**¿Qué es JavaScript?**

JavaScript es un lenguaje de programación dinámico que tiene la capacidad de ser utilizado en muchos dispositivos diferentes. Puede usarse en ordenadores personales, servidores web y teléfonos inteligentes. Es un lenguaje interpretado, orientado a objetos, débilmente tipado y dinámico.

JavaScript se emplea comúnmente para el desarrollo web front-end y más recientemente para algunos desarrollos back-end a través de frameworks como Node.Js. o Next.Js. Tiene características como la programación orientada a objetos, funciones y herencia basada en prototipos.

**¿Cómo nace Javascript?**

Nace con la necesidad de generar dinamismo en las páginas web y que a su vez los usuarios y las empresas pudieran interactuar unos con otros. Fue creado por Brendan Eich en 1995 y se convirtió en un estándar oficial del World Wide Web Consortium (W3C) en 1997.

**¿Por qué decimos que Javascript es un lenguaje dinámico?**

Corre directamente en la etapa de Runtime, sin una etapa de compilación previa. Esto permite probar nuestro código inmediatamente; pero también es lo que hace que los errores no se muestren sino hasta que se ejecuta el programa. Lo que se ve a primera vista, cuando se analiza el código, es muy probable que no sea lo que se va a obtener cuando el programa se ejecute.

JavaScript permite declarar (por ejemplo) variables cuyo valor (y tipo) solo se conocerá al momento de su ejecución en función de las condiciones dadas al momento de correrlo en un entorno real. En cambio, los lenguajes estáticos no compilarán en código ejecutable a menos que todos los valores (o tipos de valores) se conozcan por adelantado.

**¿Por qué es débilmente tipado?**

Porque los tipos de datos no están bien definidos en el lenguaje y permite, por ejemplo, operaciones entre numerosos y letras. Esto sucede porque el lenguaje asume tipos de datos que no necesariamente fueron los que se querían representar. Se pueden hacer operaciones entre tipos distintos de datos (enteros con strings, booleanos con enteros, etc.). Ejemplo:

4 + "7"; // 47

4 \* "7"; // 28

2 + **true**; // 3

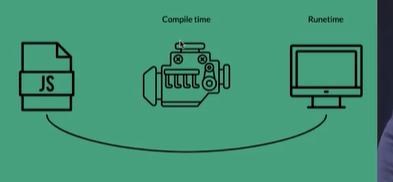
**false** - 3; // -3

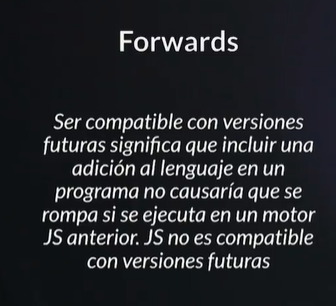
**¿Realmente es Javascript un lenguaje interpretado?**

Sí, y la razón es que el navegador lee línea por línea nuestro código, el cual le indica lo que tiene que ir haciendo, sin la necesidad de compilar. Todo esto es controlado por el motor de Javascript V8 del navegador

**Qué significa que Javascript es Backwards Compatible**

Todas las funciones nuevas que salen para Javascript no dañarán el trabajo ya hecho previamente, pero no se podrá utilizar en nuestro entorno de trabajo inmediatamente. Para solucionar esto está [Babel](https://platzi.com/blog/que-es-babel/), que permite usar las nuevas características del lenguaje, pero lo transforma a una versión que el navegador pueda entender.

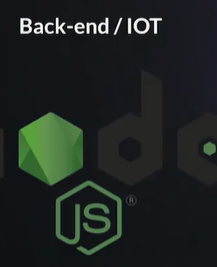






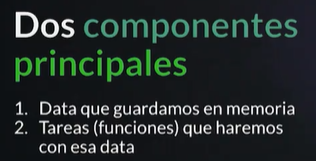


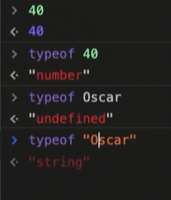




**APUNTES**  
1.- JavaScript tiene una **comunidad enorme** de desarrolladores que te pueden ir ayudando a generar diferentes cosas.

1. Si solo estuvieras interesado en trabajar **aplicaciones web** tienes muchos frameworks y librerías construidas en JavaScript que te van a ayudar a hacer proyectos de forma mucho mas rápida, eficiente y robusta (Angular, View, React,entre otros)
2. Si no quieres trabajar solo en aplicaciones Web puedes utilizar JavaScript con un framework que se llama React Native para poder **construir aplicaciones nativas** como Android y IOS.
3. Puedes construir **aplicaciones de escritorio** con JavaScript, usando un framework llamado Electron, pueden correr en Mac o Windows.
4. También puedes trabajar en la parte del **Back-end** o \*\*IOT \*\*(Internet Od Things) es un concepto que se refiere a una interconexion digital de objetos cotidianos con Internet. Esto con un Framework llamado NodeJS, el cual es un entorno de ejecución de JavaScript que corre directamente en el Back-end.





**VARIALBLES**

entro de JavaScript tenemos tres formas de declarar una variable las cuales son: **var**, **const** y **let.**

1. **Var:** Era la forma en que se declaraban las variables hasta ECMAScript 5. Casi ya no se usa porque es de forma global y tiene las siguientes características:

**o Se puede reinicializar:** osea todas las variables se inicializan, por ejemplo:  
Var pokemonType = ‘electric’ entonces reinicializar es:  
Var pokemonType = ‘grass’ osea la misma variable con diferentes datos el último dato predomina.  
**o Se puede reasignar:** osea la variable ya inicializada le reasignamos otro valor por ejemplo: inicializamos la variable: Var pokemonType = ‘electric’ ahora la reasignamos pokemonType = ‘grass’ ya no va var  
**o Su alcance es función global:** osea inicializamos la variable, pero la podemos llamar desde cualquier bloque (una llave abierta y una cerrada {}) pero hay que tener mucho cuidado con ello ya que puede haber peligro, no es recomendable usar VAR.

**const y let es la forma en que se declaran las variables a partir de ECMAScript 6,**

1. **const:** sirve para declarar variables que nunca van a ser modificadas:  
   **o No se puede reinicilizar:** es una const única no puede haber otra inicializada con el mismo nombre. const pokemonType = ‘electric’ no puede haber:  
   const pokemonType = ‘grass’  
   o **No se pude re asignar:** una vez que la hayamos inicializado no la podemos reasignar solo con su nombre: const pokemonType = ‘electric’ no puede ejecutarse:  
   pokemonType = ‘grass’  
   o **No es inmutable:** osea no puede cambiar con objetos:
2. **Let:** Son variables que pueden ser modificadas, se pueden cambiar:  
   **o No se puede reinicilizar:** es una const única no puede haber otra inicializada con el mismo nombre. let pokemonType = ‘electric’ no puede haber:  
   let pokemonType = ‘grass’  
   o **Se puede reasignar:** Osea la variable ya inicializada le reasignamos otro valor por ejemplo: inicializamos la variable: let pokemonType = ‘electric’ ahora la reasignamos pokemonType = ‘grass’  
   o **Su contexto de es bloque:** Solo funciona dentro de un bloque {}, fuera de ello no.

Funciones

Las funciones son las tareas que va a llevar a cabo el navegador. Existen 2 tipos de funciones  
1) Declarativas  
2) De expresión  
Ambas pueden llevar parámetros, que son los datos que necesitan para ejecutarse.  
Cada parámetro va separado por una coma.  
Cada instrucción que tenga la función debe terminar con ; .  
Si queremos que una función nos dé un numero o dato tenemos que usar la siguiente sintaxis:

*return El dato que queremos que nos dé;*

Las funciones declarativas tienen la siguiente sintaxis:

*function Nombre de la función (Parámetros de la función) {Instrucciones}*

Un ejemplo de una función puede ser una suma:

\_  
function suma (a,b) {return a+b;}\_

Las funciones de expresión son aquellas que guardamos en una variable, por lo tanto, no es necesario nombrarlas y tienen la siguiente sintaxis:

\_var Nombre de la variable = function(Parametros){Instrucciones}.  
\_

Un ejemplo de una función de expresión sería:

*var suma = function(a,b){return a+b;}*

Para ejecutar las funciones debemos usar la siguiente sintaxis:

\_Nombre de la funcion(Parametros función); \_

Si la función no tiene ningún parámetro, únicamente se escribe:

\_Nombre de la función(); \_

# ¿Qué es una función declarativa y una expresiva?

Cuando hablamos de funciones en JavaScript, tenemos dos tipos de funciones: Funciones Declarativas (function declaration / function statement) y Expresiones de función (function expression / funciones anónimas).

### Funciones Declarativas:

En las funciones declarativas, utilizamos la palabra reservada function al inicio para poder declarar la función:

**function** **saludar**(nombre) {

console.log(`Hola ${nombre}`);

}

saludar('Diego');

#### Expresión de función:

En la expresión de función, la declaración se inicia con la palabra reservada var, donde se generará una variable que guardará una función anónima.

**var** nombre = **function**(nombre){

console.log(`Hola ${nombre}`)

}

nombre(‘Diego’);

En la expresión de función, la función podría o no llevar nombre, aunque es más común que se hagan anónimas.

### Diferencias:

A las funciones declarativas se les aplica hoisting, y a la expresión de función, no. Ya que el hoisting solo se aplica en las palabras reservadas var y function.

Lo que quiere decir que con las funciones declarativas, podemos mandar llamar la función antes de que ésta sea declarada, y con la expresión de función, no, tendríamos que declararla primero, y después mandarla llamar.

## In this article

* [**Aprenda más**](https://developer.mozilla.org/es/docs/Glossary/Hoisting#aprenda_m%C3%A1s)

# Hoisting

Hoisting es un término que no encontrará utilizado en ninguna especificación previa a [ECMAScript® 2015 Language Specification](https://www.ecma-international.org/ecma-262/6.0/index.html). El concepto de Hoisting fue pensado como una manera general de referirse a cómo funcionan los contextos de ejecución en JavaScript (específicamente las fases de creación y ejecución). Sin embargo, el concepto puede ser un poco confuso al principio.

Conceptualmente, por ejemplo, una estricta definición de hoisting sugiere que las declaraciones de variables y funciones son físicamente movidas al comienzo del código, pero esto no es lo que ocurre en realidad. Lo que sucede es que las declaraciones de variables y funciones son **asignadas en memoria** durante la fase de compilación, pero quedan exactamente en dónde las has escrito en el código.

## [Aprenda más](https://developer.mozilla.org/es/docs/Glossary/Hoisting#aprenda_m%C3%A1s)

### [Ejemplo técnico](https://developer.mozilla.org/es/docs/Glossary/Hoisting#ejemplo_t%C3%A9cnico)

Una de las ventajas en JavaScript de colocar (**ponerlas en memoria**) las declaraciones de funciones antes de ejecutar cualquier otro segmento de código es que permite utilizar una función antes de declararla en el código. Por ejemplo:

function nombreDelGato(nombre) {

console.log("El nombre de mi gato es " + nombre);

}

nombreDelGato("Maurizzio");

/\*

El resultado del código es: "El nombre de mi gato es Maurizzio"

\*/

El código escrito arriba está escrito de la manera que sería esperada para que funcione. Ahora, veamos qué sucede cuando llamamos a la función antes de escribirla:

nombreDelGato("Dumas");

function nombreDelGato(nombre) {

console.log("El nombre de mi gato es " + nombre);

}

/\*

El resultado del código es: "El nombre de mi gato es Dumas"

\*/

Como se puede observar, aunque primero llamamos a la función en el código, antes de que sea escrita, el código aún funciona. Esto sucede por la manera en la que el contexto de ejecución trabaja en JavaScript.

Hoisting se lleva también bien con otros tipos de datos y variables. Observemos el siguiente ejemplo:

### [Ejemplo técnico](https://developer.mozilla.org/es/docs/Glossary/Hoisting#ejemplo_t%C3%A9cnico_2)

var x = 5;

(function () {

console.log("x:", x); // no obtenemos '5' sino 'undefined'

var x = 10;

console.log("x:", x); // 10

}());

¿No hemos obtenido lo esperado?, como la declaración de variables se procesa antes de ejecutar cualquier código, declarar una variable en cualquier parte del código es igual a declararla al inicio del mismo. Lo que ocurre aquí y para que se entienda, es que hipotéticamente la variable se **eleva** y pasa a declararse **al comienzo de su contexto**, en este caso la función que la contiene.

El ejemplo anterior se entiende implícitamente como:

var x = 5;

(function () {

var x; // Se elevo la declaración

console.log("x:", x); // undefined

x = 10;

console.log("x:", x); // 10

}());

JavaScript sólo utiliza el hoisting en **declaraciones**, no inicializaciones. Si está utilizando una variable que **es declarada e inicializada después**(está después en el código) **de usarla**, el valor será undefined. El siguiente ejemplo demuestra ese comportamiento:

var x = 1; // Inicializa x

console.log(x + " " + y); // '1 undefined'

var y = 2; // Inicializa y

Como se puede apreciar **la elevación afecta la declaración** de variables, pero **no su inicialización**. El valor será asignado exactamente cuando la sentencia de asignación sea alcanzada.

El ejemplo anterior se entiende implícitamente como:

var x = 1; // Inicializa x

var y;// Se elevo la declaración

console.log(x + " " + y); // '1 undefined'

y = 2; // Inicializa y

# ¿Qué es el hoisting?

En JavaScript, las declaraciones (por ejemplo, de variables o funciones) se mueven al principio de su [scope](https://www.w3schools.com/js/js_scope.asp) o ámbito. Este comportamiento se conoce como [hoisting](https://developer.mozilla.org/es/docs/Glossary/Hoisting) y es muy importante tenerlo en cuenta a la hora de programar para prevenir posibles errores.

Teniendo en cuenta cómo funciona el hoisting, **podemos llamar a una función y definirla más abajo**, porque automáticamente JS la “subirá”. Así, este código funciona correctamente:

add();function add() {  
 var myNumber = 4;  
 console.log(myNumber + myNumber);  
}

Porque, aproximadamente, JS está haciendo:

function add() {  
 var myNumber;  
 myNumber = 4;  
 console.log(myNumber + myNumber);  
}add();

En el caso de las**variables, es muy importante tener en cuenta que el hoisting solo se aplica a la declaración**, y no a su asignación.

Por lo tanto, **NO** podríamos escribir esto:

//Erroradd();function add() {  
 console.log(myNumber + myNumber);  
}var myNumber = 4;

Es esencial quedarnos con dos conceptos:

* Las funciones siempre se mueven arriba del scope. Por lo tanto, podemos elegir donde declararlas y usarlas.
* La declaración de las variables se mueven arriba del scope, pero no la asignación. Antes de usar una variable, habrá que crearla y asignarla.

# Expressions and operators

* [« Previous](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions)
* [Next »](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Numbers_and_dates)

This chapter describes JavaScript's expressions and operators, including assignment, comparison, arithmetic, bitwise, logical, string, ternary and more.

A complete and detailed list of operators and expressions is also available in the [reference](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators).

## [Operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#operators)

JavaScript has the following types of operators. This section describes the operators and contains information about operator precedence.

* [Assignment operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#assignment_operators)
* [Comparison operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#comparison_operators)
* [Arithmetic operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#arithmetic_operators)
* [Bitwise operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#bitwise_operators)
* [Logical operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#logical_operators)
* [String operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#string_operators)
* [Conditional (ternary) operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#conditional_ternary_operator)
* [Comma operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#comma_operator)
* [Unary operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#unary_operators)
* [Relational operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#relational_operators)

JavaScript has both binary and unary operators, and one special ternary operator, the conditional operator. A binary operator requires two operands, one before the operator and one after the operator:

operand1 operator operand2

For example, 3+4 or x\*y.

A unary operator requires a single operand, either before or after the operator:

operator operand

or

operand operator

For example, x++ or ++x.

### [Assignment operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#assignment_operators)

An assignment operator assigns a value to its left operand based on the value of its right operand. The simple assignment operator is equal (=), which assigns the value of its right operand to its left operand. That is, x = f() is an assignment expression that assigns the value of f() to x.

There are also compound assignment operators that are shorthand for the operations listed in the following table:

| **Name** | **Shorthand operator** | **Meaning** |
| --- | --- | --- |
| [Assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment) | x = f() | x = f() |
| [Addition assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Addition_assignment) | x += f() | x = x + f() |
| [Subtraction assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Subtraction_assignment) | x -= f() | x = x - f() |
| [Multiplication assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Multiplication_assignment) | x \*= f() | x = x \* f() |
| [Division assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Division_assignment) | x /= f() | x = x / f() |
| [Remainder assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Remainder_assignment) | x %= f() | x = x % f() |
| [Exponentiation assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Exponentiation_assignment) | x \*\*= f() | x = x \*\* f() |
| [Left shift assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Left_shift_assignment) | x <<= f() | x = x << f() |
| [Right shift assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Right_shift_assignment) | x >>= f() | x = x >> f() |
| [Unsigned right shift assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Unsigned_right_shift_assignment) | x >>>= f() | x = x >>> f() |
| [Bitwise AND assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_AND_assignment) | x &= f() | x = x & f() |
| [Bitwise XOR assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_XOR_assignment) | x ^= f() | x = x ^ f() |
| [Bitwise OR assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_OR_assignment) | x |= f() | x = x | f() |
| [Logical AND assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_AND_assignment) | x &&= f() | x && (x = f()) |
| [Logical OR assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_OR_assignment) | x ||= f() | x || (x = f()) |
| [Logical nullish assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_nullish_assignment) | x ??= f() | x ?? (x = f()) |

#### Assigning to properties

If a variable refers to an [object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects), then the left-hand side of an assignment expression may make assignments to properties of that variable. For example:

let obj = {};

obj.x = 3;

console.log(obj.x); // Prints 3.

console.log(obj); // Prints { x: 3 }.

const key = "y";

obj[key] = 5;

console.log(obj[key]); // Prints 5.

console.log(obj); // Prints { x: 3, y: 5 }.

For more information about objects, read [Working with Objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects).

#### Destructuring

For more complex assignments, the [destructuring assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment) syntax is a JavaScript expression that makes it possible to extract data from arrays or objects using a syntax that mirrors the construction of array and object literals.

var foo = ['one', 'two', 'three'];

// without destructuring

var one = foo[0];

var two = foo[1];

var three = foo[2];

// with destructuring

var [one, two, three] = foo;

#### Evaluation and nesting

In general, assignments are used within a variable declaration (i.e., with [const](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const), [let](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let), or [var](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var)) or as standalone statements).

// Declares a variable x and initializes it to the result of f().

// The result of the x = f() assignment expression is discarded.

let x = f();

// Declares a variable x and initializes it to the result of g().

// The result of the x = g() assignment expression is discarded.

x = g(); // Reassigns the variable x to the result of g().

However, like other expressions, assignment expressions like x = f() evaluate into a result value. Although this result value is usually not used, it can then be used by another expression.

Chaining assignments or nesting assignments in other expressions can result in surprising behavior. For this reason, some JavaScript style guides [discourage chaining or nesting assignments](https://github.com/airbnb/javascript/blob/master/README.md#variables--no-chain-assignment)). Nevertheless, assignment chaining and nesting may occur sometimes, so it is important to be able to understand how they work.

By chaining or nesting an assignment expression, its result can itself be assigned to another variable. It can be logged, it can be put inside an array literal or function call, and so on.

let x;

const y = (x = f()); // Or equivalently: const y = x = f();

console.log(y); // Logs the return value of the assignment x = f().

console.log(x = f()); // Logs the return value directly.

// An assignment expression can be nested in any place

// where expressions are generally allowed,

// such as array literals' elements or as function calls' arguments.

console.log([ 0, x = f(), 0 ]);

console.log(f(0, x = f(), 0));

The evaluation result matches the expression to the right of the = sign in the "Meaning" column of the table above. That means that x = f() evaluates into whatever f()'s result is, x += f() evaluates into the resulting sum x + f(), x \*\*= f() evaluates into the resulting power x \*\* y, and so on.

In the case of logical assignments, x &&= f(), x ||= f(), and x ??= f(), the return value is that of the logical operation without the assignment, so x && f(), x || f(), and x ?? f(), respectively.

When chaining these expressions without parentheses or other grouping operators like array literals, the assignment expressions are **grouped right to left** (they are [right-associative](https://en.wikipedia.org/wiki/Operator_associativity)), but they are **evaluated left to right**.

Note that, for all assignment operators other than = itself, the resulting values are always based on the operands' values before the operation.

For example, assume that the following functions f and g and the variables x and y have been declared:

function f () {

console.log('F!');

return 2;

}

function g () {

console.log('G!');

return 3;

}

let x, y;

Consider these three examples:

y = x = f()

y = [ f(), x = g() ]

x[f()] = g()

##### EVALUATION EXAMPLE 1

y = x = f() is equivalent to y = (x = f()), because the assignment operator = is [right-associative](https://en.wikipedia.org/wiki/Operator_associativity). However, it evaluates from left to right:

1. The assignment expression y = x = f() starts to evaluate.
   1. The y on this assignment's left-hand side evaluates into a reference to the variable named y.
   2. The assignment expression x = f() starts to evaluate.
      1. The x on this assignment's left-hand side evaluates into a reference to the variable named x.
      2. The function call f() prints "F!" to the console and then evaluates to the number 2.
      3. That 2 result from f() is assigned to x.
   3. The assignment expression x = f() has now finished evaluating; its result is the new value of x, which is 2.
   4. That 2 result in turn is also assigned to y.
2. The assignment expression y = x = f() has now finished evaluating; its result is the new value of y – which happens to be 2. x and y are assigned to 2, and the console has printed "F!".

##### EVALUATION EXAMPLE 2

y = [ f(), x = g() ] also evaluates from left to right:

1. The assignment expression y = [ f(), x = g() ] starts to evaluate.
   1. The y on this assignment's left-hand evaluates into a reference to the variable named y.
   2. The inner array literal [ f(), x = g() ] starts to evaluate.
      1. The function call f() prints "F!" to the console and then evaluates to the number 2.
      2. The assignment expression x = g() starts to evaluate.
         1. The x on this assignment's left-hand side evaluates into a reference to the variable named x.
         2. The function call g() prints "G!" to the console and then evaluates to the number 3.
         3. That 3 result from g() is assigned to x.
      3. The assignment expression x = g() has now finished evaluating; its result is the new value of x, which is 3. That 3 result becomes the next element in the inner array literal (after the 2 from the f()).
   3. The inner array literal [ f(), x = g() ] has now finished evaluating; its result is an array with two values: [ 2, 3 ].
   4. That [ 2, 3 ] array is now assigned to y.
2. The assignment expression y = [ f(), x = g() ] has now finished evaluating; its result is the new value of y – which happens to be [ 2, 3 ]. x is now assigned to 3, y is now assigned to [ 2, 3 ], and the console has printed "F!" then "G!".

##### EVALUATION EXAMPLE 3

x[f()] = g() also evaluates from left to right. (This example assumes that x is already assigned to some [object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects). For more information about objects, read [Working with Objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects).)

1. The assignment expression x[f()] = g() starts to evaluate.
   1. The x[f()] property access on this assignment's left-hand starts to evaluate.
      1. The x in this property access evaluates into a reference to the variable named x.
      2. Then the function call f() prints "F!" to the console and then evaluates to the number 2.
   2. The x[f()] property access on this assignment has now finished evaluating; its result is a variable property reference: x[2].
   3. Then the function call g() prints "G!" to the console and then evaluates to the number 3.
   4. That 3 is now assigned to x[2]. (This step will succeed only if x is assigned to an [object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects).)
2. The assignment expression x[f()] = g() has now finished evaluating; its result is the new value of x[2] – which happens to be 3. x[2] is now assigned to 3, and the console has printed "F!" then "G!".

#### Avoid assignment chains

Chaining assignments or nesting assignments in other expressions can result in surprising behavior. For this reason, [chaining assignments in the same statement is discouraged](https://github.com/airbnb/javascript/blob/master/README.md#variables--no-chain-assignment)).

In particular, putting a variable chain in a [const](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const), [let](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let), or [var](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var) statement often does not work. Only the outermost/leftmost variable would get declared; any other variables within the assignment chain are not declared by the const/let/var statement. For example:

let z = y = x = f();

This statement seemingly declares the variables x, y, and z. However, it only actually declares the variable z. y and x are either invalid references to nonexistent variables (in [strict mode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode)) or, worse, would implicitly create [global variables](https://developer.mozilla.org/en-US/docs/Glossary/Global_variable) for x and y in [sloppy mode](https://developer.mozilla.org/en-US/docs/Glossary/Sloppy_mode).

### [Comparison operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#comparison_operators)

A comparison operator compares its operands and returns a logical value based on whether the comparison is true. The operands can be numerical, string, logical, or [object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects) values. Strings are compared based on standard lexicographical ordering, using Unicode values. In most cases, if the two operands are not of the same type, JavaScript attempts to convert them to an appropriate type for the comparison. This behavior generally results in comparing the operands numerically. The sole exceptions to type conversion within comparisons involve the === and !== operators, which perform strict equality and inequality comparisons. These operators do not attempt to convert the operands to compatible types before checking equality. The following table describes the comparison operators in terms of this sample code:

var var1 = 3;

var var2 = 4;

| Comparison operators | | |
| --- | --- | --- |
| **Operator** | **Description** | **Examples returning true** |
| [Equal](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#equality) (==) | Returns true if the operands are equal. | 3 == var1  "3" == var1  3 == '3' |
| [Not equal](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#inequality) (!=) | Returns true if the operands are not equal. | var1 != 4 var2 != "3" |
| [Strict equal](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#identity) (===) | Returns true if the operands are equal and of the same type. See also [Object.is](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is) and [sameness in JS](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness). | 3 === var1 |
| [Strict not equal](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Strict_inequality) (!==) | Returns true if the operands are of the same type but not equal, or are of different type. | var1 !== "3" 3 !== '3' |
| [Greater than](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#greater_than_operator) (>) | Returns true if the left operand is greater than the right operand. | var2 > var1 "12" > 2 |
| [Greater than or equal](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#greater_than_or_equal_operator) (>=) | Returns true if the left operand is greater than or equal to the right operand. | var2 >= var1 var1 >= 3 |
| [Less than](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#less_than_operator) (<) | Returns true if the left operand is less than the right operand. | var1 < var2 "2" < 12 |
| [Less than or equal](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#less_than_or_equal_operator) (<=) | Returns true if the left operand is less than or equal to the right operand. | var1 <= var2 var2 <= 5 |

**Note:** => is not a comparison operator but rather is the notation for [Arrow functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions).

### [Arithmetic operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#arithmetic_operators)

An arithmetic operator takes numerical values (either literals or variables) as their operands and returns a single numerical value. The standard arithmetic operators are addition (+), subtraction (-), multiplication (\*), and division (/). These operators work as they do in most other programming languages when used with floating point numbers (in particular, note that division by zero produces [Infinity](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Infinity)). For example:

1 / 2; // 0.5

1 / 2 == 1.0 / 2.0; // this is true

In addition to the standard arithmetic operations (+, -, \*, /), JavaScript provides the arithmetic operators listed in the following table:

| Arithmetic operators | | |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| [Remainder](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Remainder) (%) | Binary operator. Returns the integer remainder of dividing the two operands. | 12 % 5 returns 2. |
| [Increment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Increment) (++) | Unary operator. Adds one to its operand. If used as a prefix operator (++x), returns the value of its operand after adding one; if used as a postfix operator (x++), returns the value of its operand before adding one. | If x is 3, then ++x sets x to 4 and returns 4, whereas x++ returns 3 and, only then, sets x to 4. |
| [Decrement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Decrement) (--) | Unary operator. Subtracts one from its operand. The return value is analogous to that for the increment operator. | If x is 3, then --x sets x to 2 and returns 2, whereas x-- returns 3 and, only then, sets x to 2. |
| [Unary negation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Unary_negation) (-) | Unary operator. Returns the negation of its operand. | If x is 3, then -x returns -3. |
| [Unary plus](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Unary_plus) (+) | Unary operator. Attempts to convert the operand to a number, if it is not already. | +"3" returns 3.  +true returns 1. |
| [Exponentiation operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Exponentiation) (\*\*) | Calculates the base to the exponent power, that is, base^exponent | 2 \*\* 3 returns 8. 10 \*\* -1 returns 0.1. |

### [Bitwise operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#bitwise_operators)

A bitwise operator treats their operands as a set of 32 bits (zeros and ones), rather than as decimal, hexadecimal, or octal numbers. For example, the decimal number nine has a binary representation of 1001. Bitwise operators perform their operations on such binary representations, but they return standard JavaScript numerical values.

The following table summarizes JavaScript's bitwise operators.

| **Operator** | **Usage** | **Description** |
| --- | --- | --- |
| [Bitwise AND](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_AND) | a & b | Returns a one in each bit position for which the corresponding bits of both operands are ones. |
| [Bitwise OR](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_OR) | a | b | Returns a zero in each bit position for which the corresponding bits of both operands are zeros. |
| [Bitwise XOR](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_XOR) | a ^ b | Returns a zero in each bit position for which the corresponding bits are the same. [Returns a one in each bit position for which the corresponding bits are different.] |
| [Bitwise NOT](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_NOT) | ~ a | Inverts the bits of its operand. |
| [Left shift](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Left_shift) | a << b | Shifts a in binary representation b bits to the left, shifting in zeros from the right. |
| [Sign-propagating right shift](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Right_shift) | a >> b | Shifts a in binary representation b bits to the right, discarding bits shifted off. |
| [Zero-fill right shift](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Unsigned_right_shift) | a >>> b | Shifts a in binary representation b bits to the right, discarding bits shifted off, and shifting in zeros from the left. |

#### Bitwise logical operators

Conceptually, the bitwise logical operators work as follows:

* The operands are converted to thirty-two-bit integers and expressed by a series of bits (zeros and ones). Numbers with more than 32 bits get their most significant bits discarded. For example, the following integer with more than 32 bits will be converted to a 32-bit integer:
* Before: 1110 0110 1111 1010 0000 0000 0000 0110 0000 0000 0001
* After: 1010 0000 0000 0000 0110 0000 0000 0001
* Each bit in the first operand is paired with the corresponding bit in the second operand: first bit to first bit, second bit to second bit, and so on.
* The operator is applied to each pair of bits, and the result is constructed bitwise.

For example, the binary representation of nine is 1001, and the binary representation of fifteen is 1111. So, when the bitwise operators are applied to these values, the results are as follows:

| **Expression** | **Result** | **Binary Description** |
| --- | --- | --- |
| 15 & 9 | 9 | 1111 & 1001 = 1001 |
| 15 | 9 | 15 | 1111 | 1001 = 1111 |
| 15 ^ 9 | 6 | 1111 ^ 1001 = 0110 |
| ~15 | -16 | ~ 0000 0000 ... 0000 1111 = 1111 1111 ... 1111 0000 |
| ~9 | -10 | ~ 0000 0000 ... 0000 1001 = 1111 1111 ... 1111 0110 |

Note that all 32 bits are inverted using the Bitwise NOT operator, and that values with the most significant (left-most) bit set to 1 represent negative numbers (two's-complement representation). ~x evaluates to the same value that -x - 1 evaluates to.

#### Bitwise shift operators

The bitwise shift operators take two operands: the first is a quantity to be shifted, and the second specifies the number of bit positions by which the first operand is to be shifted. The direction of the shift operation is controlled by the operator used.

Shift operators convert their operands to thirty-two-bit integers and return a result of either type [Number](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number) or [BigInt](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt): specifically, if the type of the left operand is [BigInt](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt), they return [BigInt](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt); otherwise, they return [Number](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number).

The shift operators are listed in the following table.

| Bitwise shift operators | | |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| [Left shift](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Left_shift) (<<) | This operator shifts the first operand the specified number of bits to the left. Excess bits shifted off to the left are discarded. Zero bits are shifted in from the right. | 9<<2 yields 36, because 1001 shifted 2 bits to the left becomes 100100, which is 36. |
| [Sign-propagating right shift](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Right_shift) (>>) | This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Copies of the leftmost bit are shifted in from the left. | 9>>2 yields 2, because 1001 shifted 2 bits to the right becomes 10, which is 2. Likewise, -9>>2 yields -3, because the sign is preserved. |
| [Zero-fill right shift](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Unsigned_right_shift) (>>>) | This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Zero bits are shifted in from the left. | 19>>>2 yields 4, because 10011 shifted 2 bits to the right becomes 100, which is 4. For non-negative numbers, zero-fill right shift and sign-propagating right shift yield the same result. |

### [Logical operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#logical_operators)

Logical operators are typically used with Boolean (logical) values; when they are, they return a Boolean value. However, the && and || operators actually return the value of one of the specified operands, so if these operators are used with non-Boolean values, they may return a non-Boolean value. The logical operators are described in the following table.

| Logical operators | | |
| --- | --- | --- |
| **Operator** | **Usage** | **Description** |
| [Logical AND](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_AND) (&&) | expr1 && expr2 | Returns expr1 if it can be converted to false; otherwise, returns expr2. Thus, when used with Boolean values, && returns true if both operands are true; otherwise, returns false. |
| [Logical OR](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_OR)(||) | expr1 || expr2 | Returns expr1 if it can be converted to true; otherwise, returns expr2. Thus, when used with Boolean values, || returns true if either operand is true; if both are false, returns false. |
| [Logical NOT](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_NOT)(!) | !expr | Returns false if its single operand that can be converted to true; otherwise, returns true. |

Examples of expressions that can be converted to false are those that evaluate to null, 0, NaN, the empty string (""), or undefined.

The following code shows examples of the && (logical AND) operator.

var a1 = true && true; // t && t returns true

var a2 = true && false; // t && f returns false

var a3 = false && true; // f && t returns false

var a4 = false && (3 == 4); // f && f returns false

var a5 = 'Cat' && 'Dog'; // t && t returns Dog

var a6 = false && 'Cat'; // f && t returns false

var a7 = 'Cat' && false; // t && f returns false

The following code shows examples of the || (logical OR) operator.

var o1 = true || true; // t || t returns true

var o2 = false || true; // f || t returns true

var o3 = true || false; // t || f returns true

var o4 = false || (3 == 4); // f || f returns false

var o5 = 'Cat' || 'Dog'; // t || t returns Cat

var o6 = false || 'Cat'; // f || t returns Cat

var o7 = 'Cat' || false; // t || f returns Cat

The following code shows examples of the ! (logical NOT) operator.

var n1 = !true; // !t returns false

var n2 = !false; // !f returns true

var n3 = !'Cat'; // !t returns false

#### Short-circuit evaluation

As logical expressions are evaluated left to right, they are tested for possible "short-circuit" evaluation using the following rules:

* false && anything is short-circuit evaluated to false.
* true || anything is short-circuit evaluated to true.

The rules of logic guarantee that these evaluations are always correct. Note that the anything part of the above expressions is not evaluated, so any side effects of doing so do not take effect.

Note that for the second case, in modern code you can use the new [Nullish coalescing operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Nullish_coalescing_operator) (??) that works like ||, but it only returns the second expression, when the first one is "[nullish](https://developer.mozilla.org/en-US/docs/Glossary/Nullish)", i.e. [null](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/null) or [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined). It is thus the better alternative to provide defaults, when values like '' or 0 are valid values for the first expression, too.

### [String operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#string_operators)

In addition to the comparison operators, which can be used on string values, the concatenation operator (+) concatenates two string values together, returning another string that is the union of the two operand strings.

For example,

console.log('my ' + 'string'); // console logs the string "my string".

The shorthand assignment operator += can also be used to concatenate strings.

For example,

var mystring = 'alpha';

mystring += 'bet'; // evaluates to "alphabet" and assigns this value to mystring.

### [Conditional (ternary) operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#conditional_ternary_operator)

The [conditional operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator) is the only JavaScript operator that takes three operands. The operator can have one of two values based on a condition. The syntax is:

condition ? val1 : val2

If condition is true, the operator has the value of val1. Otherwise it has the value of val2. You can use the conditional operator anywhere you would use a standard operator.

For example,

var status = (age >= 18) ? 'adult' : 'minor';

This statement assigns the value "adult" to the variable status if age is eighteen or more. Otherwise, it assigns the value "minor" to status.

### [Comma operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#comma_operator)

The [comma operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comma_Operator) (,) evaluates both of its operands and returns the value of the last operand. This operator is primarily used inside a for loop, to allow multiple variables to be updated each time through the loop. It is regarded bad style to use it elsewhere, when it is not necessary. Often two separate statements can and should be used instead.

For example, if a is a 2-dimensional array with 10 elements on a side, the following code uses the comma operator to update two variables at once. The code prints the values of the diagonal elements in the array:

var x = [0,1,2,3,4,5,6,7,8,9]

var a = [x, x, x, x, x];

for (var i = 0, j = 9; i <= j; i++, j--)

// ^

console.log('a[' + i + '][' + j + ']= ' + a[i][j]);

### [Unary operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#unary_operators)

A unary operation is an operation with only one operand.

#### delete

The [delete](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete) operator deletes an object's property. The syntax is:

delete object.property;

delete object[propertyKey];

delete objectName[index];

where object is the name of an object, property is an existing property, and propertyKey is a string or symbol referring to an existing property.

If the delete operator succeeds, it removes the property from the object. Trying to access it afterwards will yield undefined. The delete operator returns true if the operation is possible; it returns false if the operation is not possible.

delete Math.PI; // returns false (cannot delete non-configurable properties)

const myObj = {h: 4};

delete myObj.h; // returns true (can delete user-defined properties)

##### DELETING ARRAY ELEMENTS

Since arrays are just objects, it's technically possible to delete elements from them. This is however regarded as a bad practice, try to avoid it. When you delete an array property, the array length is not affected and other elements are not re-indexed. To achieve that behavior, it is much better to just overwrite the element with the value undefined. To actually manipulate the array, use the various array methods such as [splice](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice).

#### typeof

The [typeof operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof) is used in either of the following ways:

typeof operand

typeof (operand)

The typeof operator returns a string indicating the type of the unevaluated operand. operand is the string, variable, keyword, or object for which the type is to be returned. The parentheses are optional.

Suppose you define the following variables:

var myFun = new Function('5 + 2');

var shape = 'round';

var size = 1;

var foo = ['Apple', 'Mango', 'Orange'];

var today = new Date();

The typeof operator returns the following results for these variables:

typeof myFun; // returns "function"

typeof shape; // returns "string"

typeof size; // returns "number"

typeof foo; // returns "object"

typeof today; // returns "object"

typeof doesntExist; // returns "undefined"

For the keywords true and null, the typeof operator returns the following results:

typeof true; // returns "boolean"

typeof null; // returns "object"

For a number or string, the typeof operator returns the following results:

typeof 62; // returns "number"

typeof 'Hello world'; // returns "string"

For property values, the typeof operator returns the type of value the property contains:

typeof document.lastModified; // returns "string"

typeof window.length; // returns "number"

typeof Math.LN2; // returns "number"

For methods and functions, the typeof operator returns results as follows:

typeof blur; // returns "function"

typeof eval; // returns "function"

typeof parseInt; // returns "function"

typeof shape.split; // returns "function"

For predefined objects, the typeof operator returns results as follows:

typeof Date; // returns "function"

typeof Function; // returns "function"

typeof Math; // returns "object"

typeof Option; // returns "function"

typeof String; // returns "function"

#### void

The [void operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/void) is used in either of the following ways:

void (expression)

void expression

The void operator specifies an expression to be evaluated without returning a value. expression is a JavaScript expression to evaluate. The parentheses surrounding the expression are optional, but it is good style to use them.

### [Relational operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#relational_operators)

A relational operator compares its operands and returns a Boolean value based on whether the comparison is true.

#### in

The [in operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/in) returns true if the specified property is in the specified object. The syntax is:

propNameOrNumber in objectName

where propNameOrNumber is a string, numeric, or symbol expression representing a property name or array index, and objectName is the name of an object.

The following examples show some uses of the in operator.

// Arrays

var trees = ['redwood', 'bay', 'cedar', 'oak', 'maple'];

0 in trees; // returns true

3 in trees; // returns true

6 in trees; // returns false

'bay' in trees; // returns false (you must specify the index number,

// not the value at that index)

'length' in trees; // returns true (length is an Array property)

// built-in objects

'PI' in Math; // returns true

var myString = new String('coral');

'length' in myString; // returns true

// Custom objects

var mycar = { make: 'Honda', model: 'Accord', year: 1998 };

'make' in mycar; // returns true

'model' in mycar; // returns true

#### instanceof

The [instanceof operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/instanceof) returns true if the specified object is of the specified object type. The syntax is:

objectName instanceof objectType

where objectName is the name of the object to compare to objectType, and objectType is an object type, such as [Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date) or [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array).

Use instanceof when you need to confirm the type of an object at runtime. For example, when catching exceptions, you can branch to different exception-handling code depending on the type of exception thrown.

For example, the following code uses instanceof to determine whether theDay is a Date object. Because theDay is a Date object, the statements in the if statement execute.

var theDay = new Date(1995, 12, 17);

if (theDay instanceof Date) {

// statements to execute

}

### [Operator precedence](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#operator_precedence)

The precedence of operators determines the order they are applied when evaluating an expression. You can override operator precedence by using parentheses.

The following table describes the precedence of operators, from highest to lowest.

| **Operator type** | **Individual operators** |
| --- | --- |
| member | . [] |
| call / create instance | () new |
| negation/increment | ! ~ - + ++ -- typeof void delete |
| exponentiate | \*\* |
| multiply/divide | \* / % |
| addition/subtraction | + - |
| bitwise shift | << >> >>> |
| relational | < <= > >= in instanceof |
| equality | == != === !== |
| bitwise-and | & |
| bitwise-xor | ^ |
| bitwise-or | | |
| logical-and | && |
| logical-or | || |
| conditional | ?: |
| assignment | = += -= \*\*= \*= /= %= <<= >>= >>>= &= ^= |= &&= ||= ??= |
| comma | , |

A more detailed version of this table, complete with links to additional details about each operator, may be found in the [JavaScript Reference](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence#table).

## [Expressions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#expressions)

An expression is any valid unit of code that resolves to a value.

Every syntactically valid expression resolves to some value but conceptually, there are two types of expressions: with side effects (for example: those that assign value to a variable) and those that in some sense evaluate and therefore resolve to a value.

The expression x = 7 is an example of the first type. This expression uses the = operator to assign the value seven to the variable x. The expression itself evaluates to seven.

The code 3 + 4 is an example of the second expression type. This expression uses the + operator to add three and four together without assigning the result, seven, to a variable.

JavaScript has the following expression categories:

* Arithmetic: evaluates to a number, for example 3.14159. (Generally uses [arithmetic operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#arithmetic_operators).)
* String: evaluates to a character string, for example, "Fred" or "234". (Generally uses [string operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#string_operators).)
* Logical: evaluates to true or false. (Often involves [logical operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#logical_operators).)
* Primary expressions: Basic keywords and general expressions in JavaScript.
* Left-hand-side expressions: Left values are the destination of an assignment.

### [Primary expressions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#primary_expressions)

Basic keywords and general expressions in JavaScript.

#### this

Use the [this keyword](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this) to refer to the current object. In general, this refers to the calling object in a method. Use this either with the dot or the bracket notation:

this['propertyName']

this.propertyName

Suppose a function called validate validates an object's value property, given the object and the high and low values:

function validate(obj, lowval, hival) {

if ((obj.value < lowval) || (obj.value > hival))

console.log('Invalid Value!');

}

You could call validate in each form element's onChange event handler, using this to pass it to the form element, as in the following example:

<p>Enter a number between 18 and 99:</p>

<input type="text" name="age" size=3 onChange="validate(this, 18, 99);">

#### Grouping operator

The grouping operator ( ) controls the precedence of evaluation in expressions. For example, you can override multiplication and division first, then addition and subtraction to evaluate addition first.

var a = 1;

var b = 2;

var c = 3;

// default precedence

a + b \* c // 7

// evaluated by default like this

a + (b \* c) // 7

// now overriding precedence

// addition before multiplication

(a + b) \* c // 9

// which is equivalent to

a \* c + b \* c // 9

### [Left-hand-side expressions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#left-hand-side_expressions)

Left values are the destination of an assignment.

#### new

You can use the [new operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new) to create an instance of a user-defined object type or of one of the built-in object types. Use new as follows:

var objectName = new objectType([param1, param2, ..., paramN]);

#### super

The [super keyword](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super) is used to call functions on an object's parent. It is useful with [classes](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes) to call the parent constructor, for example.

super([arguments]); // calls the parent constructor.

super.functionOnParent([arguments]);

# Operador condicional (ternario)

El**operador condicional**(**ternario**) es el único operador en JavaScript que tiene tres operandos. Este operador se usa con frecuencia como atajo para la instrucción [if](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Sentencias/if...else).

## [Sintaxis](https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/Conditional_Operator#sintaxis)

condición ? expr1 : expr2

## [Descripción](https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/Conditional_Operator#descripci%C3%B3n)

### Parámetros

condición

Una expresión que se evalúa como true o false.

expr1, expr2

Expresión con valores de algún tipo.

Si la condición es true, el operador retorna el valor de la expr1; de lo contrario,  devuelve el valor de expr2. Por ejemplo, para mostrar un mensaje diferente en función del valor de la variable *isMember,* se puede usar esta declaración:

"La Cuota es de: " + (isMember ? "$2.00" : "$10.00")

También puedes asignar variables dependiendo del resultado de la condición ternaria:

var elvisLives = Math.PI > 4 ? "Sip" : "Nop";

También es posible realizar evaluaciones ternarias múltiples (Nota: El operador condicional es asociativo):

var firstCheck = false,

secondCheck = false,

access = firstCheck ? "Acceso Denegado" : secondCheck ? "Acceso Denegado" : "Acceso Permitido";

console.log( access ); // muestra "Acceso Permitido"

También puede usar operaciones ternarias en espacio vacío con el propósito de realizar diferentes operaciones:

var stop = false, age = 16;

age > 18 ? location.assign("continue.html") : stop = true;

También puede realizar más de una operación por caso, separándolas con una coma:

var stop = false, age = 23;

age > 18 ? (

alert("OK, puedes continuar."),

location.assign("continue.html")

) : (

stop = true,

alert("Disculpa, eres menor de edad!")

);

También puede realizar más de una operación durante la asignación de un valor. En este caso, **el último valor separado por una coma del paréntesis será el valor asignado**.

var age = 16;

var url = age > 18 ? (

alert("OK, puedes continuar."),

// alert devuelve "undefined", pero será ignorado porque

// no es el último valor separado por comas del paréntesis

"continue.html" // el valor a ser asignado si age > 18

) : (

alert("Eres menor de edad!"),

alert("Disculpa :-("),

// etc. etc.

"stop.html" // el valor a ser asignado si !(age > 18)

);

location.assign(url); // "stop.html"