

JavaScript objects and functions

JavaScript itself is a language which gives us access to DOM, BOM and provides objects and functions of its own.

JavaScript follows the ECMA-262 standard.

The global window object mixes browser window functionality (methods focus(),open() etc) with being a JavaScript global object. That's why it is both green and red.

Finding and Selecting Elements

In JavaScript, we often use DOM API to select elements within the current document, so as to access or manipulate their contents. The most commonly-used functions are:

document.getElementById(an Id)

Returns the element with the given unique id.

```
<input type="text" id="foo">
var elm = document.getElementById("foo");
var input = elm.value;
```

document.getElementsByTagName(aTagName)

Returns an array of elements with the given tag name.

```
<input type="text">
var elms = document.getElementByTagName("input");
var input = elms[0].value;
```

document.getElementsByClassName(aClassName)

Returns an array of elements with the given class attribute name.

```
<input type="text" class="bar">
var elms = document.getElementsByClassName("bar");
var input = elms[0].value;
```

document.getElementsByName(aName)

Returns an array of elements with the given name attribute.

```
<input type="checkbox" name="gender" value="m">Male
<input type="checkbox" name="gender" value="f">Female
var x = document.getElementsByName("gender");
for (var i = 0; i < x.length; ++i) {
   if (x[i].checked) {
      value = x[i].value;
      break;
   }
}
```

Beside the above selection functions, there are many other selection functions available

- **`document.images`** returns an array of all <img> elements, same as document.getElementsByTagName("img").
- **`document.forms`** return an array of all <form> elements.
- **`document.links`** and **`document.anchors`** return all the hyperlinks <a href=...> and anchors <a name=...> elements.

innerHTML Property

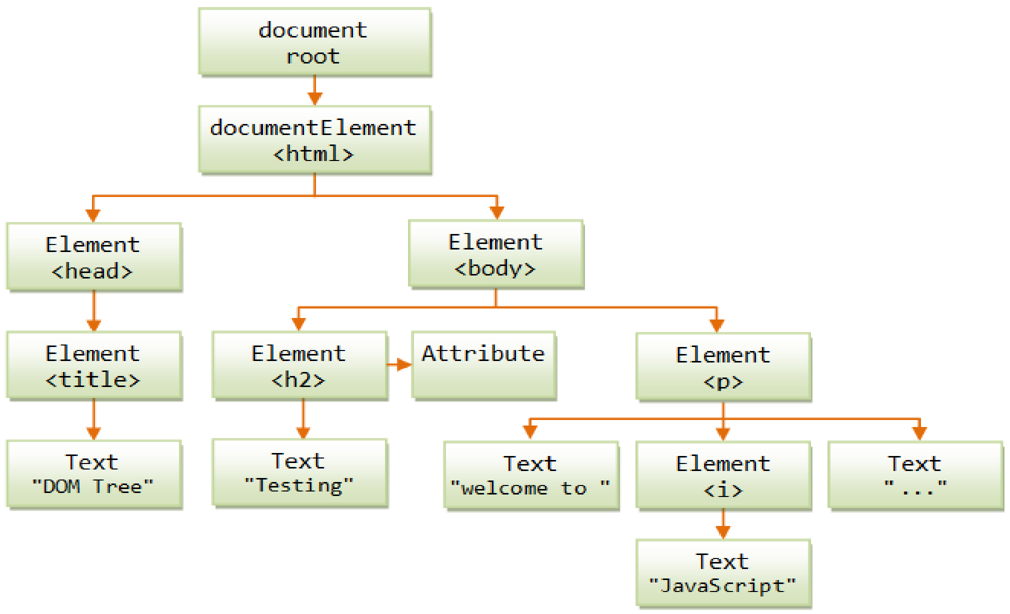
You can access and modify the content of an element via the "innerHTML" property, which contains all the texts (includes nested tags) within this element.

```
<p id="magic">Hello, <em>Paul</em></p>
<script>
  var elm = document.getElementById("magic");
  // Read content
  alert(elm.innerHTML);    // Hello, <em>Paul</em>
  // Change content
  elm.innerHTML = "Good day, <strong>Peter</strong>";
  alert(elm.innerHTML);    // Good day, <strong>Peter</strong>
</script>
```

DOM Tree & Nodes

W3C recommends that you access and manipulate HTML elements via the DOM tree and nodes. However, it is really an overkill for writing simple JavaScripts. I present them here again for completeness.

When a browser loads an HTML page, it builds DOM models a web page in a hierarchical tree-like structure composing of nodes, resembling its HTML structure.



An example of an HTML document with the corresponding DOM-tree is given follow. Take note that the text content of an element is stored in a separate Text node

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>DOM Tree</title>
</head>
<body>
  <h2 onmouseover="this.style.color='red'"
      onmouseout="this.style.color=''">Testing</h2>
  <p>welcome to <i>JavaScript</i>...</p>
</body>
</html>
```

DOM-tree types of nodes:

1. **Document Node**: the root node representing the entire HMTL document.
2. **Element node**: represents an HTML element (or tag). An element node may have child nodes, which can be either element or text node. Element node may also have attributes.
3. **Text Node**: contains the text content of an element.
4. **Others**: such as comment, attribute.

DOM node properties:

- **nodeName**: contain the name of the node, which is read-only. The `nodeName` for an Element node is the tag-name; `nodeName` for the Document node is `#document`; `nodeName` for Text nodes is `#text`.
- **nodeValue**: contain the value of the node. `nodeValue` for Text node is the text contained; `nodeValue` for Element node is undefined.

- **`nodeType`**: an integer indicating the type of the node, e.g., Element (1), Attribute (2), Text (3), Comment (8), Document (9).
- **`parentNode`**: reference to parent node. There is only one parent node in a tree structure.
- **`childNodes`**: array (or node-list) of child nodes.
- **`firstChild`**, **`lastChild`**: reference to the first and last child node.
- **`prevSibling`**, **`nextSibling`**: reference to the previous and next sibling in the same level.

Take note of the difference between singular and plural terms. For example, `parentNode` refer to *the* parent node (each node except root has one and only one parent node), `childNodes` holds an array of all the children nodes.

The root node of the DOM tree is called **`document`**. The root node `document` has only one child, called **`document.documentElement`**, representing the **`<html>`** tag, and it acts as the parent for two child nodes representing `<head>` and `<body>` tags, which in turn will have other child nodes. You can also use a special property called `document.body` to access the `<body>` tag directly.

Example

The following JavaScript lists all the nodes in the `<body>` section, in a depth-first search manner, via a recursive function.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>DOM Tree</title>
<script>
  function printNode(node) {
    document.writeln("Node name=" + node.nodeName + ", value=" + node.nodeValue + ", type=" +
node.nodeType + "<br>");
    if (node.hasChildNodes()) {
      var childs = node.childNodes;
      for (var i = 0; i < childs.length; i++) {
        printNode(childs[i]);
      }
    }
  }
</script>
</head>
<body onload="printNode(document.body)"><h2 onmouseover="this.style.color='red'"
  onmouseout="this.style.color=''">Testing</h2><p>welcome to <i>JavaScript</i>...</p></body>
</html>
```

Attribute Properties

To access an attribute of an Element node called **elementName**, you could either use:

- **property** elementName.attributeName, where attributeName is the name of the attribute, or
- **methods** elementName.getAttribute(name) and elementName.setAttribute(name, value).

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Test Attributes</title>
</head>
<body>
  <p id="magic1" align="left">Hello</p>
  <p id="magic2" align="center">Hello, again.</p>

  <script type=text/javascript>
    var node = document.getElementById("magic1");
    document.writeln(node.align);   // Get attribute "align"
    node.align = "center";          // Set attribute "align" to a new value

    node = document.getElementById("magic2");
    document.writeln(node.getAttribute("align"));  // Read attribute "align"
    node.setAttribute("align", "right");           // Write attribute "align"
  </script>
</body>
</html>
```

Attribute style (for CSS)

Element has a property called style, which models CSS style with CSS properties such as `color` and `textAlign`. For example,

```
<p id="magic">Hello</p>
......
document.getElementById("magic1").style.color="green";
document.getElementById("magic1").style.textAlign="right";
```

Manipulating Nodes

A Node object has these functions:

- ***aNode*.hasChildNodes():** returns `true` if this node has at least one child node.

Manipulating child node:

- `aParentNode.`**`insertBefore`**`(newChildNode, existingChildNode)`: insert a node before an existing child node.
- `aParentNode.`**`replaceChild`**`(newChildNode, existingChildNode)`: replace an existing child node.
- `aParentNode.`**`removeChild`**`(childNodeToRemove)`: remove the specified child node.
- `aParentNode.`**`appendChild`**`(nodeToAppend)`: append the given node as the last child.
- `aNode.`**`cloneNode`**`()`:

Creating a New Element (createElement()) and Text Node (createTextNode()), Appending a Node (appendChild())

To create new text node, you can use `document.createTextNode(text)` to create a standalone text-node, followed by an `anElementNode.appendChid(aTextNode)` to append the text node to an element.

Similarly, you can use `document.createElement(tagName)` to create a stand-alone element, followed by an `anElementNode.appendChild(elementToAppend)` to append the created element into an existing element.

For example, we shall create a new text node, as a child of a new `<p>` element. We shall then append the new `<p>` element as the last child of `<body>`.

```
<body>
<p id="magic">Hello</p>
<script>
  alert(document.getElementById("magic").innerHTML);
  var newElm = document.createElement("p");
  newElm.appendChild(document.createTextNode("Hello, again"));
  document.body.appendChild(newElm);
</script>
</body>
```

Inserting a new Node (insertBefore())

```
<body>
<p id="magic">Hello</p>
<script>
  var magicElm = document.getElementById("magic");
  alert(magicElm.innerHTML);
  var newElm = document.createElement("p");
  newElm.appendChild(document.createTextNode("Hello, again"));
  document.body.insertBefore(newElm, magicElm);
</script>
</body>
```

Replacing a Node (replaceChild())

Change the last line to document.body.replaceChild(newElm, magicElm).

Deleting a Node (removeChild())

You can remove a child node from a parent node via aParentNode.removeChild(aChildNode).

For example, let remove the last <p> from <body>.

```
<body>
<p>Hello 1</p>
<p>Hello 2</p>
<p id="magic">Hello 3</p>
<script>
  var elm = document.getElementById("magic");
  alert(elm.innerHTML);
  document.body.removeChild(elm);
</script>
</body>
```

The document object

The `document` object is the root node of the DOM-tree. It can be used to access all the elements in an HTML page.

Properties:

- documentElement, body, title: references the <html>, <body> and <title> tags respectively.
- lastModified, referrer, cookie, domain: information retrieved from the HTTP response header.
- forms[], applets[], images[], embeds[], links[], anchors[]: Arrays containing the respective HTML elements (backward compatible with DOM0).

Methods:

- **write(string), writeln(string):**

Write the specified string to the current document. writeln() (write-line) writes a newline after the string, while write() does not. Take note that browser ignores newlines in an HTML document, you need to write a <br> or <p>...</p> tag for the browser to display a line break.

- **clear():** Clear the document
- **open(), close():** Open/close the document stream
- **getElementById**(), **getElementsByName**(), **getElementsByTagName**(): Select HTML element(s) by id, name, or tag-name, respectively.

Window Object

The JavaScript window object sits at the top of the JavaScript Object hierarchy and **represents the browser window** (or windows if you have more than one browser window open at any one time).

The window object allows developers to perform tasks such as **opening** and **closing** browser windows, displaying alert and prompt dialogs and setting up timeouts (specifying an action to take place after a specified period of time).

Referencing the JavaScript window Object

It is usually necessary to use dot-notation when accessing properties or methods of an object. The window object is the top-level object of the object hierarchy. As such, whenever an object method or property is referenced in a script without the object name and dot prefix it is assumed by JavaScript to be a member of the window object. This means, for example, that when calling the window alert() method to display an alert dialog the window. prefix is not mandatory. Therefore the following method calls achieve the same thing:

```
window.alert();

alert()
```

JavaScript window Object Properties

The JavaScript window object contains a number of properties that can be inspected and used in a script:

window.closed - Used when handling multiple windows, this property indicates whether a window has been closed or not.

window.defaultstatus / window.status - *defaultstatus* specifies the default message displayed in the browser status bar. *status* specifies a temporary message to display in the browser status bar in place of the default. Disabled in many browsers.

window.frames[] - If the window contains frames this array holds the array of frame objects

window.name - Windows opened by a script must be given a name. This property contains the name of the corresponding window object.

window.opener - When a window has been opened in a script contained in another window, this property of the child window contains a reference window which opened it.

window.parent - When working with frames in a window this property contains a reference to the window object that contains the frame.

window.screen - An object which contains information about the screen on which the window is displays (properties contained in this object include *height, width, availHeight, availWidth and colorDepth*).

window.self - A reference to the current window.

window.top - A reference to the top-level window when working with frames

Opening Browser Windows using JavaScript

A new browser window can be opened from a JavaScript script using the open*()* method of the window object. The syntax for opening a new window is as follows:

```
newWindowObj = window.open("URL", "WindowName", "feature, feature, feature ... ");
```

The following provides an explanation of the arguments passed through to the *open()* method:

URL - Specifies the URL of the web page to be loaded into the new window. If no URL is specified a blank window is loaded.

WindowName - Specifies the window name and is used to refer to the window.

features - A comma separated list of features that allow you to customize the appearance of the window. Options are:

Setting	Explanation
width	Specifies the initial width of the browser client window (see innerWidth for size of content area)
height	Specifies the initial height of the browser client window (see innerHeight for size of content area)
innerWidth	Specifies the initial width of the window content area
innerHeight	Specifies the initial height of the window content area
outerWidth	Specifies the initial width of the navigator window
outerHeight	Specifies the initial height of the navigator window
toolbar	Specifies whether the window should contain the browser toolbar or not

status	Specifies whether the window should contain the browser status bar or not
dependent	Specifies whether the window should close in unison with its parent window
menubar	Specifies whether the window should contain the browser menubar
location	Specifies whether the window should contain the browser location/URL box
scrollbars	Hides/Shows browser horizontal/vertical scrollbars
resizable	Specifies whether the user is entitled to resize the window after it appears.
directories	Specifies whether the window should contain the browser personal toolbar.
copyHistory	Specifies whether the new window should contain a copy of the URL history of the invoking window
left	Specified the number of pixels from the left side of the screen to the new window
top	Specified the number of pixels from the top of the screen to the new window
alwaysLowered	Creates a new window that is always positioned beneath the other browser windows. Often used for those annoying pop-under advertisements.
alwaysRaised	Creates a new window that is always positioned over the top of the other browser windows on the screen.
z-lock	Locks the level at which the browser appears in relation to other browser windows.

The height, width and position features are set using numbers. The remaining feature options can be set using *true* or *false* values (also *yes*, *no* and *1* and *0* can be used in place of *true* and *false*). An absent attribute is false. The following example creates a new window with a menubar, specific dimension and no toolbar:

```
newWindowObj = window.open("URL", "WindowName", "toolbar=0, menubar=1, innerHeight=200,
innerWidth=300");
```

Closing Browser Windows using JavaScript

A window can be closed using the window object's *close()* method. The name of the window (specified in the *open()* method) should be referenced when performing a close so that you are certain to close the correct window. For example the following code creates a new window and creates a pushbutton which, when clicked, closes the new window:

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

newWindowObj = window.open ("", "MyWindow");

</script>

<form action="null">

  <input type="button" value="Close Window" onclick="newWindowObj.close()" />

</form>
```

It is also possible to close the window that opened the current window using the *opener* property of the current window object:

```
window.opener.close()
```

Moving and Resizing Windows

A window can be moved to specific coordinates on the screen using the window object's *moveTo()* method which takes x and y coordinates as arguments. The following example moves a new window to location 100, 200 on the screen when the "Move Window" button is pressed:

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

newWindowObj = window.open ("", "MyWindow");

</script>

<form action="null">

  <input type="button" value="Move Window" onclick="newWindowObj.moveTo(100, 200)" />

</form>
```

In addition to moving a window to a specific new location is also possible to move a window relative to its current location on the screen using the *moveBy()* method of the JavaScript window object. Once again the method takes x and y values that are added to the current x and y coordinates of the specified window. Negative values can be used to change the direction of the movement:

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

newWindowObj = window.open ("", "MyWindow");

</script>

<form action="null">

  <input type="button" value="Move Window" onclick="newWindowObj.moveTo(100, 200)" />

</form>
```

The window resizeTo() and resizeBy() methods work similarly in that they allow you to change the size of a window either to a specific size, or to a new size relative to the current size.

Changing Window Focus

When a window is the currently selected window on the screen it is said to have *focus*. Typically, clicking with the mouse pointer in a window gives that window focus. With JavaScript it is possible to programmatically change the focus of a window using the *focus()* and *blur()* methods. The following example displays a new window, blurs it so that the opening window still has focus and provides a button to switch focus to the new window:

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

newWindowObj = window.open ("", "MyWindow");

newWindowObj.blur();

</script>



<form action="null">

  <input type="button" value="Focus New Window" onclick="newWindowObj.focus()" />

</form>
```

Displaying Message Box Dialogs

The JavaScript window object provides methods to display three types of message dialogs, the alert, confirmation and prompt dialogs:

alert - intended to display a message to the user. It contains a message area where the alert message is to be displayed and an "OK" button that the user can click to dismiss the dialog. The *alert()* method takes a single argument representing the message to be displayed in the dialog. The following web page fragment displays an alert dialog with the message "You do not have a valid password" when the "Show Alert" button is clicked:

```
<form action="null">

  <input type="button" value="Show Alert" onclick="window.alert('You do not have a valid
password')" />

</form>
```

confirmation - used when a yes or no response needs to be obtained from the user. This dialog type displays with a message and "OK" and "Cancel" buttons. The *confirm()* method takes the message to be displayed to the user as an argument and returns *true* or *false* depending on whether the user pressed "OK" or "Cancel":

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


function showConfirmation()

{
        var result = confirm("Would you like to continue?");


        if (result)
                document.write("Continue");
        else
                document.write("Do not continue");
}
</script>


<form action="null">
   <input type="button" value="Show Confirmation" onclick="showConfirmation()" />
</form>
```

prompt - designed to enable information to be obtained from the user. The dialog consists of a message to the user, a text input field for the entry of data and OK and Cancel buttons. The *prompt()* method takes the message to be displayed as an argument and returns the value entered by the user:

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


function showPrompt()
{
        var userInput = prompt("Please enter your name:");


        document.write("Hello, " + userInput);
}
</script>


<form action="null">
   <input type="button" value="Show Prompt" onclick="showPrompt()" />
</form>
```