European Computer Manufacturers Association (ECMAScript) or (ES) is a standard for scripting languages like JavaScript, ActionScript and JScript. It was initially created to standardize JavaScript, which is the most popular implementation of ECMAScript. This tutorial adopts a simple and practical approach through JavaScript to describe the new features in ECMAScript 2015 (ES6), ECMAScript 2016 (ES7), ECMAScript 2017(ES8) and ECMAScript 2018 (ES9).

**Syntax** defines the set of rules for writing programs. Every language specification defines its own syntax.

A JavaScript program can be composed of −

* **Variables** − Represents a named memory block that can store values for the program.
* **Literals** − Represents constant/fixed values.
* **Operators** − Symbols that define how the operands will be processed.
* **Keywords** − Words that have a special meaning in the context of a language.

The following table lists some keywords in JavaScript. Some commonly used keywords are listed in the following table.

|  |  |  |  |
| --- | --- | --- | --- |
| break | as | any | Switch |
| case | if | throw | Else |
| var | number | string | Get |
| module | type | instanceof | Typeof |
| finally | for | enum | Export |
| while | void | this | New |
| null | super | Catch | let |
| static | return | True | False |

* **Modules** − Represents code blocks that can be reused across different programs/scripts.
* **Comments** − Used to improve code readability. These are ignored by the JavaScript engine.
* **Identifiers** − These are the names given to elements in a program like variables, functions, etc. The rules for identifiers are −
  + Identifiers can include both, characters and digits. However, the identifier cannot begin with a digit.
  + Identifiers cannot include special symbols except for underscore (\_) or a dollar sign ($).
  + Identifiers cannot be keywords. They must be unique.
  + Identifiers are case sensitive. Identifiers cannot contain spaces.

The following table illustrates some valid and invalid identifiers.

|  |  |
| --- | --- |
| **Examples of valid identifiers** | **Examples of invalid identifiers** |
| firstName  first\_name  num1  $result | Var#  first name  first-name  1number |

Whitespace and Line Breaks

ES6 ignores spaces, tabs, and newlines that appear in 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.

JavaScript is Case-sensitive

JavaScript is case-sensitive. This means that JavaScript differentiates between the uppercase and the lowercase characters.

Semicolons are Optional

Each line of instruction is called a **statement**. Semicolons are optional in JavaScript.

Example

console.log("hello world")

console.log("We are learning ES6")

A single line can contain multiple statements. However, these statements must be separated by a semicolon.

Comments in JavaScript

**Comments** are a way to improve the readability of a program. Comments can be used to include additional information about a program like the author of the code, hints about a function/construct, etc. Comments are ignored by the compiler.

JavaScript supports the following types of comments −

* **Single-line comments (//)** − Any text between a // and the end of a line is treated as a comment.
* **Multi-line comments (/\* \*/)** − These comments may span multiple lines.

Example

//this is single line comment

/\* This is a

Multi-line comment

\*/

Your First JavaScript Code

Let us start with the traditional “Hello World” example".

var message = "Hello World"

console.log(message)

The program can be analyzed as −

* Line 1 declares a variable by the name message. Variables are a mechanism to store values in a program.
* Line 2 prints the variable’s value to the prompt. Here, the console refers to the terminal window. The function log () is used to display the text on the screen.

Executing the Code

We shall use Node.js to execute our code.

* **Step 1** − Save the file as Test.js
* **Step 2** − Right-click the Test.js file under the working files option in the project-explorer window of the Visual Studio Code.
* **Step 3** − Select Open in Command Prompt option.
* **Step 4** − Type the following command in Node’s terminal window.

node Test.js

The following output is displayed on successful execution of the file.

Hello World

Node.js and JS/ES6

ECMAScript 2015(ES6) features are classified into three groups −

* **For Shipping** − These are features that V8 considers stable.
* **Staged Features** − These are almost completed features but not considered stable by the V8 team.
* **In Progress** − These features should be used only for testing purposes.

The first category of features is fully supported and turned on by default by node. Staged features require a runtime - - harmony flag to execute.

A list of component specific CLI flags for Node.js can be found here − <https://nodejs.org/api/cli.html>

The Strict Mode

The fifth edition of the ECMAScript specification introduced the Strict Mode. The Strict Mode imposes a layer of constraint on JavaScript. It makes several changes to normal JavaScript semantics.

The code can be transitioned to work in the Strict Mode by including the following −

// Whole-script strict mode syntax

"use strict";

v = "Hi! I'm a strict mode script!"; // ERROR: Variable v is not declared

In the above snippet, the entire code runs as a constrained variant of JavaScript.

JavaScript also allows to restrict, the Strict Mode within a block’s scope as that of a function. This is illustrated as follows −

v = 15

function f1() {

"use strict";

var v = "Hi! I'm a strict mode script!";

}

In, the above snippet, any code outside the function will run in the non-strict mode. All statements within the function will be executed in the Strict Mode.

ES6 and Hoisting

The JavaScript engine, by default, moves declarations to the top. This feature is termed as **hoisting**. This feature applies to variables and functions. Hoisting allows JavaScript to use a component before it has been declared. However, the concept of hoisting does not apply to scripts that are run in the Strict Mode.

A **variable**, by definition, is “a named space in the memory” that stores values. In other words, it acts as a container for values in a program. Variable names are called **identifiers**. Following are the naming rules for an identifier −

* Identifiers cannot be keywords.
* Identifiers can contain alphabets and numbers.
* Identifiers cannot contain spaces and special characters, except the underscore (\_) and the dollar ($) sign.
* Variable names cannot begin with a number.

Type Syntax

A variable must be declared before it is used. ES5 syntax used the **var** keyword to achieve the same. The ES5 syntax for declaring a variable is as follows.

//Declaration using var keyword

var variable\_name

ES6 introduces the following variable declaration syntax −

* Using the let.
* Using the const.

**Variable initialization** refers to the process of storing a value in the variable. A variable may be initialized either at the time of its declaration or at a later point in time.

The traditional ES5 type syntax for declaring and initializing a variable is as follows −

//Declaration using var keyword

var variable\_name = value

Example : Using Variables

var name = "Tom"

console.log("The value in the variable is: "+name)

The above example declares a variable and prints its value.

The following output is displayed on successful execution.

The value in the variable is Tom

JavaScript and Dynamic Typing

JavaScript is an un-typed 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. This feature is termed as **dynamic typing**.

JavaScriptVariable Scope

The scope of a variable is the region of your program in which it is defined. Traditionally, JavaScript defines only two scopes-global and local.

* **Global Scope** − A variable with global scope can be accessed from within any part of the JavaScript code.
* **Local Scope** − A variable with a local scope can be accessed from within a function where it is declared.

Example : Global vs. Local Variable

The following example declares two variables by the name **num** - one outside the function (global scope) and the other within the function (local scope).

var num = 10

function test() {

var num = 100

console.log("value of num in test() "+num)

}

console.log("value of num outside test() "+num)

test()

The variable when referred to within the function displays the value of the locally scoped variable. However, the variable **num** when accessed outside the function returns the globally scoped instance.

The following output is displayed on successful execution.

value of num outside test() 10

value of num in test() 100

ES6 defines a new variable scope - The Block scope.

The Let and Block Scope

The block scope restricts a variable’s access to the block in which it is declared. The **var** keyword assigns a function scope to the variable. Unlike the var keyword, the **let** keyword allows the script to restrict access to the variable to the nearest enclosing block.

"use strict"

function test() {

var num = 100

console.log("value of num in test() "+num) {

console.log("Inner Block begins")

let num = 200

console.log("value of num : "+num)

}

}

test()

The script declares a variable **num** within the local scope of a function and re-declares it within a block using the let keyword. The value of the locally scoped variable is printed when the variable is accessed outside the inner block, while the block scoped variable is referred to within the inner block.

**Note** − The strict mode is a way to opt in to a restricted variant of JavaScript.

The following output is displayed on successful execution.

value of num in test() 100

Inner Block begins

value of num : 200

Example: let v/s var

var no = 10;

var no = 20;

console.log(no);

The following output is displayed on successful execution of the above code.

20

Let us re-write the same code using the **let** keyword.

let no = 10;

let no = 20;

console.log(no);

The above code will throw an error: Identifier 'no' has already been declared. Any variable declared using the let keyword is assigned the block scope.

let and block level safety

If we try to declare a **let** variable twice within the same block, it will throw an error. Consider the following example −

<script>

let balance = 5000 // number type

console.log(typeof balance)

let balance = {message:"hello"} // changing number to object type

console.log(typeof balance)

</script>

The above code will result in the following error −

Uncaught SyntaxError: Identifier 'balance' has already been declared

let and multiple blocks

However, the same **let** variable can be used in different block level scopes without any syntax errors.

Example

<script>

let count = 100

for (let count = 1;count <= 10;count++){

//inside for loop brackets ,count value starts from 1

console.log("count value inside loop is ",count);

}

//outside for loop brackets ,count value is 100

console.log("count value after loop is",count);

if(count == 100){

//inside if brackets ,count value is 50

let count = 50;

console.log("count inside if block",count);

}

console.log(count);

</script>

The output of the above code will be as follows −

count value inside loop is 1

count value inside loop is 2

count value inside loop is 3

count value inside loop is 4

count value inside loop is 5

count value inside loop is 6

count value inside loop is 7

count value inside loop is 8

count value inside loop is 9

count value inside loop is 10

count value after loop is 100

count inside if block 50

100

The const

The **const** declaration creates a read-only reference to a value. It does not mean the value it holds is immutable, just that the variable identifier cannot be reassigned. Constants are block-scoped, much like variables defined using the let statement. The value of a constant cannot change through re-assignment, and it can't be re-declared.

The following rules hold true for a variable declared using the **const** keyword −

* Constants cannot be reassigned a value.
* A constant cannot be re-declared.
* A constant requires an initializer. This means constants must be initialized during its declaration.
* The value assigned to a **const** variable is mutable.

Example

const x = 10

x = 12 // will result in an error!!

The above code will return an error since constants cannot be reassigned a value. Constants variable are immutable.

Constants are Immutable

Unlike variables declared using **let** keyword, **constants** are immutable. This means its value cannot be changed. For example, if we try to change value of the constant variable, an error will be displayed.

<script>

let income = 100000

const INTEREST\_RATE = 0.08

income += 50000 // mutable

console.log("changed income value is ",income)

INTEREST\_RATE += 0.01

console.log("changed rate is ",INTEREST\_RATE) //Error: not mutable

</script>

The output of the above code will be as follows −

changed income value is 150000

Uncaught TypeError: Assignment to constant variable

const and arrays

The following example shows how to create an immutable array. New elements can be added to the array. However, reinitializing the array will result in an error as shown below −

<script>

const DEPT\_NOS = [10,20,30,50]

DEPT\_NOS.push(40)

console.log('dept numbers is ',DEPT\_NOS)

const EMP\_IDS = [1001,1002,1003]

console.log('employee ids',EMP\_IDS)

//re assigning variable employee ids

EMP\_IDS = [2001,2002,2003]

console.log('employee ids after changing',EMP\_IDS)

</script>

The output of the above code will be as shown below −

dept numbers is (5) [10, 20, 30, 50, 40]

employee ids (3) [1001, 1002, 1003]

Uncaught TypeError: Assignment to constant variable.

The var keyword

Prior to ES6, the **var** keyword was used to declare a variable in JavaScript. Variables declared using **var** do not support block level scope. This means if a variable is declared in a loop or **if block** it can be accessed outside the loop or the **if block**. This is because the variables declared using the **var** keyword support hoisting.

var and hoisting

**Variable hoisting** allows the use of a variable in a JavaScript program, even before it is declared. Such variables will be initialized to **undefined** by default. JavaScript runtime will scan for variable declarations and put them to the top of the function or script. Variables declared with **var** keyword get hoisted to the top. Consider the following example −

<script>

variable company is hoisted to top , var company = undefined

console.log(company); // using variable before declaring

var company = "TutorialsPoint"; // declare and initialized here

console.log(company);

</script>

The output of the above code will be as shown below −

undefined

TutorialsPoint

var and block scope

The **block scope** restricts a variable’s access to the block in which it is declared. The **var** keyword assigns a function scope to the variable. Variables declared using the **var** keyword do not have a block scope. Consider the following example −

<script>

//hoisted to top ; var i = undefined

for (var i = 1;i <= 5;i++){

console.log(i);

}

console.log("after the loop i value is "+i);

</script>

The output of the above code will be as follows −

1

2

3

4

5

after the loop i value is 6

The variable **i** is declared inside the for loop using the **var** keyword. The variable i is accessible outside the loop. However, at times, there might be a need to restrict a variable's access within a block. We cannot use the **var** keyword in this scenario. ES6 introduces the **let** keyword to overcome this limitation.

var and block level safety

If we declare the same **variable** twice using the **var keyword** within a block, the compiler will not throw an error. However, this may lead to unexpected logical errors at runtime.

<script>

var balance = 5000

console.log(typeof balance)

var balance = {message:"hello"}

console.log(typeof balance)

</script>

The output of the above code is as shown below −

number

object

An **expression** is a special kind of statement that evaluates to a value. Every expression is composed of −

* **Operands** − Represents the data.
* **Operator** − Defines how the operands will be processed to produce a value.

Consider the following expression- 2 + 3. Here in the expression, 2 and 3 are operands and the symbol + (plus) is the operator. JavaScript supports the following types of operators −

* Arithmetic operators
* Logical operators
* Relational operators
* Bitwise operators
* Assignment operators
* Ternary/conditional operators
* String operators
* Type operators
* The void operator

Arithmetic Operators

Assume the values in variables **a** and **b** are 10 and 5 respectively.

[Show Examples](https://www.tutorialspoint.com/es6/es6_arithmetic_operators_examples.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Function** | **Example** |
| + | **Addition**  Returns the sum of the operands. | a + b is 15 |
| - | **Subtraction**  Returns the difference of the values. | a-b is 5 |
| \* | **Multiplication**  Returns the product of the values. | a\*b is 50 |
| / | **Division**  Performs a division operation and returns the quotient. | a/b is 2 |
| % | **Modulus**  Performs a division and returns the remainder. | a%b is 0 |
| ++ | **Increment**  Increments the value of the variable by one. | a++ is 11 |
| -- | **Decrement**  Decrements the value of the variable by one. | a-- is 9 |

Relational Operators

Relational operators test or define the kind of relationship between two entities. Relational operators return a boolean value, i.e. true/false.

Assume the value of A is 10 and B is 20.

[Show Examples](https://www.tutorialspoint.com/es6/es6_relational_operators_examples.htm)

|  |  |  |
| --- | --- | --- |
| **Operators** | **Description** | **Example** |
| > | Greater than | (A > B) is False |
| < | Lesser than | (A < B) is True |
| >= | Greater than or equal to | (A >= B) is False |
| <= | Lesser than or equal to | (A <= B) is True |
| == | Equality | (A == B) is False |
| != | Not Equal | (A!= B) is True |

Logical Operators

Logical operators are used to combine two or more conditions. Logical operators, too, return a Boolean value. Assume the value of variable A is 10 and B is 20.

[Show Examples](https://www.tutorialspoint.com/es6/es6_logical_operators_examples.htm).

|  |  |  |
| --- | --- | --- |
| **Operators** | **Description** | **Example** |
| && | **And**  The operator returns true only if all the expressions specified return true. | (A > 10 && B > 10) is False |
| || | **Or**  The operator returns true if at least one of the expressions specified return true. | (A > 10 || B > 10) is True |
| ! | **Not**  The operator returns the inverse of the expression’s result. For E.g.: !(7>5) returns false. | !(A > 10) is True |

Bitwise Operators

JavaScript supports the following bitwise operators. The following table summarizes JavaScript's bitwise operators.

[Show Examples](https://www.tutorialspoint.com/es6/es6_bitwise_operators_examples.htm).

|  |  |  |
| --- | --- | --- |
| **Operators** | **Usage** | **Description** |
| Bitwise AND | a & b | Returns a one in each bit position for which the corresponding bits of both operands are ones |
| Bitwise OR | a | b | Returns a one in each bit position for which the corresponding bits of either or both operands are ones |
| Bitwise XOR | a^b | Returns a one in each bit position for which the corresponding bits of either but not both operands are ones |
| Bitwise NOT | ~ a | Inverts the bits of its operand |
| Left shift | a << b | Shifts a in binary representation b (< 32) bits to the left, shifting in zeroes from the right |
| Sign-propagating right shift | a >> b | Shifts a in binary representation b (< 32) bits to the right, discarding bits shifted off |
| Zero-fill right shift | a >>> b | Shifts a in binary representation b (< 32) bits to the right, discarding bits shifted off, and shifting in zeroes from the left |

Assignment Operators

The following table summarizes Assignment operators.

[Show Examples](https://www.tutorialspoint.com/es6/es6_assignment_operators_examples.htm).

|  |  |
| --- | --- |
| **Sr.No** | **Operator & Description** |
| 1 | **= (Simple Assignment)**  Assigns values from the right side operand to the left side operand.  **Example** − 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.  **Example** − 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.  **Example** 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.  **Example** 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. |

**Note** − The same logic applies to Bitwise operators, so they will become <<=, >>=, >>=, &=, |= and ^=.

Miscellaneous Operators

Following are some of the miscellaneous operators.

The negation operator (-)

Changes the sign of a value. The following program is an example of the same.

var x = 4

var y = -x;

console.log("value of x: ",x); //outputs 4

console.log("value of y: ",y); //outputs -4

The following output is displayed on successful execution of the above program.

value of x: 4

value of y: -4

String Operators : Concatenation operator (+)

The + operator when applied to strings appends the second string to the first. The following program helps to understand this concept.

var msg = "hello"+"world"

console.log(msg)

The following output is displayed on successful execution of the above program.

helloworld

The concatenation operation doesn’t add a space between the strings. Multiple strings can be concatenated in a single statement.

Conditional Operator (?)

This operator is used to represent a conditional expression. The conditional operator is also sometimes referred to as the ternary operator. Following is the syntax.

Test ? expr1 : expr2

Where,

**Test** − Refers to the conditional expression

**expr1** − Value returned if the condition is true

**expr2** − Value returned if the condition is false

**Example**

var num = -2

var result = num > 0 ?"positive":"non-positive"

console.log(result)

Line 2 checks whether the value in the variable num is greater than zero. If num is set to a value greater than zero, it returns the string “positive” else a “non-positive” string is returned.

The following output is displayed on successful execution of the above program.

non-positive

typeof operator

It is a unary operator. This operator returns the data type of the operand. The following table lists the data types and the values returned by the **typeof** operator in JavaScript.

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

The following example code displays the number as the output.

var num = 12

console.log(typeof num); //output: number

The following output is displayed on successful execution of the above code.

number

Spread Operator

**ES6** provides a new operator called the **spread operator**. The spread operator is represented by three dots “...” . The spread operator converts an array into individual array elements.

Spread operator and function

The following example illustrates the use of spread operators in a function

<script>

function addThreeNumbers(a,b,c){

return a+b+c;

}

const arr = [10,20,30]

console.log('sum is :',addThreeNumbers(...arr))

console.log('sum is ',addThreeNumbers(...[1,2,3]))

</script>

The output of the above code will be as seen below −

sum is : 60

sum is 6

Spread operator and Array copy and concat

The spread operator can be used to copy one array into another. It can also be used to concatenate two or more arrays. This is shown in the example below −

Example

<script>

//copy array using spread operator

let source\_arr = [10,20,30]

let dest\_arr = [...source\_arr]

console.log(dest\_arr)

//concatenate two arrays

let arr1 = [10,20,30]

let arr2 =[40,50,60]

let arr3 = [...arr1,...arr2]

console.log(arr3)

</script>

The output of the above code will be as stated below −

[10, 20, 30]

[10, 20, 30, 40, 50, 60]

Spread Operator and Object copy and concatenation

The spread operator can be used to copy one object into another. It can also be used to concatenate two or more objects. This is shown in the example below −

<script>

//copy object

let student1 ={firstName:'Mohtashim',company:'TutorialsPoint'}

let student2 ={...student1}

console.log(student2)

//concatenate objects

let student3 = {lastName:'Mohammad'}

let student4 = {...student1,...student3}

console.log(student4)

</script>

The output of the above code will be as given below −

{firstName: "Mohtashim", company: "TutorialsPoint"}

{firstName: "Mohtashim", company: "TutorialsPoint", lastName: "Mohammad"}

onmouseover and onmouseout

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

Example

<html>

<head>

<script type = "text/javascript">

function over() {

document.write ("Mouse Over");

}

function out() {

document.write ("Mouse Out");

}

</script>

</head>

<body>

<p>Bring your mouse inside the division to see the result:</p>

<div onmouseover = "over()" onmouseout = "out()">

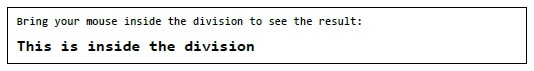
<h2> This is inside the division </h2>

</div>

</body>

</html>

The following output is displayed on successful execution of the above code.



HTML 5 Standard Events

The standard HTML 5 events are listed in the following table for your reference. The script indicates a JavaScript function to be executed against that event.

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Value** | **Description** |
| offline | script | Triggers when the document goes offline |
| onabort | script | Triggers on an abort event |
| onafterprint | script | Triggers after the document is printed |
| onbeforeonload | script | Triggers before the document load |
| onbeforeprint | script | Triggers before the document is printed |
| onblur | script | Triggers when the window loses focus |
| oncanplay | script | Triggers when the media can start play, but might have to stop for buffering |
| oncanplaythrough | script | Triggers when the media can be played to the end, without stopping for buffering |
| onchange | script | Triggers when an element changes |
| onclick | script | Triggers on a mouse click |
| oncontextmenu | script | Triggers when a context menu is triggered |
| ondblclick | script | Triggers on a mouse double-click |
| ondrag | script | Triggers when an element is dragged |
| ondragend | script | Triggers at the end of a drag operation |
| ondragenter | script | Triggers when an element has been dragged to a valid drop target |
| ondragleave | script | Triggers when an element leaves a valid drop target |
| ondragover | script | Triggers when an element is being dragged over a valid drop target |
| ondragstart | script | Triggers at the start of a drag operation |
| ondrop | script | Triggers when the dragged element is being dropped |
| ondurationchange | script | Triggers when the length of the media is changed |
| onemptied | script | Triggers when a media resource element suddenly becomes empty |
| onended | script | Triggers when the media has reached the end |
| onerror | script | Triggers when an error occurs |
| onfocus | script | Triggers when the window gets focus |
| onformchange | script | Triggers when a form changes |
| onforminput | script | Triggers when a form gets user input |
| onhaschange | script | Triggers when the document has changed |
| oninput | script | Triggers when an element gets user input |
| oninvalid | script | Triggers when an element is invalid |
| onkeydown | script | Triggers when a key is pressed |
| onkeypress | script | Triggers when a key is pressed and released |
| onkeyup | script | Triggers when a key is released |
| onload | script | Triggers when the document loads |
| onloadeddata | script | Triggers when media data is loaded |
| onloadedmetadata | script | Triggers when the duration and other media data of a media element is loaded |
| onloadstart | script | Triggers when the browser starts to load the media data |
| onmessage | script | Triggers when the message is triggered |
| onmousedown | script | Triggers when a mouse button is pressed |
| onmousemove | script | Triggers when the mouse pointer moves |
| onmouseout | script | Triggers when the mouse pointer moves out of an element |
| onmouseover | script | Triggers when the mouse pointer moves over an element |
| onmouseup | script | Triggers when a mouse button is released |
| onmousewheel | script | Triggers when the mouse wheel is being rotated |
| onoffline | script | Triggers when the document goes offline |
| ononline | script | Triggers when the document comes online |
| onpagehide | script | Triggers when the window is hidden |
| onpageshow | script | Triggers when the window becomes visible |
| onpause | script | Triggers when the media data is paused |
| onplay | script | Triggers when the media data is going to start playing |
| onplaying | script | Triggers when the media data has start playing |
| onpopstate | script | Triggers when the window's history changes |
| onprogress | script | Triggers when the browser is fetching the media data |
| onratechange | script | Triggers when the media data's playing rate has changed |
| onreadystatechange | script | Triggers when the ready-state changes |
| onredo | script | Triggers when the document performs a redo |
| onresize | script | Triggers when the window is resized |
| onscroll | script | Triggers when an element's scrollbar is being scrolled |
| onseeked | script | Triggers when a media element's seeking attribute is no longer true, and the seeking has ended |
| onseeking | script | Triggers when a media element's seeking attribute is true, and the seeking has begun |
| onselect | script | Triggers when an element is selected |
| onstalled | script | Triggers when there is an error in fetching media data |
| onstorage | script | Triggers when a document loads |
| onsubmit | script | Triggers when a form is submitted |
| onsuspend | script | Triggers when the browser has been fetching media data, but stopped before the entire media file was fetched |
| ontimeupdate | script | Triggers when the media changes its playing position |
| onundo | script | Triggers when a document performs an undo |
| onunload | script | Triggers when the user leaves the document |
| onvolumechange | script | Triggers when the media changes the volume, also when the volume is set to "mute" |
| onwaiting | script | Triggers when the media has stopped playing, but is expected to resume |

**Redirect** is a way to send both users and search engines to a different URL from the one they originally requested. Page redirection is a way to automatically redirect a web page to another web page. The redirected page is often on the same website, or it can be on a different website or a web server.

## JavaScript Page Redirection

**window.location and window.location.href**

In JavaScript, you can use many methods to redirect a web page to another one. Almost all methods are related to **window.location** object, which is a property of the Window object. It can be used to get the current URL address (web address) and to redirect the browser to a new page. Both usages are same in terms of behavior. **window.location** returns an object. If **.href** is not set, **window.location** defaults to change the parameter **.href**.

### Example

<!DOCTYPE html>

<html>

<head>

<script>

function newLocation() {

window.location = "http://www.xyz.com";

}

</script>

</head>

<body>

<input type = "button" value = "Go to new location" onclick = "newLocation()">

</body>

</html>

**location.replace()**

The other most frequently used method is the **replace()** method of window.location object, it will replace the current document with a new one. In replace() method, you can pass a new URL to replace() method and it will perform an HTTP redirect.

Following is the syntax for the same.

window.location.replace("http://www.abc.com

**location.assign()**

The location.assign() method loads a new document in the browser window.

Following is the syntax for the same.

window.location.assign("http://www.abc.org");

**assign() vs. replace()**

The difference between assign() and replace() method is that the location.replace() method deletes the current URL from the document history, so it is unable to navigate back to the original document. You can't use the browsers "Back" button in this case. If you want to avoid this situation, you should use location.assign() method, because it loads a new Document in the browser.

**location.reload()**

The location.reload() method reloads the current document in the browser window.

Following is the syntax for the same.

window.location.reload("http://www.yahoo.com");

**window.navigate()**

The window.navigate() method is similar to assigning a new value to the window.location.href property. Because it is only available in MS Internet Explorer, so you should avoid using this in cross-browser development.

Following is the syntax for the same.

window.navigate("http://www.abc.com");

## Redirection and Search Engine Optimization

If you want to notify the search engines (SEO) about your URL forwarding, you should add the rel = "canonical" meta tag to your website head part because search engines don't analyze JavaScript to check the redirection.

Following is the syntax for the same.

<link rel = "canonical" href = "http://abc.com/" />

Dialogboxes

JavaScript supports three important types of dialog boxes. These dialog boxes can be used to raise and alert, or to get confirmation on any input or to have a kind of input from the users. Here we will discuss each dialog box one by one.

Alert Dialog Box

An alert dialog box is mostly used to send a warning message to the users. For example, if one input field requires to enter some text but the user does not provide any input, then as a part of validation, you can use an alert box to send a warning message.

Nonetheless, an alert box can still be used for friendlier messages. Alert box provides only one button "OK" to select and proceed.

Example

<html>

<head>

<script type = "text/javascript">

function Warn() {

alert ("This is a warning message!");

document.write ("This is a warning message!");

}

</script>

</head>

<body>

<p>Click the following button to see the result: </p>

<form>

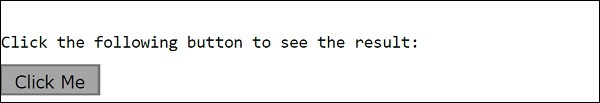
<input type = "button" value = "Click Me" onclick = "Warn();" />

</form>

</body>

</html>

The following output is displayed on successful execution of the above code.



Confirmation Dialog Box

A confirmation dialog box is mostly used to take the user's consent on any option. It displays a dialog box with two buttons: OK and Cancel.

If the user clicks on the OK button, the window method **confirm()** will return true. If the user clicks on the Cancel button, then confirm() returns false. You can use a confirmation dialog box as follows.

Example

<html>

<head>

<script type = "text/javascript">

function getConfirmation(){

var retVal = confirm("Do you want to continue ?");

if( retVal == true ){

document.write ("User wants to continue!");

return true;

} else {

Document.write ("User does not want to continue!");

return false;

}

}

</script>

</head>

<body>

<p>Click the following button to see the result: </p>

<form>

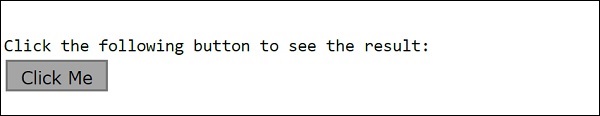
<input type = "button" value = "Click Me" onclick = "getConfirmation();" />

</form>

</body>

</html>

The following output is displayed on successful execution of the above code.



Prompt Dialog Box

The prompt dialog box is very useful when you want to pop-up a text box to get a user input. Thus, it enables you to interact with the user. The user needs to fill in the field and then click OK.

This dialog box is displayed using a method called **prompt()** which takes two parameters: (i) a label which you want to display in the text box and (ii) a default string to display in the text box.

This dialog box has two buttons: OK and Cancel. If the user clicks the OK button, the window method prompt() will return the entered value from the text box. If the user clicks the Cancel button, the window method prompt() returns null.

Example

<html>

<head>

<script type = "text/javascript">

function getValue(){

var retVal = prompt("Enter your name : ", "your name here");

document.write("You have entered : " + retVal);

}

</script>

</head>

<body>

<p>Click the following button to see the result: </p>

<form>

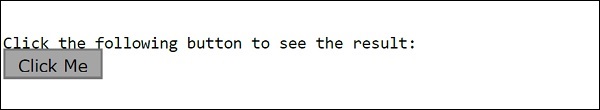
<input type = "button" value = "Click Me" onclick = "getValue();" />

</form>

</body>

</html>

The following output is displayed on successful execution of the above code.



**Objects:**

var person = {

firstname:"Tom",

lastname:"Hanks",

func:function(){return "Hello!!"},

};

//access the object values

console.log(person.firstname)

console.log(person.lastname)

console.log(person.func())

function Car() {

this.make = "Ford"

}

var obj = new Car()

obj.model = "F123"

console.log(obj.make)

console.log(obj.model)

## The Object.create Method

Objects can also be created using the **Object.create()** method. It allows you to create the prototype for the object you want, without having to define a constructor function.

### Example

var roles = {

type: "Admin", // Default value of properties

displayType : function() {

// Method which will display type of role

console.log(this.type);

}

}

// Create new role type called super\_role

var super\_role = Object.create(roles);

super\_role.displayType(); // Output:Admin

// Create new role type called Guest

var guest\_role = Object.create(roles);

guest\_role.type = "Guest";

guest\_role.displayType(); // Output:Guest

## Comparing Objects

In JavaScript, objects are a reference type. Two distinct objects are never equal, even if they have the same properties. This is because, they point to a completely different memory address. Only those objects that share a common reference yields true on comparison.

**Example 1 − Different Object References**

var val1 = {name: "Tom"};

var val2 = {name: "Tom"};

console.log(val1 == val2) // return false

console.log(val1 === val2) // return false

In the above example, **val1** and **val2** are two distinct objects that refer to two different memory addresses. Hence on comparison for equality, the operator will return false.

**Example 2 − Single Object Reference**

var val1 = {name: "Tom"};

var val2 = val1

console.log(val1 == val2) // return true

console.log(val1 === val2) // return true

Number:

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.

Following is the syntax for creating a number object.

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

|  |  |
| --- | --- |
| **Sr.No** | **Property & Description** |
| 1 | [Number.EPSILON](https://www.tutorialspoint.com/es6/es6_number_epsilon.htm)  The smallest interval between two representable numbers. |
| 2 | [Number.MAX\_SAFE\_INTEGER](https://www.tutorialspoint.com/es6/es6_number_max_safe_integer.htm)  The maximum safe integer in JavaScript (2^53 - 1). |
| 3 | [Number.MAX\_VALUE](https://www.tutorialspoint.com/es6/es6_number_max_value.htm)  The largest positive representable number. |
| 4 | [MIN\_SAFE\_INTEGER](https://www.tutorialspoint.com/es6/es6_number_min_safe_integer.htm)  The minimum safe integer in JavaScript (-(2^53 - 1)). |
| 5 | [Number.MIN\_VALUE](https://www.tutorialspoint.com/es6/es6_number_min_value.htm)  The smallest positive representable number - that is, the positive number closest to zero (without actually being zero) |
| 6 | [Number.Nan](https://www.tutorialspoint.com/es6/es6_number_nan.htm)  Special "not a number" value |
| 7 | [Number.NEGATIVE\_INFINITY](https://www.tutorialspoint.com/es6/es6_number_negative_infinity.htm)  Special value representing negative infinity; returned on overflow |
| 8 | [Number.POSITIVE\_INFINITY](https://www.tutorialspoint.com/es6/es6_number_positive_infinity.htm)  Special value representing infinity; returned on overflow |
| 9 | **Number.prototype**  Special value representing infinity; returned on overflow |

Number Methods

|  |  |
| --- | --- |
| **Sr.No** | **Method & Description** |
| 1 | [Number.isNaN()](https://www.tutorialspoint.com/es6/es6_numberisnan.htm)  Determines whether the passed value is NaN. |
| 2 | [Number.isFinite()](https://www.tutorialspoint.com/es6/es6_numberisfinite.htm)  Determines whether the passed value is a finite number. |
| 3 | [Number.isInteger()](https://www.tutorialspoint.com/es6/es6_numberisinteger.htm)  Determines whether the passed value is an integer. |
| 4 | [Number.isSafeInteger()](https://www.tutorialspoint.com/es6/es6_numberissafeinteger.htm)  Determines whether the passed value is a safe integer (number between -(253 - 1) and 253- 1) |
| 5 | [Number.parseFloat()](https://www.tutorialspoint.com/es6/es6_numberparsefloat.htm)  The value is the same as parseFloat() of the global object |
| 6 | [Number.parseInt()](https://www.tutorialspoint.com/es6/es6_numberparseint.htm)  The value is the same as parseInt() of the global object |

The String object lets you work with a series of characters; it wraps JavaScript’s string primitive data type with a number of helper methods.

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

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.

String Properties

Following is a list of the properties of String object and its description.

|  |  |
| --- | --- |
| **Sr.No** | **Property & Description** |
| 1 | [constructor](https://www.tutorialspoint.com/es6/es6_string_constructor.htm)  Returns a reference to the String function that created the object . |
| 2 | [length](https://www.tutorialspoint.com/es6/es6_string_length.htm)  Returns the length of the string. |
| 3 | [Prototype](https://www.tutorialspoint.com/es6/es6_string_prototype.htm)  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.

|  |  |
| --- | --- |
| **Sr.No** | **Method & Description** |
| 1 | [charAt()](https://www.tutorialspoint.com/es6/es6_string_charat.htm)  Returns the character at the specified index. |
| 2 | [charCodeAt()](https://www.tutorialspoint.com/es6/es6_string_charcodeat.htm)  Returns a number indicating the Unicode value of the character at the given index. |
| 3 | [concat()](https://www.tutorialspoint.com/es6/es6_string_concat.htm)  Combines the text of two strings and returns a new string. |
| 4 | [indexOf()](https://www.tutorialspoint.com/es6/es6_string_indexof.htm)  Returns the index within the calling String object of the first occurrence of the specified value, or -1 if not found. |
| 5 | [lastIndexOf()](https://www.tutorialspoint.com/es6/es6_string_method_lastindexof.htm)  Returns the index within the calling String object of the last occurrence of the specified value, or -1 if not found. |
| 6 | [localeCompare()](https://www.tutorialspoint.com/es6/es6_string_localecompare.htm)  Returns a number indicating whether a reference string comes before or after or is the same as the given string in a sorted order. |
| 7 | **match()**  Used to match a regular expression against a string. |
| 8 | [replace()](https://www.tutorialspoint.com/es6/es6_string_replace.htm)  Used to find a match between a regular expression and a string, and to replace the matched substring with a new substring. |
| 9 | [search()](https://www.tutorialspoint.com/es6/es6_string_search.htm)  Executes the search for a match between a regular expression and a specified string. |
| 10 | [slice()](https://www.tutorialspoint.com/es6/es6_string_slice.htm)  Extracts a section of a string and returns a new string. |
| 11 | [split()](https://www.tutorialspoint.com/es6/es6_string_split.htm)  Splits a String object into an array of strings by separating the string into substrings. |
| 12 | [substr()](https://www.tutorialspoint.com/es6/es6_string_substr.htm)  Returns the characters in a string beginning at the specified location through the specified number of characters. |
| 13 | [substring()](https://www.tutorialspoint.com/es6/es6_string_substring.htm)  Returns the characters in a string between two indexes into the string. |
| 14 | [toLocaleLowerCase()](https://www.tutorialspoint.com/es6/es6_string_tolocalelowercase.htm)  The characters within a string are converted to lower case while respecting the current locale. |
| 15 | **toLocaleupperCase()**  The characters within a string are converted to uppercase while respecting the current locale. |
| 16 | [toLowerCase()](https://www.tutorialspoint.com/es6/es6_string_tolowercase.htm)  Returns the calling string value converted to lowercase. |
| 17 | [toString()](https://www.tutorialspoint.com/es6/es6_methods_tostring.htm)  Returns a string representing the specified object. |
| 18 | [toUpperCase()](https://www.tutorialspoint.com/es6/es6_string_touppercase.htm)  Returns the calling string value converted to uppercase. |
| 19 | [valueOf()](https://www.tutorialspoint.com/es6/es6_string_valueof.htm)  Returns the primitive value of the specified object. |