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 is a lightweight, interpreted programming language.
- Designed for creating network-centric applications.
- Complementary to and integrated with Java.
- Complementary to and integrated with HTML.
- Open and cross-platform

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.

Advantages of JavaScript

The merits of using JavaScript are –

- **Less server interaction** – You can validate user input before sending the page off to the server. This saves server traffic, which means fewer loads on your server.

- **Immediate feedback to the visitors** – They don't have to wait for a page reload to see if they have forgotten to enter something.
- **Increased interactivity** – You can create interfaces that react when the user hovers over them with a mouse or activates them via the keyboard.
- **Richer interfaces** – You can use JavaScript to include such items as drag-and-drop components and sliders to give a Rich Interface to your site visitors.

Limitations of JavaScript

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.

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".

So your JavaScript segment will look like –

```
<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 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>
```

This code will produce the following result –

```
Hello World!
```

Whitespace and Line Breaks

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

Semicolons are Optional

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

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

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

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

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

Case Sensitivity

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

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

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

Comments in JavaScript

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

- Any text between a // and the end of a line is treated as a comment and is ignored by JavaScript.
- Any text between the characters /* and */ is treated as a comment. This may span multiple lines.

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 <head>...</head> section

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

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

This code will produce the following results –

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

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

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

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

JavaScript in <body> and <head> Sections

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

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

This code will produce the following result –

JavaScript in External File

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

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

```
<html>
```

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

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

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

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

JavaScript Data types

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

JavaScript allows you to work with three primitive data types –

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

JavaScript Variables

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>
```

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>
```

This produces the following result –

```
local
```

JavaScript Variable Names

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

- You should not use any of the JavaScript reserved keywords as a variable name. For example, **break** or **boolean** variable names are not valid.

- JavaScript variable names should not start with a numeral (0-9). They must begin with a letter or an underscore character. For example,**123test** is an invalid variable name but **_123test** is a valid one.

- JavaScript variable names are case-sensitive. For example, **Name** and **name** are two different variables.

JavaScript Reserved Words

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

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

JavaScript - Operators

JavaScript supports the following types of operators.

- Arithmetic Operators
- Comparison 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>
   </body>
</html>
```

Output

```
a + b = 43
a - b = 23
a / b = 3.3
a % b = 3
a + b + c = 43Test
++a = 35
--b = 8
```

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>

   </body>
</html>
```

Output

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

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>

  </body>
```

```
</html>
```

Output

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

Bitwise Operators

JavaScript supports the following bitwise operators –

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

Sr.No	Operator and Description
1	**& (Bitwise AND)** It performs a Boolean AND operation on each bit of its integer arguments. **Ex:** (A & B) is 2.
2	**| (BitWise OR)** It performs a Boolean OR operation on each bit of its integer arguments. **Ex:** (A | B) is 3.
3	**^ (Bitwise XOR)** It performs a Boolean exclusive OR operation on each bit of its integer arguments. Exclusive OR means that either operand one is true or operand two is true, but not both. **Ex:** (A ^ B) is 1.

4	**~ (Bitwise Not)** It is a unary operator and operates by reversing all the bits in the operand. **Ex:** (~B) is -4.
5	**<< (Left Shift)** It moves all the bits in its first operand to the left by the number of places specified in the second operand. New bits are filled with zeros. Shifting a value left by one position is equivalent to multiplying it by 2, shifting two positions is equivalent to multiplying by 4, and so on. **Ex:** (A << 1) is 4.
6	**>> (Right Shift)** Binary Right Shift Operator. The left operand's value is moved right by the number of bits specified by the right operand. **Ex:** (A >> 1) is 1.
7	**>>> (Right shift with Zero)** This operator is just like the >> operator, except that the bits shifted in on the left are always zero. **Ex:** (A >>> 1) is 1.

Example

Try the following code to implement Bitwise operator in JavaScript.

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

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

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

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

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

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

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

```
            document.write(linebreak);

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

    </body>
</html>
```

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

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

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

Example

Try the following code to implement assignment operator in JavaScript.

```
<html>
  <body>

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

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

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

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

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

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

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

Output

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

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>

  </body>
</html>
```

Output

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

JavaScript conditional statements

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.

Syntax

The syntax for a basic if statement is as follows –

```
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.

Example

```
<html>
  <body>

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

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


    </body>
</html>
```

Output

```
Qualifies for driving
```

if...else statement:

Syntax

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


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

Here JavaScript expression is evaluated. If the resulting value is true, 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.

Example

```
<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>
```

Output

```
Does not qualify for driving
Set the variable to different value and then try...
```

if...else if... statement

Syntax

The syntax of an if-else-if statement is as follows –

```
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.

Example

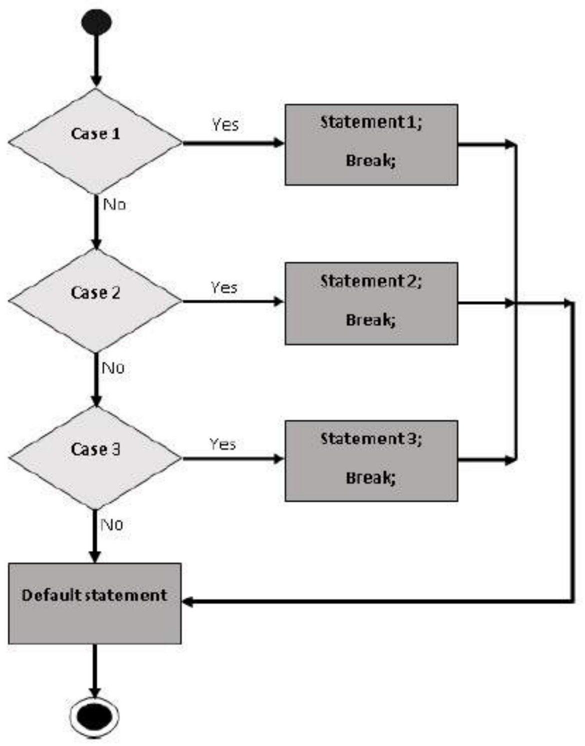
```
<html>
  <body>
      <script type="text/javascript">
      <!--
        var book = "maths";
        if( book == "history" ){
          document.write("<b>History Book</b>");
        }
        else if( book == "maths" ){
          document.write("<b>Maths Book</b>");
        }
        else if( book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

Output

```
Maths Book
```

Switch statement

The following flow chart explains a switch-case statement works.



Syntax

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.

Example

Try the following example to implement switch-case statement.

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

Output

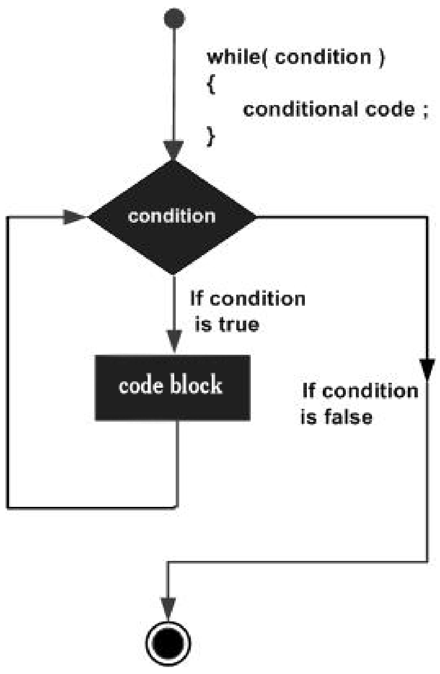
```
Entering switch block
Good job
Exiting switch block
```

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.

Flow Chart

The flow chart of **while loop** looks as follows –



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>
```

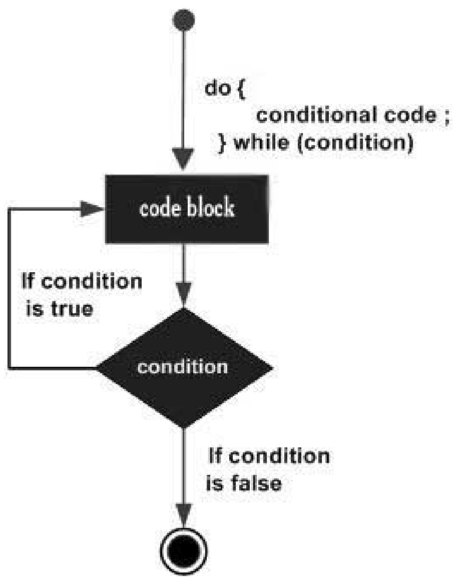
Output

```
Starting Loop
Current Count : 0
Current Count : 1
Current Count : 2
Current Count : 3
Current Count : 4
Current Count : 5
Current Count : 6
Current Count : 7
Current Count : 8
Current Count : 9
Loop stopped!
Set the variable to different value and then try...
```

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

Flow Chart

The flow chart of a **do-while** loop would be as follows –



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

Try the following example to learn how to implement a **do-while** loop in JavaScript.

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

Output

```
Starting Loop
Current Count : 0
Current Count : 1
Current Count : 2
Current Count : 3
Current Count : 4
Loop Stopped!
```

For Loop:

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

Try the following example to learn how a for loop works in JavaScript.

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

Output

```
Starting Loop
```

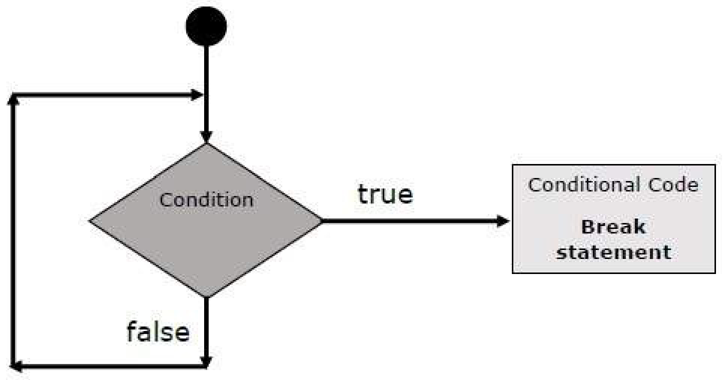
```
Current Count : 0
Current Count : 1
Current Count : 2
Current Count : 3
Current Count : 4
Current Count : 5
Current Count : 6
Current Count : 7
Current Count : 8
Current Count : 9
Loop stopped!
```

break Statement:

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

Flow Chart

The flow chart of a break statement would look as follows –



Example

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

```
<html>
  <body>
    <script type="text/javascript">
      var x = 1;
      document.write("Entering the loop<br /> ");
      while (x < 20)
      {
        if (x == 5){
          break; // breaks out of loop completely
        }
        x = x + 1;
        document.write( x + "<br />");
      }
      document.write("Exiting the loop!<br /> ");
    </script>
    <p>Set the variable to different value and then try...</p>
  </body>
</html>
```

Output

```
Entering the loop
2
3
4
5
Exiting the loop!
```

```
Set the variable to different value and then try...
```

We already have seen the usage of **break** statement inside **a switch** statement.

Continue Statement

The **continue** statement tells the interpreter to immediately start the next iteration of the loop and skip the remaining code block. When a **continue** statement is encountered, the program flow moves to the loop check expression immediately and if the condition remains true, then it starts the next iteration, otherwise the control comes out of the loop.

Example

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

```
<html>
  <body>
    <script type="text/javascript">
        var x = 1;
        document.write("Entering the loop<br /> ");
        while (x < 10)
        {
          x = x + 1;
          if (x == 5){
            continue; // skill rest of the loop body
          }
          document.write( x + "<br />");
        }
        document.write("Exiting the loop!<br /> ");
    </script>
    <p>Set the variable to different value and then try...</p>
  </body>
</html>
```

Output

```
Entering the loop
2
3
4
6
7
8
9
10
Exiting the loop!
```

JavaScript - Functions

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

Function Definition

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

Syntax

The basic syntax is shown here.

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

Example

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

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

Calling a Function:

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

```
<html>
  <head>
      <script type="text/javascript">
      function sayHello()
      {
        document.write ("Hello there!");
      }
    </script>
  </head>
  <body>
    <p>Click the following button to call the function</p>
    <form>
      <input type="button" onclick="sayHello()" value="Say Hello">
    </form>
    <p>Use different text in write method and then try...</p>
  </body>
</html>
```

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-Events

What is an Event?

JavaScript's interaction with HTML is handled through events that occur when the user or the browser manipulates a page. When the page loads, it is called an event. When the user clicks a button, that click too is an event. Other examples include events like pressing any key, closing a window, resizing a window, etc.

HTML Events

- HTML events are **"things"** that happen to HTML elements.
- When JavaScript is used in HTML pages, JavaScript can **"react"** on these events.

Here are some examples of HTML events:

- An HTML web page has finished loading
- An HTML input field was changed
- An HTML button was clicked

Often, when events happen, you may want to do something. JavaScript lets you execute code when events are detected.

In the following example, an **onclick** attribute (with code), is added to a button element:

Example

```
<button onclick='getElementById("demo").innerHTML=Date()'>The time is?</button>
```

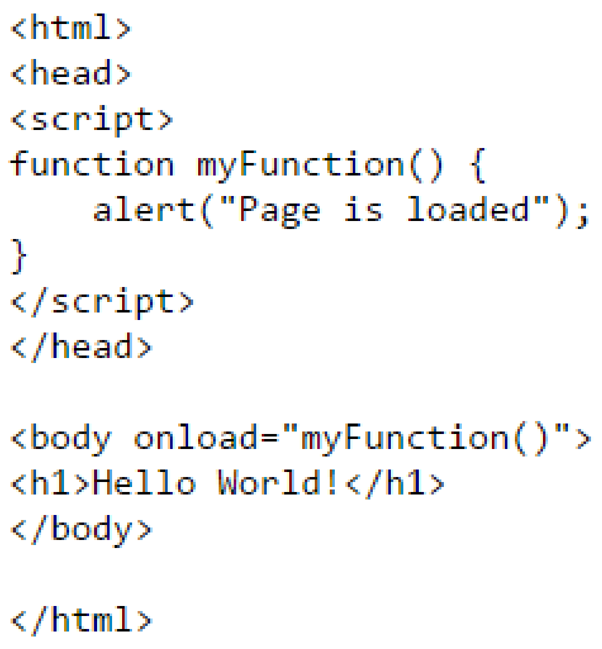
In the example above, the JavaScript code changes the content of the element with id="demo".

JavaScript Events:

Here is a list of some HTML events:

- **onload** : it fires when the page has been loaded
- **onerror**: it fires when an error occurs in a web page
- **onclick** : it fires when the user clicks on an element or a button
- **onmouseover** : it fires when the mouse passes over an element
- **onmouseout** : it fires when the mouse pointer exit from an element.
- **onmousedown** : it fires when the mouse is clicked on an element.
- **onmouseup** : it fires when the releasing a mouse button.
- **onfocus** : it fires when an input field gets focus.
- **onblur** : it fires when a user leaves an input field.
- **onsubmit** :it fires when a user clicks the submit button.
- **onreset** : it fires when a user clicks the reset button.
- **onkeypress** / **onkeydown** : it fires when a user is pressing/holding down a key.
- **onkeyup** : it fires when a user releases a key.

onload Event example:

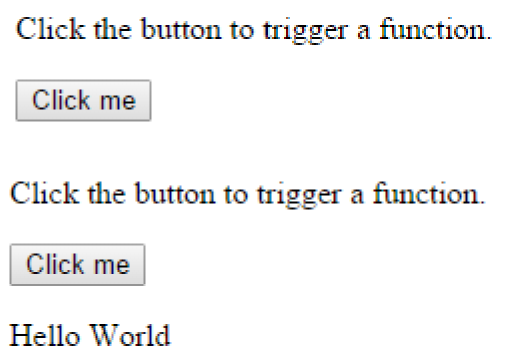


```
<html>
<head>
<script>
function myFunction() {
    alert("Page is loaded");
}
</script>
</head>

<body onload="myFunction()">
<h1>Hello World!</h1>
</body>

</html>
```

onclick Event example:

```
<html>
<head>
<script>
function myFunction() {
    document.getElementById("demo").innerHTML = "Hello World";
}
</script>
</head>
<body>

<p>Click the button to trigger a function.</p>

<button onclick="myFunction()">Click me</button>

<p id="demo"></p>

</body>
</html>
```

Result;



Click the button to trigger a function.

Click the button to trigger a function.

Hello World

onmouseover and onmouseout events example:

```
<html>
<head><title>Hai</title>
     <link rel="icon" href="D:\E Drive Back Up\Wallpapers\111.jpg" type="image/x-icon">

</head>

<body>
<marquee id="test"  direction="left"   width="50%" onmouseover="this.stop()" onmouseout="this.start()">

This is my test
       </marquee>
</body>
</html>
```

onmousedown and onmouseup events example:

```
<html>
<head>
<script>
function myFunction(elmnt, clr) {
    elmnt.style.color = clr;
}
</script>
</head>
<body>

<p onmousedown="myFunction(this,'red')" onmouseup="myFunction(this,'green')">
Click the text to change the color. A function, with parameters, is triggered
when the mouse button is pressed down, and again, with other parameters, when
the mouse button is released.
</p>

</body>
</html>
```

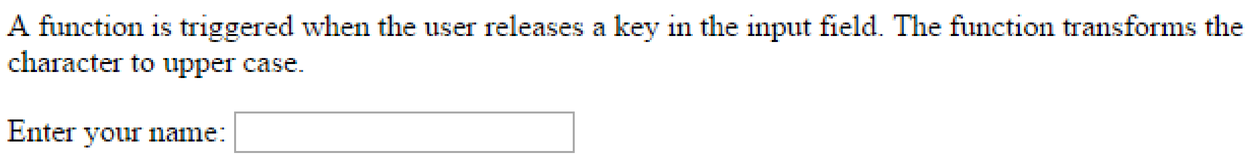
onkeypress / onkeydown events example:

```
<html>
<head>
<script>
function myFunction() {
    alert("You pressed a key inside the input field");
}
</script>
</head>
<body>

<p>A function is triggered when the user is pressing a key in the input field.
</p>

<input type="text" onkeydown="myFunction()">

</body>
</html>
```

onkeyup event example:

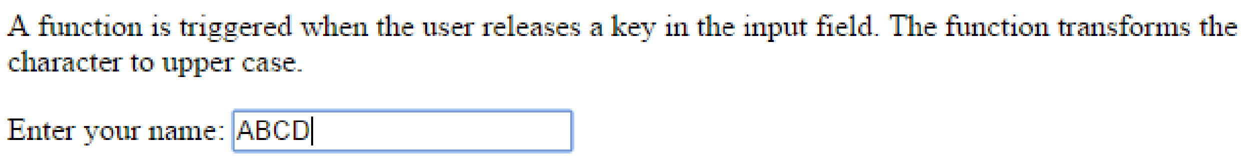
```
<html>
<head>
<script>
function myFunction() {
    var x = document.getElementById("fname");
    x.value = x.value.toUpperCase();
}
</script>
</head>
<body>

<p>A function is triggered when the user releases a key in the input field.
The function transforms the character to upper case.</p>
Enter your name: <input type="text" id="fname" onkeyup="myFunction()">

</body>
</html>
```

Result:



A function is triggered when the user releases a key in the input field. The function transforms the character to upper case.

Enter your name:



A function is triggered when the user releases a key in the input field. The function transforms the character to upper case.

Enter your name: ABCD

JavaScript - Objects

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

- **Encapsulation** – the capability to store related information, whether data or methods, together in an object.
- **Aggregation** – the capability to store one object inside another object.
- **Inheritance** – the capability of a class to rely upon another class (or number of classes) for some of its properties and methods.
- **Polymorphism** – the capability to write one function or method that works in a variety of different ways.
- Objects are composed of attributes. If an attribute contains a function, it is considered to be a method of the object, otherwise the attribute is considered a property.

Object Properties

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

The syntax for adding a property to an object is –

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

For example – The following code gets the document title using the **"title"** property of the **document** object.

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

Object Methods

Methods are the functions that let the object do something or let something be done to it. There is a small difference between a function and a method – at a function is a standalone unit of statements and a method is attached to an object and can be referenced by the **this** keyword.

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

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

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

User-Defined Objects

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

The new Operator

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

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

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

JavaScript Native Objects

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

Here is the list of all important JavaScript Native Objects –

- JavaScript Number Object
- JavaScript Boolean Object
- JavaScript String Object

- JavaScript Array Object
- JavaScript Date Object
- JavaScript Math Object

JavaScript - The Number Object

The **Number** object represents numerical date, either integers or floating-point numbers. In general, you do not need to worry about **Number** objects because the browser automatically converts number literals to instances of the number class.

Syntax

The syntax for creating a **number** object is as follows –

```
var val = new Number(number);
```

In the place of number, if you provide any non-number argument, then the argument cannot be converted into a number, it returns **NaN** (Not-a-Number).

Number Properties

Here is a list of each property and their description.

Property	Description
MAX_VALUE	The largest possible value a number in JavaScript can have 1.7976931348623157E+308
MIN_VALUE	The smallest possible value a number in JavaScript can have 5E-324
NaN	Equal to a value that is not a number.
NEGATIVE_INFINITY	A value that is less than MIN_VALUE.
POSITIVE_INFINITY	A value that is greater than MAX_VALUE

Number Methods

The Number object contains only the default methods that are a part of every object's definition.

Method	Description
toString()	Returns the string representation of the number's value.
valueOf()	Returns the number's value.

JavaScript - The Boolean Object

The **Boolean** object represents two values, either "true" or "false". If *value* parameter is omitted or is 0, -0, null, false, **NaN,** undefined, or the empty string (""), the object has an initial value of false.

Syntax

Use the following syntax to create a **boolean** object.

```
var val = new Boolean(value);
```

Boolean Properties

Here is a list of the properties of Boolean object –

Property	Description
constructor	Returns a reference to the Boolean function that created the object.
prototype	The prototype property allows you to add properties and methods to an object.

Boolean Methods

Here is a list of the methods of Boolean object and their description.

Method	Description
toString()	Returns a string of either "true" or "false" depending upon the value of the object.
valueOf()	Returns the primitive value of the Boolean object.

JavaScript - The Strings Object

Use the following syntax to create a String object –

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

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

String Properties

Here is a list of the properties of String object and their description.

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

String Methods

Here is a list of the methods available in String object along with their description.

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

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

String HTML Wrappers

Here is a list of the methods that return a copy of the string wrapped inside an appropriate HTML tag.

Method	**Description**
anchor()	Creates an HTML anchor that is used as a hypertext target.
big()	Creates a string to be displayed in a big font as if it were in a <big> tag.
blink()	Creates a string to blink as if it were in a <blink> tag.

bold()	Creates a string to be displayed as bold as if it were in a <b> tag.
fixed()	Causes a string to be displayed in fixed-pitch font as if it were in a <tt> tag
fontcolor()	Causes a string to be displayed in the specified color as if it were in a <font color="color"> tag.
fontsize()	Causes a string to be displayed in the specified font size as if it were in a <font size="size"> tag.
italics()	Causes a string to be italic, as if it were in an <i> tag.
link()	Creates an HTML hypertext link that requests another URL.
small()	Causes a string to be displayed in a small font, as if it were in a <small> tag.
strike()	Causes a string to be displayed as struck-out text, as if it were in a <strike> tag.
sub()	Causes a string to be displayed as a subscript, as if it were in a <sub> tag
sup()	Causes a string to be displayed as a superscript, as if it were in a <sup> tag

JavaScript - The Date Object

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

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

Syntax

You can use any of the following syntaxes to create a Date object using Date() constructor.

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

Note – Parameters in the brackets are always optional.

Here is a description of the parameters –

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

- **minute** – Integer value representing the minute segment of a time reading.
- **second** – Integer value representing the second segment of a time reading.
- **millisecond** – Integer value representing the millisecond segment of a time reading.

Date Properties

Here is a list of the properties of the Date object along with their description.

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

Date Methods

Here is a list of the methods used with **Date** and their description.

Method	Description
Date()	Returns today's date and time
getDate()	Returns the day of the month for the specified date according to local time.
getDay()	Returns the day of the week for the specified date according to local time.
getFullYear()	Returns the year of the specified date according to local time.

getHours()	Returns the hour in the specified date according to local time.
getMilliseconds()	Returns the milliseconds in the specified date according to local time.
getMinutes()	Returns the minutes in the specified date according to local time.
getMonth()	Returns the month in the specified date according to local time.
getSeconds()	Returns the seconds in the specified date according to local time.
getTime()	Returns the numeric value of the specified date as the number of milliseconds since January 1, 1970, 00:00:00 UTC.
getTimezoneOffset()	Returns the time-zone offset in minutes for the current locale.
getUTCDate()	Returns the day (date) of the month in the specified date according to universal time.
getUTCDay()	Returns the day of the week in the specified date according to universal time.
getUTCFullYear()	Returns the year in the specified date according to universal time.
getUTCHours()	Returns the hours in the specified date according to universal time.
getUTCMilliseconds()	Returns the milliseconds in the specified date according to universal time.
getUTCMinutes()	Returns the minutes in the specified date according to universal time.

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

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

toString()	Returns a string representing the specified Date object.
toTimeString()	Returns the "time" portion of the Date as a human-readable string.
toUTCString()	Converts a date to a string, using the universal time convention.
valueOf()	Returns the primitive value of a Date object.

JavaScript - The Math Object

The **math** object provides you properties and methods for mathematical constants and functions. Unlike other global objects, **Math** is not a constructor. All the properties and methods of **Math** are static and can be called by using Math as an object without creating it.

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

Syntax

The syntax to call the properties and methods of Math are as follows

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

Math Properties

Here is a list of all the properties of Math and their description.

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

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

Math Methods

Here is a list of the methods associated with Math object and their description

Method	Description
abs()	Returns the absolute value of a number.
acos()	Returns the arccosine (in radians) of a number.
asin()	Returns the arcsine (in radians) of a number.
atan()	Returns the arctangent (in radians) of a number.
atan2()	Returns the arctangent of the quotient of its arguments.

ceil()	Returns the smallest integer greater than or equal to a number.
cos()	Returns the cosine of a number.
exp()	Returns E^N, where N is the argument, and E is Euler's constant, the base of the natural logarithm.
floor()	Returns the largest integer less than or equal to a number.
log()	Returns the natural logarithm (base E) of a number.
max()	Returns the largest of zero or more numbers.
min()	Returns the smallest of zero or more numbers.
pow()	Returns base to the exponent power, that is, base exponent.
random()	Returns a pseudo-random number between 0 and 1.
round()	Returns the value of a number rounded to the nearest integer.
sin()	Returns the sine of a number.
sqrt()	Returns the square root of a number.
tan()	Returns the tangent of a number.

toSource()	Returns the string "Math".

JavaScript HTML DOM

Every web page resides inside a browser window which can be considered as an object. A Document object represents the HTML document that is displayed in that window. The Document object has various properties that refer to other objects which allow access to and modification of document content. The way a document content is accessed and modified is called the **Document Object Model**, or **DOM**. The Objects are organized in a hierarchy.

With the object model, JavaScript gets all the power it needs to create dynamic HTML:

- JavaScript can change all the HTML elements in the page
- JavaScript can change all the HTML attributes in the page
- JavaScript can change all the CSS styles in the page
- JavaScript can remove existing HTML elements and attributes
- JavaScript can add new HTML elements and attributes
- JavaScript can react to all existing HTML events in the page
- JavaScript can create new HTML events in the page

What is the DOM?

- The DOM is a W3C (World Wide Web Consortium) standard.
- The DOM defines a standard for accessing documents:
- *"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."*

The W3C DOM standard is separated into 3 different parts:

- Core DOM - standard model for all document types
- XML DOM - standard model for XML documents
- HTML DOM - standard model for HTML documents

What is the HTML DOM?

The HTML DOM is a standard **object** model and **programming interface** for HTML. It defines:

- The HTML elements as **objects**
- The **properties** of all HTML elements
- The **methods** to access all HTML elements
- The **events** for all HTML elements

The HTML DOM Document Object

- The document object represents your web page.
- If you want to access any element in an HTML page, you always start with accessing the document object.

Finding HTML Elements

Method	**Description**
document.getElementById(*id*)	Find an element by element id
document.getElementsByTagName(*name*)	Find elements by tag name
document.getElementsByClassName(*name*)	Find elements by class name

Changing HTML Elements

Method	Description
element.innerHTML = *new html content*	Change the inner HTML of an element
element.*attribute = new value*	Change the attribute value of an HTML element
element.setAttribute*(attribute, value)*	Change the attribute value of an HTML element
element.style.*property = new style*	Change the style of an HTML element

Adding and Deleting Elements

Method	Description
document.createElement(*element*)	Create an HTML element
document.removeChild(*element*)	Remove an HTML element
document.appendChild(*element*)	Add an HTML element

document.replaceChild(*element*)	Replace an HTML element
document.write(*text*)	Write into the HTML output stream

Adding Events Handlers

Method	**Description**
document.getElementById(*id*).onclick = function(){*code*}	Adding event handler code to an onclick event

JavaScript - The Arrays

The **Array** object lets you store multiple values in a single variable. It stores a fixed-size sequential collection of elements of the same type.

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 Properties

Here is a list of the properties of the Array object along with their description.

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

Array Methods

Here is a list of the methods of the Array object along with their description.

Method	Description
concat()	Returns a new array comprised of this array joined with other array(s) and/or value(s).

every()	Returns true if every element in this array satisfies the provided testing function.
filter()	Creates a new array with all of the elements of this array for which the provided filtering function returns true.
forEach()	Calls a function for each element in the array.
indexOf()	Returns the first (least) index of an element within the array equal to the specified value, or -1 if none is found.
join()	Joins all elements of an array into a string.
lastIndexOf()	Returns the last (greatest) index of an element within the array equal to the specified value, or -1 if none is found.
map()	Creates a new array with the results of calling a provided function on every element in this array.
pop()	Removes the last element from an array and returns that element.
push()	Adds one or more elements to the end of an array and returns the new length of the array.
reduce()	Apply a function simultaneously against two values of the array (from left-to-right) as to reduce it to a single value.
reduceRight()	Apply a function simultaneously against two values of the array (from right-to-left) as to reduce it to a single value.
reverse()	Reverses the order of the elements of an array -- the first becomes the last, and the last becomes the first.

shift()	Removes the first element from an array and returns that element.
slice()	Extracts a section of an array and returns a new array.
some()	Returns true if at least one element in this array satisfies the provided testing function.
toSource()	Represents the source code of an object
sort()	Represents the source code of an object
splice()	Adds and/or removes elements from an array.
toString()	Returns a string representing the array and its elements.
unshift()	Adds one or more elements to the front of an array and returns the new length of the array.

Length :

Description

Javascript array **length** property returns an unsigned, 32-bit integer that specifies the number of elements in an array.

Syntax

Its syntax is as follows – array.length

Return Value: Returns the length of the array.

Example

```
<html>
```

```
  <head>
    <title>JavaScript Array length Property</title>
  </head>
    <body>
    <script type="text/javascript">
      var arr = new Array( 10, 20, 30 );
      document.write("arr.length is : " + arr.length);
    </script>
  </body>
</html>
```

Output

```
arr.length is : 3
```