**¿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

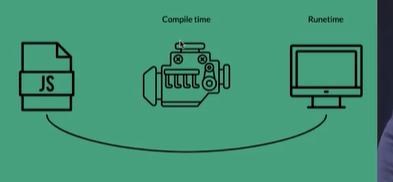
**false** - 3; // -3

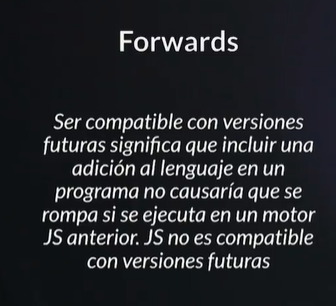
**¿Realmente es Javascript un lenguaje interpretado?**

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.

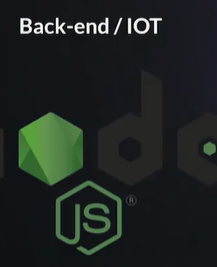






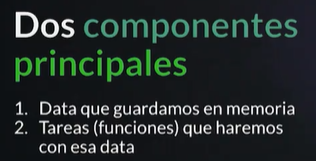


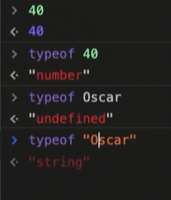




**APUNTES**  
1.- JavaScript tiene una **comunidad enorme** de desarrolladores que te pueden ir ayudando a generar diferentes cosas.

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.