**Curso Básico de JavaScript.**

**¿Qué es JavaScript?**

**¿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.

**¿Qué es Javascript?**  
Es un lenguaje interpretado, orientado a objetos, débilmente tipado y dinámico.

**Débilmente tipado**  
Se pueden hacer operaciones entre tipos distintos de datos (enteros con strings, booleanos con enteros, etc). Ejemplo:



**Dinámico**  
Corre directamente en la etapa de Runetime sin una etapa de compilación previa. Esto permite probar nuestro código inmediatamente; pero también es lo que hace que los errores se muestren hasta que se ejecuta el programa.

**¿Realmente es Javascript un lenguaje interpretado?**  
Si, y la razón es que el navegador lee línea por línea nuestro código el cuál le indica lo que tiene que hacer, sin la necesidad de compilar. Todo esto es controlado por el motor de Javascript V8 del navegador

**Javascript es Backwards Compatible**  
Todas las funciones nuevas que salen de Javascript no dañarán el trabajo ya hecho, pero no se podrá utilizar en nuestro entorno de trabajo inmediatamente. Para solucionar esto, está **Babel** que permite utilizar las nuevas características del lenguaje pero lo transforma a una versión que el navegador pueda entender.

# ¿Por qué JavaScript?

.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 más 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 of Things) es un concepto que se refiere a una interconexión 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.

# 

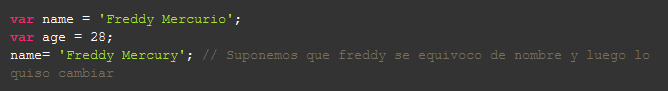
# Elementos de un Lenguaje de Programación: Variables, Funciones y Sintaxis.

# 

# 

# Variables.

**var**: Es una variable que puede cambiar su valor en un futuro y su alcance es local. Surgió en ECMAScript5.  
Ejemplo:



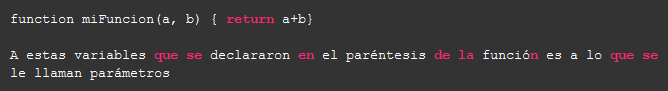
**const**: Es una variable constante que no puede cambiar nunca su valor en un futuro.  
Ejemplo:

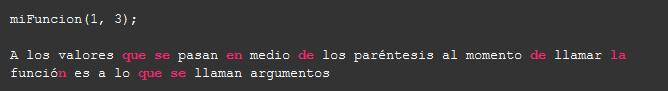


**let**: Es una variable que puede cambiar su valor, pero solo funciona en un bloque donde se declare ( **{let}** )

# 

# 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*** *dato;*

# ¿Cuándo utilizar 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:



#### **Expresión de función:**

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



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.

# Scope.

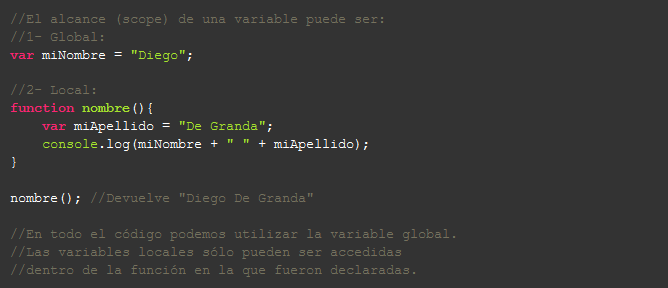
# Scope: Alcance que tienen las variables en el código.

# Existen 2 tipos de sopes.

# Global: Puede ser llamada a lo largo de nuestro programa.

# Local: Solo puede ser llamada dentro del bloque de código en el que se declaro

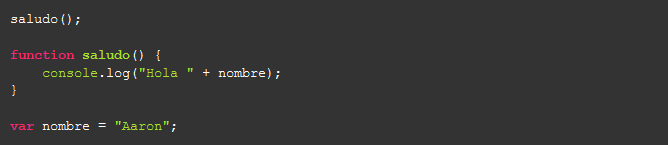
# 



# Hoisting.

El **Hoisting** es un proceso del compilador de JavaScript, que consiste en que la *declaración de las variables* y las *funciones* son llevadas al inicio del código, sin importar su posición, para su procesamiento, sin embargo, la inicialización de las variables no es llevada al inicio del código para su compilación, sino solo su declaración, lo cual suele dar espacio a errores cuando se declara una variable sin inicializarla y se procesa en el código antes de haber llegado a su inicialización, lo cual nos suele dar una variable con valor ***undefined***, ya que la variable sí fue almacenada en memoria, pero no se le asigno un valor hasta después de su ejecución.

Aquí un ejemplo de esto:



El output de este código sería el siguiente:



Debido a que como lo hemos dicho, la variable se tomó en cuenta y se le asigno memoria, sin embargo, el compilador no la inicializo y se le dio el valor de *undefined*, y con ese valor se ingresó a la función, y ya después de correr la función se le asigno el valor.

Este comportamiento se puede entender fácilmente si se comprenden estos dos puntos esenciales:

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

# Coerción.

# 

# Coerción es la forma en la que podemos cambiar un tipo de valor a otro, existen dos tipos de coerción:

# Coerción implícita: es cuando el lenguaje nos ayuda a cambiar el tipo de valor.

# Coerción explicita : es cuando obligamos a que cambie el tipo de valor.



# Valores: Truthy y Falsy.

***¿Qué tipos por default son verdaderos y falsos?***

Usamos la función de JS que es ***Boolean()*** dentro del paréntesis ponemos el valor y nos dice si el mismo el False o True.

–> ***Falsy***

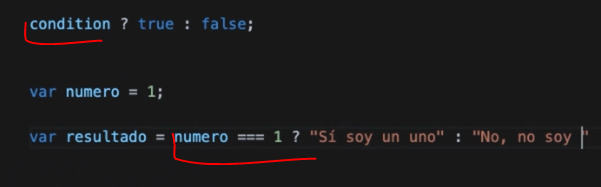
* Boolean() —> sin ningún valor es false
* Boolean(0) —> false
* Boolean(null) —> false
* Boolean(NaN) —> false // NaN es Not and Number
* Boolean(Undefined) —> false
* Boolean(false) —> false
* Boolean("") —> false

–> ***Truthy***

* Boolean(1) —> true //cualquier número que no sea igual a cero es true
* Boolean(“a”) —> true
* Boolean(" ") —> true // siendo un espacio el valor es true
* Boolean([]) —> true // un array nos da un true
* Boolean({}) —> true // un objeto nos da el valor de true
* Boolean(function() {}) —> true //una funcion también es true
* Boolean(true) —> true

Todo esto lo vamos a usar en condiciones esto valida si es verdadero o falso para ejecutar cierta acción.

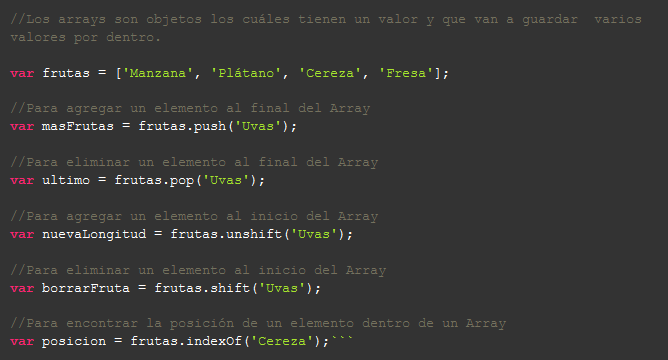
# Otra forma de realizar un If.



# Operadores: Asignación, Comparación y Aritméticos.

# 

# Arrays.



**Mayor documentación:**

<https://devcode.la/tutoriales/manejo-de-arrays-en-javascript/>

<https://dev.to/lukocastillo/time-complexity-big-0-for-javascript-array-methods-and-examples-mlg>

# Loops: For y For...of



# Objects.

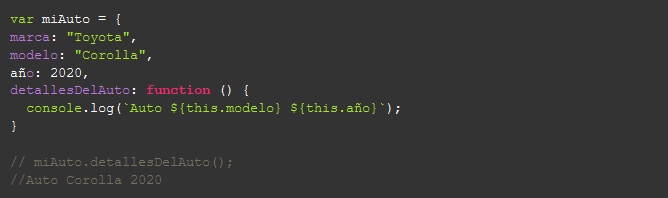
# Para definir un objeto hacemos uso de {}

# 

# Acceder a una propiedad del objeto:

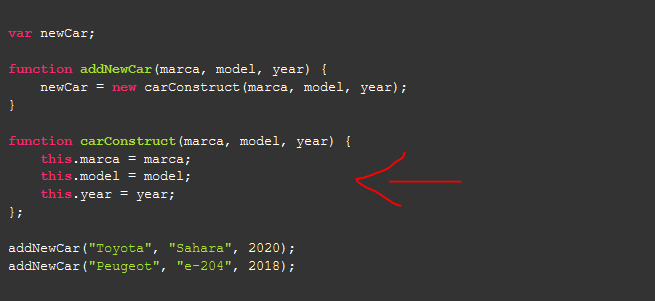


Se pueden agregar propiedades que van a ser una función, se les llama métodos de objetos.

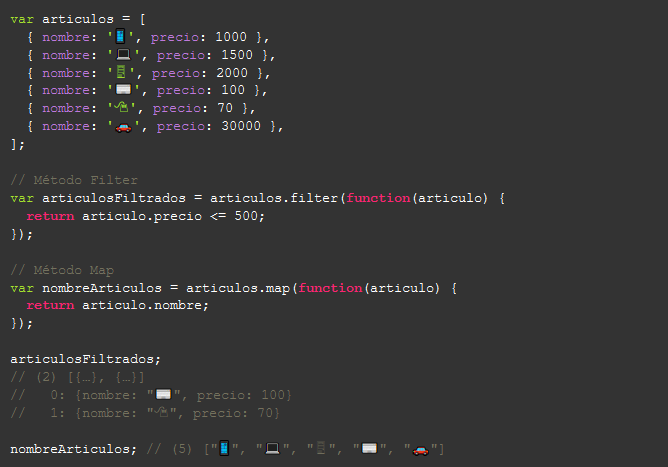


¿Qué es *this*?  
Es una variable que hace referencia al objeto. En este caso: this = miAuto.

# Objects: Función constructora



# Métodos para recorridos Arrays.



Otra forma también valida de usar el *filter* es la siguiente:



Es la misma forma, pero un poco más recortada.

# Recorriendo Arrays con .find(), .forEach() y .some()

# 

Por si a alguien le quedó alguna duda con respecto a la diferencia entre find y filter:

El método *find ()* devuelve el primer valor que coincide de la colección. Una vez que coincida con el valor en los resultados, no verificará los valores restantes en la colección de matriz.

El método *filter ()* devuelve los valores coincidentes en una matriz de la colección. Verificará todos los valores de la colección y devolverá los valores coincidentes en una matriz.

A modo de recopilación:

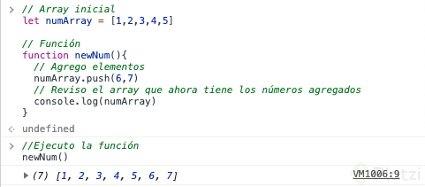
* **find()** : Devuelve el primer elemento del array que cumpla con la condición dada
* **foreach()** : Ejecuta lo que le definamos una vez por cada elemento de nuestro array
* **some()** : Comprueba si al menos un elemento del array cumple con la condición que le damos
* **filter()** : Devuelve todos los elementos del array que cumplan con la condición dada  
  Acá te dejo la documentación de cada uno: [**find()**](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/find) - [**foreach()**](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/forEach) - [**some()**](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/some) - [**filter()**](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/filter)

# Eliminando elementos de un Array.

# .push()

El método .push() nos permite agregar uno o más elementos al final de un array.

A continuación, veremos un ejemplo aplicado con un array que contiene números:



Como podemos ver, al momento de ejecutar la función se agregan los números 6 y 7 al array.

Ahora revisemos un ejemplo con strings:



Como podemos ver, agregamos dos cadenas de strings al ejecutar la función donde tenemos txtArray.push(). Es decir, indico el array al que voy a agregar elementos, uso el método .push(), y dentro de .push() indico los elementos que quiero agregar al string. Finalmente, el console.log() lo uso para revisar en la consola si esto sucedió o no.

# .shift()

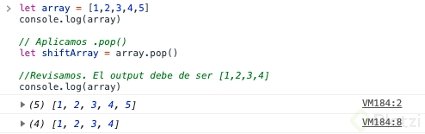
Ahora pasemos a la otra cara de la moneda donde necesitamos eliminar un elemento del array. .shift() eliminar el primer elemento de un array, es decir, elimina el elemento que esté en el índice 0.



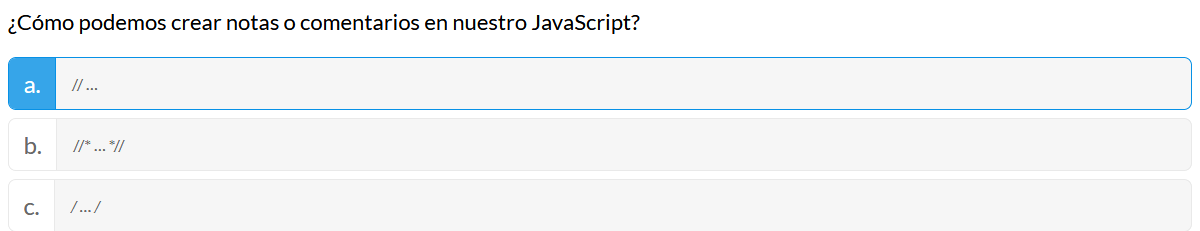
Como vemos, luego de aplicar .shift() se eliminó exitosamente el primer elemento del array. ¿Y si quisiéramos eliminar el último elemento? Pasemos al bonus track de esta clase 🙌🏼.

# Bonus Track

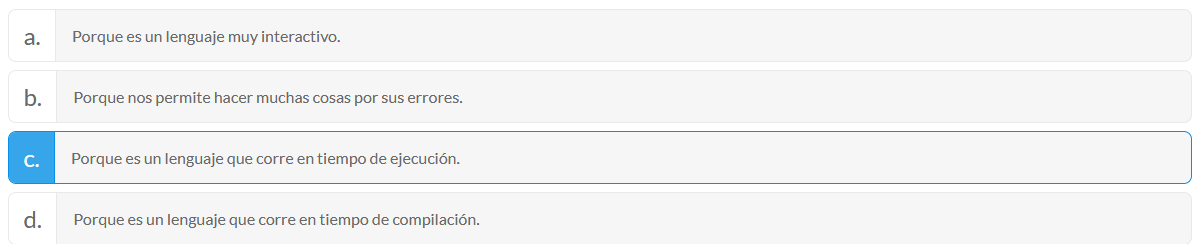
Si ya entendiste cómo funciona .shift() aplicar .pop() te será pan comido 🍞. El método .pop() eliminará el último elemento de un array. En este sentido, si tenemos un array de 5 elementos, pop() eliminará el elemento en el índice 4. Usemos el mismo ejemplo, pero usando este método.

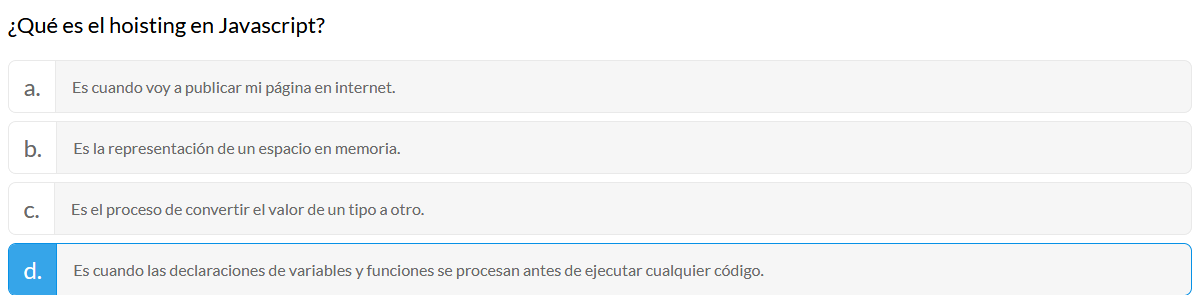


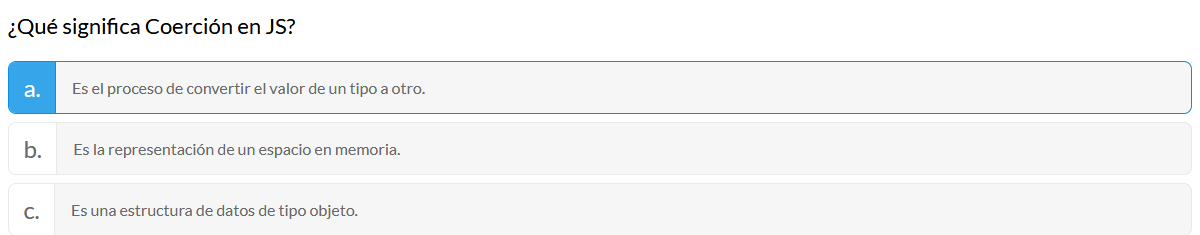
***Questions and Answers***



¿Por qué decimos que JavaScript es un lenguaje dinámico?







**Para practicar JS.**

**https://jsconsole.com/**

**Curso de Fundamentos de JavaScript.**

***Notas del Curso.***

**Bienvenidos al Curso de Fundamentos de JavaScript.**

A lo largo del curso aprenderemos las bases sólidas para dominar el lenguaje JavaScript. Veremos desde lo más básico como: variables, tipos de datos y operadores, e iremos avanzando con los condicionales y las estructuras de control. Luego veremos los conceptos relacionados con asincronismo, uno de los aspectos fundamentales de JavaScript, usando callbacks, promesas y async await. Y finalmente crearemos un juego de Simón dice en el que pondremos en práctica todo lo aprendido.

**LIBROS**

<https://eloquentjavascript.net/00_intro.html>

<https://jsparagatos.com/>

No hay nada mejor que los libros para aprender, así que aquí les algunos de los libros que he leído que en lo personal me han parecido muy buenos

You Don’t Know JavaScript por Kyle Simpson - Link -> (<https://github.com/getify/You-Dont-Know-JS>) – (GRATIS, pero puedes comprar los libros en físico)

Esta serie de libros es un must si quieres dominar los conceptos clave de JavaScript, yo lo he leído varias veces ya que te ayuda a entender cómo funciona el lenguaje a un nivel profundo, es un poco más avanzado, pero nada complejo, ya que siempre parte de temas básicos para que puedas entender con mayor facilidad. Este es un gran libro que les recomiendo lean mientras ven el curso, ya que aquí puede que encuentren varias respuestas a dudas que les puedan surgir durante el curso. Es un excelente complemento, que te ayudara  
a entender mejor como funciona JavaScript, les aseguro que les va a ser de mucha ayuda.

**Eloquent JavaScript por Marijn Haverbeke** - Link -> (<https://eloquentjavascript.net/>) – (GRATIS, pero puedes comprar los libros en físico)

# Variables.

Las variables se declaran colocando la palabra reservada (keyword) **var**

A las variables se les puede asignar el valor que queremos que esta tenga directamente en la misma línea en la que la creamos.

Ej: var nombre = 'Jeremy'

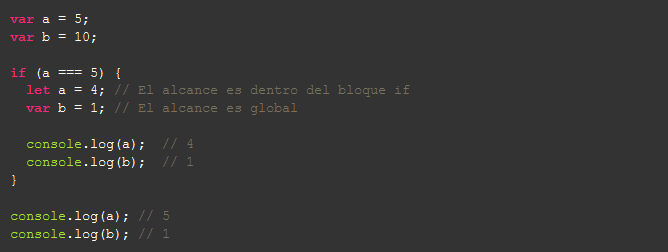
Se pueden declarar múltiples variables en una misma línea de código haciendo uso de la coma.

Ej: var nombre = 'Jeremy', apellido = 'Barroso'

**Diferencias entre LET vs VAR**

let permite declarar variables limitando su alcance (Scope) al bloque, declaración, o expresión donde se está usando. Lo anterior diferencia la expresión let de la palabra reservada var , la cual define una variable global o local en una función sin importar el ámbito del bloque.

Cuando usamos let dentro de un bloque, podemos limitar el alcance de la variable a dicho bloque. Notemos la diferencia a con var, cuyo alcance reside dentro de la función donde ha sido declarada la variable.



# Variables: Strings.

Los strings son cadenas de texto. Para indicar que estamos usando una cadena de texto debemos de colocar las comillas simples.

En esta sección utilizaremos los métodos:

* toUpperCase, que sirve para transformar un String a mayúsculas.
* toLowerCase, que sirve para transformar el string a minúsculas.

También usaremos el atributo:

* length, que nos indica la cantidad de caracteres que tiene un string.

Para concatenar dos strings se utiliza el símbolo (+)  
 *var nombreCompleto = nombre + ’ ’ + apellido*

* **str.toUpperCase()** - Cambia el string a mayúsculas
* **str.toLowerCase()** - Cambia el string a minúsculas
* **str.charAt()** - Trae el carácter en la posición en, requiere el parámetro de la posición del carácter al que queremos acceder.

\*\*\* El 1er carácter de todo string se encuentra en la posición 0. str.charAt(0) .

* **str.length** - Se llama al atributo length, que entrega la cantidad de letras que contiene un string.
* **Interpolación de variables** :

Se realiza con las **comillas invertidas** “``”:

${var1} ${var2}

\*\*\* Dentro de las llaves se puede escribir código js.

${str.toUpperCase()}

* **str.substr(posición, número de caracteres a traer)** - trae una subcadena, la cual es cortada dependiendo del 1er parámetro que es desde donde queremos que empiece y del 2ndo parámetro que es cuántos caracteres queremos que traiga la subcadena.

**Desafío:** Encuentra la última letra de su nombre

**Variables: Números.**

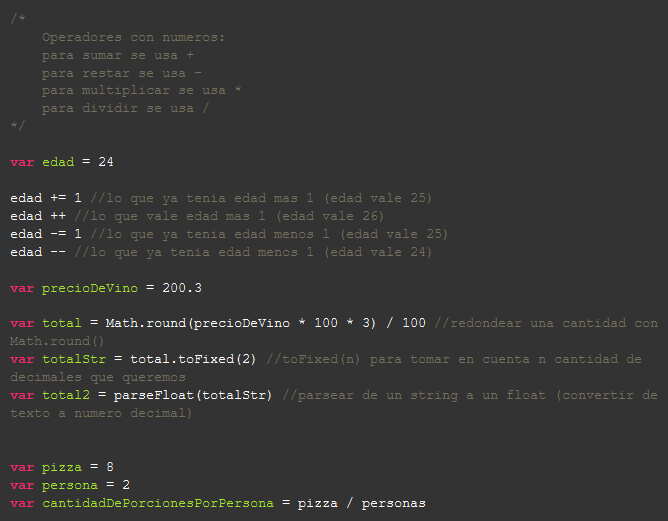
En esta clase aprenderemos variables de tipo números y las operaciones más comunes en este tipo de variables.

En esta clase veremos operadores matemáticos como:

* suma ( + )
* resta ( - )
* multiplicación
* división

Recuerda que en operaciones con decimales debemos realizar operaciones adicionales para conseguir un resultado preciso. Ej. var total = ( precioDeVino \* 100 \* 3) / 100

Para redondear una operación se utiliza la función: Math.round



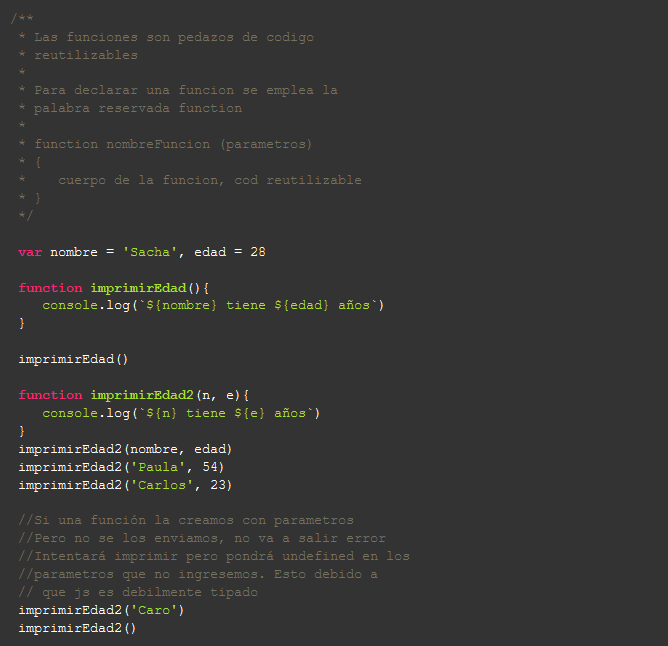


# Funciones.

Las funciones son fracciones de código reutilizable. En esta clase aprenderemos a definir e invocar nuestras funciones. Para definir una función utilizaremos la palabra reservada “***function***”.

Delimitamos el cuerpo de la función usando llaves { }. Los parámetros de la función son variables que se pasan a la función escribiéndolos entre paréntesis ()

Definir funciones nos sirve para reutilizar código. JavaScript es un lenguaje interpretado, esto quiere decir que intentará ejecutar el código sin importar si los parámetros que le pasemos a la función estén invertidos o incluso incompletos.



# El alcance de las funciones.

En esta clase hablaremos del alcance que tienen las funciones. Si una variable no está definida dentro del cuerpo de una función hablamos de una variable global. Por el contrario, una variable definida dentro de una función es una variable local.

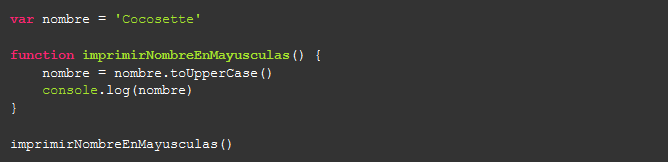
Para que la ejecución de una función no modifique una variable global usamos parámetros en lugar de pasar directamente la variable.

Es posible utilizar el mismo nombre para una variable global y para el parámetro de una función con un alcance local.

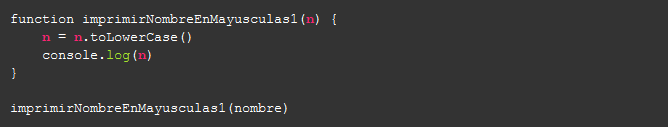
* ***Variable global***: Variable que no está definida dentro de una funcion sino por fuera de ella.

Al hacer referencia directa a una variable global dentro de una funcion, podremos modificar su valor, esto se le denomina sideEfect, y esto lo vamos a querer evitar.

Para acceder a la variable global, hacemos referencia al objeto global, en este caso por estar en el browser accedemos con window.nombre



El alcance de la siguiente funcion solo es para la variable local que le llega por parámetro.

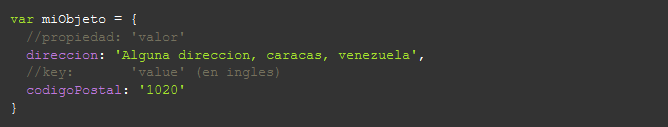


# Objetos.

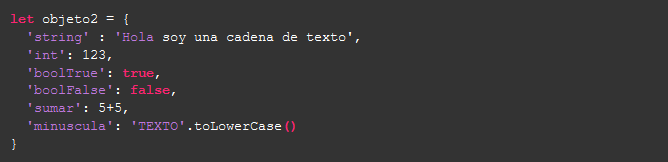
Vamos a empezar a trabajar con objetos, veremos cómo declararlos, cuáles son sus ventajas, cómo asignarles atributos y cómo trabajar con ellos dentro de las funciones.

Los objetos se definen con {} ***(llaves)*** para asignarlo a una variable se hace de esta manera var ejemplo = {}

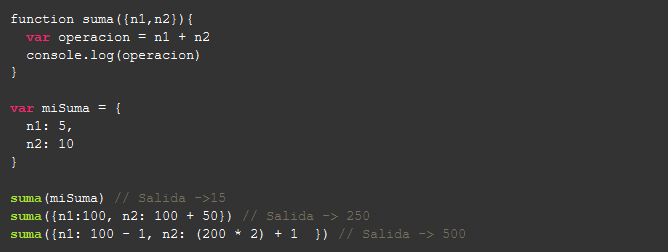
Los objetos se componen por propiedades y valores, se utiliza ***’,'*** para separar cada propiedad.



Los objectos pueden recibir como valor en sus propiedades, cadenas de texto (string), valores numéricos (int) y booleanos. También puedes realizar operaciones dentro de los valores. Ejemplo:



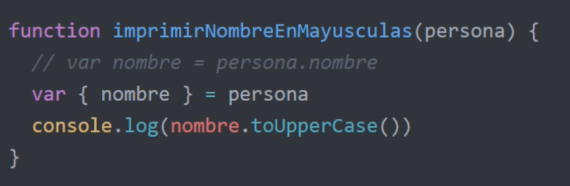
Los objetos se pueden enviar por para parámetro a alguna función de esta manera:



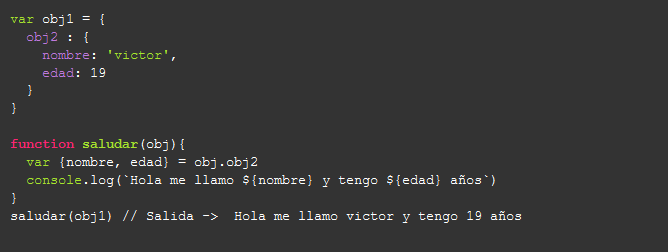
# Desestructurar objetos.

En esta clase aprenderemos otra forma de acceder a los atributos de los objetos que es la desestructuración de los mismos.  
Para no duplicar las variables introducir el nombre de la variable como parámetro de la segunda variable.

Ej var{nombre} = persona



***T***ambién se puede desestructurar un objeto de un objeto para obtener sus valores ejemplo:



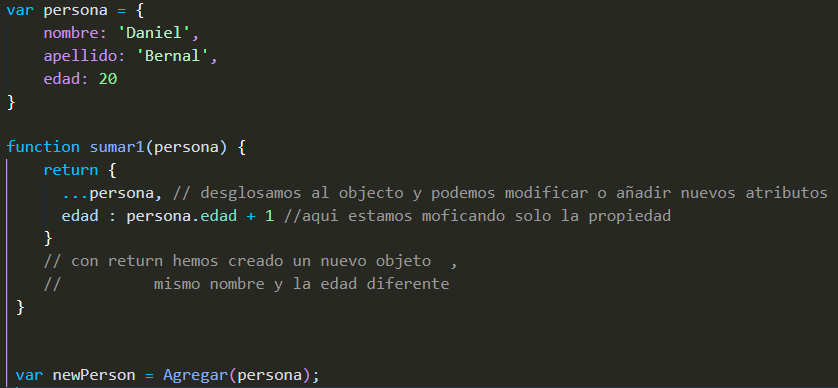
# Parámetros como referencia o como valor.

Javascript se comporta de manera distinta cuando le pasamos un objeto como parámetro.

Cuando los objetos se pasan como una referencia, estos se modifican fuera de la función. Para solucionar esto se puede crear un objeto diferente. Esto lo podemos hacer colocando tres puntos antes del nombre.

Ej …persona.

A los tres puntos “…” se llama operador de propagación (**Spread operator**), es una nueva implementación en **ES6**. Básicamente lo que hace es crear un duplicado de nuestro objeto para luego modificarlo según lo que necesitemos.



**JS se comporta diferente con los objetos** , los objetos que pasamos por **parámetro** a una función, \*\* se pasan por referencia\*\* , esto quiere decir que si modificamos al objeto dentro del cuerpo de la función su valor se verá modificado fuera de ésta, produciéndose un **Side Effect** .

**Comparaciones en JavaScript**

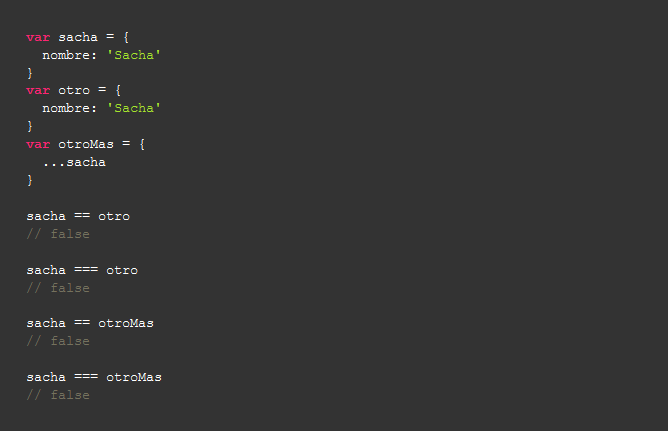
Existen varias maneras de comparar variables u objetos dentro de javascript. En el primer ejemplo le asignamos a X un valor numérico y a Y un string. Para poder compararlos debemos agregar dos signos de igual (==). Esto los convierte al mismo tipo de valor y permite que se puedan comparar.

Cuando realizamos operaciones es recomendable usar tres símbolos de igual (===). Esto permite que JavasScript no iguale las variables que son de distinto tipo. Te recomendamos que uses el triple igual siempre que estés comparando variables.

Existen cinco tipos de datos que son primitivos:

1. Boolean
2. Null
3. Undefined
4. Number
5. String

Al comparar objetos JS tiene en cuenta también el nombre del objeto, por lo tanto, se remite a comparar el nombre de las variables además del valor de los atributos.  
Con objetos literales desglosados (otroMas en este caso), pasa lo mismo y la comparación da false ya que lo que se genera es un nuevo objeto a partir del desglosado.



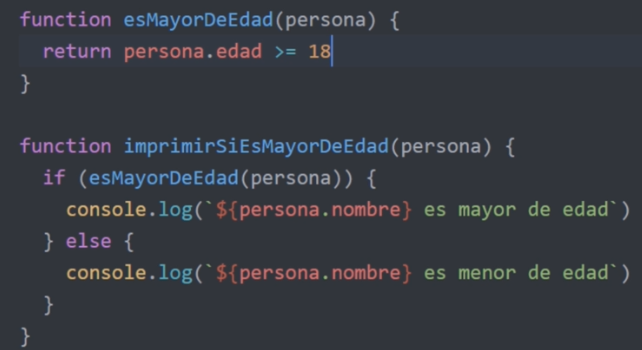
# Funciones que retornan valores

En esta clase seguiremos trabajando con condicionales para desglosar las funciones en funciones más pequeñas que retornen un valor.

Debemos de tener en cuenta que el número 18 está incluido dentro del rango de edad. Para ello utilizamos los símbolos >=.

Return detiene la ejecución de una función y devuelve el valor de esa función.

Las variables definidas con const se comportan como las variables, excepto que no pueden ser reasignadas. Las constantes pueden ser declaradas en mayúsculas o minúsculas. Pero por convención, para distinguirlas del resto de variables, se escribe todo en mayúsculas.

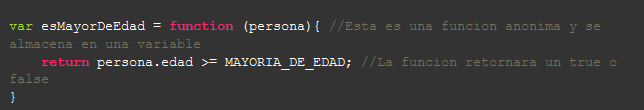


# Arrow functions

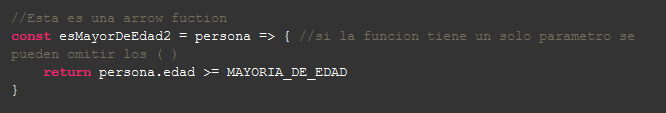
En esta clase aprenderemos a utilizar Arrow Functions que permiten una nomenclatura más corta para escribir expresiones de funciones. Este tipo de funciones deben definirse antes de ser utilizadas.

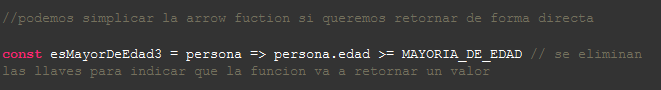
Al escribir las Arrow Functions no es necesario escribir la palabra function, la palabra return, ni las llaves.

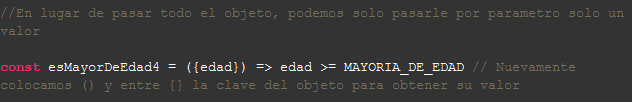
***Anonymous Function***



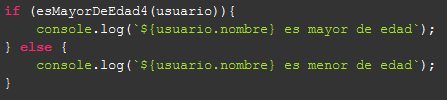
***Arrow Function***







***Ahora para llamarla.***



# Estructuras repetitivas: for

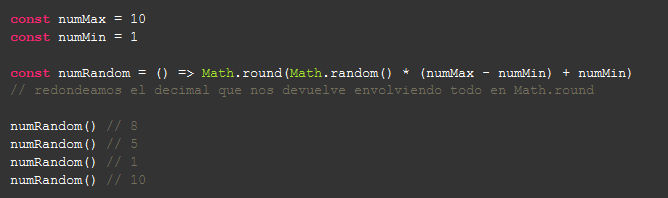
En esta clase estudiaremos una de las estructuras básicas de control. El bucle for, se utiliza para repetir una o más instrucciones un determinado número de veces.

Para escribir un bucle for se coloca la palabra for seguida de paréntesis y llaves.  
Ej. for( ){ }. Dentro de los paréntesis irán las condiciones para ejecutar el bucle, y dentro las llaves irán las instrucciones que se deben repetir.

En este ejemplo la variable i la utilizamos como contador.

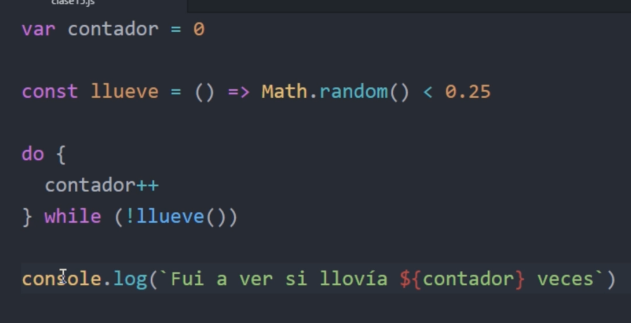


***Aquí como generar un numero aleatorio.***



# Estructuras repetitivas: do-while

Otra estructura repetitiva es el do-while. A diferencia de la instrucción while, un bucle do…while se ejecuta una vez antes de que se evalúe la expresión condicional.



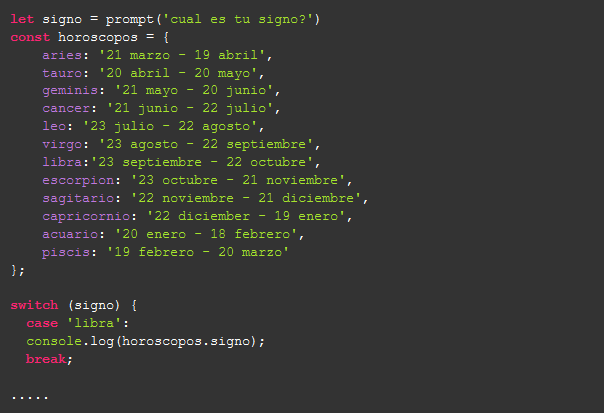


# Condicional múltiple: switch

Una última estructura de control se llama switch. Switch se utiliza para realizar diferentes acciones basadas en múltiples condiciones.

Prompt, muestra un cuadro de mensaje que le pide al usuario que ingrese alguna información.

Break, sirve para que el browser se salte un bucle.



# Introducción a Arrays

Los Arrays son estructuras que nos permiten organizar elementos dentro de una colección. Estos elementos pueden ser números, strings, booleanos, objetos, etc.

***var personas = []***

Incluso podemos mezclar diferentes tipos de datos.

Si queremos acceder al 1er elemento de un array , escribimos el nombre del array seguido de los corchetes y el índice 0:

personas[0]

Si queremos acceder a los atributos de alguno de los elementos después del índice del elemento se agrega un “.” seguido del atributo:

personas[0].altura

Otra forma de acceder a los atributos es utilizando corchetes y el atributo pasarlo como un string:

personas[0].[‘altura’]

Se debe tomar en cuenta que estamos utilizando objetos, no colecciones, pero podemos acceder a los atributos.

**NOTA**: Los Arrays como los strings tienen la propiedad length, indicándonos cuantos elementos contiene.

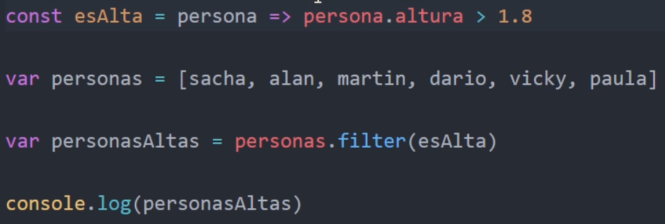
# Filtrar un array

En esta clase aprenderemos como filtrar los Arrays. Para filtrar siempre necesitamos establecer una condición. En este ejemplo nuestra condición es que la estatura de las personas sea mayor de 1.80mts.

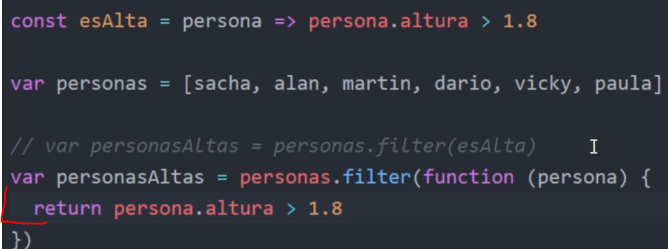
El método ***filter ( )*** crea una nueva matriz con todos los elementos que pasan la prueba implementada por la función proporcionada.

Recuerda que si no hay elementos que pasen la prueba, filter devuelve un array vacío.

**Reto**: escribe el filtrado de personas bajas.

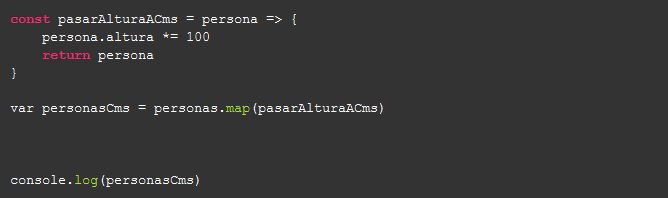


**Otra forma de hacerlo:**

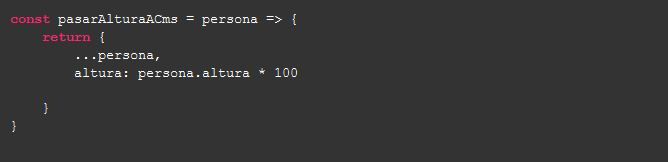


# Transformar un array

En esta clase veremos cómo transformar un array. El método ***map()*** itera sobre los elementos de un array en el orden de inserción y devuelve un array nuevo con los elementos modificados.



Al hacer esto map nos pasará la altura de las personas convertida a centímetros, y nos modificará nuestro array principal personas, así que para evitarlo hacemos uso de [Spread-operator](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Operadores/Spread_operator):



**Las diferencia entre filter() y map() radica en sus callbacks, o sea en lo que retornan.**

*filter()*: retornará 1 subconjunto de un Array.

*map()*: retornará un nuevo Array basándose en otro de referencia.



# Reducir un array a un valor

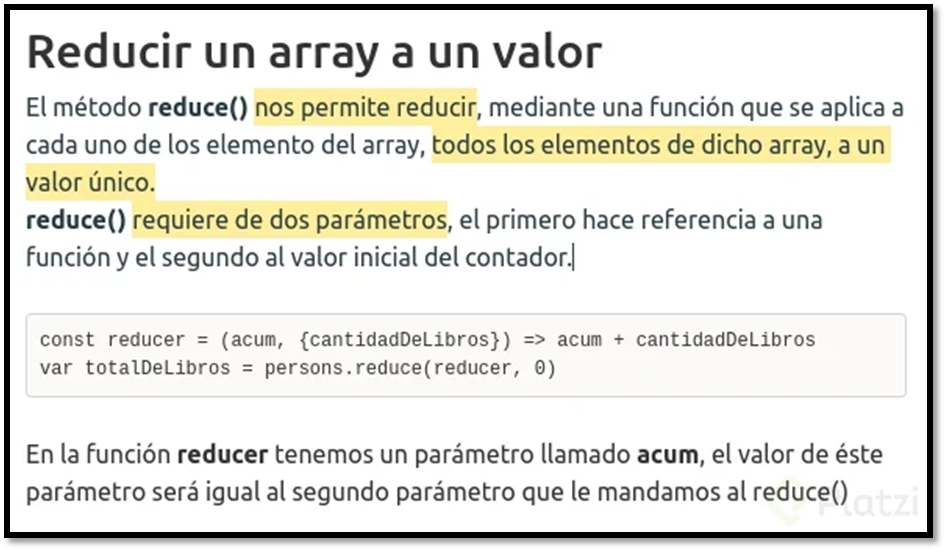
El método ***reduce()*** nos permite reducir, mediante una función que se aplica a cada uno de los elemento del array, todos los elementos de dicho array, a un valor único.

La Función Reduce, reduce un array a un valor único.

Para reducir un array se necesitan 2 cosas**:** Una función y el valor original / valor inicial del acumulador.

La función va pasando el **valor acumulado** por cada elemento del array.





**Cómo funcionan las clases en JavaScript**

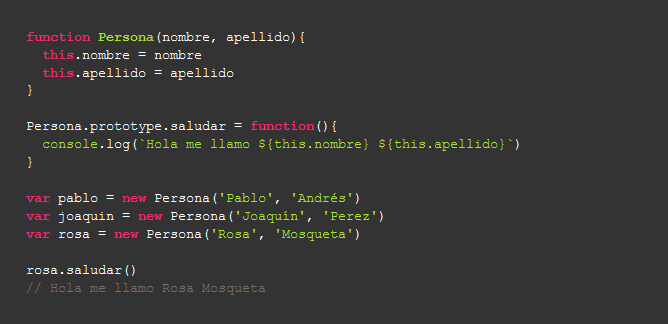
Las clases son funciones cuya sintaxis tiene dos componentes:

* expresiones
* declaraciones

En esta clase veremos el uso de this. Dentro de una función, el valor de this depende de cómo es llamada ésta.

**Reto:** agrega el atributo altura y la función soyAlto.

* Hablar de objetos en javascript es hablar de prototipos.
* *this* hace referencia al nuevo objeto que se acaba de crear.
* la función que define al prototipo retorna implícitamente *this*, es decir retorna el nuevo objeto que se creó.
* La palabra reservada *new* se utiliza para crear un nuevo objeto con el prototipo indicado.



# Modificando un prototipo

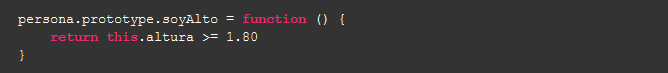
En esta clase veremos cómo se modifican las clases de herencias. JavaScript funciona con una estructura orientada a objetos y cada objeto tiene una propiedad privada que mantiene un enlace a otro objeto llamado prototipo.

**El prototipo es un objeto más de JS** , a partir de que lo modifiquemos, éste va a quedar modificado.

Es de mucha importancia **donde colocamos las funciones que va a tener el prototipo** . Estos **se deben de colocar juntos**, ya que con ellos se va creando **el prototipo del objeto**, y deseamos que tenga juntas todas sus funciones.

**Por ello definimos los prototipos arriba, y los utilizamos a lo largo del código.**

Acá en este código agregaremos la funcion de “soyAlto” para las personas de la siguiente manera:



Al probar en la consola, las personas con esta función, nos ira dando true or false de quien es alto y quien no, lo importante acá es saber que recién estamos modificando el prototipo, y en caso de llamar esta función al final del código esto no nos permitirá acceder a la función. Debe ir al principio del código con su prototipo.

**NOTA:** Cuando tenemos un error en JS, éste no se sigue ejecutando, se corta ahí la ejecución del código.

# El contexto de las funciones: quién es this

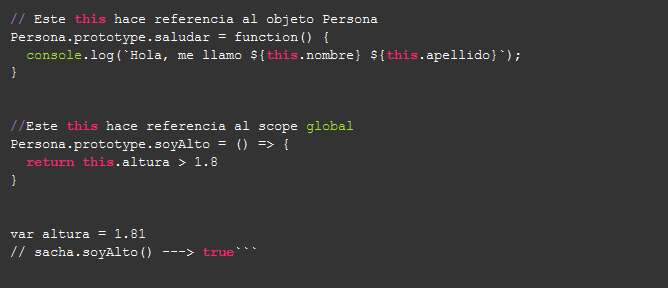
En esta clase explicamos por qué al introducir el arrow function salió un error. El error del contexto de this en JavaScript es uno de los errores más comunes.

Recuerda que dentro de la arrow function, this está haciendo referencia al espacio global, a ***window***.

En las **Arrow functions, se asigna la función** , pero **cambia** el **this dentro de la función** , ya que trata de tomar el this global, es decir, el que está fuera de la Arrow function.

**Para comprobarlo en consola:**

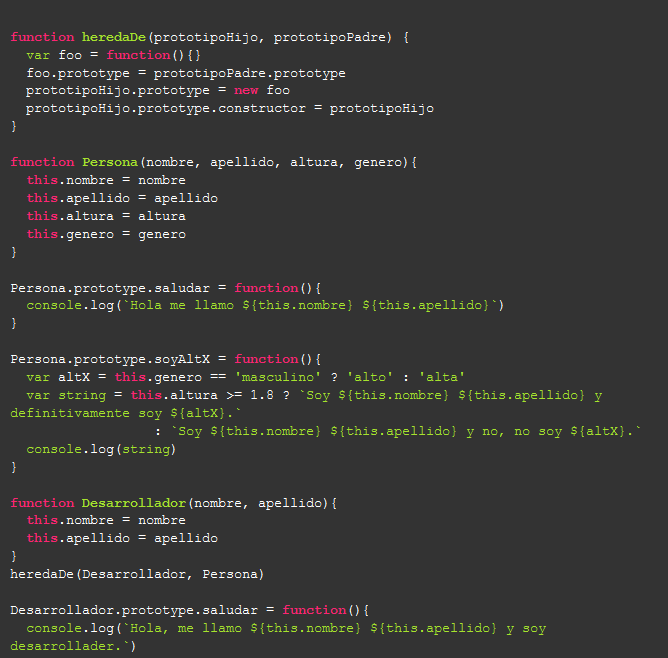
this === window



# La verdad oculta sobre las clases en JavaScript

Los objetos en JavaScript son “contenedores” dinámicos de propiedades. Estos objetos poseen un enlace a un objeto prototipo. Cuando intentamos acceder a la propiedad de un objeto, la propiedad no sólo se busca en el propio objeto sino también en el prototipo del objeto, en el prototipo del prototipo, y así sucesivamente hasta que se encuentre una propiedad que coincida con el nombre o se alcance el final de la cadena de prototipos.

Aquí estaríamos hablando de **herencia prototipal.**

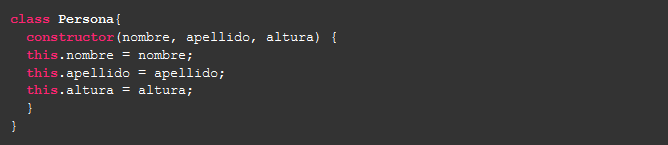


# Clases en JavaScript

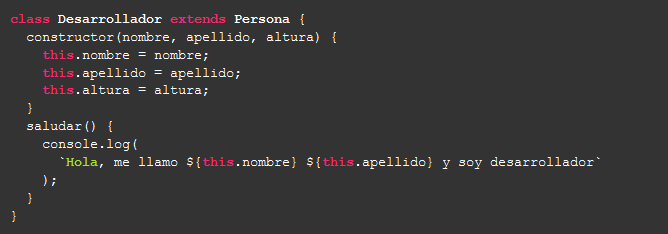
Las clases de JavaScript son introducidas en el ECMAScript 2015 y son una mejora en la sintaxis sobre la herencia basada en prototipos de JavaScript.

La palabra clave ***extends*** se usa en declaraciones de clase o expresiones de clase para crear una clase que es hija de otra clase.

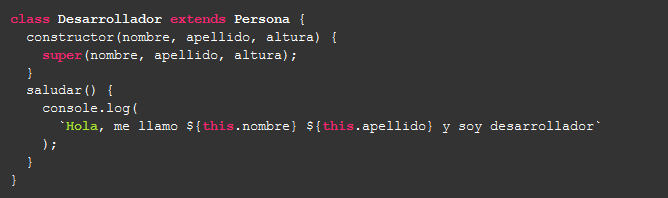
El método constructor es un método especial para crear e inicializar un objeto creado a partir de una clase. Ej:



Luego se crea la clase **Desarrollador** que va a heredar de la principal **Persona**, en este caso se utilizará la palabra **extends**

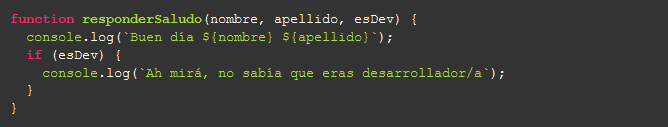


Sin embargo, aquí nos generará un error y es que para heredar de forma correcta se debe utilizar la palabra super quedando así:

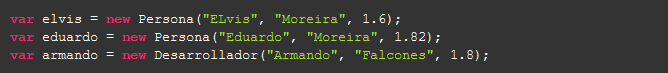


# Funciones como parámetros

En JavaScript, los parámetros de funciones son por defecto undefined. De todos modos, en algunas situaciones puede ser útil colocar un valor por defecto diferente que lo evalúe como verdadero.

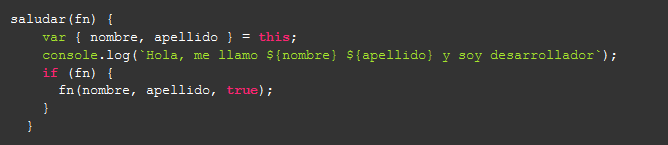


Esta función devuelve el saludo, es el objetivo a conseguir. Luego declaramos los objetos:



Ejecutaremos la función saludar pero a 2 de ellas le vamos a querer que devuelva el saludo



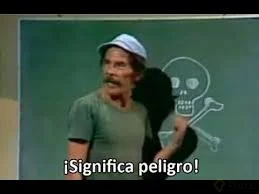


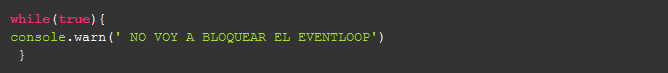
# Cómo funciona el asincronismo en JavaScript

JavaScript sólo puede hacer una cosa a la vez, sin embargo; es capaz de delegar la ejecución de ciertas funciones a otros procesos. Este modelo de concurrencia se llama EventLoop.

JavaScript delega en el navegador ciertas tareas y les asocia funciones que deberán ser ejecutadas al ser completadas. Estas funciones se llaman callbacks, y una vez que el navegador ha regresado con la respuesta, el callback asociado pasa a la cola de tareas para ser ejecutado una vez que JavaScript haya terminado todas las instrucciones que están en la pila de ejecución.

Si se acumulan funciones en la cola de tareas y JavaScript se encuentra ejecutando procesos muy pesados, el EventLoop quedará bloqueado y esas funciones pudieran tardar demasiado en ejecutarse.

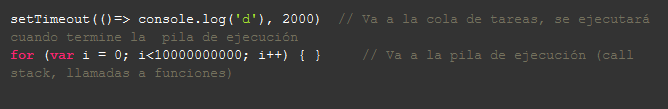




Esto lo que hará es entrar en un bucle infinito que me imprima infinitas veces sin llegar a un fin y llenando nuestra pila de ejecución bloqueando nuestro navegador.

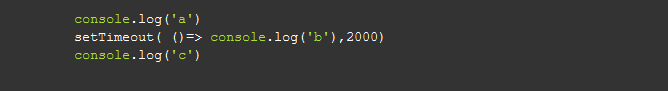
# Cómo funciona el tiempo en JavaScript

En principio, cualquier tarea que se haya delegado al navegador a través de un callback, deberá esperar hasta que todas las instrucciones del programa principal se hayan ejecutado. Por esta razón el tiempo de espera definido en funciones como setTimeout, no garantizan que el callback se ejecute en ese tiempo exactamente, sino en cualquier momento a partir de allí, sólo cuando la cola de tareas se haya vaciado.





También podría ser expresado de la siguiente manera:



*(2.000 miliSegundos =2 segundos)*

**Operaciones que realizamos de manera asíncrona**:

* Modificación del DOM(Un elemento, un título, una clase de css)
* Mostrar un alert
* Petición de datos a una API externa
* Request externo

# ¿Qué pasó con swapi.co?

## [SWAPI.co](http://SWAPI.co) ha desaparecido

Desafortunadamente, [swapi.co](http://swapi.co) ya no se mantiene y el servicio está actualmente inactivo. El autor del proyecto, Paul Hallett, quien creó y le dio mantenimiento ha desactivado desde hace tiempo esta API que muchos utilizamos en nuestros proyectos de JavaScript para aprender a integrar un backend a un frontend.

## SWAPI.dev, una nueva solución

Juriy Bura, junto a otros desarrolladores, han publicado una versión idéntica a la API utilizada por [Swapi.io](http://Swapi.io), la cual está disponible desde el dominio **swapi.dev**. Por lo tanto, de ahora en adelante, para continuar el curso sin problemas solo debes reemplazar [swapi.io](http://swapi.io), la URL obliterada (la que ya no funciona), por la nueva versión que sí está disponible: **swapi.dev**

Este proyecto es mantenido por la comunidad y gracias a la filosofía del código libre es posible tener una nueva versión de la API para nuestros proyectos.

## **Crea tu propia API de Star Wars**

**¿Quieres tener tu propia versión de SWAPI?**

El código fuente del proyecto está disponible en GitHub, lo que te permite crear tu propia versión con solo realizar un Fork y subirlo a un servidor o consumirla en tu localhost.  
Este es el repositorio del proyecto original: <https://github.com/phalt/swapi>.

¡Compártenos en los comentarios en link a tu repositorio fork de la API de Star Wars y qué cambios hiciste o planeas hacer en tu versión de este proyecto!

# Callbacks

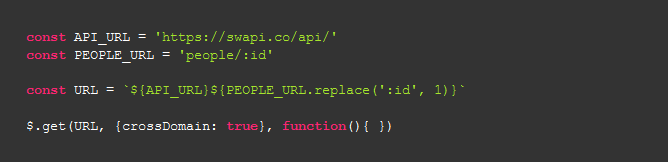
# Un callback es una función que se ejecutara solo cuando otra la llame, esta llamada solo ocurre cuando la primera logra ejecutase.

# 

Utilizaremos una librería externa, *jQuery*, con un fin específico que es el de realizar un request y obtener datos de una API externa. Utilizamos la versión CDN de JQuery.  
*Nota: un CDN es un Content Delivery Network. Un servidor en el planeta que nos va a conectar con la versión de jQuery más cercana a nuestra locación.*

La API que usaremos es la de Star Wars.  
Implementamos la llamada a la librería jquery.minified en el html antes de llamar nuestro archivo de funciones. Este lo usaremos para hacer requests a la api de ‘[swapi.co](http://swapi.co)’.

Realizamos el request. En este caso:



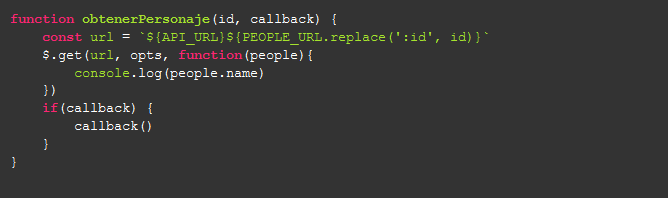
* $.get es el método de jQuery para realizar el callback. (Referencia <http://api.jquery.com/jQuery.get/>)

# Manejando el Orden y el Asincronismo en JavaScript

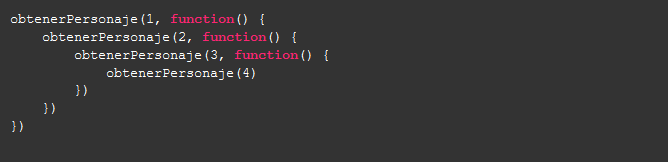
Una manera de asegurar que se respete la secuencia en que hemos realizado múltiples tareas es utilizando callbacks, con lo que se ejecutará luego, en cada llamada. Lo importante es que el llamado al callback se haga a través de una función anónima. Sin embargo, al hacerlo de esta manera generamos una situación poco deseada llamada ***CallbackHell***.

Si queremos que lleguen en orden tenemos que hacer los requests uno después del otro y no en paralelo como los veníamos haciendo.

Para eso usamos el segundo parámetro en la función.



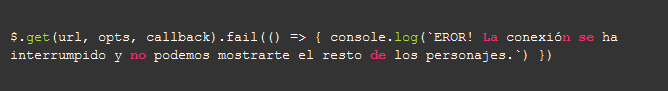
Así podemos invocar la función del callback de la siguiente manera:



Pero esto trae la problemática del anidamiento infinito llamado ***CallbackHell***.

# Manejo de errores con callbacks

Para solucionar el problema de quedarnos sin conexión, u otro error similar, en medio de una sucesión de callbacks utilizamos el método ***fail().***



DATOS ADICIONALES DE LA CLASE:

* **Caché**: Son archivos **(vídeos,audios,imágenes,textos,etc)** pertenecientes a cierto ordenador **(que no es el tuyo)** de una página web pero que una parte de dichos archivos se guardan en tu ordenador con el objetivo de que cuando vuelvas otra vez a dicha página web **tengas la sensación que la pagina haya cargado mucho más rápido**.  
  La ventaja del caché es la velocidad con la que carga la página web y su desventaja es que puede ocupar bastante espacio en tu memoria del CPU.
* **Network**: También llamado **red**, es básicamente la comunicación entre ordenadores a través de cables,señales,ondas,etc con el objetivo de transmitir datos , recursos , servicios, etc entre sí. Por ejemplo , la mayoría de las personas , en la actualidad, utilizamos lo que es **Internet** que también es una red que une computadores, cuyo intercambio de información lo hacemos a través de páginas web o ciertos programas.

**Promesas**

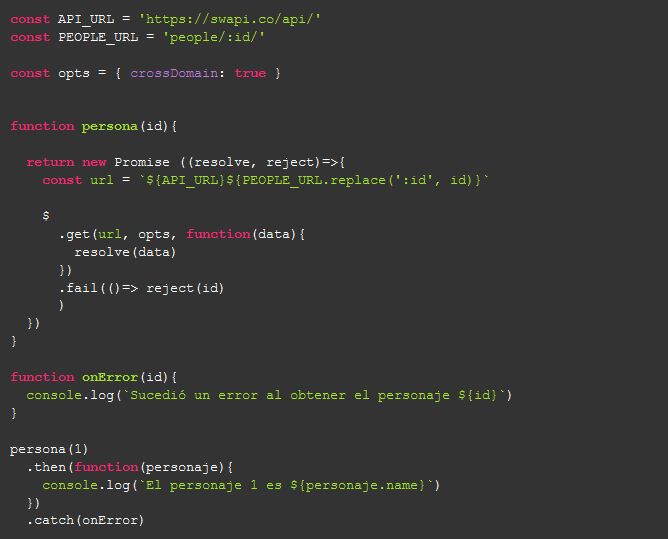
En esta clase veremos las promesas, que son valores que aún no conocemos. Las promesas tienen tres estados:

* pending
* fullfilled
* rejected

Las promesas se invocan de la siguiente forma:



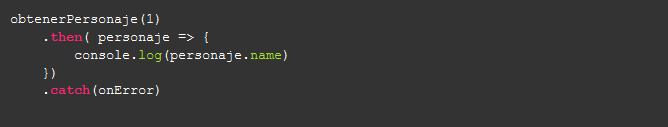
Ejemplo:



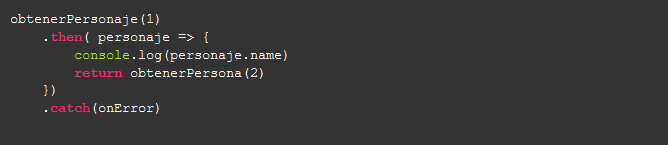
# Promesas Encadenadas

A diferencia de los callbacks en el CallbackHell, que terminan estando anidados unos dentro de otros, cuando se usan Promesas la ejecución de las llamadas no se hacen de manera anidada sino de manera encadenada, al mismo nivel una debajo de la otra, lo que hace que el código sea mucho más legible y mantenible.

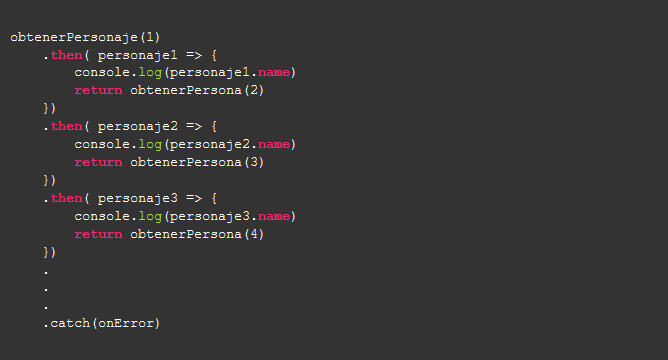
Encadenar promesas es mucho más limpio que con el método anterior.  
Primero escribimos la invocación de la promesa con un arrow function:



Al resolver esta promesa vamos a retornar otra promesa invocando dentro del .*then* nuevamente la función *obtenerPersona()* con el id del siguiente personaje:



Y para obtener los valores de esta promesa encadenamos otro *.then* y copiamos la función parámetro cambiando el valor del id.



# Múltiples promesas en paralelo.

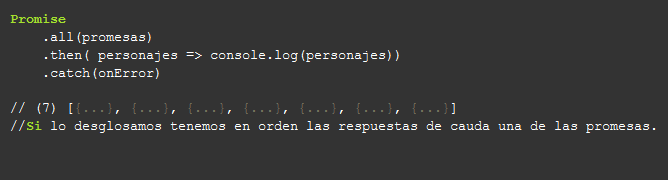
Con promises podemos hacer los requests en paralelo sin alterar el orden de los objetos, lo que mejoraría mucho nuestro código y performance.

Generamos un array con los ids de los personajes que queremos obtener. Y a partir de este vamos a generar otro array con múltiples promesas, donde cada elemento sea una promesa, la promesa de obtener un personaje con su id.  
Con el método *map()* vamos a recorrer el array *ids* y por cada elemento de este vamos a generar uno nuevo que va a ser una *promesa*.  
Estas promesas las guardamos en una variable ‘*promesas*’. A partir de cada objeto del array *ids* (de cada id) obtenemos una nueva promesa con la función obtenerPersonaje(id).

# 

*¿Cómo obtenemos los valores de estas promesas cuando se resuelvan?*

Para esto podemos llamar a un método que tiene la clase de promesas llamado *'Promise.all()'*  
A este le pasamos el *array ‘promesas’*, le encadenamos el *.then()* que nos entrega los objetos y después encadenamos el *.catch()* que se va a ejecutar si cualquiera de las promesas que tenemos en el array falla.



# Async-await: lo último en asincronismo

Async-await es la manera más simple y clara de realizar tareas asíncronas. Await detiene la ejecución del programa hasta que todas las promesas sean resueltas. Para poder utilizar esta forma, hay que colocar async antes de la definición de la función, y encerrar el llamado a Promises.all() dentro de un bloque try … catch.



Para visualizar mejor organizados los arrays:

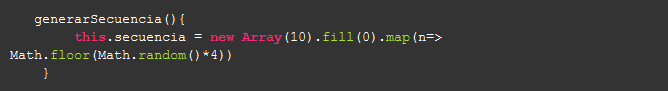


# Generando una secuencia de números

Para generar la secuencia del juego usaremos un array con números aleatorios, que representarán el color del botón que se iluminará cada vez. Usamos new Array() para crear el arreglo de manera dinámica, y llamamos al método ***fill()*** para rellenar ese array con ceros y poder luego iterar sobre éste con ***map()***



Vamos a clases y creamos en el constructor “this.generarSecuencia” para llamar la función;



Como podemos observar en esta función vamos a generar la secuencia y lo haremos creando un array, de una manera totalmente nueva cual no habíamos implementado antes, es muy importante acá usar la función “fill” para darle un valor a los arrays porque sino no va a funcionar nuestra función “map” para poder trabajar con los arrays y crear la secuencia, para ellos usamos una arrow function dentro de map, y usaremos el módulo de matemáticas universal de JavaScript y usaremos la función “floor” de este que lo que hará es redondearnos hacia abajo el número que tengamos, esto lo haremos porque al usar math.random() como lo haremos a continuación, este crea un numero aleatorio entre 0 y 1 y debido a que lo multiplicamos x4 nos dará varios números entre 0 y 4 por lo que los queremos redondeados y para ello usamos esta función.

*Complementos.*

[*https://www.freecodecamp.org/news/https-medium-com-gladchinda-hacks-for-creating-javascript-arrays-a1b80cb372b/*](https://www.freecodecamp.org/news/https-medium-com-gladchinda-hacks-for-creating-javascript-arrays-a1b80cb372b/)

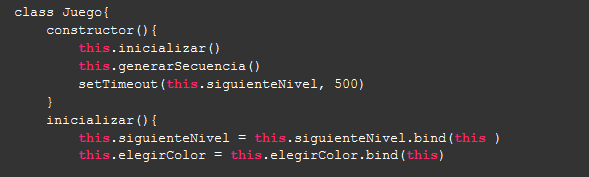
[*https://dmitripavlutin.com/when-not-to-use-arrow-functions-in-javascript/*](https://dmitripavlutin.com/when-not-to-use-arrow-functions-in-javascript/)

# Agregando la verificación del color elegido

Para agregar atributos al objeto principal en el que está nuestro código, basta con usar this, haciendo referencia al contexto de la clase, y agregar los atributos con un punto: this.atributo = valor

La verificación del color elegido la haremos creando y removiendo los eventos del click al pasar el juego a cada nuevo nivel.

Nota: es conveniente colocar desde el constructor del objeto la función bind, para evitar que en las futuras iteraciones la referencia this se dirija a Window en lugar de al objeto (en este caso el juego).



# Diferencias entre var, let y const

“**var**” es la manera más **antigua** de declarar variables. No es muy estricta en cuanto al alcance, ya que, al declarar variables de esta forma, dichas variables podrán ser **accedidas**, e incluso modificadas, **tanto dentro como fuera** de los bloques internos en una función.

Con “**let**” por otra parte, el alcance se reduce al **bloque** (las llaves) en el cual la variable fue declarada. **Fuera** de este bloque la variable **no existe**. Una vez declarada la variable con let, no se puede volver a declarar con en ninguna otra parte de la función.

“**const**” al igual que “**let**” se define en el contexto o alcance de un bloque, a diferencia de let y var, las variables definidas como constantes (const), ya no podrán ser modificadas ni declaradas nuevamente, en ninguna otra parte de la función o el contexto en el que ya existen.

La recomendación es reducir siempre al mínimo el alcance de nuestras variables, por lo que se debe usar let en lugar de var mientras sea posible.

**Scopes**

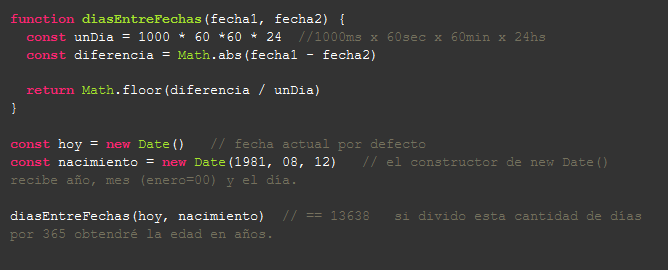
* **scope local**: es el bloque en que se está ejecutando una instrucción.
* **scope de una función**: es el bloque que está delimitado por una función.
* **scope global**: es el que está accesible sin importar en qué bloque se esté.  
  Todos los scope pueden tener otros scopes dentro.

**Declaraciones**

* **var**: se declara en el scope de la función con valor undefined, se inicializa al momento de ejecutarse cualquier línea donde se haya definido (dentro del scope de la función), si no se encuentra dentro de ninguna función entonces se declara en el scope global, puede ser modificada dentro del scope de la función/global desde el inicio del mismo, la primera inicialización debe ser con var, puede ser inicializada con var varias veces, una por cada scope local que esté contenido dentro del scope de la función/global.
* **let**: se declara dentro del scope local al momento de la ejecución de la línea, deja de estar disponible cuando termina el scope local, se puede modificar dentro del scope local, a partir de la línea en la que se declara.
* **const**: se declara dentro del scope local al momento de la ejecución de la línea, deja de estar disponible cuando termina el scope local, no puede ser modificada una vez declarada.

# ¿Hace cuántos días naciste?

Con variables de tipo Date, se pueden realizar operaciones de suma y resta similares a las que se realizan con números. El resultado que se obtiene está en milisegundos, por lo que luego hay que hacer algunas operaciones adicionales para llevarlos a días, meses o años según queramos. También aplica para Horas, Minutos, Segundos y Milisegundos.



# Funciones recursivas

La recursividad es un concepto muy importante en cualquier lenguaje de programación. Una función recursiva es básicamente aquella que se llama (o se ejecuta) a sí misma de forma controlada, hasta que sucede una condición base.

Para realizar recursividad necesitamos 2 cosa; 1 caso base y 1 caso recursivo.  
Usaremos una división para mostrar las funciones recursivas.

Algoritmo de nuestro ejercicio:

13 /*4*\_

13 - 4 = 9 1 => *caso recursivo*  
9 - 4 = 5 1 => *caso recursivo*  
5 - 4 = 1 1 => *caso recursivo*  
1 - 4 = -3 0 => *caso base*

La función:



**¿Cuándo hace falta poner el punto y coma al final de la línea?**

El punto y coma es opcional en JavaScript, excepto en algunos casos:

* Cuando usamos varias instrucciones en una misma línea
* Al comenzar la próxima línea con un array
* Al comenzar la próxima línea con un template string

# Memorización: ahorrando cómputo

La memorización es una técnica de programación que nos permite ahorrar cómputo o procesamiento en JavaScript, al ir almacenando el resultado invariable de una función para que no sea necesario volver a ejecutar todas las instrucciones de nuevo, cuando se vuelva a llamar con los mismos parámetros. Es similar a usar memoria cache.

Este proceso nos va a permitir ahorrar procesamiento, ahorrar cómputo, guardando ciertos resultados de algunas cuentas.

Usaremos otro ejemplo de recursividad:

Factoriales:

!6 = 6\* 5 \* 4 \* 3 \* 2 \* 1 = 720

!12 = 12 \* 11 \* 10 \* 9 \* … \* 1 == 12 \* 11 \* 10 \* 9 \* 8 \* 7 \* !6

Cómo guardamos los resultados para no tener que volver a realizar cuentas ya hechas.



Ahora guardamos en una cache los resultados de las operaciones ya hechas.



# Entiende los closures de JavaScript

# Clousure: Uso de variable como función, con información previa guardada

Un closure, básicamente, es una función que recuerda el estado de las variables al momento de ser invocada, y conserva este estado a través de reiteradas ejecuciones. Un aspecto fundamental de los closures es que son funciones que retornan otras funciones.

**Closures**

Un *closure* es una función que recuerda el estado de las cosas cuando fue creada.  
Una función que devuelve otra función con parámetros invocados en dos veces; primero el de la función ‘*padre*’ y luego el de la función ‘*hijo*’.

Para ejemplificar generemos una función que va a crear saludos. En este caso hagamos un saludo argentino, uno mexicano y otro para colombia.

La función ‘padre’ es generadora o creadora de otras funciones y la función ‘hijo’ es anónima, es la que nos va a devolver el resultado.  
Vamos a llamar esta función para crear constantes.

Entonces ahora podemos llamar a la función nuevamente a través de cada variable constante y pasando el parámetro de la función ‘hijo’ esta vez, el parámetro ‘nombre’ para generar el saludo de la siguiente manera.



**Cambiando de contexto al llamar a una función**

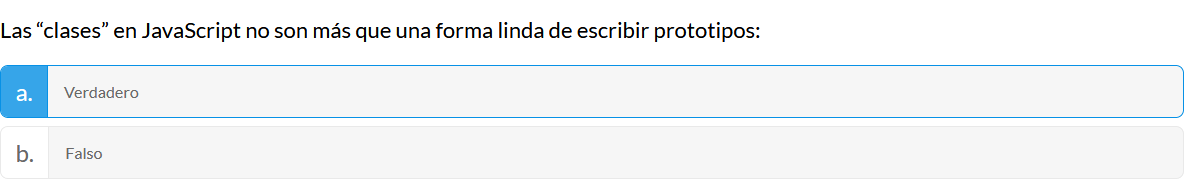
El contexto (o alcance) de una función es por lo general, ***window***. Así que, en ciertos casos cuando intentamos referirnos a ***this*** en alguna parte del código, es posible que tengamos un comportamiento inesperado, porque el contexto quizás no sea el que esperamos.

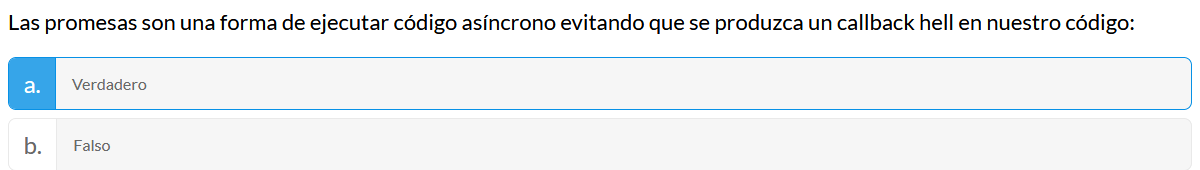
Existen al menos tres maneras de cambiar el contexto de una función.

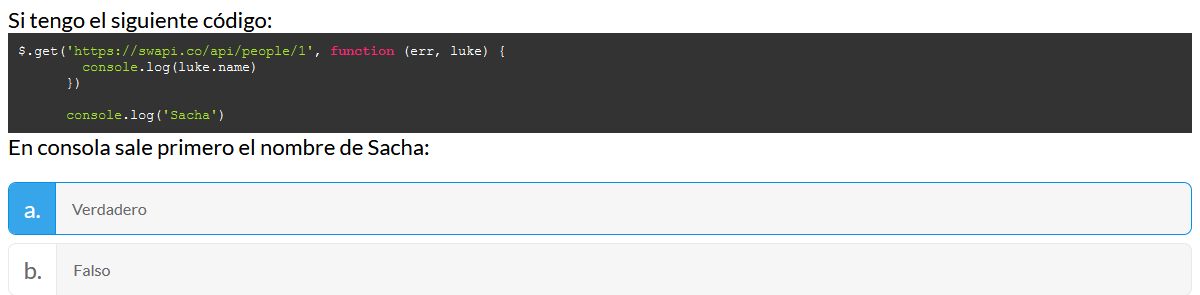
* Usando el método .**bind**, enviamos la referencia a la función sin ejecutarla, pasando el contexto como parámetro.
* Usando el método .**call**, ejecutamos inmediatamente la función con el contexto indicado
* Usando el método .**apply**, es similar a .**call** pero los parámetros adicionales se pasan como un arreglo de valores

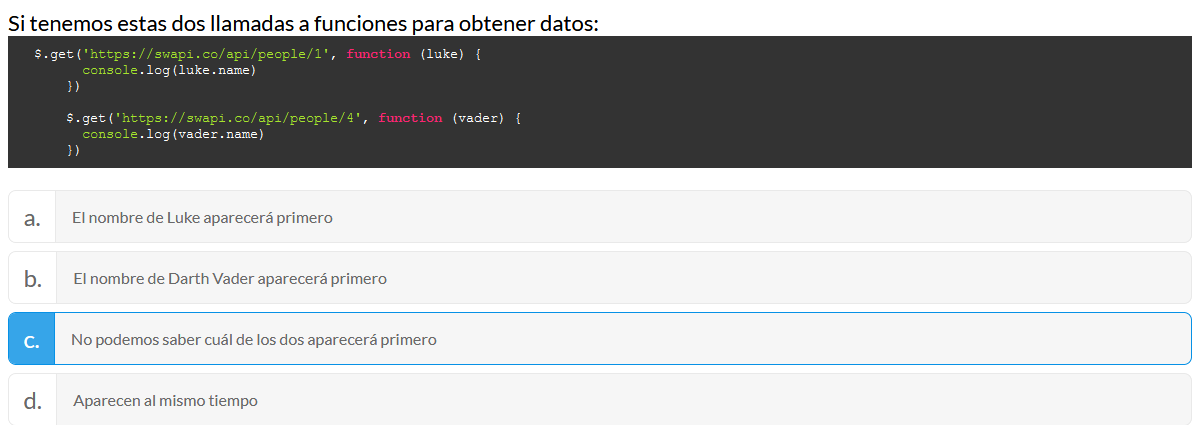


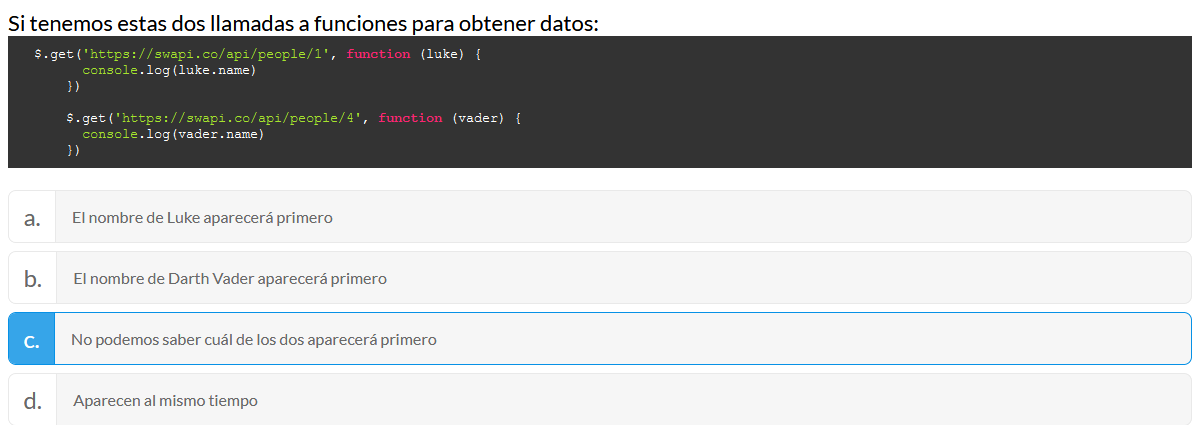
*Importants Questions and Answers.*

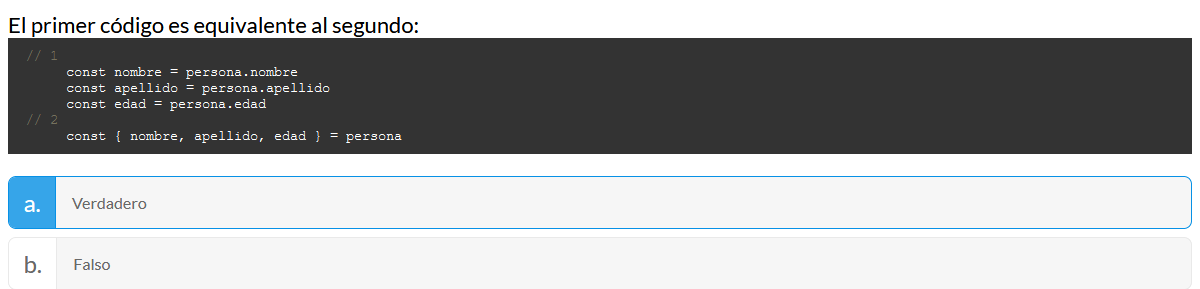


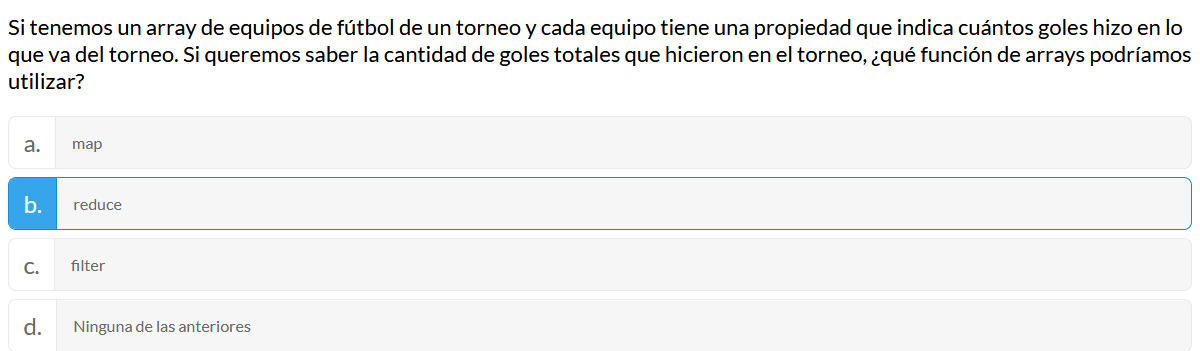
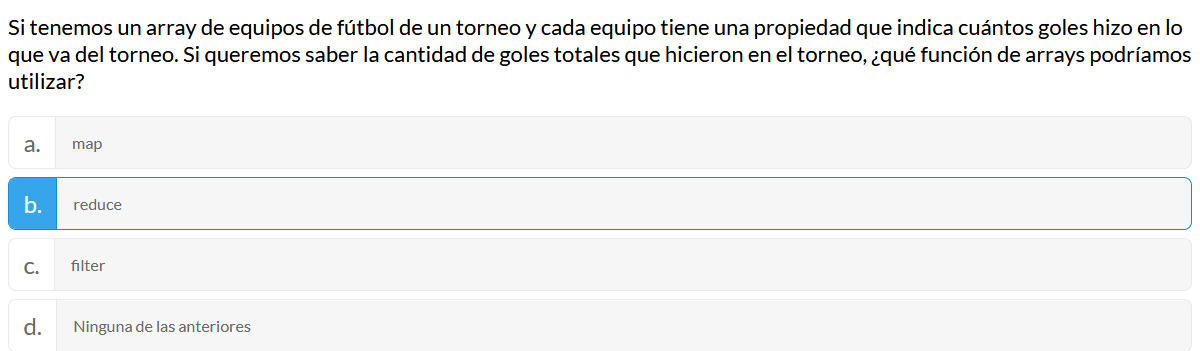


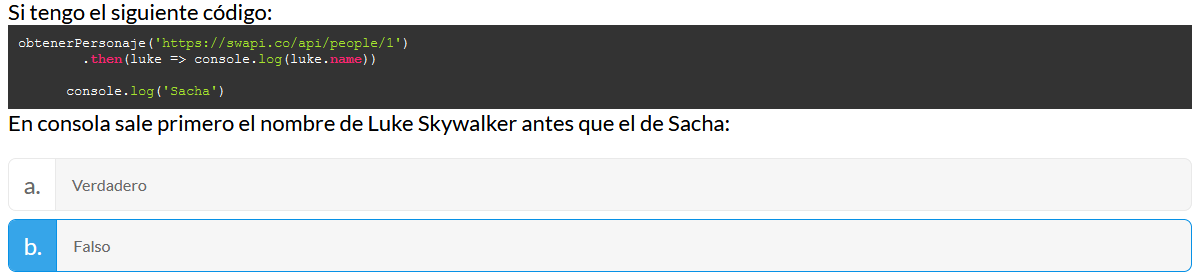


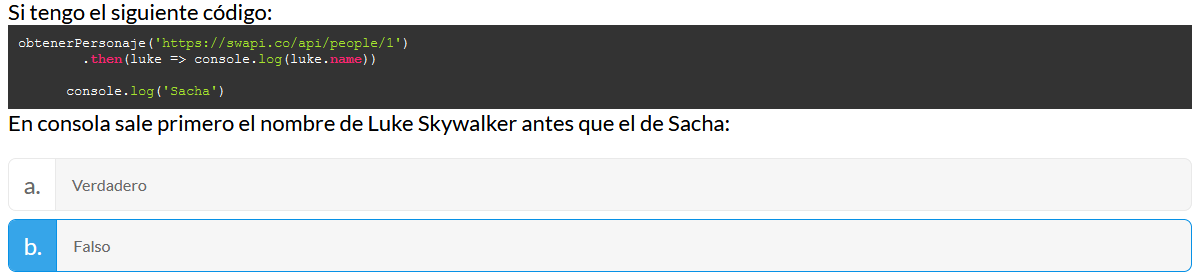


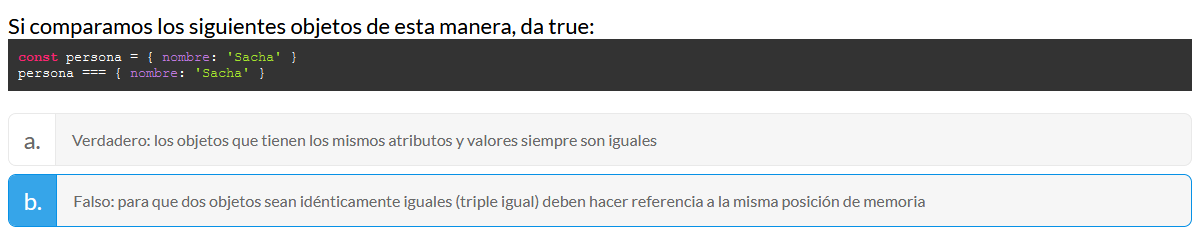
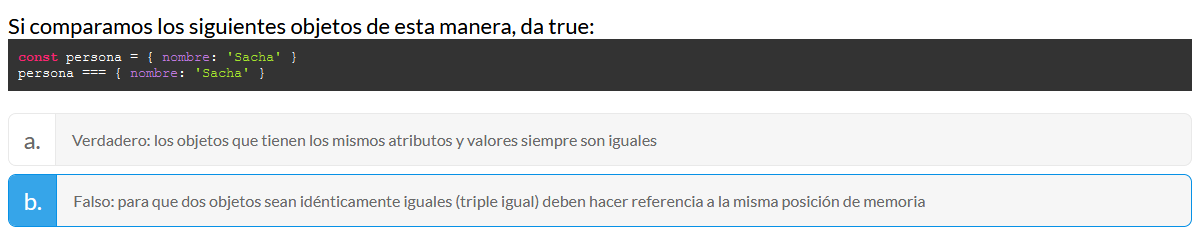


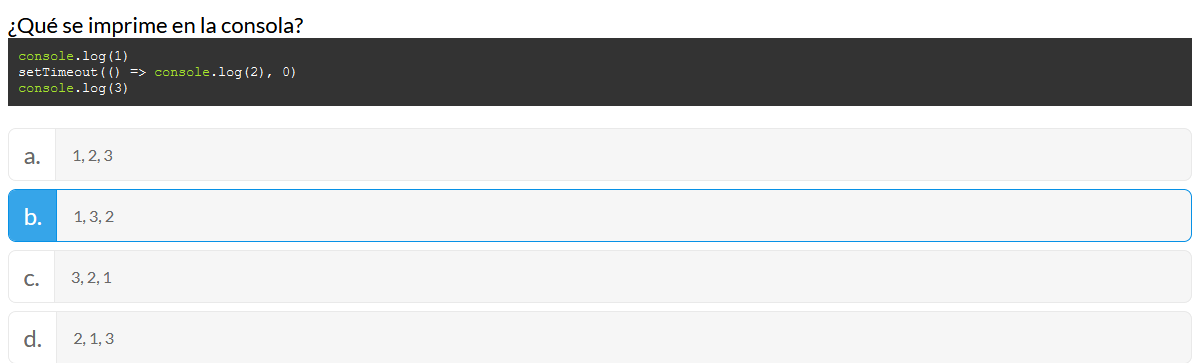
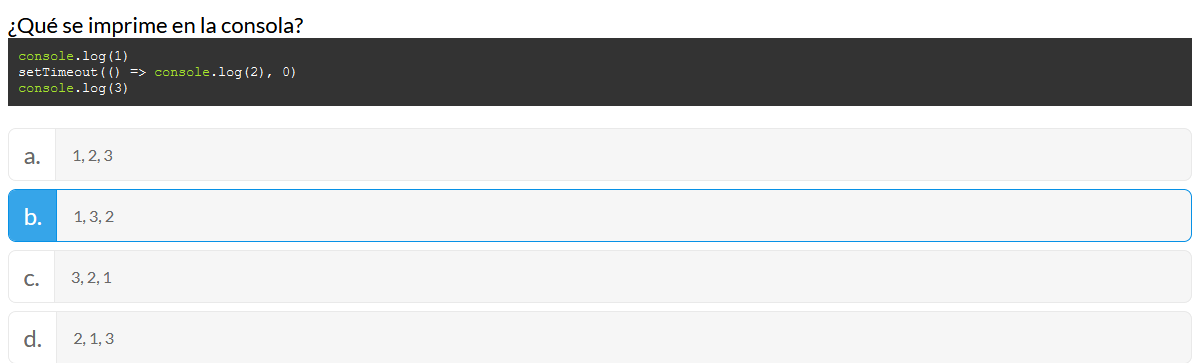


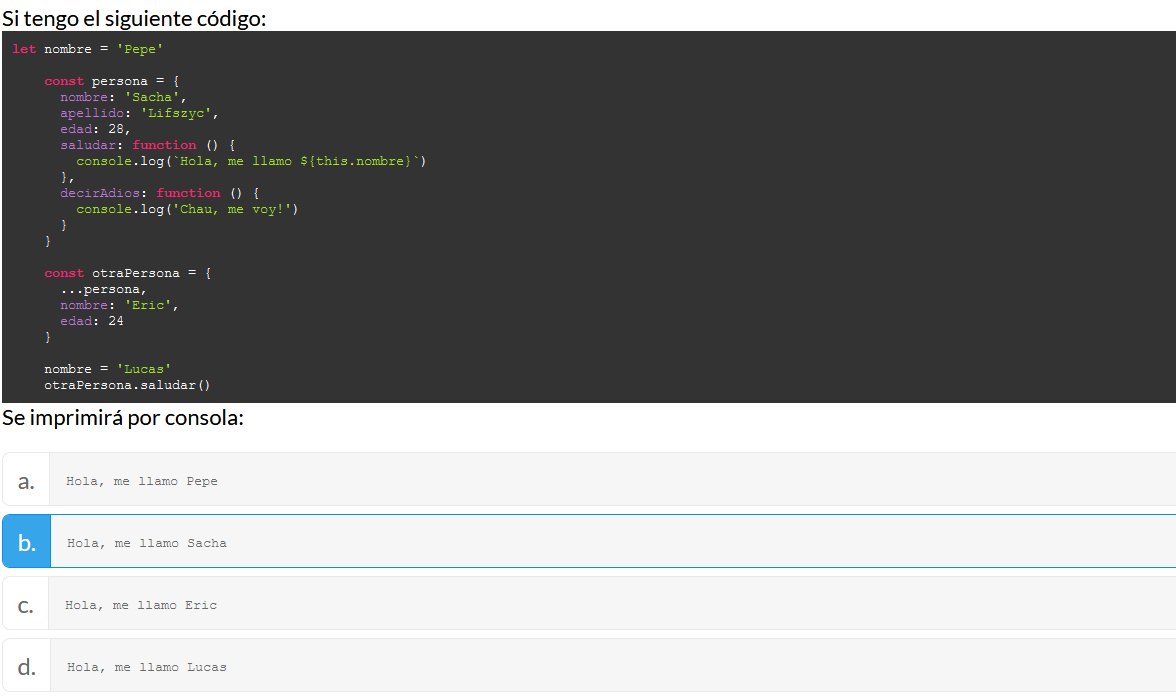


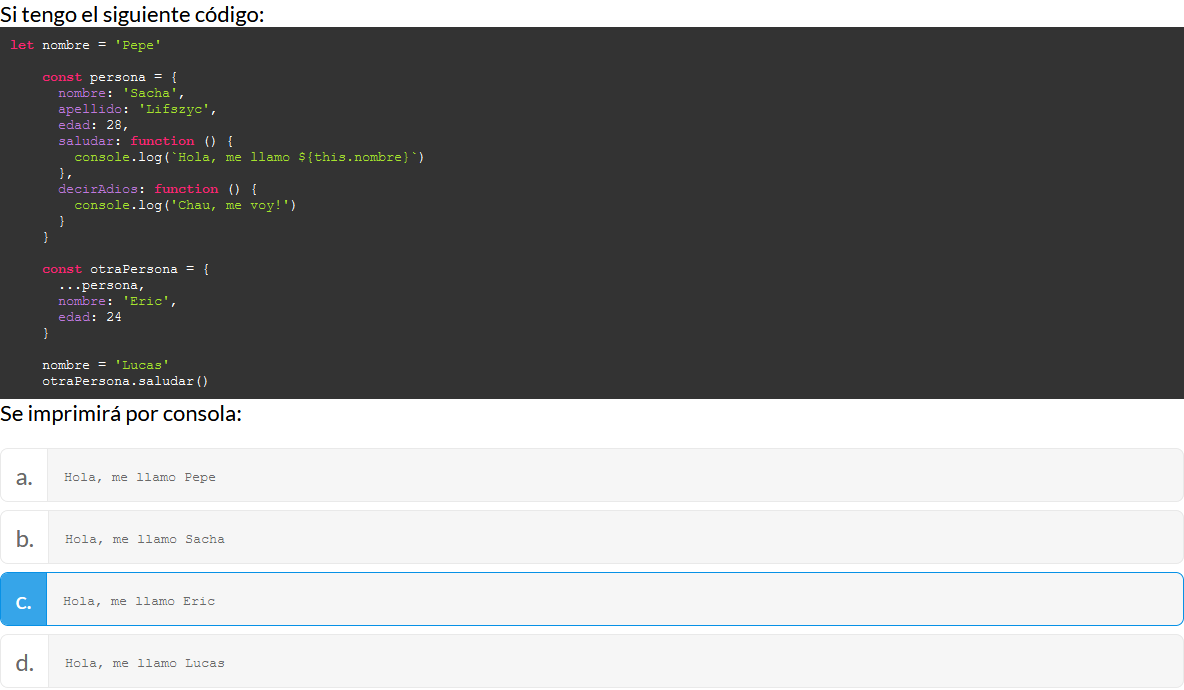






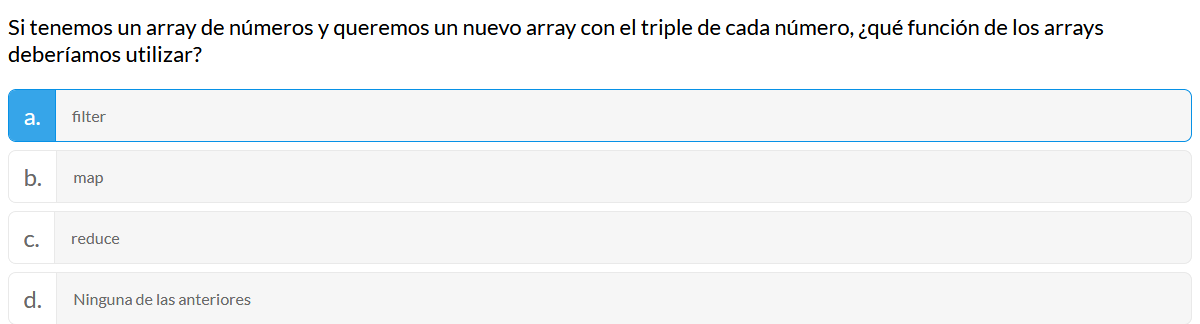


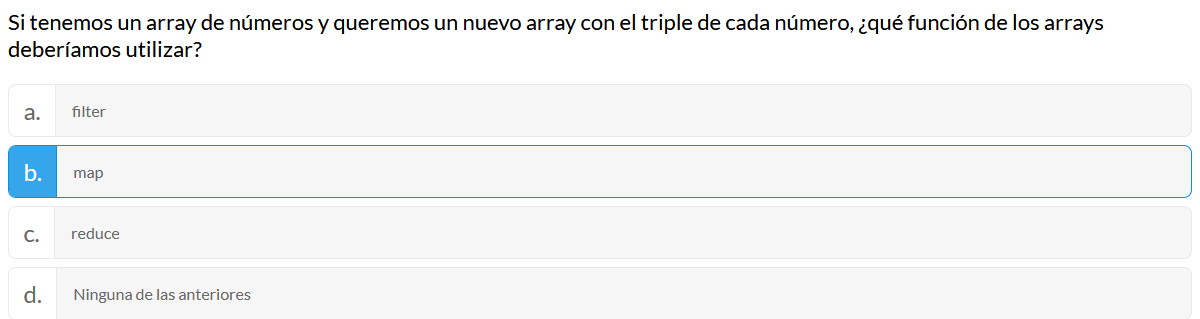


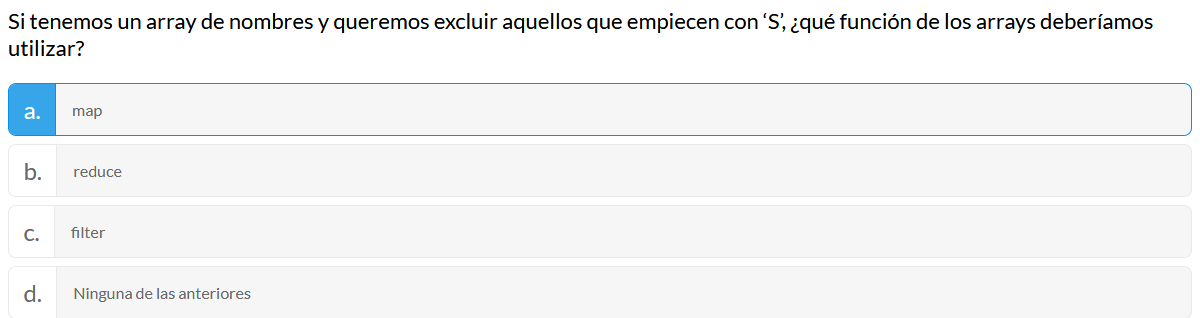


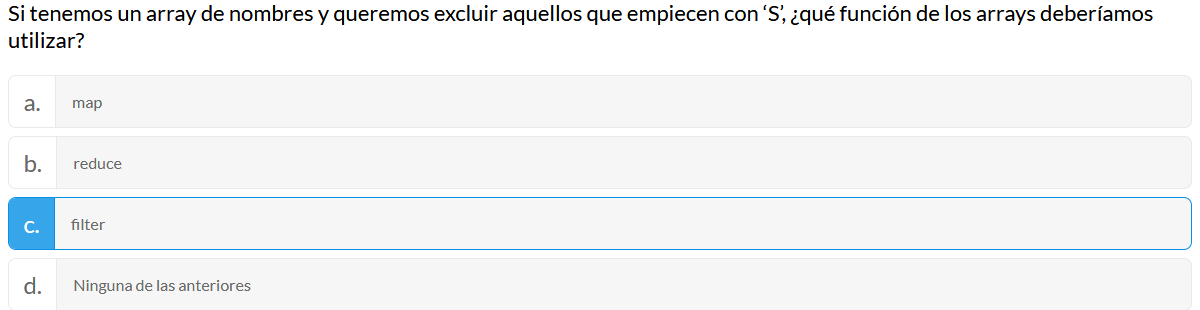












*Important to Read.*

[*https://eloquentjavascript.net/01\_values.html*](https://eloquentjavascript.net/01_values.html)

**Curso de jQuery a JavaScript.**

**La historia de jQuery.**

jQuery es una librería de JavaScript que hizo su lanzamiento en el año 2006 con el fin de resolver diferentes problemáticas:

* Una misma forma de acceder al DOM **$('selector'):** En ese tiempo, todos los navegadores accedían al DOM de una forma diferente.
* Poder interactuar con datos de un servidor **$.ajax():** jQuery permite hacer llamadas al servidor.
* Crear animaciones **$.animate():** En ese tiempo crear animaciones era muy difícil.

***Problema.***

El problema con jQuery surgió cuando se empezó a exagerar su uso y darle menor importancia a aprender JavaScript. Esto genero malos hábitos de aprendizaje y hasta en algunos casos no diferenciar jQuery de JavaScript.

A la par de que algunos se quedaban en jQuery, estaba ocurriendo la revolución de JavaScript trayendo consigo librerías que resolvían problemas específicos.

Aun con estas nuevas librerías, seguía sin resolverse el problema de hacer que uno aprenda más JavaScript que librerías o frameworks pues uno igual puede abusar de Vue, React o Angular.

* Se empezó a abusar de la librería y se empezó a usar sin ver si realmente era necesario su uso o no.
* Uno se hacía dependiente de jQuery y sus sub-librerías a tal punto que cuando se quería modificar algo de una librería no sabía cómo hacerlo.
* No se diferenciaba qué era jQuery y qué JavaScript.
* Mientras unos se quedaban en jQuery, la revolución de JavaScript estaba sucediendo.
* La idea es no depender de una librería sino aprender la tecnología que está detrás de cada librería.

## **Ventajas de usar JavaScript**

* Reutilizar conocimiento en otros lados de tu aplicación.
* Poder implementar soluciones sin depender de una librería.
* Estar más capacitado para grandes empresas.

# Promesas

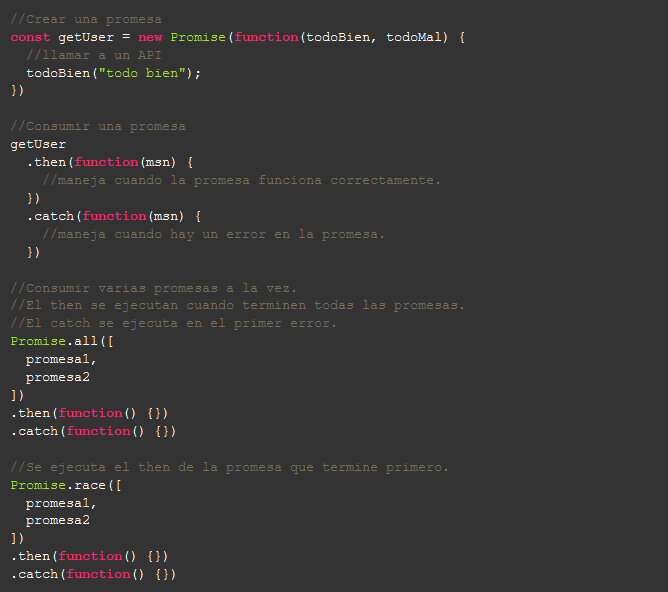
“Una Promesa es un objeto que representa la terminación o el fracaso eventual de una operación asíncrona”, o dicho de forma más cotidiana, se va a mandar una función para ver si falla o se ejecuta con éxito. Al crear una Promesa debemos pasarle por argumento la función que vamos a ejecutar de forma asíncrona, dicha función va a recibir dos parámetros para evaluar si se ejecutó bien la función o si fallo.

Dentro de JavaScript tenemos dos funciones para ejecutar una función después de algún tiempo, estas funciones son:

• **setInterval**: ejecutara una función cada x tiempo.  
• **setTimeout**: ejecutara una función después de x tiempo.

Si queremos resolver **varias promesas** a la misma vez, Promise cuenta con un método llamado **all** que recibe un array de promesas como parámetro.

Promise también cuenta con el método **race** que te regresa los resultados de la promesa que termine primero.



# Tutorial de Ajax en jQuery y Javascript

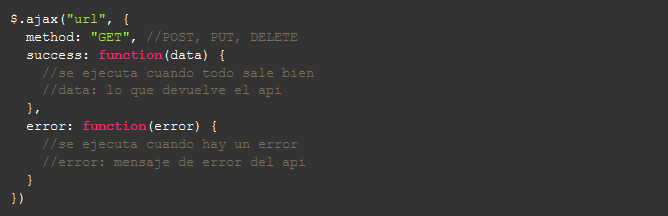
Una característica muy solicitada en cualquier sitio dinámico es **solicitar datos a un servidor**, denominado **API**. Para esto normalmente se utiliza Ajax.

**Ajax** recibe dos parámetros los cuales son la url de la API y un objeto donde pondrás la configuración que se usara para realizar la petición. En la configuración se añaden dos funciones para manejar cuando la petición se realizó correctamente y cuando falla.

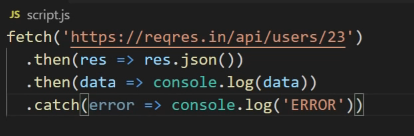
JavaScript internamente cuenta con una función llamada **fetch** que también realiza peticiones a una API. Al igual que Ajax necesita dos parámetros, una url y una configuración, pero si solo le mandas la url fetch usará una configuración por defecto donde el método HTTP será GET.  
**fetch** te regresa una **promesa**, esa promesa al resolverse te da los datos de respuesta y tiene un método llamado json que te regresa otra promesa con los datos en formato JSON.

Las promesas resuelven el problema del Callback Hell haciendo que una promesa pueda devolver otra promesa y en lugar de ser anidadas como los callback, estas promesas son encadenadas.

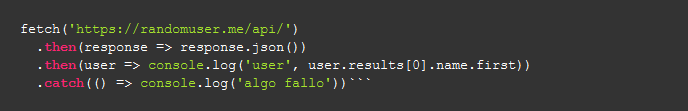
***Ajax - JQuery***



***Fetch-JavaScript***

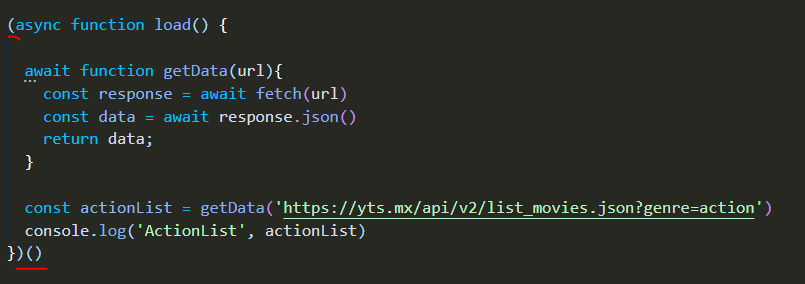


***Fetch con Arrow Functions.***



***Crear una función que se llame a si misma como un Load()***

Se debe agregar toda la función dentro de paréntesis () y luego se llaman los paréntesis de una Tipica función. (**funcion**)();



# Funciones asíncronas

Una función asíncrona va a ser como una función normal, pero poniendo código asíncrono de forma que sea más fácil de leer de forma síncrona.

Para declarar una función asíncrona se usa la palabra reservada async, luego de eso declaras tu función de forma normal. Dentro de una función asíncrona cuentas con otra palabra reservada llamada await, lo que hará esta palabra es indicar que se debe esperar a que termine de ejecutarse ese fragmento de código antes de continuar.

Sin el uso de funciones asíncronas para cada fetch tendríamos que usar los métodos then y catch, en cambio gracias a async/await solo debemos escribir la palabra await antes de cada promesa.



# Selectores

Un selector nos sirve para poder manipular un objeto del DOM, puedes buscar dicho objeto ya sea por su id, clase, atributo, etc.

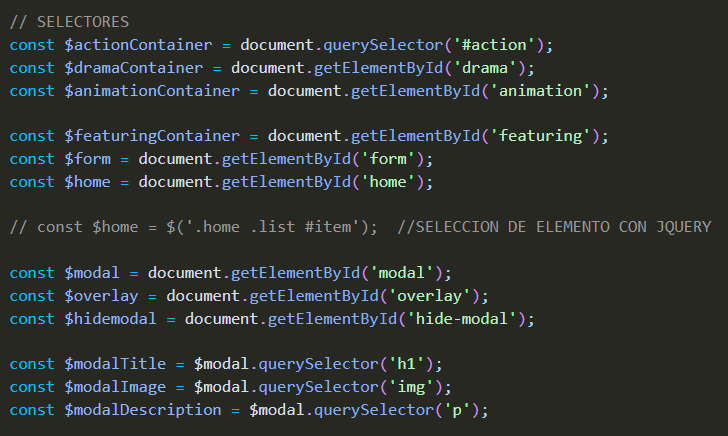
En jQuery hacemos un selector de la siguiente forma:



Por convención una variable que este represente un objeto del DOM lleva el signo $, esto es para tener claro que estamos manipulando un objeto del DOM y no algún tipo de información o dato.

Dentro de JavaScript existen distintas funciones para hacer selectores, entre ellas se encuentra:

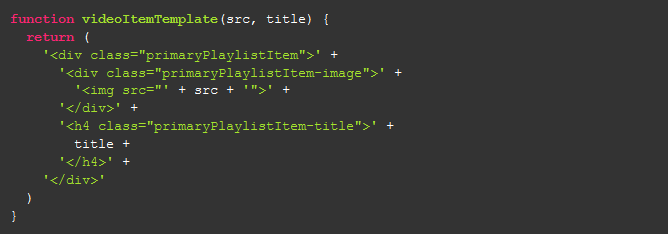
• **getElementById**: recibe como parámetro el **id** del objeto del DOM que estás buscando. Te regresa un solo objeto.  
• **getElementsByTagName**: recibe como parámetro el **tag** que estás buscando y te regresa una colección html de los elementos que tengan ese tag.  
• **getElementsByClassName**: recibe como parámetro la **clase** y te regresa una colección html de los elementos que tengan esa clase.  
• **querySelector**: va a buscar el **primer elemento** que coincida con el selector que le pases como parámetro.  
• **querySelectorAll**: va a buscar **todos los elementos** que coincidan con el selector que le pases como parámetro.



# Creación de templates.

# Vamos a crear una plantilla con nuestro elemento HTML base, dicha plantilla recibirá valores dinámicos.

Dentro de **jQuery**, la creación de un template seria con un texto base y si nuestro texto cuenta con distintas líneas más aparte tuviera valores dinámicos se vería de la siguiente forma:

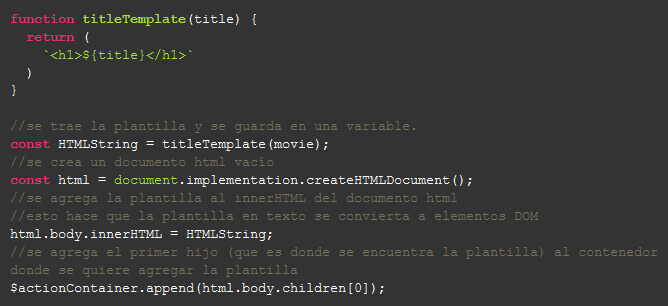


Desde ECMAScript 6 contamos con una nueva característica llamada **template literals** que se representan con las comillas invertidas ``, el ejemplo anterior pasaría a verse de esta forma:

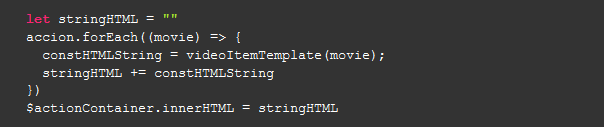


# Creación de DOM

La plantilla que creamos la clase anterior de momento es puro texto, no es un elemento HTML que podamos poner dentro del navegador pues si los imprimimos en el navegador lo único que veremos es texto.  
Vamos a insertar la plantilla dentro de nuestro container. Para convertir nuestra plantilla de texto a un Document Object Model necesitamos crear dentro de memoria un documento HTML, esto es posible gracias al método **document.implementation.createHTMLDocument**. A este documento HTML le vamos a añadir al body, mediante innerHTML, nuestra plantilla de texto. Una vez añadida le pedimos al body el primer elemento hijo que tenga y este lo añadimos a nuestro container.  
Este flujo es la magia que hay detrás de varias librerías y frameworks que nos ayudan a crear interfaces.



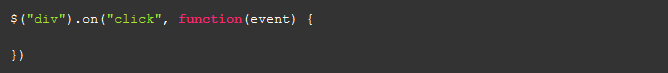
Todo esto de crear un nuevo documento HTML por motivos de reutilización, sin embargo podría quedar también de la siguiente forma:



# Eventos

Toda aplicación web necesita lidiar con interacciones del usuario, desde un click hasta arrastrar algún elemento, estas interacciones son escuchadas por el navegador mediante algo llamado eventos. Existen muchos tipos de eventos, el más común es el evento de click.  
En esta clase vamos a trabajar con el evento click y submit.  
Para que un elemento HTML pueda escuchar algún evento debemos usar el método **addEventListener**. Este método recibe dos parámetros, el **nombre del evento** que va a escuchar y la **función** que se va a ejecutar al momento de que se accione el evento.  
La página se recarga al momento de ejecutarse el evento submit, para evitar esto debemos quitarle la acción por defecto que viene en submit usando el método **event.preventDefault().**

**jQuery**



**JavaScript**



**Nota**: cuando se activa el evento submit, el browser de refresca por defecto. Para evitar esto se usa event.preventDefault().

# Clases y estilos CSS

En esta clase vamos a aprender a manipular las clases de CSS y estilos de nuestros elementos mediante JavaScript.  
Dentro de cada elemento tenemos un método llamado **classList**, con este podemos ver las clases que tiene nuestro elemento y además llamar a otros métodos para añadir, borrar o hacer **toggle** a alguna clase.  
De igual forma podemos acceder a todas las propiedades de CSS algún elemento mediante **element.style.**

**Clases.**



**Estilos Online.**



**Truco con el Chrome/Firefox Dev Tools**

$0 para coger al elemento seleccionado desde consola. 😄

# Creación de elementos y asignación de atributos.

# Vamos a crear un elemento HTML sin usar un template string. Para crear el elemento desde cero vamos a usar el método document.createElement, este recibe como parámetro la etiqueta html del elemento que se quiere crear, no funciona mandándole el template string.

**Creación.**



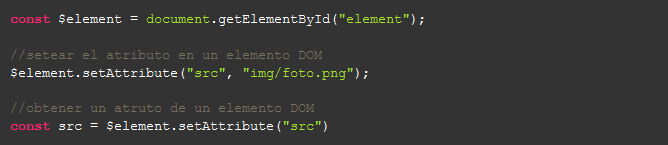
**Asignación de Atributos.**

# Para añadirle un atributo al elemento que acabamos de crear haremos uso del método setAttribute. Este recibe dos parámetros, uno indicando el nombre del atributo que vamos a añadir y el segundo parámetro indicando el valor de dicho atributo.

**jQuery**



**JavaScript**

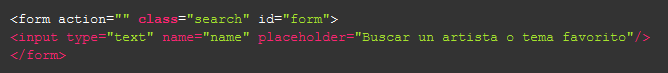


# También es posible crear una función para poder añadir múltiples atributos a un solo elemento.

# 

**Obtener Info de Formularios.**

Podemos tomar la información que se deja en un formulario de un sitio web al fijarla cuando escribimos sobre ella. Uno de los atributos de form o un input en HTML es name, en ella guarda en memoria la información de que dejamos escrito.

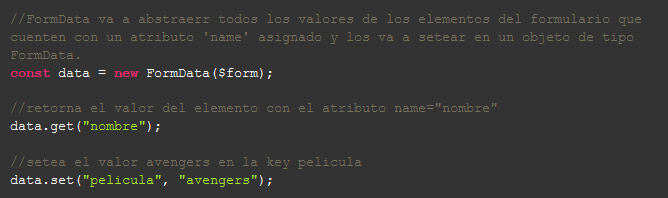


Para obtener toda la información a javaScript la conseguimos creando un nuevo objeto FormData:





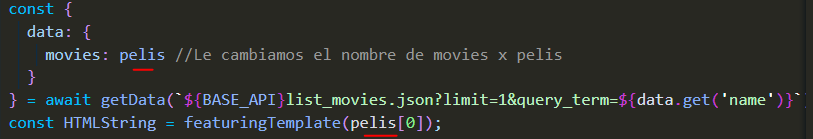
**FormData()** es una interfaz que te permite obtener los valores de un formulario.



# Desestructuración de objetos

Destructuring assignment permite entrar a un objeto o lista y poder sacar un dato para asignarlo a otra variable.

**EJ 1.**



Que seria lo mismo que:



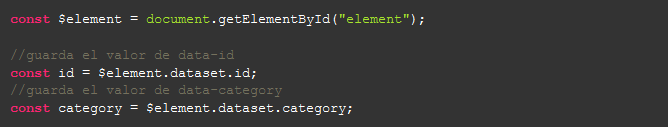
**EJ 2.**



## **Dataset**

Dataset permite acceder a un objeto con todos los atributos data de un elemento DOM.





La parte de la asignación de: id y category se puede hacer también con Destructuring assignment y queda mucho mejor.



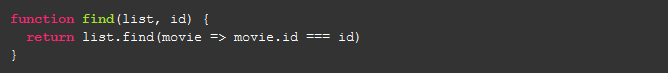
## **Transformar tipos de datos**

Cambiar un string a entero.



## **Encontrando elementos en lista (find)**

**find()** devuelve el primer elemento de un array que cumpla con el criterio de búsqueda. Si no se encuentra ningún elemento devuelve undefined.



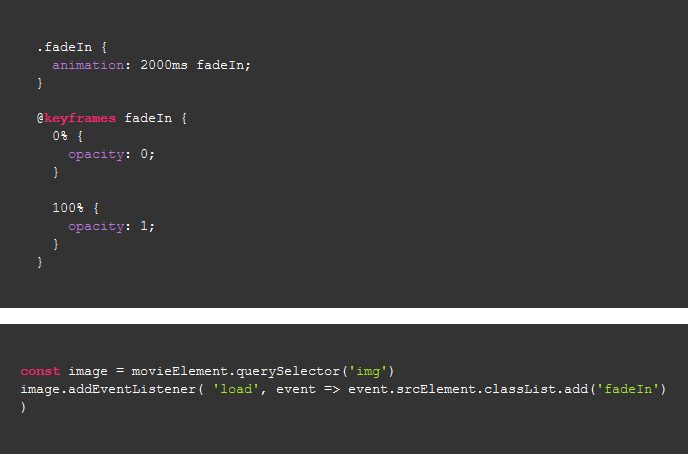
## **Switch**



# Animaciones

Tanto como en CSS puro, con javaScript también podemos hacer efectos de fade con CSS.

En esta clase obtenemos el objeto image del template que generamos para cada película y le agregamos una clase previamente declarada en nuestro CSS.  
Usamos el eventListener ‘load’ para imprimir todas las imágenes juntas una vez traídas de la API.



# Manejo de errores

El manejo de errores se hace con un bloque **try/catch**. Se intenta ejecutar un bloque de instrucciones (try) y se especifica una respuesta en caso suceda un error (catch).



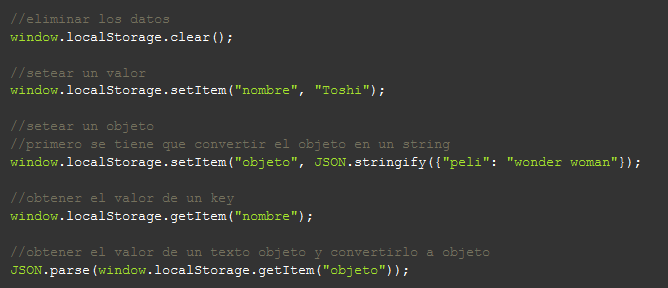
* Se puede crear un error customizado con **new Error**().
* Se puede lanzar un error con **throw**.



# LocalStorage

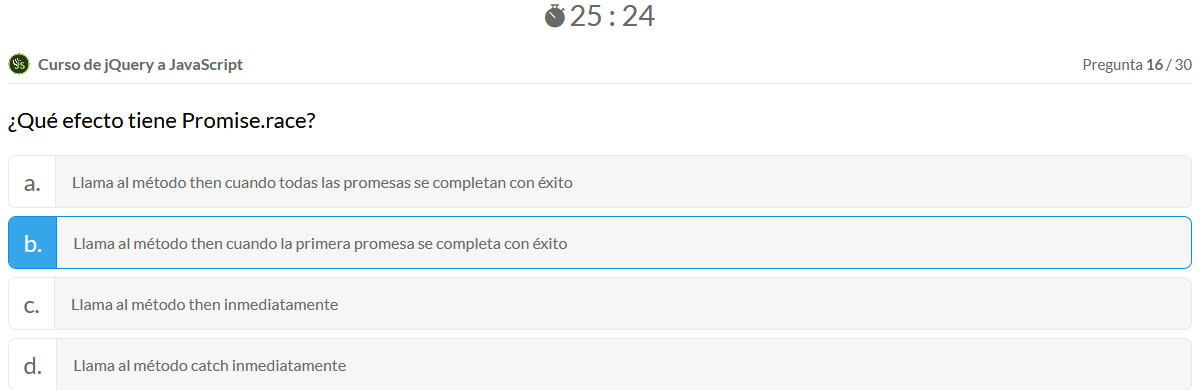
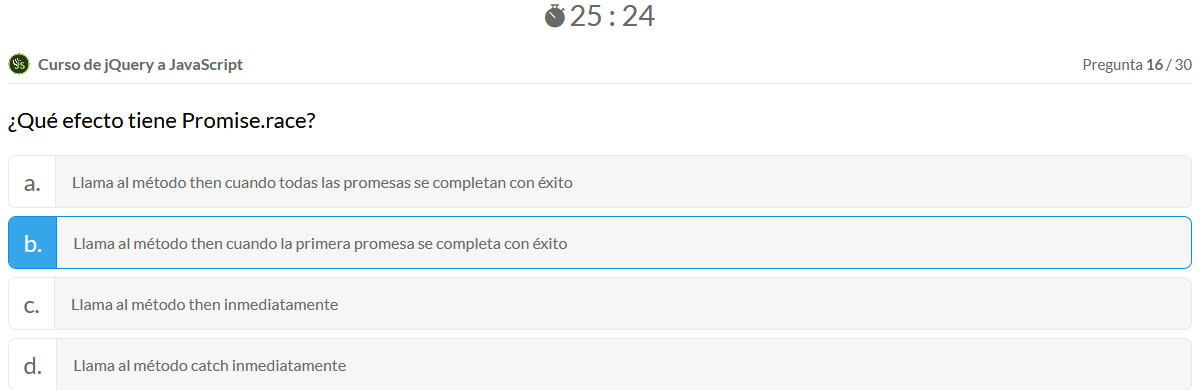
* localStorage permite almacenar datos sin tiempo de expiración
* sessionStorage permite almacenar datos. Estos datos se van a borrar cuando se termine la sesión del navegador

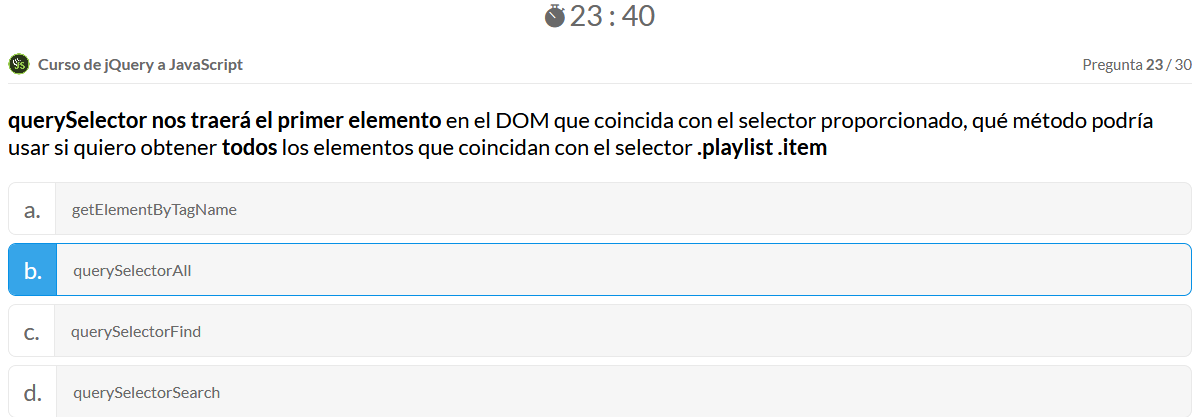
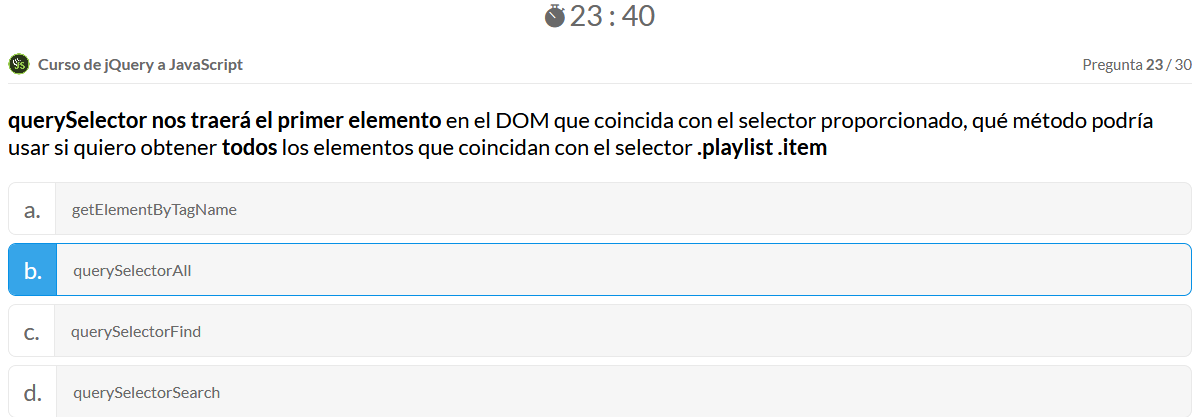
En local storage solo se puede guardar texto plano. No se pueden guardar objetos.

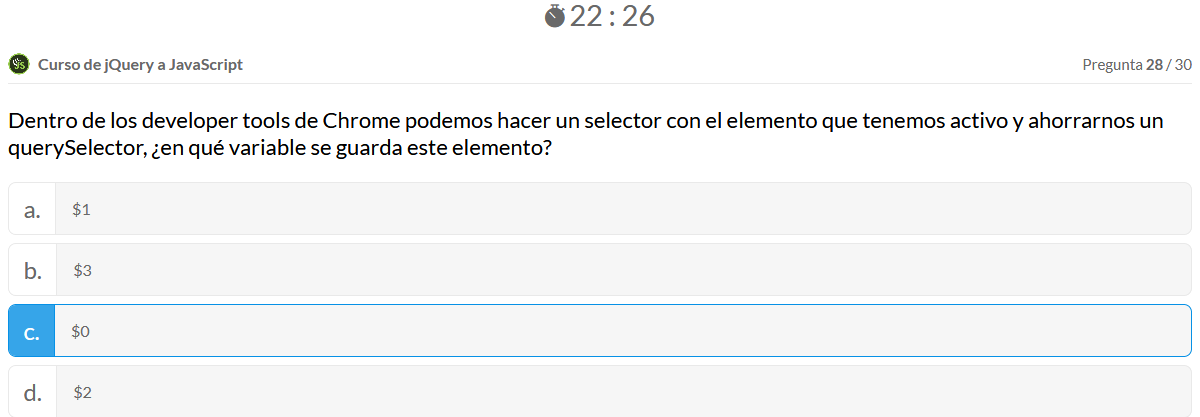
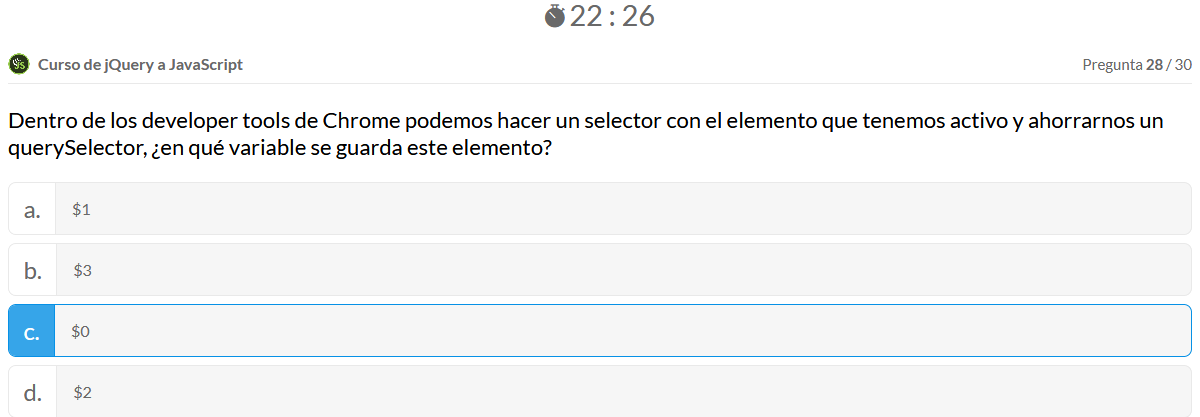


*Importants Questions and Answers.*









# Arreglar Error ‘Access-Control-Allow-Origin’



**Recursos Útiles:**

<https://www.freecodecamp.org/news/https-medium-com-gladchinda-hacks-for-creating-javascript-arrays-a1b80cb372b/>

<https://filisantillan.com/blog/this-en-diferentes-situaciones-y-su-comportamiento/>

<https://dmitripavlutin.com/when-not-to-use-arrow-functions-in-javascript/>

<https://github.com/MineiToshio/CursosPlatzi>

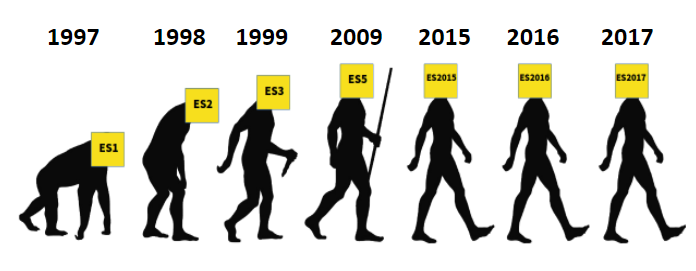
**Curso de ECMACScript6+**

**Qué es ECMAScript.**

ECMAScript es la especificación del lenguaje JavaScript propuesto por ECMA Internacional, que es la institución encargada de los estándares, y JavaScript, es el lenguaje de programación que utiliza las especificaciones propuestas, que van siendo añadidas cada año a partir del 2015, cuando fue lanzado ES6.

ECMA es una institución que dicta estándares o funcionalidades que deben adaptarse o añadirse a los engines de Js. Puesto que Js es un lenguaje interpretado y con muchos motores que lo interpretan/compilan, se vió la necesidad de hacer un estándar que todos esos motores siguieran, para tener consistencia en nuestros desarrollo a lo largo de todos los navegadores.

Esto nos ha permitido no solo tener funcionalidades nuevas, si no también que funcione de la misma manera en donde sea que corramos nuestro proyecto. Aunque también es importante decir, que por ser un estándar y no una empresa que está detrás del desarrollo del motor de js en sí, se suele tardar meses incluso años en ver reflejadas las nuevas características en los motores, o ver que esas características llegan primero a uno y luego a otros. Como el caso de Node.js, que incluso 5 años después de ECMAScript 2015 (ES6), aún no podemos usar ECMA import/export en él de manera normal.



# Default Parameters .

# Una de las características añadidas a ecmascript6 son los parámetros por defecto. Esto significa que podemos establecer ciertos valores que le pasamos a una función de una manera por defecto.

# Antiguamente se utilizaba este método:

# 

# Ahora:

# 

# Concatenación (template literals / template strings)

# Otra de las novedades es la concatenación

# Antes para concatenar algo simple como un “Hello world”

# 

# Ahora:

# 

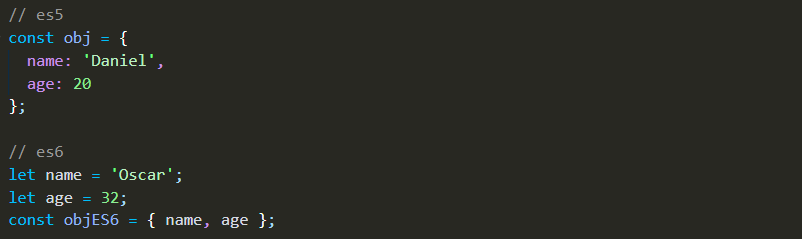
# Multilínea en los Strings.

# 

# LET / CONST / VAR.

# 

# Parámetros en objetos.



# Promesas

# 

# Arrow Functions.

# 

# Clases.

# 

# Módulos.

# Export default

# Con export default podemos exportar una variable, o función por defecto de un archivo así:

# 

# Para importarla no necesitamos hacerlo con su nombre, podemos usar cualquier nombre y por defecto nos traerá la única variable o función que exportó así:

# 

# Export nombrado

# A diferencia del export default en esta ocasión podemos exportar más de una variable o función que se encuentre en nuestro archivo, pero para importarlas debemos hacerlo con el nombre exacto de esa variable entre llaves.

# Para exportar lo hacemos así:

# 

# Podemos importar solo lo que necesitemos así:

# 

# También podemos importar más de un elemento nombrando cada uno

# 

# Podemos cambiarles los nombres.

# 

# Y podemos importar todas las funciones de una sola vez

# 

## **Anterior sintaxis**

Cuando no nos funcione esta sintaxis porque no la tengamos soportada podemos usar la anterior

Para exportar lo hacemos así:

Para importar así:

Con la sintaxis antigua también podemos exportar más de una variable o función pasándolas como un objeto.

# Generadores.

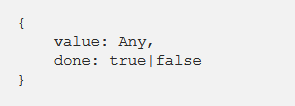
A diferencia de las funciones normales, una función generadora puede ser detenida en medio de su ejecución y posteriormente retomarla desde el punto en que se detuvo mediante la ejecución del callback que proporcionan. Es decir:

* Nos van a permitir simplificar codificar iteradores (pues las funciones generadoras permiten devolver múltiples resultados).
* Podemos obtener secuencias de resultados en vez de un único resultado.

Las características de este tipo de funciones son las siguientes:

* Son declaradas mediante un \* después de la palabra clave function para diferenciarlas de las funciones normales.
* Devuelven un objeto sobre el que podemos invocar el método next() .

Cada vez que invocamos next sobre el objeto devuelto por la función generadora obtenemos un nuevo objeto con la siguiente estructura:



La propiedad value es el valor devuelto por la función en ese paso mientras que done indica si la función ha dado por concluida su ejecución o por el contrario tiene más elementos que devolver. En el momento en que se devuelve false Javascript considera que la ejecución ha terminado.

# 

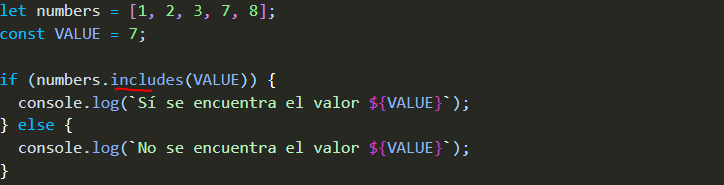
En este ejemplo declaramos la función generadora generatorFunction cuya primera diferencia con una función normal es que en vez de emplear return para devolver el valor final usa la palabra clave yield .

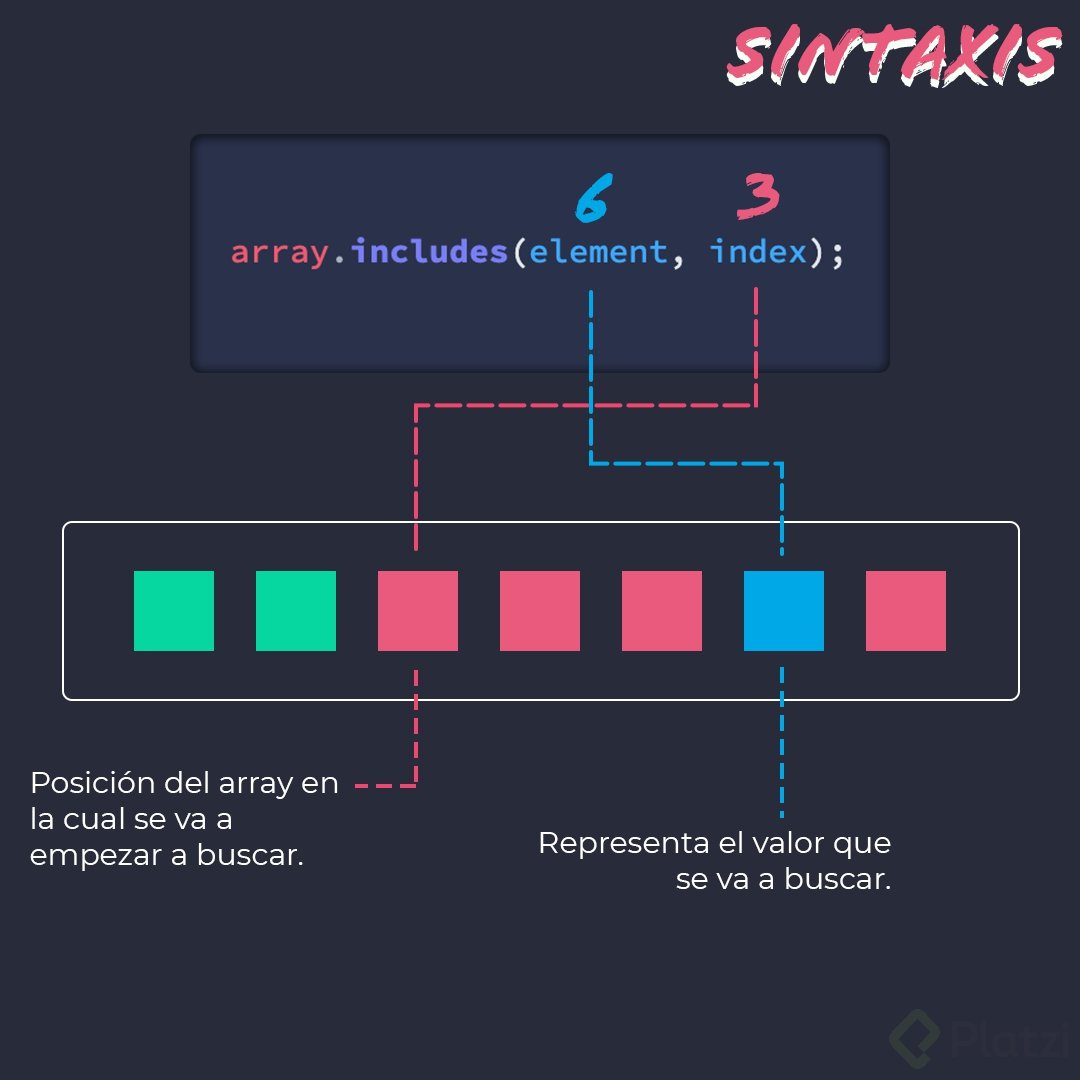
**Yield** es la forma de devolver valores dentro de una función generadora de modo que cuando un valor es devuelto de este modo la ejecución de la función generadora se detiene hasta que next es vuelta a llamar.

# <https://latteandcode.medium.com/javascript-todo-lo-que-necesitas-saber-sobre-generadores-5f2a6d42afc4>

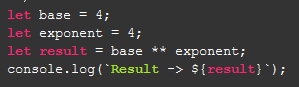
# ¿Qué se implementó en ES7?

***Includes() para un array.***





***Usar Exponencial. \*\****



***Result -> 256***

# ¿Qué se implementó en ES8?

# Object entries (Obtener propiedades de un objeto en un array)

# 

# 

# Object Values (Obtener valores de un objeto en un array)

# 

# 

# Padding (Añadir caracteres a un string)

# El primer parámetro que recibe es la cantidad de caracteres que podría recibir y el siguiente parámetro es el String que va a recibir.

# 

# Async Await.

# 

# ¿Qué se implementó en ES9?

**Propagation Properties (obtener las propiedades de un objeto y añadirlas en otro / Obtener una propiedad concreta)**

# 

# Finally de una promesa.

# 

# RegEx con JavaScript.

# 

# Otra de las nuevas características de ES9 en la captura de grupos es nombrar nuestros grupos para encontrarlos de una forma más fácil. El código de arriba seria:

# 

# ¿Qué se implementó en ES10 (Jun 2019)?

# ****Flat en arrays (nivel de profundidad: un nuevo método que nos permite aplanar arreglos.)****

# 

# 

# ****FlatMap en arrays**(** lo mismo que flat con el beneficio de que primero manipula la data para luego poder aplanar)

# 

# 

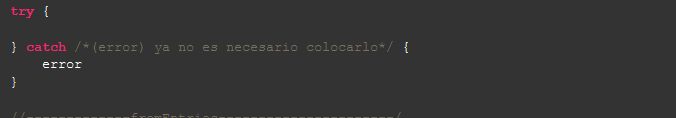
# ****TrimStart / trimEnd / Trim (Quitar espacios en blanco)****

# 

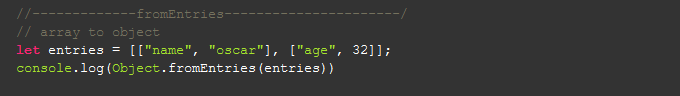
# Trim() quita los espacios tanto en el inicio como en el final

# 

**Optional catch bilding.**

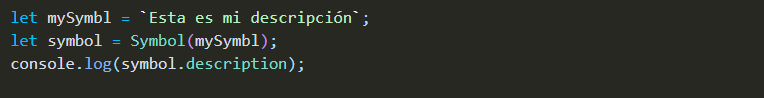


# Object fromEntries (Convertir un array en un objeto)





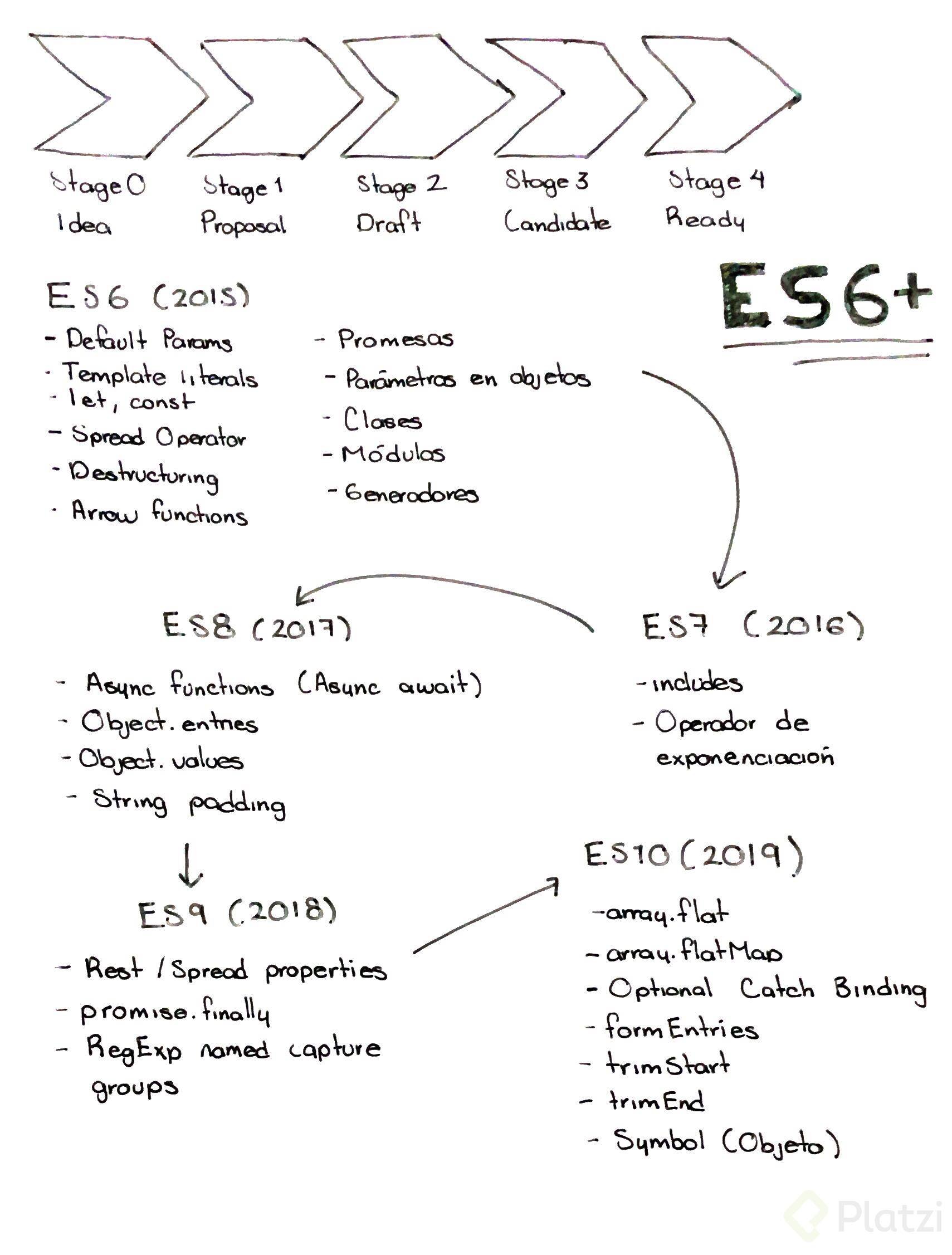
**Symbols Objects (Permite regresar la descripción opcional del Symbol)**



**Output:** Esta es mi descripción

<https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Symbol/description>

<https://tc39.es/>



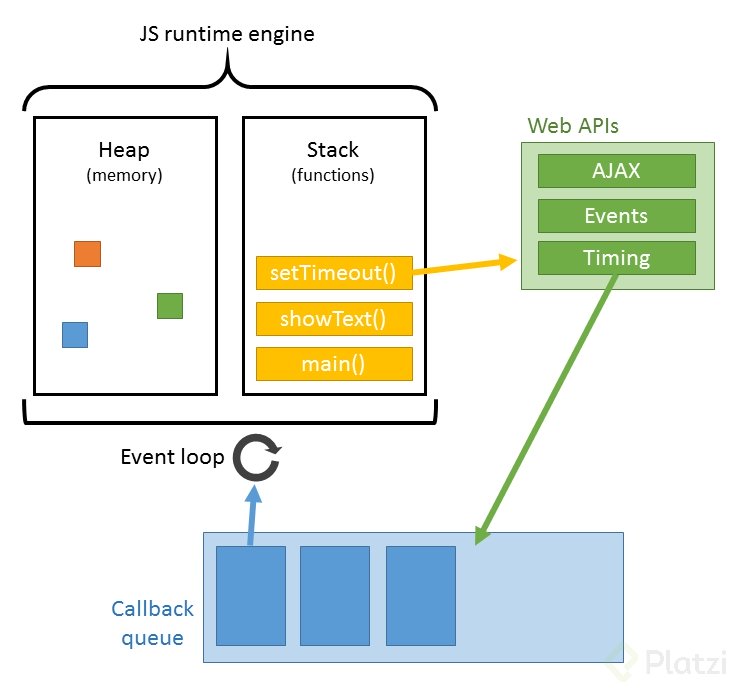
**Curso de Asincronismo con JavaScript.**

**Introducción al asincronismo.**

El asincronismo es básicamente una manera de aprovechar el tiempo y los recursos de nuestra aplicación, ejecutando tareas y procesos mientras otros son resueltos en background (como la llegada de la información de una API), para posteriormente continuar con las tareas que requerían esa información que no tenías de manera instantánea.

Un ejemplo fácil es comparando asincronismo vs sincronismo: En lenguajes síncronos al hacer un temporizador para ejecutar una función, todo el código se pausa hasta terminar el tiempo, mientras que en Javascript u otros lenguajes asíncronos, podemos estar aprovechando ese tiempo para ejecutar otros procesos hasta que ese tiempo finaliza.

**Event Loop**  
• El Event Loop hace que Javascript parezca ser multihilo a pesar de que corre en un solo proceso.  
• Javascript se organiza usando las siguientes estructuras de datos:  
• **Stack**. Va apilando de forma organizada las diferentes instrucciones que se llaman. Lleva así un rastro de dónde está el programa, en qué punto de ejecución nos encontramos.  
• **Memory Heap**. De forma desorganizada se guarda información de las variables y del scope.  
• **Schedule Tasks**. Aquí se agregan a la cola, las tareas programadas para su ejecución.  
• **Task Queue**. Aquí se agregan las tareas que ya están listas para pasar al stack y ser ejecutadas. El stack debe estar vacío para que esto suceda.  
• **MicroTask Queue**. Aquí se agregan las promesas. Esta Queue es la que tiene mayor prioridad.  
• El Event Loop es un loop que está ejecutando todo el tiempo y pasa periódicamente revisando las queues y el stack moviendo tareas entre estas dos estructuras.



**Conceptos Importantes.**

**API**

Interfaz de programación de aplicaciones (Application Programming Interface). Es un conjunto de rutinas que provee acceso a funciones de un determinado software.

**Concurrencia**

Cuando dos o más tareas progresan simultáneamente.

**Paralelismo**

Cuando dos o más tareas se ejecutan, literalmente, a la vez, en el mismo instante de tiempo.

**Bloqueante**

Una llamada u operación bloqueante no devuelve el control a nuestra aplicación hasta que se ha completado. Por tanto el thread queda bloqueado en estado de espera.

**Síncrono**

Es frecuente emplear ‘bloqueante’ y ‘síncrono’ como sinónimos, dando a entender que toda la operación de entrada/salida se ejecuta de forma secuencial y, por tanto, debemos esperar a que se complete para procesar el resultado.

**Asíncrono**

La finalización de la operación I/O se señaliza más tarde, mediante un mecanismo específico como por ejemplo un callback, una promesa o un evento, lo que hace posible que la respuesta sea procesada en diferido.

**Call Stack**

La pila de llamadas se encarga de albergar las instrucciones que deben ejecutarse. Nos indica en que punto del programa estamos, por donde vamos.

**Heap**

Región de memoria libre, normalmente de gran tamaño, dedicada al alojamiento dinámico de objetos. Es compartida por todo el programa y controlada por un recolector de basura que se encarga de liberar aquello que no se necesita.

**Cola o Queue**

Cada vez que nuestro programa recibe una notificación del exterior o de otro contexto distinto al de la aplicación, el mensaje se inserta en una cola de mensajes pendientes y se registra su callback correspondiente.

**EventLoop o Loop de eventos**

Cuando la pila de llamadas (call stack) se vacía, es decir, no hay nada más que ejecutar, se

procesan los mensajes de la cola. Con cada ‘tick’ del bucle de eventos, se procesa un nuevo

mensaje.

**Hoisting**

Sugiere que las declaraciones de variables y funciones son físicamente movidas al comienzo del código en tiempo de compilación.

**DOM**

DOM permite acceder y manipular las páginas XHTML como si fueran documentos XML. De hecho, DOM se diseñó originalmente para manipular de forma sencilla los documentos XML.

**XML**

Lenguaje de marcado creado para la transferencia de información, legible tanto para seres

humanos como para aplicaciones informáticas, y basado en una sencillez extrema y una rígida sintaxis. Así como el HTML estaba basado y era un subconjunto de SGML, la reformulación del primero bajo la sintaxis de XML dio lugar al XHTML; XHTML es, por tanto, un subconjunto de XML.

**Events**

Comportamientos del usuario que interactúa con una página que pueden detectarse para lanzar una acción, como por ejemplo que el usuario haga click en un elemento (onclick), que elija una opción de un desplegable (onselect), que pase el ratón sobre un objeto (onmouseover), etc.

**Compilar**

Compilar es generar código ejecutable por una máquina, que puede ser física o abstracta como la máquina virtual de Java.

**Transpilar**

Transpilar es generar a partir de código en un lenguaje código en otro lenguaje. Es decir, un

programa produce otro programa en otro lenguaje cuyo comportamiento es el mismo que el

original.

Lista de APIs

<https://github.com/public-apis/public-apis>

<https://rickandmortyapi.com/>

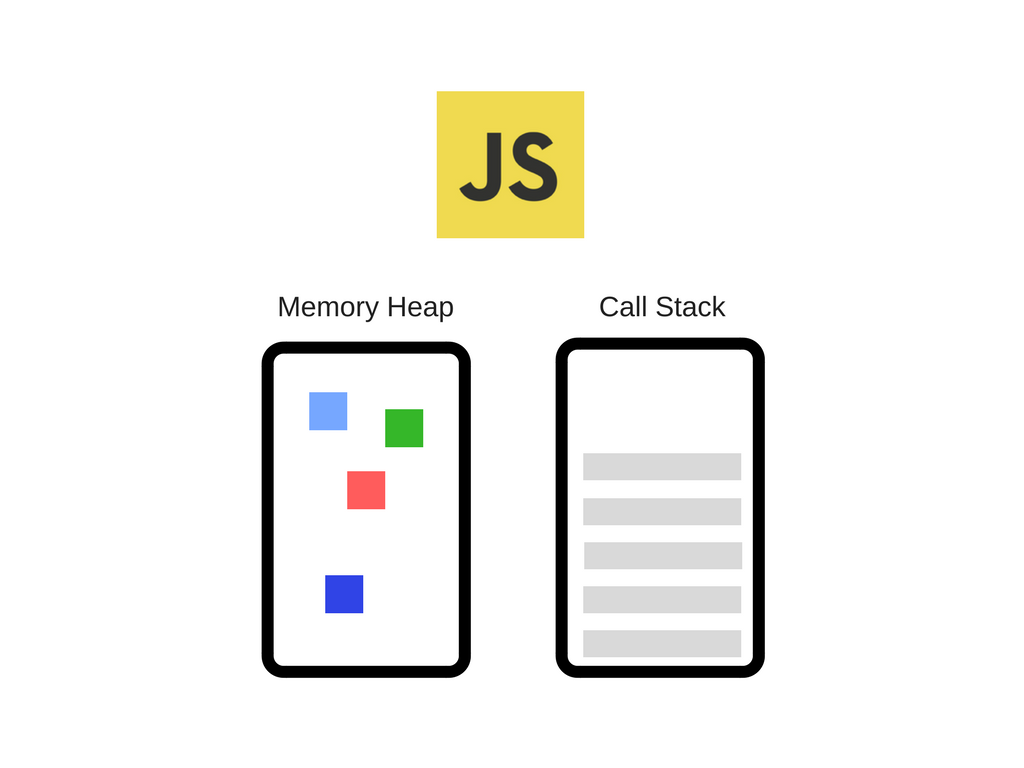
# Event Loop: la naturaleza asincrónica de Javascript

Decidí escribir este artículo ya que pienso que es importante comprender todos los aspectos del lenguaje para convertirse en un desarrollador profesional (o al menos uno bueno). Es algo que, lamentablemente, en javascript no es común.

Debido a la gran cantidad de librerías y herramientas presentes en este lenguaje, muchos programadores comienzan a desarrollar aplicaciones sin tener un entendimiento real de cómo es que las cosas funcionan “por debajo”.

En este caso voy a hablar de uno de estos aspectos, el cual es muy importante. El famoso event loop.

Antes que nada miremos un dibujo que representa el runtime de v8 (el runtime que usa Chrome y node)



Como se puede ver en la imagen, el engine consiste de dos elementos principales

* Memory Heap: es donde se realiza la alocación de memoria
* Call Stack: es donde el runtime mantiene un track de las llamadas a las funciones

Solo hablaremos de la call stack, que es la que se relaciona con el Event Loop.

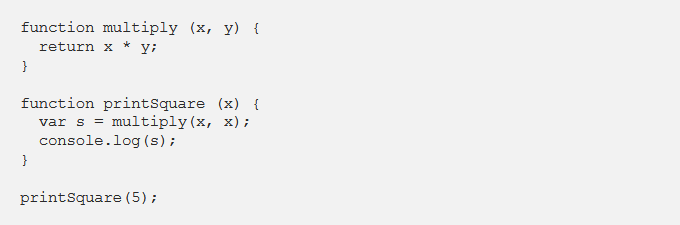
# Call Stack

Para los que no sepan un stack (también llamado pila) es una estructura simple, similar a un arreglo en el que solo se puede agregar ítems al final (push) , y remover el último (pop).

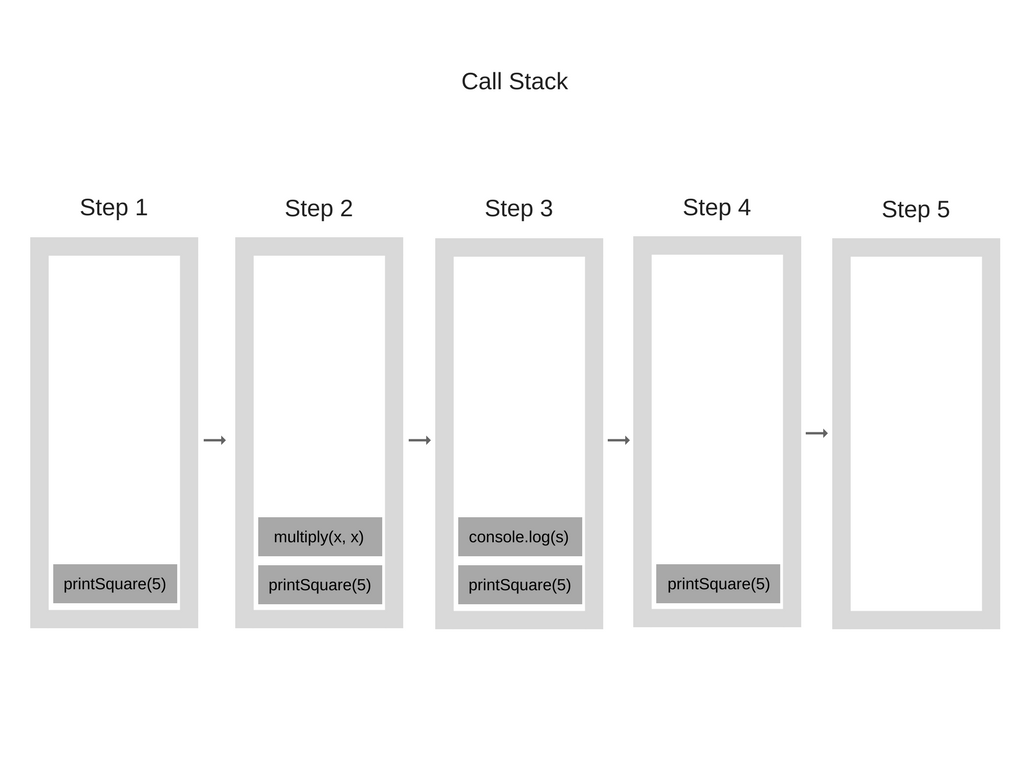
El proceso que realiza el call stack es simple, cuando se está a punto de ejecutar una función, esta es añadida al stack. Si la función llama a su vez, a otra función, es agregada sobre la anterior. Si en algún momento de la ejecución hay un error, este se imprimirá en la consola con un mensaje y el estado del call stack al momento en que ocurrió.

**Javascript** es un lenguaje **single threaded**. Esto quiere decir que durante la ejecución de un script existe un solo thread que ejecuta el código. Por lo tanto solo se cuenta con un call stack

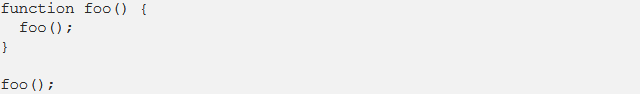
Veamos un ejemplo:

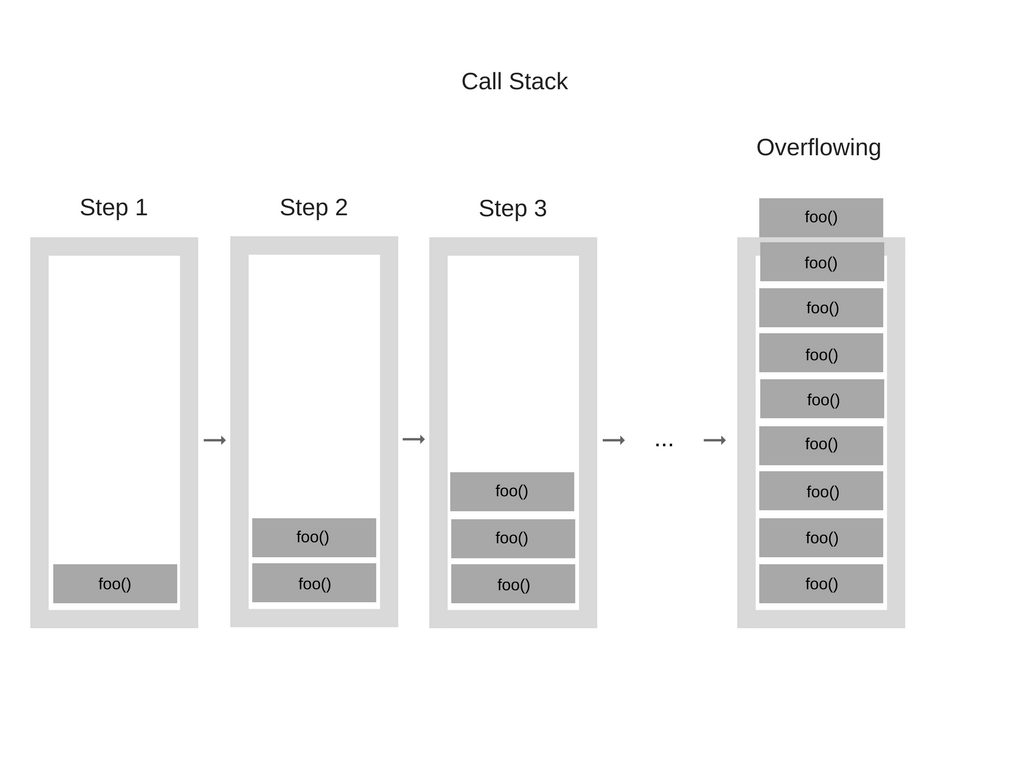


**Los estados del call stack serían:**



Y que pasa si tenemos una función de esta manera:





Lo que sucedería es que en algún momento la cantidad de funciones llamadas excede el tamaño del stack , por lo que el navegador mostrará este error:

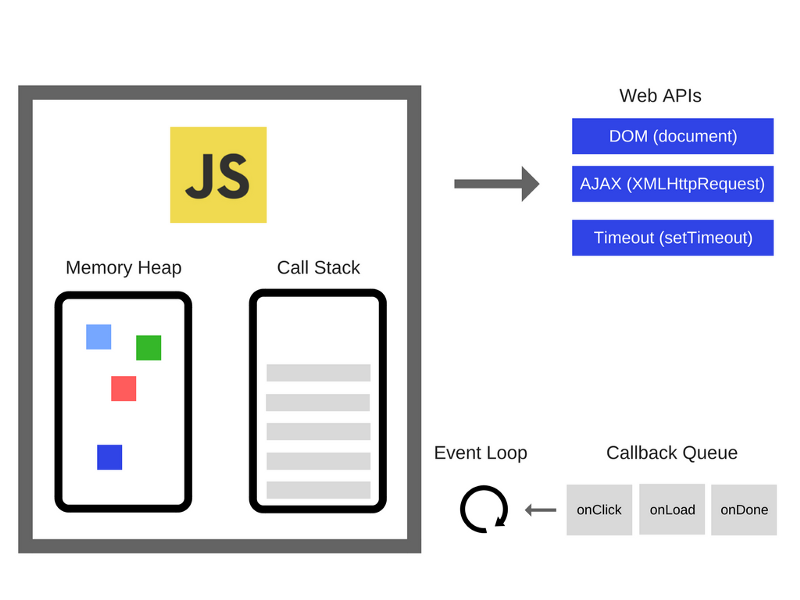
Image for post

Pero qué pasa si llamamos a un timeout o hacemos un request con AJAX a un servidor. Al ser un solo thread, hay un solo call stack y por lo tanto solo se puede ejecutar una cosa a la vez. Es decir el navegador debería congelarse, no podría hacer más nada, no podría renderizar, hasta que la llamada termine de ejecutarse. Sin embargo esto no es así, javascript es asincrónico y no bloqueante. Esto es gracias al Event Loop.

# Event Loop

Algo interesante acerca de javascript, o mejor dicho de los runtime de javascript, es que no cuentan nativamente con cosas como setTimeout, DOM, o HTTP request. Estas son llamadas web APIs, que el mismo navegador provee, pero no están dentro del runtime JS.

Por lo tanto este es el gráfico que muestra una visión más abarcativa de javascript. En este se puede ver el runtime, más las Web APIs y el callback Queue del cual hablaremos más adelante.

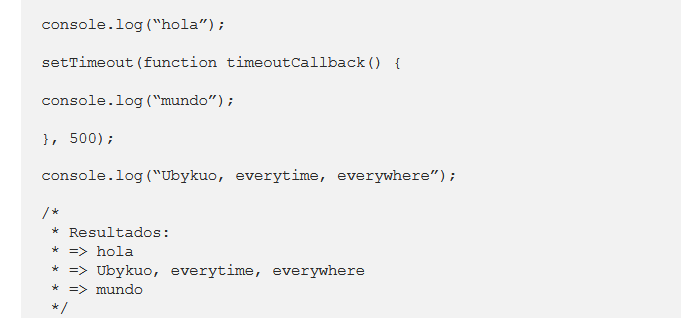


Al haber un solo thread es importante no escribir código bloqueante para la UI no quede bloqueada.

Pero ¿Cómo hacemos para escribir código no bloqueante?

La solución son **callbacks** asincrónicas. Para esto combinamos el uso de callbacks (**funciones que pasamos como parámetros a otras funciones**) con las WEB APIs.

Por ejemplo:



Como pueden ver la ejecución no se queda bloqueada en *setTimeout()* ya que imprime la instrucción que le sigue primero) ¿Pero entonces cómo es que posible que esto sea así si solo existe un solo thread? ¿ Cómo es que la ejecución continua y al mismo tiempo el setTimeout hace la cuenta regresiva para ejecutar la función pasada como callback?

Esto es porque, como mencione anteriormente, el **setTimeout** NO es parte del runtime. Sino que es provista por el navegador como **WEB APIs** ( o en el caso de Node por c++ APIs). Los cuales SI se ejecutan en un **thread distinto**.

**¿Como se maneja esto con una única call stack?**

Existe otra estructura donde se guardan las funciones que deben ser ejecutadas luego de ciertos **eventos** (OnLoad, OnClick, OnDone), en el caso del código de ejemplo de arriba se guarda que, cuando el timeout termine se debe ejecutar la función timeoutCallback(). Tener en cuenta que cuando sucede el evento, esta estructura no es la que la ejecuta y tampoco las agrega al call stack ya que sino podría pasar que la función se ejecutará en medio de otro código. Lo que hace es enviarla al **Callback Queue**.

Lo que hace el event loop es fijarse el call stack, y si está vacío (es decir no hay nada ejecutándose) envía la primera función que esté en la callback queue al call stack y comienza a ejecutarse.

Luego de terminar la cuenta regresiva del setTimeout() (que no es ejecutada en el runtime de javascript), timeoutCallback() será enviada a la callback queue. El event loop chequeará el Call Stack, si este está vacío enviará timeoutCallback() al call stack para su ejecución.

Para verlo visualmente.

[https://miro.medium.com/max/700/1\*7TBEc7ozzrMZmVgvI3s0mQ.gif](https://miro.medium.com/max/700/1*7TBEc7ozzrMZmVgvI3s0mQ.gif)

De esta manera se logra que el código sea no bloqueante, en vez de un setTimeout podría ser una llamada a un servidor, en donde habría que esperar que se procese nuestra solicitud y nos envíe una respuesta , el cual sería tiempo ocioso si no contáramos con callbacks asincrónicas, de modo que el runtime pueda seguir con otro código. Una vez que la respuesta haya llegado del servidor y Call Stack esté vacío, se podrá procesar la respuesta (mediante la función pasada como callback ) y hacer algo con ella , por ejemplo mostrarla al usuario.

**¿Por qué si bloqueamos el call stack la ui ya no responde más?**

Esto se debe a que el navegador intenta realizar un proceso de renderizado cada cierto tiempo. Pero este no puede realizarse si hay código en el stack. El proceso de renderizado es similar a una callback asincrónica , ya que debe esperar a que el stack está vacío, es como una función más en la Callback Queue (aunque con cierta prioridad). Por lo que sí hay código bloqueante , el proceso de renderizado tardará más en realizarse y el usuario no podrá hacer nada, no podrá seleccionar texto, no podrá ingresar texto, no podrá apretar un botón.

¿Qué pasaría si a un usuario que interactuando con nuestra página le sucediera esto?

Lo más probable es que cierre el navegador y nunca más vuelva a entrar a nuestra página. No es algo que queremos que suceda.

# Datos curiosos.

Algunos “hacks” interesantes para entender Javascript.

Una forma ingeniosa de ejecutar una función asincrónicamente es usar setTimeout (funcionAEjecutar, 0). Si bien el timeout es 0 , al llamar a una Web Api esta es enviada a la callback queue y será ejecutada cuando el stack esté vacío.

Otra hack interesante es para evitar el error de overflow del call stack. Una forma de permitir una cantidad absurda de llamadas recursivas, es la de envolver la función en un setTimeout. Esto permitirá que las funciones vayan al Callback Queue, evitando que se apilen en el Call Stack. Claramente esta no es una buena práctica , pero es un buen ejemplo del comportamiento de Javascript.

# XMLHttpRequest (Using callbacks).

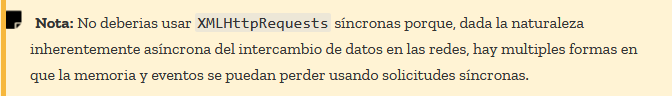
En esta guía le echaremos un vistazo a cómo usar [XMLHttpRequest](https://developer.mozilla.org/es/docs/Web/API/XMLHttpRequest) para enviar solicitudes [HTTP](https://wiki.developer.mozilla.org/en-US/docs/Web/HTTP) con el objetivo de intercambiar datos entre el sitio web y el servidor. Se incluyen ejemplos, tanto para los casos de uso comunes de XMLHttpRequest, como para los más inusuales.

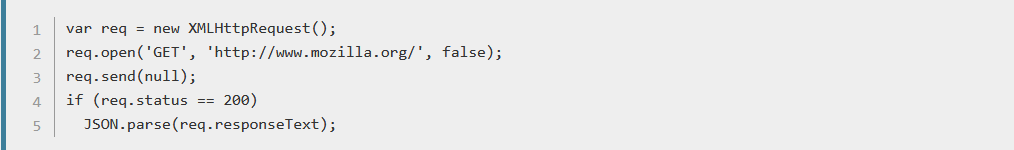
Para enviar una solicitud HTTP, cree un objeto XMLHttpRequest, abra una URL y envíe la solicitud. Una vez que la transacción haya sido completada, el objeto contendrá información útil tal como el cuerpo de la respuesta y el estado [HTTP](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status) del resultado.



### **Ejemplo: Solicitud síncrona**

Este ejemplo demuestra cómo hacer una solicitud síncrona.





En la línea 1 se instancia un objeto XMLHttpRequest.  Después en la línea 2 se abre una nueva solicitud, especificando que una solicitud GET se utilizará para extraer la página de inicio de Mozilla.org, y que la operación no debe ser asíncrona.

En la línea 3 se envía la solicitud.  El parámetro null indica que la solicitud GET no necesita contenido en el cuerpo. En la línea 4 se verifica el código de estatus después de que la transacción se completa.  Si el resultado es 200 -- El código HTTP para resultado "OK"-- el contenido de la solicitud que llega en texto, lo convertimos en JSON.

### **Ejemplo: Solicitud Asíncrona con callback(Funcion enviada como parámetro)**



<https://www.w3schools.com/xml/ajax_xmlhttprequest_response.asp>

<https://developer.mozilla.org/es/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest>

# Promesas.

# 

# 

# Async/await.

# Async/await no es más que Syntax Sugar. Es una manera más bonita de hacer lo mismo que estábamos haciendo con .then(). La clave es recordar que si una función regresa un promesa, podemos usar el keyword await, que le indicia al navegador: “Espera a que la promesa se resuelva y almacena su resultado en esta variable”. Todo esto toma lugar dentro de una función asíncrona, así que usamos async para lograr esto.

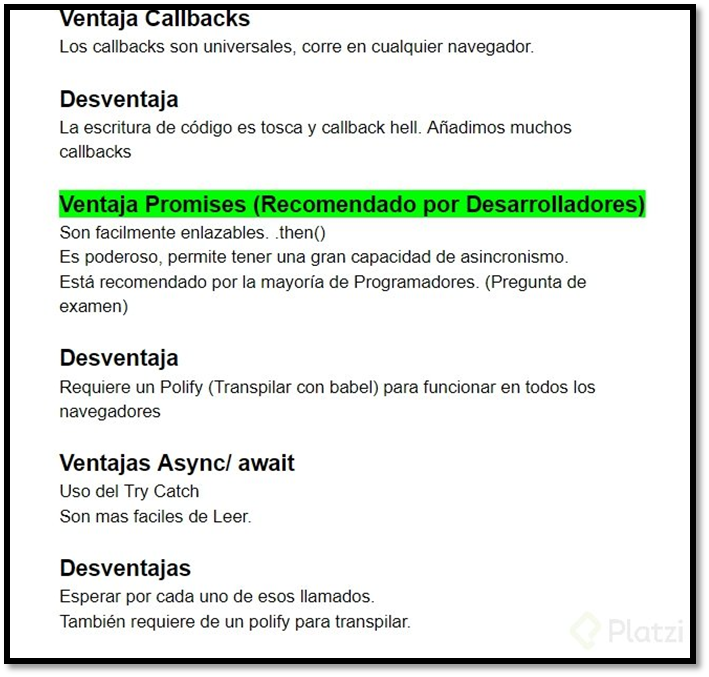
# 

# Callbacks Vs Promesas Vs Async/Await

**Callbacks** --> Ventajas: Simple(una función que recibe otra función). Son universales, corren en cualquier navegador.  
Desventajas: Composición tediosa, anidando cada vez más elementos. Caer en Callback Hell.

**Promesas** --> Ventajas: Fácilmente enlazables .Then( return… ).Then - Fácil e intuitivo de leer.  
Desventajas: Posible error si no se retorna el siguiente llamado. No corre en todos los navegadores.

**Async-Await** --> Ventajas: Se puede usar try-catch . Código más ordenado e intuitivo.  
Desventajas: No corre en todos los navegadores (se requiere un transpilador).



Importants Questions And Answers.

