Índice

[**Deuda técnica y refactorización de código** 1](#_Toc132564105)

[**Tipos de deuda técnica** 1](#_Toc132564106)

[**¿Cómo pagar las deudas?** 1](#_Toc132564107)

[**¿Cuándo refactorizar?** 2](#_Toc132564108)

[**Reglas del diseño simple** 2](#_Toc132564109)

[**Qué es Clean Code** 2](#_Toc132564110)

[**Uso correcto de var, let y const** 2](#_Toc132564111)

[**Reglas para la nomenclatura** 4](#_Toc132564112)

[**Cómo nombrar según el tipo de dato** 4](#_Toc132564113)

[**Declaración y expresión de funciones** 5](#_Toc132564114)

[**Declaración de funciones** 5](#_Toc132564115)

[**Expresión de funciones** 6](#_Toc132564116)

[**Función de expresión utilizando arrow function** 6](#_Toc132564117)

[**Diferencias** 6](#_Toc132564118)

[**Parámetros y argumentos** 6](#_Toc132564119)

[**Rest Operator** 7](#_Toc132564120)

[**Funciones de flecha y el this** 8](#_Toc132564121)

[**POO con ES6 (constructores, métodos y herencia)** 10](#_Toc132564122)

[**Clase** 10](#_Toc132564123)

[**Herencia en JavaScript** 11](#_Toc132564124)

[**Tamaño reducido (responsabilidad única)** 13](#_Toc132564125)

[**Organización de clases** 13](#_Toc132564126)

[**Cuando usar comentarios** 14](#_Toc132564127)

[**¿Se debe o no utilizar comentarios?** 14](#_Toc132564128)

[**Formato coherente (codear en equipos)** 15](#_Toc132564129)

[**Reglas para trabajar en equipo** 15](#_Toc132564130)

[**Principio DRY** 15](#_Toc132564131)

[**Notación big O** 17](#_Toc132564132)

[**Algoritmo de tiempo constante: O(1)** 17](#_Toc132564133)

[**Algoritmo de tiempo lineal: O(N)** 18](#_Toc132564134)

[**Algoritmo de tiempo cuadrático: O(N²)** 19](#_Toc132564135)

[**Algoritmo de tiempo logarítmico: O(n log n)** 20](#_Toc132564136)

# **Deuda técnica y refactorización de código**

Nuestro código debe ser **simple** y **directo**, debería leerse con la misma facilidad que un texto bien escrito.

**Tipos de deuda técnica**

* **Imprudente y deliberada**: Se da cuando el desarrollador actúa consiente de lo que van hacer y deforma imprudente, porque no toman en cuenta el factor de deuda que están dejando en el código.
* **Imprudente e inadvertida**: Esto es muy común en los perfiles con poco conocimiento técnico, es decir, no se sabe que se esta dejando una deuda por la falta de conocimientos que se tiene.
* **Prudente y deliberada**: Está deuda técnica en si misma, no es mala, pero muchas veces se toman decisiones de inducir ciertos tipos de deudas para hacer un desarrollo más ágil y tener un producto mucho más rápido.
* **Prudente e inadvertida**: Es posible qué, esta deuda sea la más común de todas, porque como desarrolladores, vamos aprendiendo a lo largo de cada proyecto y puede ser que los conocimientos que teníamos en un inicio del proyecto, creer que la forma en la que se hizo en ese momento, era la más eficiente, pero al final, cuando tenemos muchos más conocimientos que cuando iniciamos, nos damos cuenta que quizás lo podríamos a ver hecho de una mejor forma.

**¿Cómo pagar las deudas?**

Refactorizando el código, esto es mejorar el código **sin alterar** **su comportamiento** para que sea más entendible y tolerante a cambios. Y es importante que el código tenga tests (units o integration tests) automáticos que validen el comportamiento del código.

**¿Cuándo refactorizar?**

* Cuando hay **código de** **baja calidad** (duplicación de código, funciones con más de una acción)
* Cuando se detecta cualquier otro tipo de **code smell** (código duplicado, métodos o clases demasiado grandes y complejos, falta de cohesión entre diferentes partes del código, uso excesivo de condicionales y bucles anidados.)

# **Reglas del diseño simple**

* El código pasa correctamente los test.
* Revela la intención del diseño. El código debe autoexplicarse.
* Respeta el principio **DRY**: “don’t repeat yourself”, es decir, en español, no te repitas a ti mismo.
* Tiene el menor número posible de elementos. Es decir, 1 función = 1 acción.

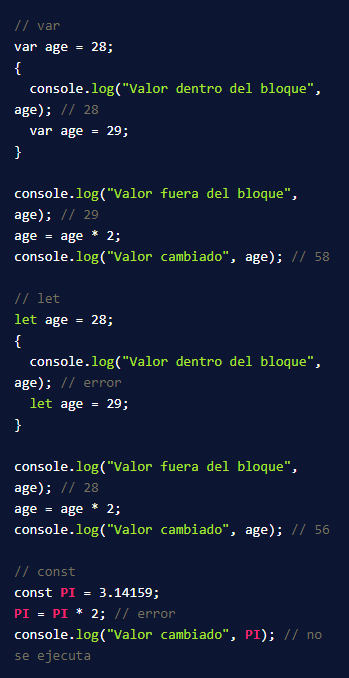
# **Qué es Clean Code**

**El código limpio es aquel que se ha escrito con la intención que otra persona lo entienda**.

# **Uso correcto de var, let y const**

ECMAScript 6 2015: Introdujo **let** y **const**.

* **var**: No respeta ámbitos de bloque.
* **let**: Si respeta ámbitos de bloque.
* **const**: Valores que no cambian NUNCA, pero pueden cambiar por **referencia**, pero la constante en sí no puede ser reasignada.



# **Reglas para la nomenclatura**

¿Cómo nombro correctamente mis variables, funciones, etc…?

* **Nombres pronunciables y expresivos**: Preferentemente en inglés usando camelCase🐫, evitando guiones - o \_ y abreviaturas.
* **Nombres sin información técnica**: Evitar nombres con relación a la tecnología (tipo de datos, clases, etc). Ej. “arrayNames” --> “namesList”
* **Usar lenguaje ubicuo**: Aquel que se construye a partir del lenguaje que usan los expertos, es decir: crear un lenguaje en común para desarrolladores e interesados, determinando palabras de uso común.

# **Cómo nombrar según el tipo de dato**

* **Arrays**: Plural.
* **Booleanos**: Con prefijos **is**, **has** y **can**. (“es verdadero”, “tiene/contiene x”, “puede hacer x”).
* **Números**: Con prefijos **min**, **max** y **total**.
* **Funciones**: Verbo + sustantivo/s (ya que son acciones).
* **…de Acceso**: get + sustantivo
* **…de Modificación**: set + sustantivo;
* **…de Predicado**: is + sustantivo;
* **Clases**: Sustantivos (no genéricos).

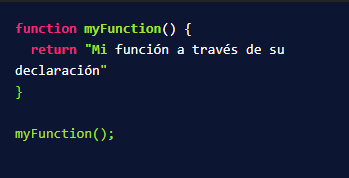
# **Declaración y expresión de funciones**

Las funciones son entidades organizativas de cualquier lenguaje de programación.

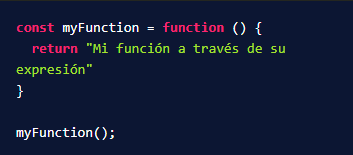
Cuando hablamos de funciones en JavaScript, tenemos dos tipos de funciones:

* **Funciones Declarativas** (function declaration / function statement)
* **Expresiones de función** (function expression / funciones anónimas).

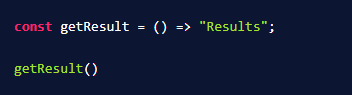
## **Declaración de funciones**



## **Expresión de funciones**



## **Función de expresión utilizando arrow function**



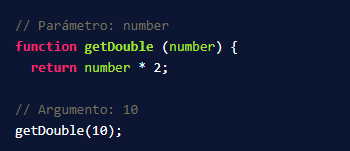
## **Diferencias**

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**.

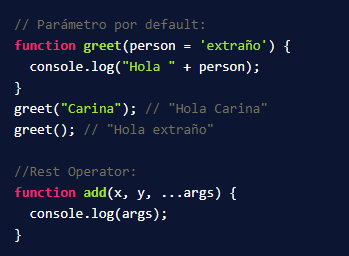
Lo que quiere decir que, con las **funciones declarativas**, podemos mandar llamar la función **antes** de que ésta sea declarada, y con la expresión de función, **no**, tendríamos que declararla primero, y después mandarla llamar.

# **Parámetros y argumentos**

* **Argumentos**: Se utilizan en las llamadas de la función para ejecutarla.
  + Se recomiendan máximo 3 argumentos por función.
* **Parámetros**: Se utilizan en la función para trabajar con elementos externos.



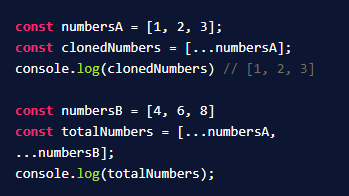
Se pueden establecer parámetros **default** que ayudan a mantener la integridad del código. ¡Aunque no hay que abusar de su uso!



## **Rest Operator o Spread Operator**

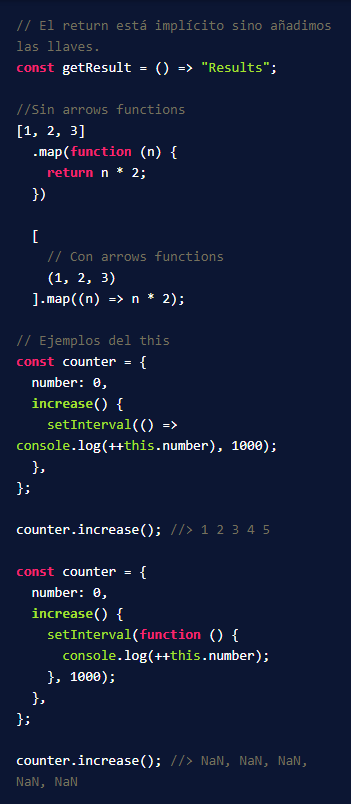
Si se usa como parámetro, permite tener en cuenta parámetros adicionales y poder utilizarlos (¡y debe ser el último parámetro!)

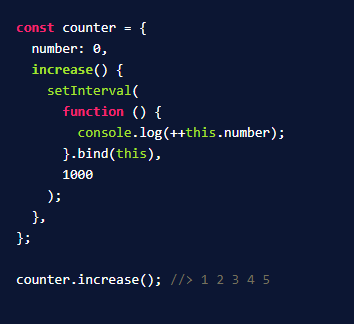
Si se usa como argumento, expande los elementos de un **array**/**objeto**.



# **Funciones de flecha y el this**

* **Una expresión de función flecha** es una alternativa compacta a una expresión de función tradicional, pero es limitada y no se puede utilizar en todas las situaciones.
* **Diferencias y limitaciones**: No tiene sus propios enlaces a **this** o **super** y no se debe usar como métodos.
* Esta funcionalidad viene con el EMAC6.
* Cuando creas una **arrow functions** su **this** queda relacionado al ámbito externo.





# **POO con ES6 (constructores, métodos y herencia)**

## **Clase**

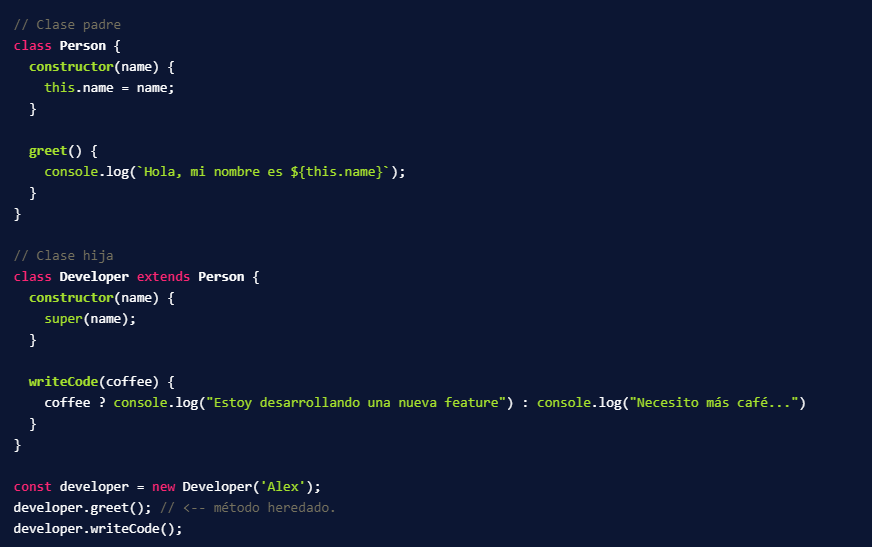
* A raíz de EMAC6 podemos usarlo.
* Antes se debía usar el prototype para poder emular clases.
* Recuerda una clase es una abstracción para representar ciertos elementos del mundo real.
* Nos ayudar organizar mucho nuestro código usando clases.



# **Herencia en JavaScript**

La herencia es cuando se heredan las propiedades, métodos y comportamientos desde una clase padre.

* Desde el EMAC6 podemos usar la palabra reservada **extends** para crear herencias.
* Anteriormente se usaba **prototype** para generar herencia.



# **Tamaño reducido (responsabilidad única)**

Este concepto viene del manifiesto **SOLID** que son las bases para la Programación POO.

* **S**: El principio de responsabilidad única: Nunca debe haber más de una razón para cambiar una clase. En otras palabras, cada clase debe tener **una sola** **responsabilidad**.
* **O**: El principio abierto-cerrado: Las entidades de software, deben estar abiertas a la extensión, pero cerradas a la modificación.
* **L**: El principio de sustitución de Liskov: Las funciones que usan punteros o referencias a clases base deben poder usar objetos de clases derivadas sin saberlo.
* **I**: El principio de segregación de interfaces: Muchas interfaces específicas del cliente son mejores que una interfaz de propósito general.
* **D**: El principio de inversión de dependencia: Depende de abstracciones, no de concreciones.

# **Organización de clases**

Los códigos de las clases se deben organizar de la siguiente manera y en el siguiente orden:

* **Variables**: Son conocidas como **propiedades** en el mundo POO, son todos esos elementos donde vamos a guardar información.
* **Constantes**: Son todos esos elementos que no van a cambiar su valor, también son conocidas como **propiedades**.
* **Variables estáticas**: Son elementos que pueden o no estar en la clase.
* **Métodos**: Tus métodos son tus funciones, solo que en POO se le reconoce así, no olvides que dentro de una clase los métodos deben ser ordenados de **mayor** **importancia** a **menor importancia**. Lo ordenas de la siguiente manera, creas una clase y esta clase tiene un método funcional y los otros métodos que se crean es para apoyar el primer método.
* **Funciones estáticas**: Se le dicen estáticas, ya que estas no deben ser heredadas a otra clase, solo se usan en la clase principal.
* **Getters y setters**: Estas permiten alterar y obtener alguna propiedad de tu clase. Recuerda esto se usa para no poder acceder directamente a la propiedad, a pesar que javascript si lo permite, pero **NO** es buena práctica.

# **Cuando usar comentarios**

* El comentario no debe explicar:
  + ❌ El ¿**Qué**? ⇒ Porque el código debe ser Autodescriptivo.
  + ❌ El ¿**Cómo**? ⇒ El código debe ser Autoexplicativo.
  + ✅ Sino el ¿**Por qué**? ⇒ Comentar explicando el por que se agrego tal librería, por que se tomo la decisión de resolver el problema de cierta forma.

(Kernighan, No comentes código mal escrito, reescríbelo.)

* Si bien, al usar correctamente la nomenclatura, el ordenamiento de nuestras variables, funciones y clases y respetar el principio de responsabilidad única, va a hacer que nuestro código sea mucho más fácil de entender, no significa que se deba dejar de añadir comentarios. Por ejemplo, añadir un comentario sobre alguna librería externa que se haya utilizado.

## **¿Se debe o no utilizar comentarios?**

La respuesta es un gran **depende...**

# **Formato coherente (codear en equipos)**

## **Reglas para trabajar en equipo**

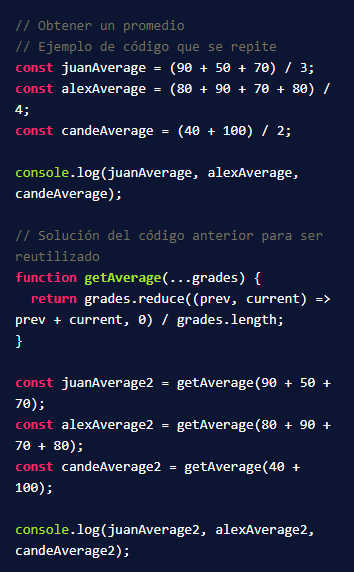
Para tener un formato coherente es recomendable:

* **Problemas similares, soluciones similares**: Seguir los mismos patrones de soluciones.
* **Densidad, apertura y distancia vertical**: Todo lo que esté relacionado, mantenlo junto en las líneas de código.
* **Lo más importante va primero**: De la funcionalidad principal se van a llamar otras funciones o métodos y se desglosarán a partir de ahí, es más intuitivo.
* **Indentación**: Respetar mismos espacios de tabulación (2 o 4).

# **Principio DRY**

DRY: **D**on’t **r**epeat **y**ourself (No te repitas a ti mismo).

No repetir código, sino extraerlo a una **clase** o **función** para poder reutilizarlo.



# **Notación big O**

La **Big O** **notation** o también conocida como **la notación Big O** es la expresión matemática de cuánto se tarda en ejecutar un algoritmo en función de la **longitud de** **entrada**, normalmente hablando del **peor de los casos**.

En la práctica se utiliza Big O para clasificar los algoritmos en función de cómo responden a los cambios en el tamaño de la **entrada**, por lo que los algoritmos con la misma tasa de crecimiento se representan con la misma notación. El uso de la letra **O** es porque la tasa de crecimiento de una función también se llama orden de la función.

Conocer Big O ayuda y facilita tu trabajo como desarrollador/a al ser consciente de la eficiencia de un algoritmo que se traduce en poder crear aplicaciones con un buen rendimiento.

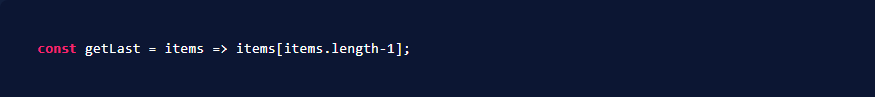
Ahora exploremos los tipos más comunes de Big O al utilizar JavaScript **ES6+**.

## **Algoritmo de tiempo constante: O(1)**

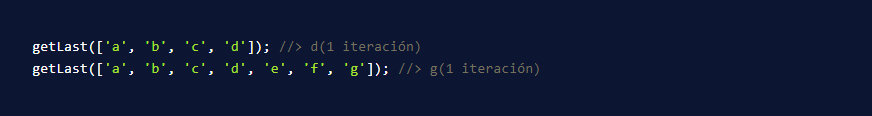
Este algoritmo es de orden uno y en este orden, independientemente de la complejidad, o sea, el número de elementos, **el tiempo es constante**.

Esto se puede observar en los algoritmos que devuelven un elemento en una posición ya conocida de un array sin importar el tipo o la longitud.

Código



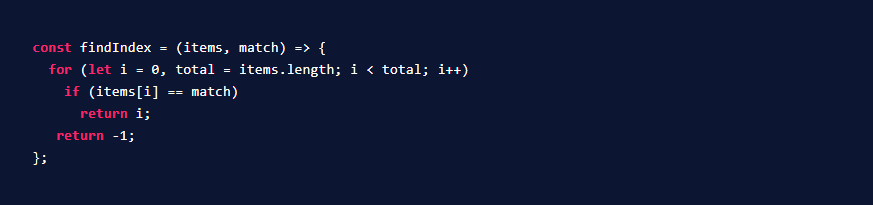
Implementación



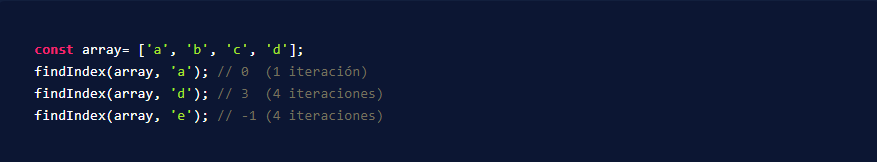
## **Algoritmo de tiempo lineal: O(N)**

En este tipo de algoritmos, el tiempo en el peor de los casos **crece a la par que el número de elementos**. Es decir, para N elementos se necesitarán N iteraciones.

Código



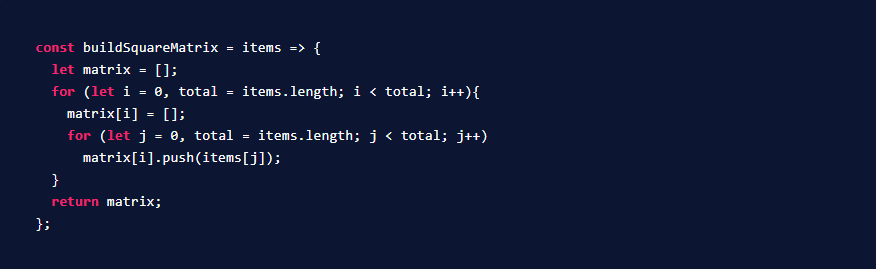
Implementación



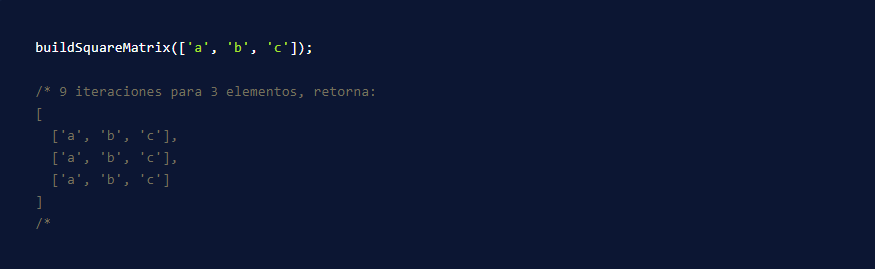
## **Algoritmo de tiempo cuadrático: O(N²)**

En este tipo de algoritmo, el tiempo en el peor de los casos **es el cuadrado del número de entradas**. Esto quiere decir que el tiempo crece **exponencialmente** en relación con el número de entradas.

Código



Implementación



## **Algoritmo de tiempo logarítmico: O(n log n)**

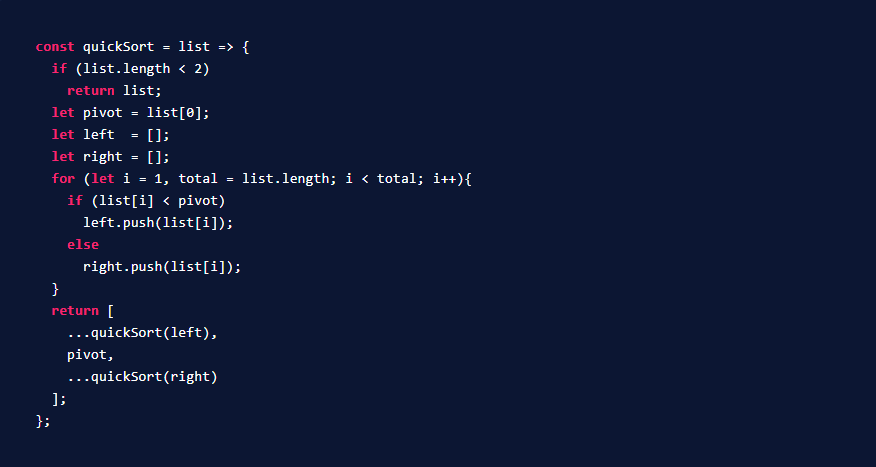
Si se trabaja con algoritmos de **búsqueda** y **ordenación**, el enfoque suele buscar ser más eficiente cuando se trata de grandes colecciones. En lugar de buscar los elementos uno por uno, se divide los datos y se descartan muchos datos en cada iteración, por lo general la mitad o log base 2.

Si consideramos que estamos un orden **log base 2**, se podría idealmente encontrar un elemento específico en una colección de un millón de elementos con menos de 20 iteraciones, si se escala el tamaño de la colección a mil millones, entonces se requerían menos de 30 iteraciones.

Si eres un/a entusiasta del big data, entonces será fácil que imagines la ventaja que tienen este tipo de algoritmos, ya que cuanto mayor sea la colección, mayor será la eficiencia relativa que proporcionen.

Entre estos algoritmos, el más popular que podemos encontrar es el de **Quicksort**, el cual puede utilizarse para encontrar un elemento específico y ordenar una lista de forma muy eficiente.

Código



Implementación

