**JavaScript Profesional**

**Tabla de Contenido**

* [¿Como llegá un script al navegador?](https://github.com/JasanHdz/javascript-professional/tree/master/notes#como-lleg%C3%A1-un-script-al-navegador)
* [Scope](https://github.com/JasanHdz/javascript-professional/tree/master/notes#scope)
* [Clouseres](https://github.com/JasanHdz/javascript-professional/tree/master/notes#clouseres)
* [¿Quién es This?](https://github.com/JasanHdz/javascript-professional/tree/master/notes#qui%C3%A9n-es-this)
* [Métodos Call y Apply](https://github.com/JasanHdz/javascript-professional/tree/master/notes#m%C3%A9todos-call-y-apply)
  + [Evitar confundir Apply y Call](https://github.com/JasanHdz/javascript-professional/tree/master/notes#evitar-confundir-apply-y-call)
* [Bind](https://github.com/JasanHdz/javascript-professional/tree/master/notes#bind)
* [Diferencias Apply, Call y Bind](https://github.com/JasanHdz/javascript-professional/tree/master/notes#diferencias-apply-call-y-bind)
* [Prototype](https://github.com/JasanHdz/javascript-professional/tree/master/notes#prototype)
* [Object create](https://github.com/JasanHdz/javascript-professional/tree/master/notes#object-create)
* [Herencia Prototipal](https://github.com/JasanHdz/javascript-professional/tree/master/notes#herencia-prototipal)
* [¿Como funciona Javascript?](https://github.com/JasanHdz/javascript-professional/tree/master/notes#como-funciona-javascript)
  + [Analizador y Abstract Syntax Tree](https://github.com/JasanHdz/javascript-professional/tree/master/notes#analizador-y-abstract-syntax-tree)
  + [Fallo en el Parset](https://github.com/JasanHdz/javascript-professional/tree/master/notes#fallo-en-el-parset)
  + [Eager Parsing (Parser de V8)](https://github.com/JasanHdz/javascript-professional/tree/master/notes#eager-parsing-parser-de-v8))
  + [Lazy Parsing](https://github.com/JasanHdz/javascript-professional/tree/master/notes#lazy-parsing)
  + [Tokens](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tokens)
  + [Abstract Syntax Tree](https://github.com/JasanHdz/javascript-professional/tree/master/notes#abstract-syntax-tree)
* [Abstract Syntax Tree en Práctica](https://github.com/JasanHdz/javascript-professional/tree/master/notes#abstract-syntax-tree-en-pr%C3%A1ctica)
* [Como funciona el Javascript Engine](https://github.com/JasanHdz/javascript-professional/tree/master/notes#como-funciona-el-javascript-engine)
  + [Bytecode vs Machine Code](https://github.com/JasanHdz/javascript-professional/tree/master/notes#bytecode-vs-machine-code)
  + [SpiderMonkey vs V8](https://github.com/JasanHdz/javascript-professional/tree/master/notes#spidermonkey-vs-v8)
* [Event Loop](https://github.com/JasanHdz/javascript-professional/tree/master/notes#event-loop)
  + [EventLoop con Asincronia](https://github.com/JasanHdz/javascript-professional/tree/master/notes#eventloop-con-asincronia)
  + [Task Queue](https://github.com/JasanHdz/javascript-professional/tree/master/notes#task-queue)
  + [Event Loop con Promesas](https://github.com/JasanHdz/javascript-professional/tree/master/notes#event-loop-con-promesas)
* [Promesas](https://github.com/JasanHdz/javascript-professional/tree/master/notes#promesas)
* [Getters y Setters](https://github.com/JasanHdz/javascript-professional/tree/master/notes#getters-y-setters)
* [Proxy](https://github.com/JasanHdz/javascript-professional/tree/master/notes#proxy)
* [Generadores](https://github.com/JasanHdz/javascript-professional/tree/master/notes#generadores)
* [Como cancelar peticiones Fetch](https://github.com/JasanHdz/javascript-professional/tree/master/notes#como-cancelar-peticiones-fetch)
* [Intersection Observer API](https://github.com/JasanHdz/javascript-professional/tree/master/notes#intersection-observer-api)
  + [Creando un Intersection Observer](https://github.com/JasanHdz/javascript-professional/tree/master/notes#creando-un-intersection-observer)
  + [Opciones de Intersection observer](https://github.com/JasanHdz/javascript-professional/tree/master/notes#opciones-de-intersection-observer)
  + [Determinando un elemento para ser observado](https://github.com/JasanHdz/javascript-professional/tree/master/notes#determinando-un-elemento-para-ser-observado)
* [Creación de Plugin para IntersectionObserver de nuestro videoplayer](https://github.com/JasanHdz/javascript-professional/tree/master/notes#creaci%C3%B3n-de-plugin-para-intersectionobserver-de-nuestro-videoplayer)
* [Visibility Change](https://github.com/JasanHdz/javascript-professional/tree/master/notes#visibility-change)
* [Service worker](https://github.com/JasanHdz/javascript-professional/tree/master/notes#service-worker)
* [Typescript](https://github.com/JasanHdz/javascript-professional/tree/master/notes#typescript)
  + [Tipos básicos en Typescript](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tipos-b%C3%A1sicos-en-typescript)
  + [Funciones en Typescript](https://github.com/JasanHdz/javascript-professional/tree/master/notes#funciones-en-typescript)
  + [Interfaces en Typescript](https://github.com/JasanHdz/javascript-professional/tree/master/notes#interfaces-en-typescript)
  + [Clases en Typescript](https://github.com/JasanHdz/javascript-professional/tree/master/notes#clases-en-typescript)
  + [Herencia Typescript](https://github.com/JasanHdz/javascript-professional/tree/master/notes#herencia-typescript)
* [Modificadores de Acceso en Typescript](https://github.com/JasanHdz/javascript-professional/tree/master/notes#modificadores-de-acceso-en-typescript)
  + [Público por defecto](https://github.com/JasanHdz/javascript-professional/tree/master/notes#p%C3%BAblico-por-defecto)
  + [Private](https://github.com/JasanHdz/javascript-professional/tree/master/notes#private)
  + [Protected](https://github.com/JasanHdz/javascript-professional/tree/master/notes#protected)
* [Convertir Proyecto a Typescript](https://github.com/JasanHdz/javascript-professional/tree/master/notes#convertir-proyecto-a-typescript)
  + [Refactorización](https://github.com/JasanHdz/javascript-professional/tree/master/notes#refactorizaci%C3%B3n)
* [Patrones de diseño](https://github.com/JasanHdz/javascript-professional/tree/master/notes#patrones-de-dise%C3%B1o)
* [Objetivos de los patrones de diseño](https://github.com/JasanHdz/javascript-professional/tree/master/notes#objetivos-de-los-patrones-de-dise%C3%B1o)
  + [Historia sobre Patrones de diseño](https://github.com/JasanHdz/javascript-professional/tree/master/notes#historia-sobre-patrones-de-dise%C3%B1o)
  + [Beneficios de utilizar patrones de diseño](https://github.com/JasanHdz/javascript-professional/tree/master/notes#beneficios-de-utilizar-patrones-de-dise%C3%B1o)
  + [Desventajas de utilizar patrones de diseño](https://github.com/JasanHdz/javascript-professional/tree/master/notes#desventajas-de-utilizar-patrones-de-dise%C3%B1o)
* [Categorias de patrones de diseño](https://github.com/JasanHdz/javascript-professional/tree/master/notes#categorias-de-patrones-de-dise%C3%B1o)
  + [Patrones Creacionales](https://github.com/JasanHdz/javascript-professional/tree/master/notes#patrones-creacionales)
  + [Patrones estructurales](https://github.com/JasanHdz/javascript-professional/tree/master/notes#patrones-estructurales)
  + [Patrones de comportamiento](https://github.com/JasanHdz/javascript-professional/tree/master/notes#patrones-de-comportamiento)
* [Patrón Singleton y casos de uso](https://github.com/JasanHdz/javascript-professional/tree/master/notes#patr%C3%B3n-singleton-y-casos-de-uso)
* [Implementación del patrón Singleton con Typescript](https://github.com/JasanHdz/javascript-professional/tree/master/notes#implementaci%C3%B3n-del-patr%C3%B3n-singleton-con-typescript)
  + [Es genial con Typescript](https://github.com/JasanHdz/javascript-professional/tree/master/notes#es-genial-con-typescript)
  + [Observer (patrón de diseño)](https://github.com/JasanHdz/javascript-professional/tree/master/notes#observer-patr%C3%B3n-de-dise%C3%B1o)
  + [Objetivo](https://github.com/JasanHdz/javascript-professional/tree/master/notes#objetivo)
  + [Motivación](https://github.com/JasanHdz/javascript-professional/tree/master/notes#motivaci%C3%B3n)
  + [Participantes](https://github.com/JasanHdz/javascript-professional/tree/master/notes#participantes)
* [Implicaciones sobre Observer](https://github.com/JasanHdz/javascript-professional/tree/master/notes#implicaciones-sobre-observer)
* [Implementación del patrón Observer con Typescript](https://github.com/JasanHdz/javascript-professional/tree/master/notes#implementaci%C3%B3n-del-patr%C3%B3n-observer-con-typescript)
* [Casos de uso del patrón Observer: Redux](https://github.com/JasanHdz/javascript-professional/tree/master/notes#casos-de-uso-del-patr%C3%B3n-observer-redux)
  + [Conceptos básicos](https://github.com/JasanHdz/javascript-professional/tree/master/notes#conceptos-b%C3%A1sicos)
* [Patrón Decoratory casos de uso](https://github.com/JasanHdz/javascript-professional/tree/master/notes#patr%C3%B3n-decoratory-casos-de-uso)
  + [Motivation](https://github.com/JasanHdz/javascript-professional/tree/master/notes#decorator-motivation)
  + [Aplicabilidad](https://github.com/JasanHdz/javascript-professional/tree/master/notes#decorator-aplicabilidad)
  + [Estructura](https://github.com/JasanHdz/javascript-professional/tree/master/notes#decorator-estructura)
  + [Participantes](https://github.com/JasanHdz/javascript-professional/tree/master/notes#decorator-participantes)
  + [Colaboraciones](https://github.com/JasanHdz/javascript-professional/tree/master/notes#decorator-colaboraciones)
  + [Consecuencias](https://github.com/JasanHdz/javascript-professional/tree/master/notes#decorator-consecuencias)
  + [Implementación](https://github.com/JasanHdz/javascript-professional/tree/master/notes#decorator-implementaci%C3%B3n)

**¿Como llegá un script al navegador?**

El **DOM** es la representación que hace el navegador de un documento HTML.

El navegador interpreta el archivo HTML y cuando termina de transformarlo al DOM se dispara el evento DOMContentLoaded lo que significa que todo el documento está disponible para ser manipulado.

Todo script que carguemos en nuestra página tiene un llamado y una ejecución.

Tanto con async como defer podemos hacer llamados asíncronos pero tiene sus diferencias:

* async. Con async podemos hacer la petición de forma asíncrona y no vamos a detener la carga del DOM hasta que se haga la ejecución del código.
* defer. La petición es igual asíncrona como en el async pero va a deferir la ejecución del Javascript hasta el final de que se cargue todo el documento.
* scripts embebidos: el navgador carga linea a linea el html y cuando se encuentra un codigo entre scripts va a deter su ejecución hasta que hallá procesado todo el script.

Hay que tener en cuenta que cuando carga una página y se encuentra un script a ejecutar toda la carga se detiene. Por eso se recomienda agregar tus scripts justo antes de cerrar el body para que todo el documento esté disponible.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Scope**

El Scope o ámbito es lo que define el tiempo de vida de una variable, en que partes de nuestro código pueden ser usadas.

**Global Scope** Variables disponibles de forma global se usa la palabra var, son accesibles por todos los scripts que se cargan en la página. Aquí hay mucho riesgo de sobreescritura.

**Function Scope** Variables declaradas dentro de una función sólo visibles dentro de ella misma (incluyendo los argumentos que se pasan a la función).

**Block Scope** Variables definidas dentro de un bloque, por ejemplo variables declaradas dentro un loop while o for. Se usa let y const para declarar este tipo de variables.

**Module Scope** Cuando se denota un script de tipo module con el atributo type="module las variables son limitadas al archivo en el que están declaradas.

Esto va a declarar que esté archivo es un modulo, estó no estan en todos los navegadores pero si en los más modernos. Los valores que teníamos escritos ahora no los vamos a poder leer en consola porque el module scope está limitando el alcance

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Clouseres**

El scope cuando lo juntamos con funciones podemos lograr algo que se llamán clouseres o clausuras, para entender bien vemos un ejemplo:

// Clouseres

// printColor

let color = 'green';

function printColor() {

console.log(color);

}

Podemos observar que la variable global está en el scope global, para evitar eso lo que vamos a hacer es crear una función que se va a llamár autómaticamente

// IIFE: immediately invoked function expression

(function () {

let color = 'green';

function printColor() {

console.log(color);

}

printColor();

})();

Sacamos esté codigó del entorno global, lo pasamos a uno de una función y entonces color no cruza al scope global. Cuando tenemos está declaración y está ejecución que se declara afuera de l función interna, **estó es un Closures**. Es la combinación del scope de una función y el scope donde fue definida, donde el scope de la función es la función *IIFE* la función principal, y adentro la función que fue definida dentro de ese scopeque tiene acceso a lo que estaba afuera.

Clouseres nos va a permitir tener una funcionalidad o feture que el lenguaje no trae: variables privadas

// Clousers Jasan Hernández.

// ¿Variables privadas?

const counter1 = {

count: 3

}

// count esta en el scope Global Window

console.log(counter1.count);

// Podemos modificar su valor si quisieramos

counter1.count = 99;

console.log(counter1.count);

// Clouseres - creamos un function scope

function makeCounter(n) {

// count ya no existe en window, ahora

// solo pertenece a la función

let count = n;

return {

increase: function () { count += 1; },

decrease: function () { count -= 1},

getCount: function () { return count; },

}

}

let counter = makeCounter(7);

console.log('This count is:', counter.getCount());

console.log('This count is:', counter.increase());

console.log('This count is:', counter.decrease());

// No podemos cambiar el valor de count porque no

// está en nuestro alcance.

// ERROR FATAL

counter.count = 99;

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**¿Quién es This?**

This se refiere a un objeto. Ese objeto es el que actualmente está ejecutando un pedazó de código.

this es un concepto que tienen muchos lenguajes de programación sobre todo aquellos que son orientados a objetos, quizás si conoces un poco de Java sabes que cuando escribes *this* en una clase, this se va a referir a la instancia de esa clase cuando creas ese objeto. En JavaScript estó también es cierto pero también hay otros contextos de los cuales hay que conocer para saber en que momentos this se comporta así y en que otros momentos this tiene valores diferentes.

* Global Scope, como se comporta this con el global-scope.

// this en el scope global

console.log(`this: ${this}`);

// this en el scope de una función

function whoIsThis() {

return this;

}

console.log(whoIsThis());

Cuando llamamos a la función whoIsThis() directamente el motor de javascript le va asignar que this va ha ser window, ya que es un valor que se tiene que asignar por default, exepto si estamos usando javascript en un modo el **'strict mode'**, esté *strict mode* que puedes encender manualmente solo escribiendo una cadena que digá "use strict", con estó comienza el modo estricto.

* This en el Scope de una Función strict mode

Ahora usando el modo **strict mode** dentro de la siguiente función que también retorna **this** nos imprime que ahora es **undefine**, y esté es justo el comportamiento que sucede cuando usamos strict mode, *strict mode* nos ayuda a evitar algunos errores que le pueden pasar a cualquier otro programador pero el motor de javascript va a tratar de ayudarnos y estos errores nos van a salir temprano durante la fase de desarrollo y no en producción y un error es mejor tenerlo en desarrollo que en producción.

// this en el scope de una función strict mode

function whoIsThisStrictMode() {

"use strict";

return this;

}

// undefine

console.log(`whoIsThisStrictMode: ${whoIsThisStrictMode()}`);

* This en el contexto de un objeto

this se refiere al objeto que actualmente está ejecutando un pedazo de código, por lo tanto this va a ser todo esté objeto ejemplo:

// this en el constexto de un objeto

const person = {

name: "Gabriel",

saludar: function () {

console.log(`hola soy ${this.name}`);

}

}

console.log(person.saludar());

* This cuando sacamos una función de un objeto.

const person = {

name: "Gabriel",

saludar: function () {

console.log(`hola soy ${this.name}`);

}

}

// this cuando sacamos una función de un objeto

const accion = person.saludar;

accion();

Ahora nos aparece undefine porque acción no se está llamando dentro del contexto de un objeto simplemente se está llamado directamente similar como lo hicimos arriba en las funciones directas, por lo tanto el mensaje se mostra incompleto cuando se intenta usar this.

* This en el contexto de una clase

Las clases como tál no existen en javascript al menos no son como las clases de java o de c++ pero resulta útil llamarles clases a estas funciones especiales que llamamos con new.

Todas las funciones tienen un valor de this lo que sucede es que ese valor de this es un objeto vacío, en otras palabras this comienza siendo estó this = {}. No puedes asignar this directamente pero si puedes asignar una propiedad de this.

// This en el contexto de una Clase(Prototype);

function Person(name, lastName) {

this.name = name;

this.lastName = lastName;

}

Person.prototype.saludar = function () {

console.log(`Me llamó ${this.name} ${this.lastName}`);

}

const Jasan = new Person('Jasan', 'Hernández');

Jasan.saludar();

En el contexto de objetos que fueron instanciados de una clase, **this** se va a referir a la instancia de ese objeto no se va ha referir a Person, ni a Person.prototype.saludar, Sino que se va a referir a Jasan que es el objeto que está ejecutando a saludar.

Y cuando asignamos los valores de this.name = name lo estamos haciendo sobre la instancia, no sobre el objeto prototipal.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Métodos Call y Apply.**

This no es un valor que podemos asignar directamente, no podemos escribir this = 'valor', pero si existen unos métodos que son parte del prototipo de function *call, apply, bind*. Estos 3 métodos nos van ha ayudar a establecer cual es el this que va ha ser el contexto de la llamada a una función .

* Establece this usando 'call'

// Establece this usando 'call'

function saludar() {

console.log(`Hola soy ${this.name} ${this.lastname}`)

}

Tenemos una función que no está en el contexto de un objeto ni de niguna clase, con la ayuda de call vamos a establecer cual va a ser el this.

function saludar() {

console.log(`Hola soy ${this.name} ${this.lastname}`)

}

const richard = {

name: 'Richard',

lastname: 'Lopez'

}

saludar.call(richard)

Aquí lo que hicimos fue establecer el valor de this para la función saludar, pero aveces las funciones si reciben argumentos.

* Establece this usando 'call' y pasar argumentos a la función.

const richard = {

name: 'Richard',

lastname: 'Lopez'

}

function caminar(metros, direccion) {

console.log(`${this.name} camina ${metos} metros hacia ${direccion}`);

}

caminar.call(richard, 400, 'norte');

Lo primero es establecer el contexto del this para caminar usando call(richard) pero además tenemos algunos argumentos. Cuando estó no sucede tenemos que pasar los argumentos además de pasar el contexto.

* Establece this usando 'apply' y pasar argumentos a la función.

**apply** hace la misma funcionalidad que *call* pero sus argumentos los pasamos de una forma ligeramente diferente, así que vamos a seguir usando el ejemplo de caminar y vamos usar apply que usa el mismo argumento que tiene call que en esté caso el this será richart. Pero en lugar de pasar los argumentos separados por comas, vamos a pasarlo como **parte de un arreglo** y en ese arreglo van todos los argumentos que queramos.

const richard = {

name: 'Richard',

lastname: 'Lopez'

}

function caminar(metros, direccion) {

console.log(`${this.name} camina ${metos} metros hacia ${direccion}`);

}

caminar.apply(richard, [400, 'norte']);

// también podemos pasarle un arreglo

const valores = [200, 'sur'];

caminar.apply(richard, valores)

Es lo mismo pero nos podemos enfrentar donde hay situaciones donde es más fácil usar el call y hay lugares donde es más fácil usar el apply. Sobre todo si son valores que son una lista de valores.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Evitar confundir Apply y Call**

Call = Commas Apply = Areglo

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Bind**

bind() es parte de esté conjunto de: call, apply, bind. Sin embargo bind no va a llamar la función automáticamente sino que va a construir una nueva función, está nueva función va a tener el this que le pasamos ya integrado y cuando llamemos ha está nueva función se va a ejecutar.

// Establecer una nueva función usando bind()

function saludar() {

console.log(`Hola soy ${this.name} ${this.lastname}`)

}

const daniel = {

name: "Daniel",

lastname: "Sanchez"

}

const danielSaluda = saludar.bind(daniel);

danielSaluda();

* ¿Como le haríamos si necesitaramos pasar argumentos a está función a la que estamos bindeando?

function saludar() {

console.log(`Hola soy ${this.name} ${this.lastname}`)

}

function caminar(metros, direccion) {

console.log(`${this.name} camina ${metos} metros hacia ${direccion}`);

}

const daniel = {

name: "Daniel",

lastname: "Sanchez"

}

const danielSaluda = saludar.bind(daniel);

danielSaluda();

const danielCamina = caminar.bind(daniel);

danielCamina(1000, 'SurOeste');

Existe otra forma de pasar estos argumentos y es muy interesantes porque permite harcodear estos argumentos, y en lugar de escribirlos dentro de la nueva función los escribimos despues del **this**.

const danielCamina = caminar.bind(daniel, 1000, 'SurOeste');

danielCamina();

Esto se vuelve más interesante porque podemos guardar argumentos parciales, es decir poner argumentos en ambas funciones, tanto el funció que genera como a la llamada a la nueva función:

const danielCamina = caminar.bind(daniel, 1000);

danielCamina('SurOeste');

Está técnica se llama kurin donde guardamos parcialmente algunos argumentos y luego llenamos los demás, es una ténica funcional que es muy poderosa y que te va a permitir hacer funciones reutilizables, para contextos donde solo parcialmente esta establecido un valor.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Diferencias Apply, Call y Bind**

Call y Apply van a establecer el this y va a llamar la función inmediatamente, bind va a crear una nueva función donde this estará guardado y luego tocará guardar y ejecutar esa nueva función.

Existen veces que tenemos objetos que se parecen a otros, por ejemplo *cuando usamos un getElementByClassName* del *DOM*. Lo que nos regresa es un tipo de de objeto **nodeList**, **no es un array**, por lo tanto **no tienen** todos lo **métodos** que tiene el array como el **foreach**

DOM:

<body>

<ul>

<button class="call-to-action">Aprender</button>

<button class="call-to-action">Aprender Más</button>

<button class="call-to-action">¡Nunca pares de aprender!</button>

</ul>

<script src="call-apply-bind.js"></script>

</body>

Obteniendo los botones del Dom:

// Cuando es útil uno de estós métodos

const buttons = document.getElementsByClassName("call-to-action");

buttons.forEach(button => {

button.onclick = () => alert('Nunca pares de aprender');

});

// No es una funcion forEach porque button es NodeList

No es una funcion forEach porque button es NodeList es por eso que el código de arriba nos marcará un error.

Los nodeList se parecen a los arreglos pero no del todo, lo que si sabemos es que tienen una propiedad se llama leght y está es suficiente para hacer alguna de las operaciones de arreglos.

Ahora lo que podemos hacer es lo siguiente:

Vamos a llamar al forEach através del array porque foreach porque foreach es una propiedad del prototype que entonces le podemos cambiar el this usando call, donde le pasamos el nuevo this y ejecutamos la función por cada elemento como si usaramos un forEach de un arreglo.

// Cuando es útil uno de estós métodos

const buttons = document.getElementsByClassName("call-to-action");

// Obtenemos la función foreach de Un arreglo y le cambiamos this. para que hagá referencia a buttons y de esté modo recorrer el arreglo.

Array.prototype.forEach.call(buttons, button => {

button.onclick = () => alert('Nunca pares de aprender');

})

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Prototype**

Si hay un temaque hace que a todos los desarrolladores de javascript nos explote la cabeza de lo dificíl que es, es este: **Prototype** es un concepto casi único de javascript que se aparte de como normalmente manejamos lenguajes y objetos en lenguajes de programación, las clases son como un plano en Java lo escribimos y luego insanciamos objetos, si queremos crear una clase que hereda de otra clase y volvemos a instanciar otros objetos, pero en javascript todos son objetos.

* Ejemplo de objeto comú y corriente:

// Un objeto común y corriente

const zelda = {

name: "Zelda"

}

zelda.saludar = function () {

console.log(`Hola soy ${this.name}`);

}

zelda.saludar();

const link = {

name: "Link"

}

link.saludar = function () {

console.log(`Hola soy ${this.name}`);

}

link.saludar();

Esto es algó inficiente, estamos creando una función por cada uno de los 2 objetos y además los objetos son similares y los tenemos que escribir a mano.

* Escribiendo una función que haga más eficiente a nuestro objeto.

// Seamos un poco más eficientes

function Hero(name) {

const hero = {

name: name,

}

hero.saludar = function () {

console.log(`Hola soy ${this.name}`);

}

return hero;

}

const zelda = Hero('Zelda');

zelda.saludar();

const link = Hero("Link");

link.saludar();

Aquí hay un poco de ineficiencia, ya que está función saludar la estamos definiendo cada vez llamamos a Hero.

Podemos ser un poco más eficientes y tener un método que tenga la colección de métodos para los Heros

* Aun podemos mejorar más y evitar tener que crear la misma función cada vez

Lo logramos creando un objeto externo llamado **heroMethods** el cual tiene una función saludar como propiedad, la cual solo se está definiedo una sola vez. Y a nuestro objeto hero en su método saludar solo le pasamos la referencia de el objeto externo *heroMethods.saludar*

// Aun podemos mejorar más y evitar tener que crear la misma función cada vez

const heroMethods = {

saludar: function () {

console.log(`Me llamó ${this.name}`);

}

}

function Hero(name) {

const hero = {

name: name,

}

hero.saludar = heroMethods.saludar;

return hero;

}

const zelda = Hero('Zelda');

zelda.saludar();

const link = Hero("Link");

link.saludar();

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Object create**

Recibe un objeto y lo que hace es crear un nuevo objeto como lo dice su nombre

const nuevoObjeto = Object.create({});

Esté nuevo objeto va a contender todas las propiedades que esté objeto tiene definido, esto la da algó parecido a 'super poderes'. Usando el ejemplo de prototypes quedaría de a siguiente manera:

// Object create

const heroMethods = {

saludar: function () {

console.log(`Soy un super heroe ${this.name}`);

}

}

function Hero(name) {

const hero = Object.create(heroMethods);

hero.name = name;

return hero;

}

const zelda = Hero('Zelda');

zelda.saludar();

const link = Hero("Link");

link.saludar();

Analisemos lo que esta haciendo object create, porque va más allá de copiar propiedades de un objeto a uno nuevo. Si nostros ejecutamos al objeto zelda y a heroMethods nos aparecerán 2 objetos tal cual como los habiamos definido.

Si intentamos crear un nuevo objeto directo desde la consola del navegador, pasará algo interesante.

zelda

heroMethods

const newHero = Object.create(heroMethods);

Cuando nosotros creamos un nuevo objeto con Object.create nuestro objeto aparentemente se mostrará vacío, pero si nosotros intentamos ingresar a una propiedad del objeto si aparecerá definido. Estó es gracias a un objeto que si se nos imprimio el cual aparece dentro del nuevo objeto con un color azul desvanecido el cual se llama ***proto***.

Nostros accedemos a newHero.saludar sin estar definido dentro del objeto mismo, esto sucede gracias a la **herencia prototipal**. Por ahora basta con que entendamos que **todo lo que estaba en heroMethods** paso al ***proto*** de **newHero**

Modificaremos el objet Hero agregando la función saludar de heroMethods porque la propiedad saludar le debería pertenecer a Hero ya que es una función que esta ocupando el objeto. El lugar para hacer estó que se nos recomienda es hacerlo dentro de su método prototype, inicialmente prototype es un objeto vacío. Pero como es un objeto le podemos añadir nuevas propiedades

// Métodos de Hero dentro de Hero

function Hero(name) {

const hero = Object.create(Hero.prototype);

hero.name = name;

return hero;

}

Hero.prototype.saludar = function () {

console.log(`Soy una super heroina ${this.name}`);

}

const zelda = Hero('Zelda');

zelda.saludar();

const link = Hero("Link");

link.saludar();

* new es un atajo (azucar sintactica) para llevar Hero.prototype al objeto

Seguro haz usado un keyword que se llamá **new**. Desde ahora te advierto que new es un atajo lo que le llamamos azúcar sintactica, es decir, es algó que le añadimos al lenguaje para facilitar algunos procesos pero son cosas que ya podemos hacer y justo tiene que ver con Object.create.

Primero comenzamos añadiendo new cada vez que queremos instanciar un nuevo objeto.

const zelda = new Hero('Zelda');

zelda.saludar();

new es lo mismo que hacer estó:

function Hero(name) {

const hero = Object.create(Hero.prototype);

hero.name = name;

return hero;

}

const zelda = new Hero('Zelda');

zelda.saludar();

Cada ve que usamos **new** esté atajo de Object.create ocurre autómaticamente:

function Hero(name) {

// Object.create Ocurre autómaticamente cada vez que utilizamos new y no necesitamos ponerlo

// const hero = Object.create(Hero.prototype);

this = Object.create(Hero.prototype);

hero.name = name;

return this;

}

const zelda = new Hero('Zelda');

zelda.saludar();

La regla que usa new cuando hace el Object.create es que siempre va a sacar el prototype de lo que sea el constructor, si despues de new dice hero new Hero

Lo que hará será un Hero.prototype no existe otro. Y en lugar de guardarlo en hero sirve imaginarnos que lo va a guardar en *this*. Esté this se va a inicializar, no lo haremos nosotros directamente, lo está haciendo el lenguaje, entonces si se vale y en lugar de decir hero.name hacemos: this.name. Siempre la clase o función clase lo que hará siempre será retornarnos this implicitamente.

Lo que acabamos de hacer es: tienes un objeto simple donde nos las inventamos para crear nuevos objetos y fuimos arreglando algunas inficiencias hasta llegar a Object.create y new que solo es sugar-sintaxs para Object.create. Esto es lo que ocurre cada vez que construimos un constructor en una función o cuando escribimos Hero.prototype.

Hoy en dia hay nuevas formas de hacerlo, usando el keyword class internamente, definimos los métodos, definimos un constructor, todo estó es tambipen sugar-sintaxs que vale la pena y es muy importante entender como funciona nativamente para ver como es que nuestro lenguaje se está comportando.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Herencia Prototipal**

Por default los objetos en JavaScript tienen cómo prototipo a Object que es el punto de partida de todos los objetos, es el prototipo padre. Object es la raíz de todo, por lo tanto tiene un prototipo padre undefined.

Cuando se llama a una función o variable que no se encuentra en el mismo objeto que la llamó, se busca en toda la prototype chain hasta encontrarla o regresar undefined.

La función hasOwnProperty sirve para verificar si una propiedad es parte del objeto o si viene heredada desde su prototype chain.

function Hero(name) {

this.name = name;

}

Hero.prototype.saludar = function () {

console.log(`Hola, soy ${this.name}`);

}

const zelda = new Hero('Zelda');

// Propiedades de la instancia

console.log(`New: ${zelda.name} `);

// Propiedades de la "clase".

console.log(`saludar: ${zelda.saludar} `);

// Propiedades heredadas de ej: toString

console.log('toString: ', zelda.toString);

// hasOwnProperty (de dónde sale toString o esto?)

console.log('zelda.hasOwnProperty("name"): ', zelda.hasOwnProperty('name'));

console.log('zelda.hasOwnProperty("saludar"): ', zelda.hasOwnProperty('saludar'));

Proto **proto** es el lugar donde calleron todas las propiedades que estaban en Hero.prototype, si seguimos por ahí, hay otro **proto** que viene desde Object y aquí esta hasOwnProperty y toString.

Nosotros podemos escribrir zelda.\_proto\_ y va funcionar, pero esa propiedad **proro** es algo que puede variar no es algo que está especificado en el lenguaje es un detalle de implementación que ponen los browsers o node o cualquier otro entorno. La forma correcta o sugerida es utilizando **Object.getPrototypeOf(zelda)**, el cual nos va a regresar ese prototipo.

si comparamos:

const prototypeOfZelda = Object.getPrototypeOf(zelda);

prototypeOfZelda === Hero.prototype;

// true

Esté objeto es identico al que está a prototypeOfZelda, no solo porque tiene el mismo contenido sino porque es el mismo objeto. Cuando utilizamos tripe = es porque estamos comparando la referencia en memoria es decir: el mismo lugar en memoria. Si esto es cierto nosotros podemos hacer lo siguiente:

const prototypeOfZelda = Object.getPrototypeOf(zelda);

prototypeOfZelda === Hero.prototype;

// true

Hero.prototype.fight = function () {console.log("FIGHT")}

zelda.fight();

// Ejecuta el menssage

Es una referencia desde zelda hasta prototype, el lenguaje lo está encadenando, porque si vamos a zelda.hasOwnProperty("fight"); es false porque no le pertenece, le pertence al prototype.

Lo que hace el lenguaje para encontrarlo es preguntar: -- ¿zelda.fight existe? -- No existe -- Okay, vamos a buscarlo en su proto -- ¿zelda.**proto**.fight existe? -- Si -- Ejecutamos!!

**Object** es el punto de partida de todos los objetos en javascript esto incluye las funciones que también son objetos en javascript

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**¿Como funciona Javascript?**

¿Qué pasa cuando llega un script al navegador?

Comienza un proceso ejecutado por el motor de javascript que va a tomar ese código y lo va analizar y lo va a deconstruir y reconstruir, despues ejecuta y luego lo optimiza.

Acerca de la web...

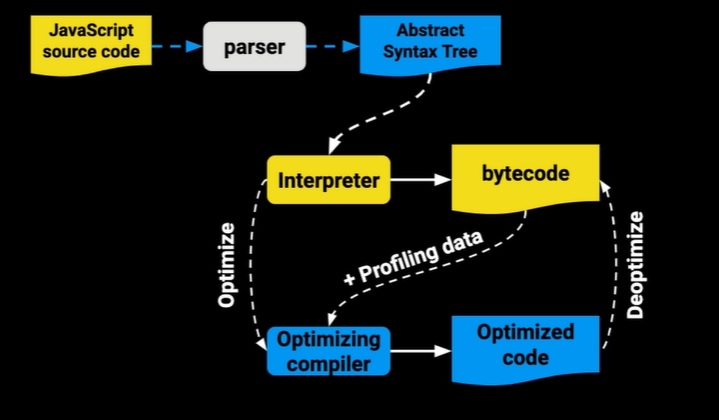
La web no siempre ha sido igual y cuando llegó javascript llegó con **Netscape**. En ese entonces lo que haciamos eran cosas muy simples donde abriamos un cajon donde el usuario ingresaba información y luego el programa lo leía y lo seguia ejecutando, pero eran cosas muy simples, javascript lo que haciamos es que ibamos leyendo linea por linea y lo ibamos interpretando un paso a la vez, y eso sigue siendo cierto pero de una manera totalmente nueva, y está nueva manera llegó con google chrome.

Google una compañía con productos sumamente complejos como google maps, necesitaba un navegador que pudiera ejecutar las cosas más eficientemente y por eso reinvento como funcionaban los motores de javascript en resumen estó es lo que hacen:

* Reciben un código fuente lo analizan y lo deconstruyen en el AST(Abstract Syntax Tree). Esté AST lo toma un interpretador y lo convierte en bytecode que es lo que se va a ejecutar, el programa va a empezar ejecutando bytecode y eventualmente va a tener suficiente información para ejecutar machine code que es el mismo código pero automizado.

1. recibe código fuente,
2. Parsea código y produce Abstract Syntax Tree(AST)
3. Se compila a bytecode y se ejecuta.
4. Se optimiza a machine code y se remplaza el código base.

Mirando un grafo con este orden usando V8.

[](https://github.com/JasanHdz/javascript-professional/blob/master/assets/v8.jpg)

**Javascript source code** pasa por el **parset** donde obtenesmos el AST, despues el **AST** se lo damos al **interpretador** que va a producir **bytecode** ahí es donde comienza a ejecutarse nuestro programa, bytecode es un lenguaje menor nivel pero va a permitir que se ejecute más rápido, **mientras se va ejecutando** hay un proceso que se llama el **profiling data** que va a estar analizando la ejecución va a encontrar los puntos donde el programa se puede optimizar y eventualemente va a producir el machine code, esto hace el *optimizing compiler*\* el **compilador de optimizaciones** y despues tenemos el **Optimized code**.

Hay veces en que estas asunciones fallan ahí deoptimize(deoptimizamos) el código.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Analizador y Abstract Syntax Tree**

**Parser:** un parser va a tomar tu código fuente y lo va a leer, pero lo que tu estas escribiendo no es la que la computadora entiende así tal cual como lo escribiste, primero lo tiene que descomponer y esa descomposicion o esos pedazos que van a salir se llamán **tokens**. Tokens identifican que let es una palabra clave o new es una palabra clave, que el simbolo de + es un operador y que lo que está aun lado y al otro o quizas son número u otro tipo de variable y una vez que tenemos esos tokens es cuando vamos a hacer el AST(Abstract Syntax Tree).

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Fallo en el Parset**

Que paso si el parser esta analizando tu programa y de momento hay algó que no hace sentido, justo es cuando ocurre un **sintax error**

Un **SyntaxError** es lanzado cuando el motor de javascript se encuentra con parte de código que no forman parte de la sintaxis del lenguaje al momento de analizar código.

El proceso de parsing es muy importante que se haga bien.

**Google dice:**

* Parsing es 15% - 20% del proceso de ejecución.
* La mayoría del Javascript en una página nunca se ejecuta.
* Esto hace que **bundling y code splitting** sea muy importante!

La tercer parte significa que el código lo tenemos que empaquetar de una forma eficiente donde hay unos archivos de código separados lógicamente y que solamente vamos a cargar cuando sea necesario, esta es una modalidad que se esta volviendo muy frecuentemente de una solo página **Single Pages Apps**.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Eager Parsing (Parser de V8):**

Cuando sea hace este parsing vamos a encontrar todos los errores de sintaxis en el código que se esta analizando y vamos a crear el AST. Que siemplemente es un árbol o arquitectura en forma de árbol que representa tu programa, y además va ha construir los scopes. En este momento vamos a saber que variables se pueden leer en que partes del código

* Encuentrar errores de sintaxis
* Crea el AST
* Construye Scopes

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Lazy Parsing**

Cuando hacemos esto parsing estamos retrazando alguna parte del código porque no hace falta analizarla y puede esperar, esto tiene una ventaja y es que es el doble de rápido por lo tanto si las cosas tardaban 20% si logramos que ocurra mucho lazy parsing podemos retrazar ese analisis, una consecuencia es que el AST no se construye y los Scopes se construyen parcialmente.

* Doble de rápido que el eager parser
* No crea el AST
* Construye los scopes parcialmente.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Tokens**

Accediendo al siguiente enlce podemos ver con ejemplos la manera en como una sentencia de javascript se transpila a un token. [url](https://esprima.org/demo/parse.html) o ingresando a [esprima](https://esprima.org/)

Parser produces the (beautiful) syntax tree

// Life, Universe, and Everything

var answer = "hola";

[

{

"type": "Keyword",

"value": "var"

},

{

"type": "Identifier",

"value": "answer"

},

{

"type": "Punctuator",

"value": "="

},

{

"type": "String",

"value": "\"hola\""

},

{

"type": "Punctuator",

"value": ";"

}

]

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Abstract Syntax Tree**

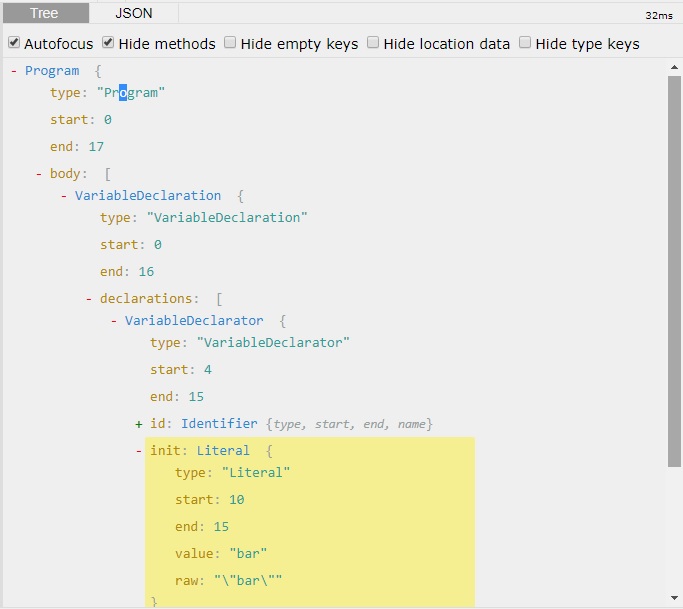
El AST es un gráfo (estructura en forma de árbol). Donde vamos a tener una raíz que será nuestro programa y lo vamos a ir descomponiendo en partes, todo esto lo vamos a poder hacer siguiendo los tokens que produce el parser, esto se usa en muchisimos sitios, no solo para ejecutar un programa javascript, tembién lo usamos para transformar código de una forma a otra que es como lo que hace **babel** o **priged**

Se usa en:

* Javascript Engine
* Bundlers: Webpack, Rollup, Parcel
* Transpilers: Babel
* Linters: ESLint, Prettify
* Type Checkers: TypeScript, Flow
* Syntax Highlighters

Demo de AST

let foo = "bar";

[](https://github.com/JasanHdz/javascript-professional/blob/master/assets/demoast.jpg)

También puede construir tu propio ejemplo en [AST Explorer](https://astexplorer.net/)

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Abstract Syntax Tree en Práctica**

Usemos el AST para crear una regla para Eslint, que analiza estáticamente nuestro código para ver si encuentra errores o si hay que leventar warnings porque estamos violando alguna regla de estilo o simplemente nuestro código esta vilolando la sintaxis del lenguaje, muchas de estas reglas ya vienen con Eslint pero también podemos desarrollar nuevas, para hacer eso vamos a desarrollar una herramienta que se llama [AST Explorer](https://astexplorer.net/) En esté explorer tenemos que asegurarnos que la configuración es la correcta, si aquí no dice babel-eslint vamos a seleccionarla y vamos a ver como vamos a transformar el código. En esté caso lo vamos a procesar y vamos usar la última versión de eslint v4.

En la parte de abajo del lado izquierdo vamos a escribir esa regla y en la parte de abajo del lado derecho vamos a ver como esa regla esta funcionando.

La regla la vamos a definir dentro de una función y el nombre de esa función va a ser el nombre del nodo que queremos corregir. Si vamos al AST vemos que está parte del código trabaja con declaraciones de variables, cuando hay una declaración la queremos entender y si encontramos que el nombre esta en lowerCase que lo que hace es guardar un número, lo queremos corregir, así que vamos a corregir estó y copiaremos en esté caso el nombre del nodo "VariableDeclaration", está función es la que va a recibir un nodo.

Código al que queremos establecer reglas:

const pi = 3.1415;

const halft\_pi = 1.356;

// Variables constantes

// Variables que guarden un número

// El nombre de la variable tiene que estar en UPPER CASE

Reglas que establecemos para el código usando EsLint

export default function(context) {

return {

VariableDeclaration(node) {

// Tipo de variable const

if(node.kind === "const") {

const declaration = node.declarations[0];

// asegurarnos que el valor es un número

if(typeof declaration.init.value === "number") {

if(declaration.id.name !== declaration.id.name.toUpperCase()) {

context.report({

node: declaration.id,

message: "El nombre de la constante debe estar en Mayúsculas",

fix: function(fixer) {

return fixer.replaceText(declaration.id, declaration.id.name.toUpperCase());

}

})

}

}

}

}

};

};

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Como funciona el Javascript Engine**

Despues de que el parser hizo su trabajo y nos dio el AST continua el interpretador, el compilador para obtener código optimizado.

* Recibe código fuente
* Parsea el código y produce un Abstract Syntax Tree(AST)
* Se compila a bytecode
* Se optimiza a machine code y se remplaza el código base.

Cuando el bytecode se está ejecutando hay un observador, un programa que observa el bytecode y va a estar tomando notas, cuando tiene suficientes notas es capaz de tomar decisiones para optimizar tu código y así obtener código optimizado, si alguna de estas observaciones en el futuro resulta ser falsa y que todavía no se cumple, no pasa nada, tu código va a regresar a una versión ya optmizada donde va a seguir corriendo aunque no tan rápido.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Bytecode vs Machine Code**

El bytecode es algo parecido a assembly y assembly es un lenguaje de muy bajo nivel pero que aún se puede escribir, son palabras clave que le dicen al procesador que hacer. Es similar a Assembly excepto que en lugar de operar sobre un procesador va a operar sobre algo que se llama la virtual machine que es un programa que ejecuta bytecode.

El Machine Code es lo más bajo nivel, ni tu ni yo queremos escribir machine code a mano porque se trata básicamente de ceros y de unos, es código binario, pero los procesadores si lo necesitan así, es el código que les va a llegar a ellos y va a volar y es muy rápido porque el machine code no hay que traducirlo, ya esta traducido.

Cuando el motor de Javascript V8 que es el que ocupa chrome y node, produce esté código va a crearlo a machine code ya el bytecode que corre la máquina virtual no se va a ocupar.

El profiler es un programa que esta en medio del bytecode y el optimizador, este programa lo que hace es que toma una onbservación de la ejecución del bytecode y cuando ve que las funciones se estan llamando igual, una vez tras otra y todas la llamadas son iguales, puede hacer unas optimizaciones, ejemplo:

function add(a, b) {

return a + b;

}

for (let i = 0; i < 1000; i++) {

add(i, i);

}

Cuando el código se halla ejecutado ejemplo 50 veces, el código va a empezar a ponerse caliente, luego 100 iteraciones más y sera más caliente y así sucesivamente hasta que esta super caliente y esta listo para ser optimizado, en terminos del V8 le llamos hot function(función caliente). Esto significa que el optimizador de código ya esta listo para optimizar esa parte, esta seguro que la ejecución siempre es similar, recibe numeros, regresa numeros, entonces podemos optimizarla a machine code.

Que pasa si por la razón que sea, ya no estamos pasando 2 números como argumento, ahora estamos pasando 2 números y 1 cadena, el resultado sería el número 1 más el string todo pegado. Pero esto va a confundir el optimizador, y lo regresara a bytecode y lo que pasa es que se va a deoptimizar el código, tu puedes seguir programando, no pasara nada pero esto demuestra un poco de ineficiencia. Por esto es bueno que las funciones se llamen igual, si le pasamos objetos que tengan una forma diferente o argumentos de tipos diferentes o no se va a poder optimizar tu función o se va ha deoptimizar, pero esto es así.

Los motores de Javascript funcionan ligeramente diferente, cada browser tiene su propia implementación aunque todos siguen una estructura parecida, el que acabamos de analizar fue V8.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**SpiderMonkey vs V8**

SpiderMonkey tiene 2 capas de optimización, Chakra también y recibe información de profiler y de varios lugares. **JavascriptCore tiene 3 capas de optimización**. ¿Esto significa que safari es más rápido que chrome? No necesariamente, en el desarrollo de programas o de ingeniería siempre se trata traetors, intercambio de costos beneficios, esto quizas no sea perfecto, pero luego saldra mejor, y en javascriptCore puede ser que nuestro programa tarde un poquito más en comenzar, pero una vez que se inicialice el profiler comienza a actuar, optimiza poco y luego otro poco, asi sucesivamente.

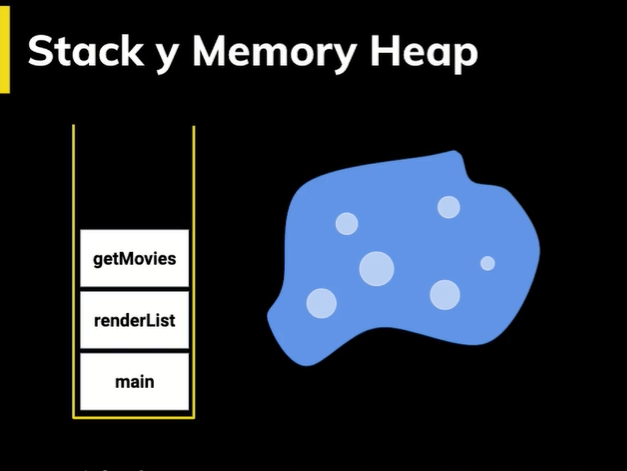
Mientras que en chrome nuestro programa empieza a ejecutarse rápido y quizas toma un poco más de tiempo en que se optmize pero así es como funcionan los motores de javascript.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Event Loop**

Si lo tuvieramos que describir en una oración: El eventLoop es lo que hace que javascript parezca ser multihilo cuando realmente es un solo hilo, entonces como rayos es que podemos hacer scroll, click, cargar un imagen, hacer una petición, miles de cosas haciendo eso una sola vez.

Tenemos que saber que Javascript se organiza usando 2 estructuras de datos, es el Stack y el Heap.

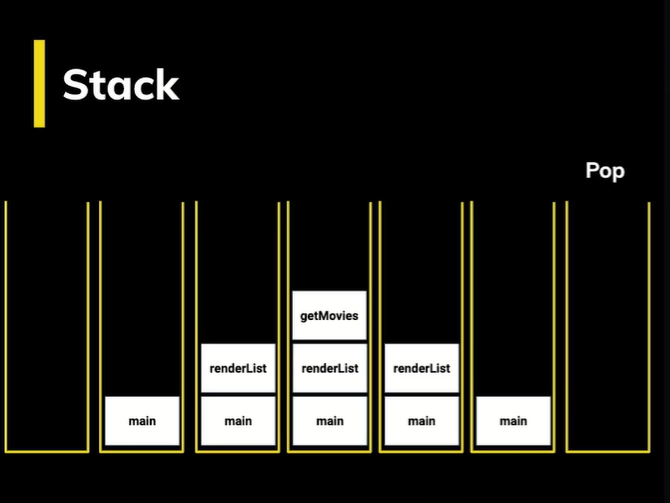
[](https://github.com/JasanHdz/javascript-professional/blob/master/assets/stackheap.png)

El stack es una estructura de datos que lleva rastro de donde está el programa, si un programa comienza con una función main, a su vez llama a renderList y renderList llamá a getMovies, es Stack se vería algo así.

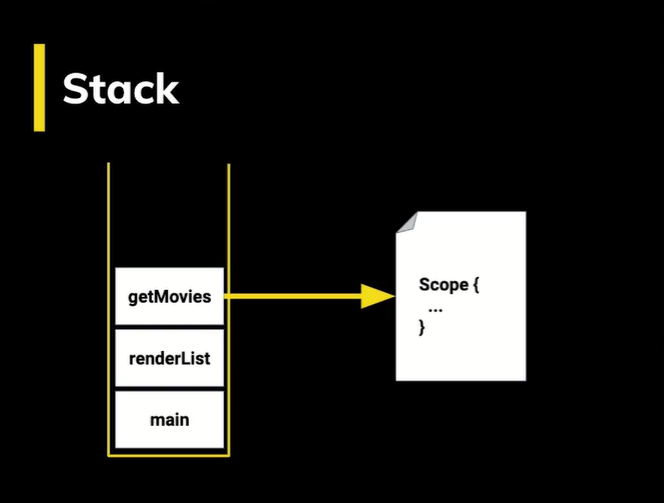
También tenemos el memory heap, el memory heap es una estructura desorganizada, en el stack hay un orden, una función dentro de una función, una dentro de otra, el memory Heap es completamente aleatoria y ahí es donde se guarda la información de las variables, el scope, etc.

El stack comenzará vacío, pero vamos a hacer una operación que se llama un push y vamos a poner como si ponemos un plato, entonces en ese contenedor que solo esta abierto para arriba, hay un plato, si vovemos a hacer un push pusimos otro plato encima, hacemos push, otro plato encima, ahora que pasa si tu quisieras quitar el plato que estaba hasta abajo, no puedes porque no hay forma de sacarlo por enfrente, ni por debajo, solamente por arriba, por lo tanto tienes que sacar el plato que esta hasta arriba, para sacar otro plato y por fin sacar otro plato.

Esa operación de sacar se llama pop, si hacemos pop sale getMovies, si volvemos a hacer pop sale renderList, y si volvemos a hacer pop sale mainy el obtenemos el stack vacío.

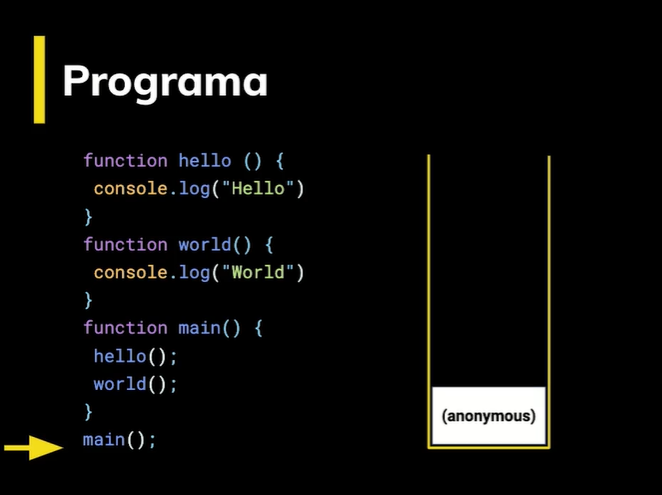
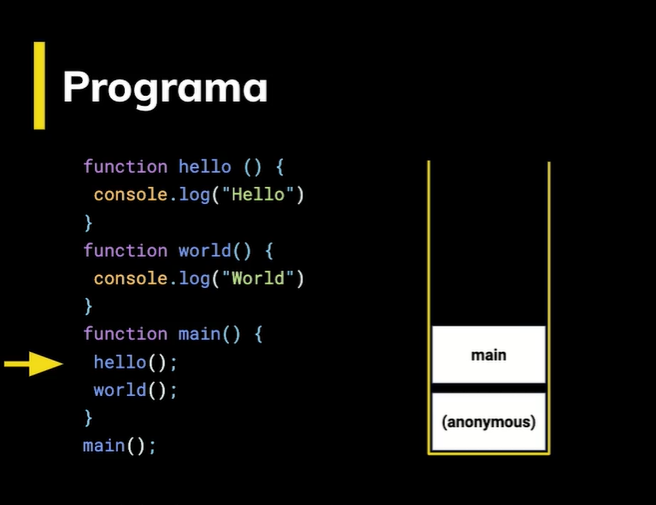
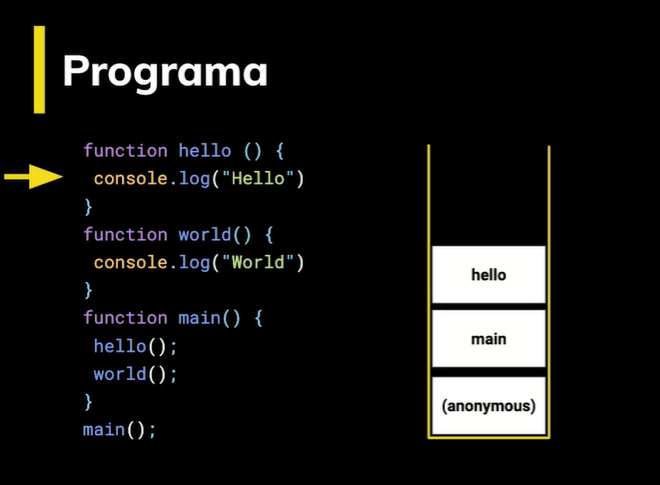
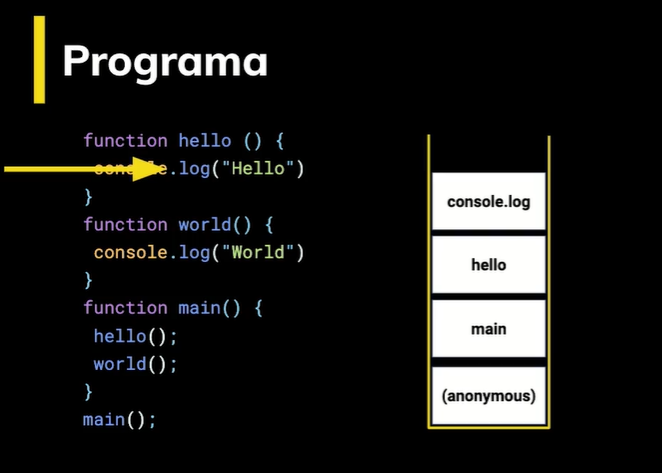
[](https://github.com/JasanHdz/javascript-professional/blob/master/assets/stack.png)

El stack es donde están nuestras funciones, es el registro de como esta operando nuestro programa, apunta a variables como el scope.

[](https://github.com/JasanHdz/javascript-professional/blob/master/assets/scope.png)

Es donde dice estas cosas en está función tienen acceso al entorno global, esto tiene acceso al scope de la función, esto tiene acceso al scope de un bloque y nos guarda esa información.

Entonces si tenemos un programa como este:

[](https://github.com/JasanHdz/javascript-professional/blob/master/assets/stackstart.png) [](https://github.com/JasanHdz/javascript-professional/blob/master/assets/stackmain.png) [](https://github.com/JasanHdz/javascript-professional/blob/master/assets/stackhello.png) [](https://github.com/JasanHdz/javascript-professional/blob/master/assets/stackconsole.png)

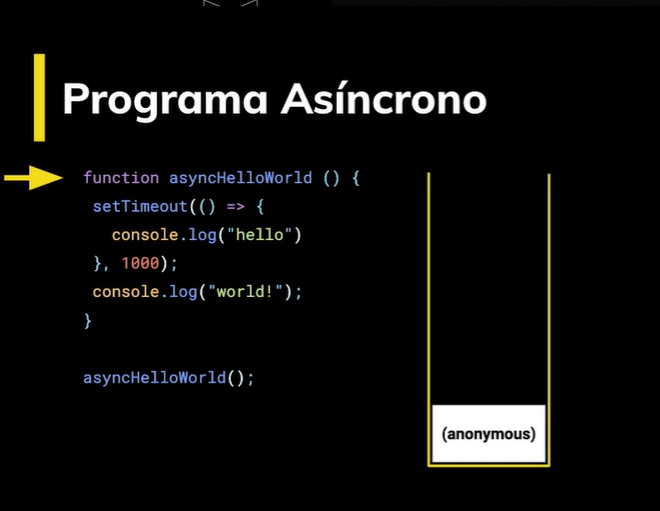
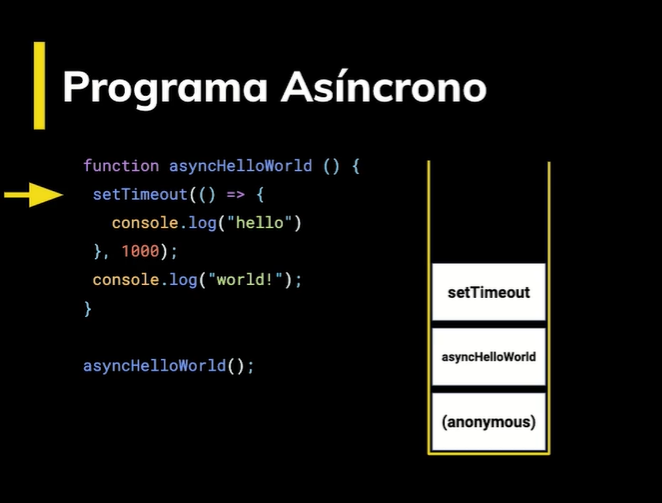
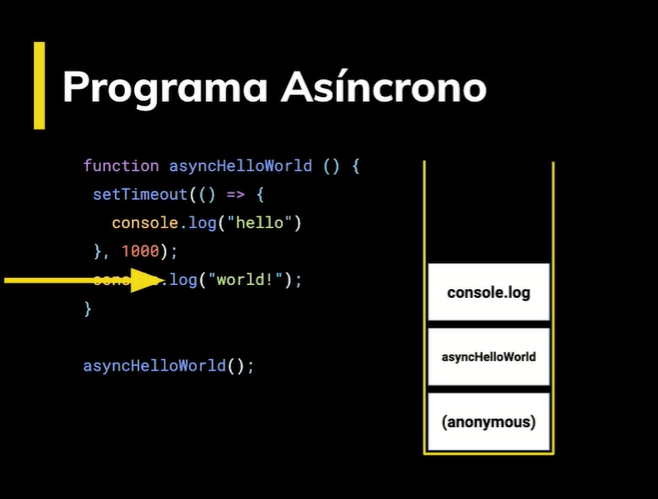
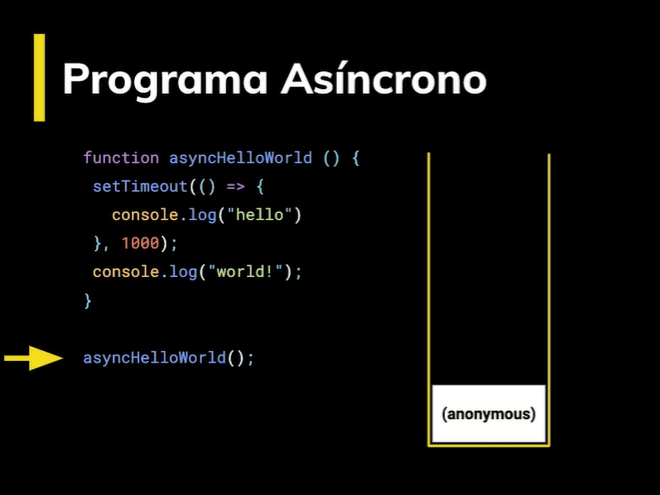
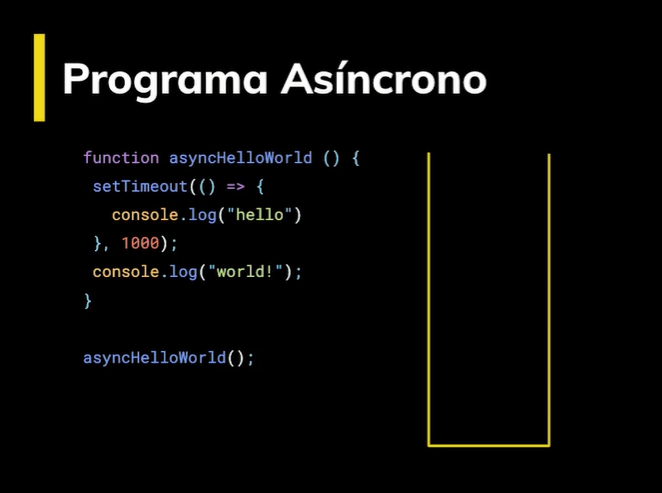
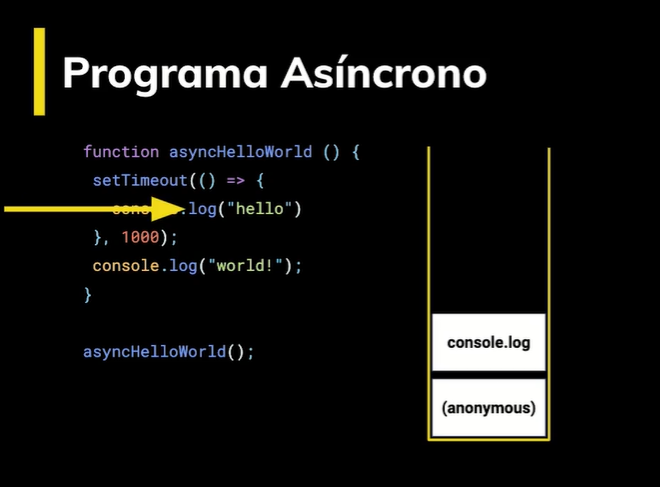
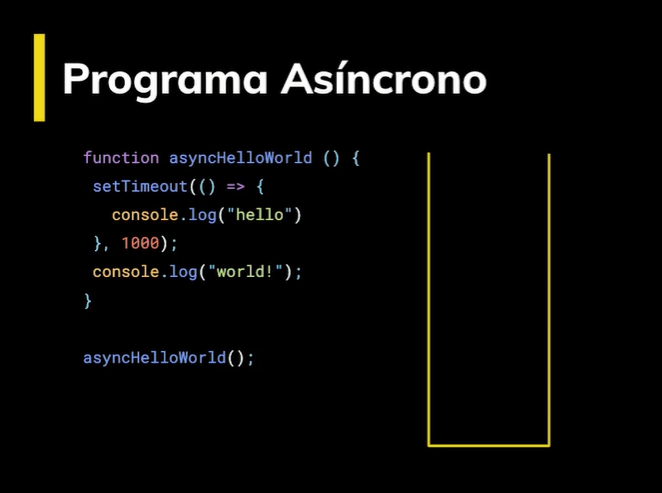
Y así sucesivamente van agregando y quitando ejecuciones en el orden correspondiete.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**EventLoop con Asincronia**

Cuando se ejecuta una función asincrona, como por ejemplo un setTimeout, lo reconoce pero no lo ejecuta, sigue con su proceso normal y luego aparece otra vez para ejecutar la función que ejecutaría un setTimeout.

Parece raro pero esta es la asincronia, cosas que van a pasar eventualmente, pues eventualmente pasarán, pero aún no les toca.

[](https://github.com/JasanHdz/javascript-professional/blob/master/assets/stackasyncrono.png) [](https://github.com/JasanHdz/javascript-professional/blob/master/assets/stackasyncronosettimeout.png) [](https://github.com/JasanHdz/javascript-professional/blob/master/assets/stackasyncronoconsole.png) [](https://github.com/JasanHdz/javascript-professional/blob/master/assets/stackasyncronofin.png) [](https://github.com/JasanHdz/javascript-professional/blob/master/assets/stackasyncronoclean.png) [](https://github.com/JasanHdz/javascript-professional/blob/master/assets/stacksettimeout.png) [](https://github.com/JasanHdz/javascript-professional/blob/master/assets/stackasyncronoclean.png)

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

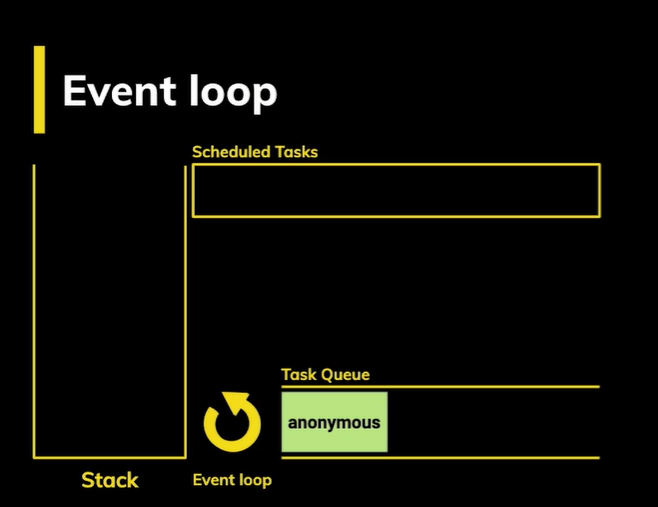
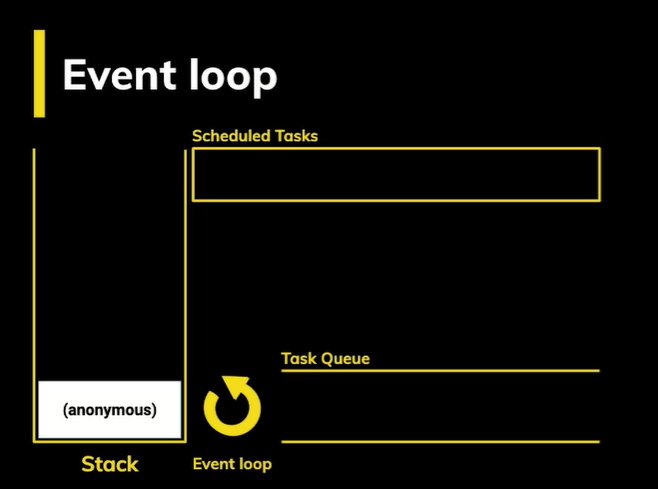
**Task Queue**

**¿Como funciona estó?**

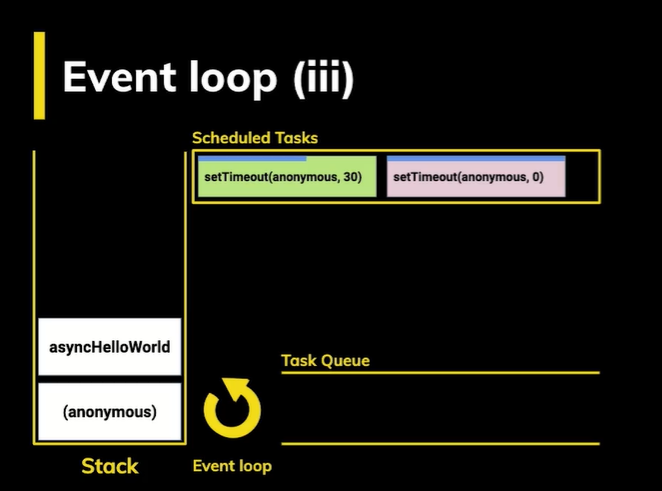
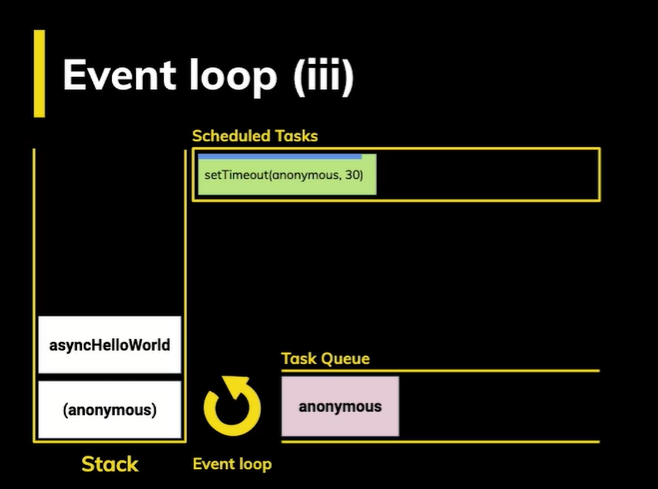
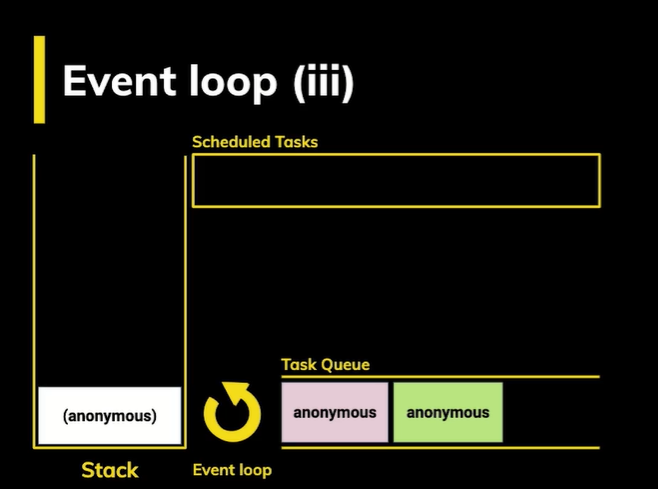
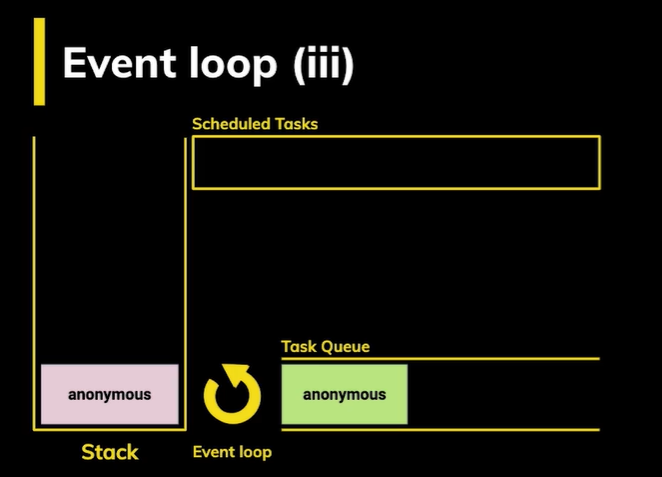
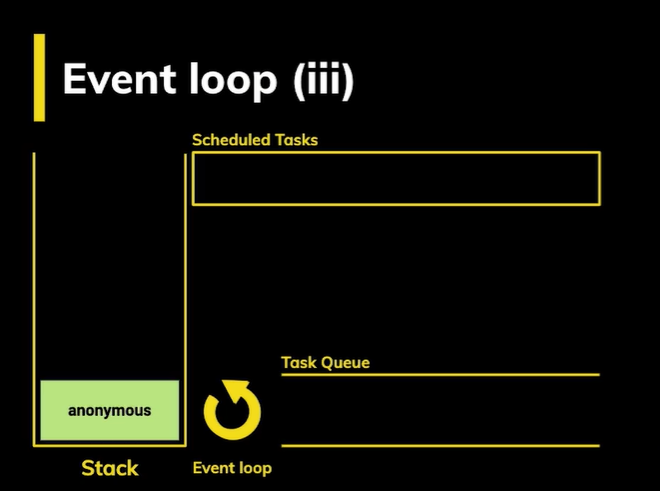
Para poder entenderlo tenemos que hablar sobre **Queue**: es una estructura de datos igual que el stack donde si el stack las cosas, lo primero que entra es lo último que sale. En el **Queue lo primero que entra es lo primero que sale**, es como ir al banco y hacemos una fila, el que llegó primero es al que van a antender primero

Teniendo en cuenta esto vamos a hablar sobre la cola de tareas.

Cuando teniamos el setTimeout encolamos una tarea que ibamos a hacer en 1000 milisegundos.

[](https://github.com/JasanHdz/javascript-professional/blob/master/assets/eventloop.png) [](https://github.com/JasanHdz/javascript-professional/blob/master/assets/eventloopstack.png)

¿Qué pasa si el stack no está vacío?

[](https://github.com/JasanHdz/javascript-professional/blob/master/assets/eventloopstackloadtwo.png) [](https://github.com/JasanHdz/javascript-professional/blob/master/assets/eventloopstackloadtwotask.png) [](https://github.com/JasanHdz/javascript-professional/blob/master/assets/eventloopstackloadtwotaskall.png) [](https://github.com/JasanHdz/javascript-professional/blob/master/assets/eventloopstackiii.png) [](https://github.com/JasanHdz/javascript-professional/blob/master/assets/eventloopstackfintwo.png)

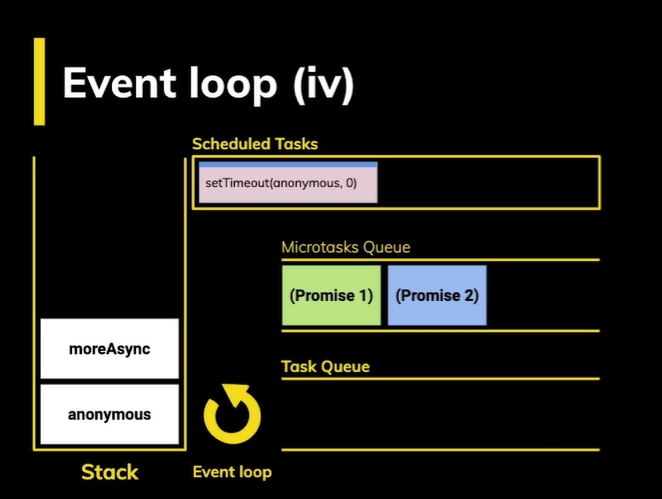
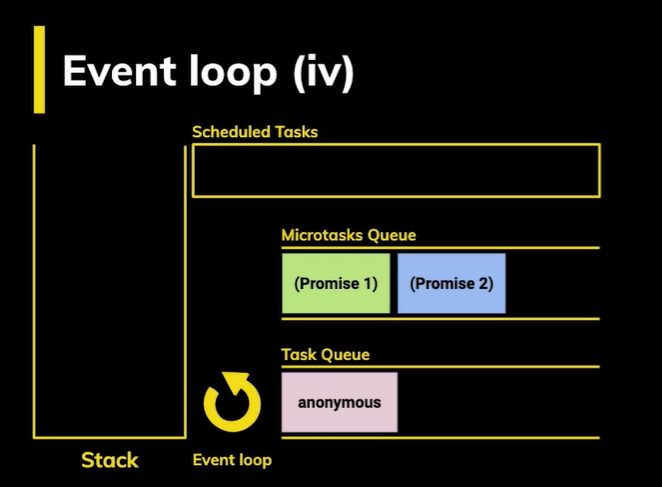
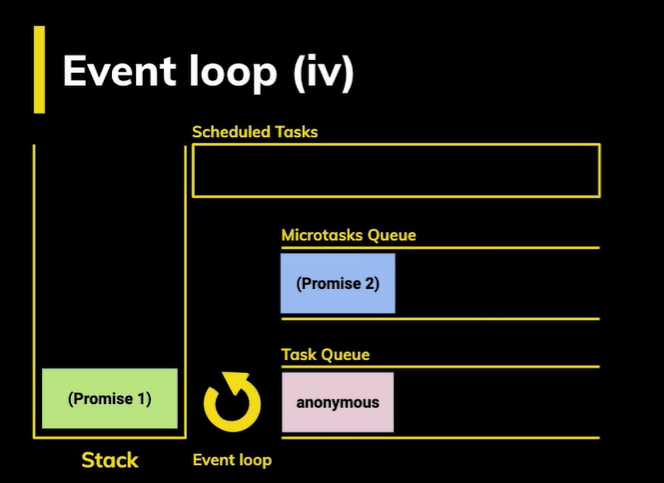
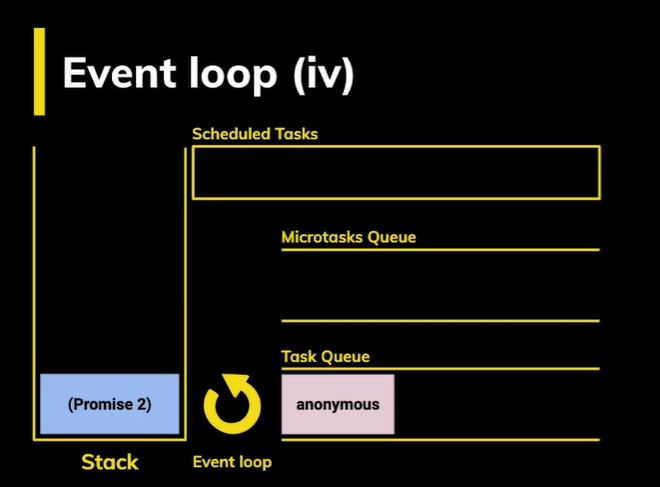
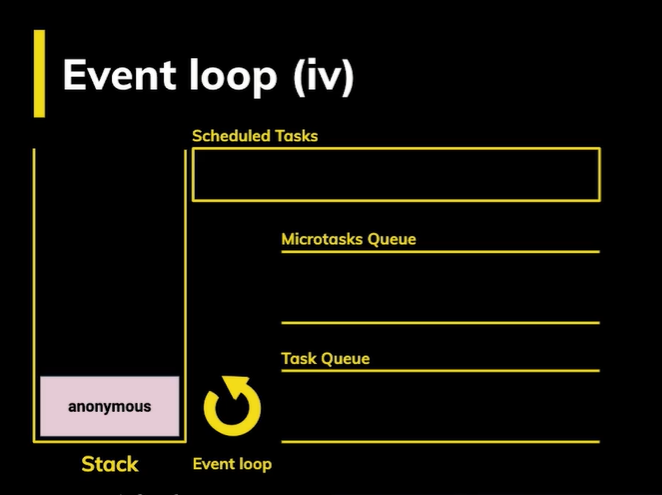
[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Event Loop con Promesas**

Las promesas son algo que eventualmente va a pasar, se puede resolver una promesa o se puede rechazar pero lo importante es que eventualmente, esto es asincrono, entonces llevamos este programa de nombre moreAsync.

[](https://github.com/JasanHdz/javascript-professional/blob/master/assets/promiseeventloop.png)

Resulta que las promesas van en otra cola, la cola de microtareas **Microtask Queue**. Las microtareas son de mayor preoridad y seimpre van primero sobre el scheduled task.

[](https://github.com/JasanHdz/javascript-professional/blob/master/assets/microtask.png) [](https://github.com/JasanHdz/javascript-professional/blob/master/assets/microtaskespera.png) [](https://github.com/JasanHdz/javascript-professional/blob/master/assets/microtaskpromise.png) [](https://github.com/JasanHdz/javascript-professional/blob/master/assets/microtaskpromise2.png) [](https://github.com/JasanHdz/javascript-professional/blob/master/assets/microtaskpromisefin.png)

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Promesas**

Ya vimos como el event loop procesa las promesas, ahora vamos a volver a las promesas, peroe sta vez vamos a ver como funciona el patrón de **.then** lo vamos a convertir a **async await** y también vamos a aprender diferentes patrones cuando escribimos funciones que nos regresan una promesa, todo esto para facilitar el desarrollo de nuestras apps, todo esto lo vamos a hacer con una api que es libre que se llamá [themoviedb](https://www.themoviedb.org/?language=es-ES).

// The Movie Database API: https://developers.themoviedb.org/3/getting-started/introduction

const apiKey = 'b89fc45c2067cbd33560270639722eae';

function getMovie(id) {

const url = `https://api.themoviedb.org/3/movie/${id}?api\_key=${apiKey}`;

return fetch(url).then(response => response.json());

}

async function getPopularMovies() {

const url = `https://api.themoviedb.org/3/discover/movie?sort\_by=popularity.desc&api\_key=${apiKey}`;

const response = await fetch(url);

const data = await response.json();

return fetch(url)

.then(response => response.json())

.then(data => data.results);

}

async function getTopMoviesIds(n = 3) {

return getPopularMovies().then(popularMovies => {

popularMovies.slice(0, n).map(movie => movie.id);

})

}

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Getters y Setters**

Uno de los features modernos que trae javascript son getters y setters, son funciones que podemos utilizar dentro de objetos que nos permiten tener propiedades virtuales, es decir no es una propiedad que existe directamente en el objeto, pero atravez de un getter o setter podemos correr una fución que va ha calcular estos valores o va a mostrar una valor para establecer esté nuevo valor.

Los getters los vamos a escribir usando el keyword get seguido de la propiedad virtual

let persona = {

nombre: 'Yeison',

apellido: 'Daza',

get nombreCompleto() {

return`${nombre}${apellido}`

},

set nombreCompleto(nom) {

const palabras = nom.split(' ');

this.nombre = palabras[0] || '';

this.apellido = palabras[1] || '';

}

}

persona.nombreCompleto = 'Camilo Sanchez'

console.log(persona.nombre); //camilo

console.log(persona.apellido); //sanchez

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Proxy**

Igual que los getters y setters el proxy es uno de los fetures más recientes del lenguaje, también igual que los getters y setters podemos intersectar, algunas llamadas a un objeto, sin embargo más alla de get y set, podemos intersectar muchisimas otras cosas. Si vamos a la documentación de proxy en [MDN](https://github.com/JasanHdz/javascript-professional/blob/master/notes) vamos a encontrar una sección que dice *Methods of the handler object* (métodos del objeto manejador). Aquí vamos a encontrar a get y set, decimos que son trampas, cuando hay una llamada, la llamada va a caer en estas trampas si las tenemos definidas, en la trampa de get y de set, también hay trampas para ver el getPrototypeOf, handler.apply, handler.constructor, etc.

Esto nos va a permitir que antes de la llamada llegue al objeto al que tiene que llegar podemos manipularla. Hay una idea que se me hace muy interesante muy divertida, es un feature que tienen algunos programas como por ejemplo git, si vamos a la consola y escribimos mal el comando, no se ejecutara la instrucción pero nos devolvera una sugerencia a lo que escribimos o en dado caso de no tener una sugerencia, nos dara una lista de posibles comandos.

Vamos a hacer esto mismo, pero en Javascript, que será interceptar las llamadas, si la propiedad que esta buscando el usuario no existe en un objeto, vamos a ver cuales son las que si existen para sugerir una.

Para este ejemplo nos vamos a apoyar de una librería que se llama [fast-levenshtein](https://www.jsdelivr.com/package/npm/fast-levenshtein). Leveshtein es un algoritmo que va a encontrar la distancia entre 2 cadenas. Es decir si tenemos 2 cadenas y se diferencian por 1 sola letra, esa sería una distancia de 1, si se diferencian por 2 campos, sería una distancia de 2.

// target es mi objeto a supervisar (sus propiedades pueden ser objetos, array, funciones, u otro proxy)

const target = {

red: 'Rojo',

green: 'Verde',

blue: 'Azul'

}

// handler es un objeto con funciones (trampa) que definen las acciones a seguir cuando se accede al objeto supervisado

const handler = {

get(obj, prop) {

if (prop in obj) {

// si la propiedad existe, pues retornamos su valor

return obj[prop]

}

// Si llega hasta aqui, vamos a ver si podemos retornar una sugerencia

const suggetion = Object.keys(obj).find(key => {

// creo un objeto con todas mis claves del objeto supervisado, y retorno aquella (nombre)

// que su distancia sea <= 3 tomando como base la propiedad invocada

return Levenshtein.get(key, prop) <= 3

})

if (suggetion) {

console.log(`${prop} no se encontró. ¿Quisiste decir ${suggetion}?`);

}

return obj[prop];

}

}

const p = new Proxy(target, handler);

p.red; // "Rojo"

p.green; // "Verder"

p.reed //reee no se encontró. ¿Quisiste decir red?

p.geen //geen no se encontró. ¿Quisiste decir green?

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Generadores**

Los generadores sons funciones especiales, podemos iniciar su ejecución y detenerla a mitad, nuestro programa continua por otro sitio y luego podemos regresar a esta función generador y continuar su ejecución donde la dejamos, lo que esta muy interesante es que los generadores cuando los detemos se recuerdan de su contexto, cuales eran las variables que tenían en su scoope, veamos un ejemplo de como son las funciones.

function\* simpleGenerator() {

console.log("GENERATOR START");

console.log("GENERATOR END");

}

const gen = simpleGenerator();

Los generadores que se crean traen una función que se llama next(), es una forma de decirle al generador "continua tu ejecucíon" porque ahora esta suspendido, si hacemos next, escribira el generador en consola. Pero también regresa un objeto con los valores [value: undefine, y done: true], este es el valor de retorno del generador. Cuando donde es true quiere decir que el generador termino su ejecución.

Para obtener value definido podemos utilizar un keyword que se llama yield(seder) y si ejecutamos next el yield cortara la ejecución y ahi terminará la ejecución y si queremos volver a ejecutar las instrucciones pendientes tenemos que volver a lanzar a next().

function\* simpleGenerator() {

console.log("GENERATOR START");

yield;

console.log("GENERATOR END");

}

const gen = simpleGenerator();

gen.next()

// GENERATOR START

gen.next();

// GENERATOR END

Algó muy interesantes es que cuando hacemos **yield** podemos regresar un valor

function\* simpleGenerator() {

console.log("GENERATOR START");

yield 1; // {value: 1, done: false}

yield 2; // {value: 2, done: false}

yield 3; // {value: 3, done: false}

console.log("GENERATOR END");

}

const gen = simpleGenerator();

gen.next()

// GENERATOR START

gen.next();

// GENERATOR END

Generadores inifitos

// Podemos hacer generadores infinitos

function\* idMaker() {

let id = 1;

while (true) {

yield id;

id += 1;

}

}

Cuando llamamos a next también podemos pasar valores que la función recibe.

function\* idMakerWithReset() {

let id = 1;

let reset;

while (true) {

reset = yield id;

if (reset) {

id = 1;

} else {

id += 1;

}

}

}

Los generadores se prestan para crear funciones eficientes en memoria, vamos a escribir la secuencia fibonacci, una función que imprima la secuencia, que lo que hace es sumar los dos número anteriores para generar uno nuevo.

// Ahora hagamos un ejemplo un poco más complejo: la secuencia fibonacci

function\* fibonacci() {

let a = 1, b = 1;

while (true) {

const nextNumber = a + b;

a = b;

b = nextNumber;

yield nextNumber;

}

}

Los generadores son funciones especiales cuya ejecución podemos comenzar y detener a mitad de vuelo y cuando queramos continuarla podemos llamar a next, podemos pasarle un valor al generador si hace falta y su ejecución va a continuar siempre recordanse del scope en el que estaba.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Como cancelar peticiones Fetch**

La peticiones [AJAX](https://github.com/JasanHdz/javascript-professional/blob/master/notes) permitieron en su tiempo hacer peticiones asíncronas al servidor sin tener que detener la carga de la página. Hoy en día se utiliza la función fetch para esto.

Con fetch tenemos algo llamado AbortController que nos permite enviar una señal a una petición en plena ejecución para detenerla.

AbortController nos va ha dar 2 controllers para poder detener una ejecución en este caso va ha ser la del fetch, la cual es una clase que trae el motor de javascript, la cual tenemos que instanciarla.

<html>

<head>

<title>Abort Fetch</title>

</head>

<body>

<a href="/ejercicios/">Go back</a>

<p><em>Abre la consola</em></p>

<img id="huge-image" height="400" />

<button id="load">Load HUGE Image</button>

<button id="stop" disabled>Stop Fetching</button>

<script src="abort-fetch.js"></script>

</body>

</html>

const url =

'https://images.pexels.com/photos/974470/nature-stars-milky-way-galaxy-974470.jpeg?q=100';

// Elementos del DOM imagen y 2 botones

const img = document.getElementById('huge-image');

const loadButton = document.getElementById('load');

const stopButton = document.getElementById('stop');

let controller;

// Función que habilita o desabilita un boton

function startLoading() {

loadButton.disabled = true;

// Camnbia el texto de su contenido

loadButton.innerText = 'Loading...';

stopButton.disabled = false;

}

// Funcíon que desabilita el boton de carga

function stopLoading() {

loadButton.disabled = false;

loadButton.innerText = 'Load HUGE Image';

stopButton.disabled = true;

}

loadButton.onclick = async function() {

// Se ejecuta startLoading que lo unico que hace es cambiar la apariencia del boton

// Para que se vea que esta cargando

startLoading();

// Declaramos la variable antes para despues tener acceso a ella

// en el boton de cancelar petición fetch

controller = new AbortController();

try {

// Hacemos la petición asincrona a la URL usando Async await

// Vamos a añadirle un objeto de configuración al fetch

// Esté objeto de configuración le vamos a pasar un objeto que se llama la señal

// La señal va a venir del abort controller

const response = await fetch(url, { signal: controller.signal });

// Vamos a obtener el binario de la imagen con blob img en forma binaria

const blob = await response.blob();

// Convertimos el blob binario a una URL, el navegador se encarga de asignar el blob una url

const imgUrl = URL.createObjectURL(blob);

// Ahora el src se lo asignamos a la url de la imagen

img.src = imgUrl;

} catch (error) {

console.log(error.message);

}

// Cuando la función asincrona se falle vamos a cambiar el boton a stop

stopLoading();

};

stopButton.onclick = function() {

// Si deseamos detener tenemos que llamar al abort controller.container

// El abort envia una señal al fetch y hace que la petición se cancele

controller.abort();

stopLoading();

};

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes" \l "tabla-de-contenido)

**Intersection Observer API**

La API Observador de Intersección, provee una vía para, de forma asíncrona, observar cambios en la intersección de un elemento con un elemento ancestro o con el viewport del documento de nivel superior.

La información sobre intersección es necesaria por muchas razones, tales como:

Lazy-loading de imágenes u otro contenido a medida que la página se desplaza. Implementación de “scroll infinito” de sitios web, donde más y más contenido se carga y muestra a medida que usted hace scroll, de forma que el usuario no tiene que pasar páginas. Informes de visualizaciones de anuncios para calcular ingresos por publicidad. Decidir si deben realizarse tareas o procesos de animación basados en si el usuario verá o no el resultado.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Creando un Intersection Observer**

Crear el intersection observer llamando a su constructor y pasándole una función callback para que se ejecute cuando un nivel (threshold) sea cruzado en una u otra dirección:

var options = {

root: document.querySelector('#scrollArea'),

rootMargin: '0px',

threshold: 1.0

}

var observer = new IntersectionObserver(callback, options);

Un threshold de 1.0 significa que cuando el 100% del elemento target está visible dentro del elemento especificado por la opción root, la función callback es invocada.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Opciones de Intersection observer**

El objeto options pasado al constructor [IntersectionObserver()](https://developer.mozilla.org/es/docs/Web/API/IntersectionObserver/IntersectionObserver) le deja controlar las circunstancias bajo las cuales la función callback del observer es invocada. Tiene los siguientes campos:

**root**

El elemento que es usado como viewport para comprobar la visibilidad de elemento target. Debe ser un elemento ascendiente del target. Por defecto se toma el viewport del navegador si no se especifica o si se especifica como null.

**rootMargin**

Margen alrededor del elemeto root. Puede tener valores similares a los de CSS [margin](https://developer.mozilla.org/es/docs/Web/CSS/margin) property, e.g. "10px 20px 30px 40px" (top, right, bottom, left). Los valores pueden ser porcentajes. Este conjunto de valores sirve para aumentar o encoger cada lado del cuadro delimitador del elemento root antes de calcular las intersecciones. Por defecto son todos cero.

**threshold**

Es un número o un array de números que indican a que porcentaje de visibilidad del elemento target, la función callback del observer debería ser ejecutada. Si usted quiere que se detecte cuando la visibilidad pasa la marca del 50%, debería usar un valor de 0.5. Si quiere ejecutar la función callback cada vez que la visibilidad pase otro 25%, usted debería especificar el array [0, 0.25, 0.5, 0.75, 1]. El valor por defecto es 0 (lo que significa que tan pronto como un píxel sea visible, la función callback será ejecutada). Un valor de 1.0 significa que el umbral no se considera pasado hasta que todos los pixels son visibles.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Determinando un elemento para ser observado**

Una vez usted ha creado el observer, necesita darle un elemento target para observar:

var target = document.querySelector('#listItem');

observer.observe(target);

Cuando el elemento target encuentra un threshold especificado por el IntersectionObserver, la función callback es invocada. La función callback recibe una lista de objetos [IntersectionObserverEntry](https://developer.mozilla.org/es/docs/Web/API/IntersectionObserverEntry) y el observer:

var callback = function(entries, observer) {

entries.forEach(entry => {

// Cada entry describe un cambio en la intersección para

// un elemento observado

// entry.boundingClientRect

// entry.intersectionRatio

// entry.intersectionRect

// entry.isIntersecting

// entry.rootBounds

// entry.target

// entry.time

});

};

Asegúrese de que su función callback se ejecute sobre el hilo principal. Debería operar tan rápidamente como sea posible; si alguna cosa necesita tiempo extra para ser realizada, use [Window.requestIdleCallback()](https://developer.mozilla.org/es/docs/Web/API/Window/requestIdleCallback).

También, note que si especifica la opción root, el elemento target debe ser un descendiente del elemento root.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Creación de Plugin para IntersectionObserver de nuestro videoplayer**

class AutoPause {

constructor() {

this.threshold = 0.25;

this.handlerIntersection = this.handlerIntersection.bind(this)

}

run(player) {

this.player = player;

// const observer = new IntersectionObserver(handler, config)

const observer = new IntersectionObserver(this.handlerIntersection, {

// threshold: umbral define que porciento del elemento tiene que tener interseccion

threshold: this.threshold

})

observer.observe(this.player.media)

}

// Cuando intersectionObserver llame a handlerIntersection le va a pasar una lista de entries

// los entries son todos los objetos que estamos observando

handlerIntersection(entries) {

const entry = entries[0];

console.log(entry);

const isVisible = entry.intersectionRatio >= this.threshold

if (isVisible) {

this.player.play();

} else {

this.player.pause();

}

}

}

export default AutoPause;

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes" \l "tabla-de-contenido)

**Visibility Change**

El visibilityChange forma parte del API del DOM llamado Page Visibility y nos deja saber si el elemento es visible, pude ser usado para ejecutar una acción cuando cambiamos de pestaña. Así podemos ahorrar batería y mejorar la UX.

El documento DOM ahora tiene un elemento que podemos escuchar.

document.addEventListener('visibilitychange', () => {

console.log(document.visibilityState);

})

Usando este evento nosotros podemos salirnos del navegador, también podemos cambiar de pestaña y el DOM lo sabrá. Usemos esto en nuestro plugin para que cuando cambiemos de tab el video se detenga. En el método run es cuando los plugins se echan a correr, aquí es un buen momento, para conectarnos a este evento y que cuando suceda tomar acción.

run(player) {

this.player = player;

// const observer = new IntersectionObserver(handler, config)

const observer = new IntersectionObserver(this.handlerIntersection, {

// threshold: umbral define que porciento del elemento tiene que tener interseccion

threshold: this.threshold

})

observer.observe(this.player.media)

// Ejecutamos el evento VisiblityChange y ejecutamos una función

document.addEventListener('visibilitychange', this.handleVisibilityChange)

}

handleVisibilityChange() {

const isVisible = document.visibilityState === "visible";

if (isVisible) {

this.player.play();

} else {

this.player.pause();

}

}

El evento visibilityChange es un evento muy simple pero muy util, nos deja saber si el tab es el que esta hasta el frente, el tab que el usuario esta viendo, si cambiamos de tab nos permite cambiar acción, no solo nos permite ver un video, también pudiera ser cambiar el titulo de la pestaña, y asi decirle al dom que haga otras acciones, que pueden ahorrar bateria o mejorar el rendimiento de nuestras aplicaciones.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Service worker**

Una de las nuevas tendencias en el desarrollo web, son las pwa o *progressive web apps*, dentro de las varias cosas que ofrecen está que tu app funcione offline, esto lo hacemos posible usando los services workers, **services workers es una capa que va a vivir entre el navegador y el internet**. Lo que van a hacer es algo parecido a los proxys, van a interceptar peticiones, en esté caso las peticiones vamos a tener la oportunidad de hagarrar la petición, buscar la respuesta pero antes de regresarla al browser la vamos a guardar en cache

**¿Qué pasa una vez que lo tenemos en cache?**

La proxima vez que ocurra una petición, en lugar de tener que ir a internet, ya tenemos la respuesta, así que nadamas la regresamos, imaginate un usuario que va dentro del metro se mete en un tunel y pierde conectividad, va a seguir utilizando tu aplicación porque usando service workers, va a funcionar offline.

Vamos a añadirselo a nuestro VideoPlayer.

1. Primero vamos a crear una condición, está condición nos va a servir si el navegador del usuario le da apoyo a los services workets, como es un feature reciente, no todos los navegadores tienen service workets.

if ('serviceworkets' in navigator)

1. Despues dentro del service workets vamos a registrar un archivo, esté archivo va a ser el serice worker tal cual, pero es posible que dentro del registro ocurra un error, asi que es importante ver ese error.

navigator.serviceWorker.register('/sw.js')

.catch(error => {

console.log(error.message);

})

1. En el nivel más alto del proyecto vamos a escribir el archivo del services worker. Aquí es donde vamos a escribir el código.

Los services workers se instalan, el navegador lo va a instalar en la computadora del usuario, no es lo mismo que una aplicación, pero si va a vivir dentro del navegador, entonces cada vez que nostros hagamos cambios hay que volver a instalarlos, esto va suceder cuando el usuario este usando la aplicación en producción pero cuando estamos en desarrollo queremos que esto suceda rápido, no con la lentitud que pueda suceder en producción. Para hacerlo hay que activar *updated on reload* en las devtools

1. Vamos a escribir el código del service worker

// Self hace refencia al service worker es como this a los objetos

self.addEventListener('install', event => {

// Creamos un precache con una lista de recursos que queremos que mantenga en cache

event.waitUntil(precache());

})

// Cuando ocurra una petición queremos a ir al cache para ver si encontramos la respuesta

self.addEventListener('fetch', event => {

// Extraemos la petición

const request = event.request;

// Solo queremos hacer algo con las peticiones que son GET

if (request.method !== "GET")

return;

// actualizar el cache

event.waitUntil(updateCache(cache))

// Buscamos en el cache

// event tiene otra función que se llamá responder con responseWith

// vamos a responder con una respuesta cacheada

event.respondWith(cachedResponse(request))

})

// Escribimos la función del precache

async function precache() {

// Para trabajar con cache tenemos que trabajar con una parte

// de la api del dom que se llamá caches, y lo que hay que hacer es abrir un cache en especifico

// Creamos una instancia de cache que le va a pertenecer o se va a llamar v1,

// podemos ponerle como queramos porque apenas estamos haciendo una instancia,

// este cache regresa una promesa, por lo cual hay que esperarla

const cache = await caches.open("v1");

// Una vez tenemos la instancia de cache queremos añadir varios recursos

// añadirmos todos nuestro recursos, los cuales son todos lo archivos que hemos escrito

// Tenemos que regresarlo porque devuelve una promesa

return cache.addAll([

// Es muy importante asignarne este request

'/',

'index.html',

'styles.css',

'MediaPlayer.js',

'index.js',

'plugins/AutoPause.js',

'plugins/AutoPlay.js',

'buckbunny.mp4'

]);

}

// vamos a pasarle el request

async function cachedResponse(request) {

// Comenzamos abriendo el cache

const cache = await caches.open("v1");

// debemos checar si en el cache tenemos la contestanción al request

// Para hacer eso vamos a guardalo en el response

// Estamos preguntando al cache

// ¿Ya tienes una copia que le corresponse al request?

const response = await cache.match(request)

// Como es posible que este response sea undefine, tenemos que contestar con lo que nos de la red

return response || fetch(request);

}

async function updateCache(request) {

const cache = await caches.open("v1");

const response = await fetch(request);

return cache.put(request, response)

}

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Typescript**

TypeScript es un superset de JavaScript que añade tipos a nuestras variables ayudando así a la detección de errores de forma temprana y mejorando el autocompletado.

Los navegadores no entienden TypeScript así que lo vamos a transpilar a JavaScript usando **Parcel**.

yarn add parcel-bundler --dev

Agregamos un par de configuraciones a nuestro packages.json

"scripts": {

"start": "parcel index.html player-video/index.html player-video/\*\*/\*.html"

},

"browserslist": [

"last 1 Chrome version"

]

TypeScript es un lenguaje de programación libre y de **código abierto** desarrollado y mantenido por **Microsoft**. Es un superconjunto de JavaScript, que esencialmente añade tipos estáticos y objetos basados en clases. [Anders Hejlsberg](https://es.wikipedia.org/wiki/Anders_Hejlsberg), diseñador de C# y creador de Delphi y Turbo Pascal, ha trabajado en el desarrollo de TypeScript.1​ TypeScript puede ser usado para desarrollar aplicaciones JavaScript que se ejecutarán en el lado del cliente o del servidor (**Node.js**).

TypeScript extiende la sintaxis de JavaScript, por tanto cualquier código JavaScript existente debería funcionar sin problemas. Está pensado para grandes proyectos, los cuales a través de un compilador de TypeScript se traducen a código JavaScript original.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Tipos básicos en Typescript**

* boolean. Valor verdadero o falso.
* number. Números.
* string. Cadenas de texto.
* string[]. Arreglo del tipo cadena de texto.
* Array. Arreglo multi-tipo, acepta cadenas de texto o números.
* enum. Es un tipo especial llamado enumeración.
* any. Cualquier tipo.
* object. Del tipo objeto.

// TypeScript por Jasan Hernández

// Boolean

let muted: boolean = true;

muted = false;

// Numbers

let numerador: number = 42;

let denomindador: number = 6;

let resultado = numerador / denomindador;

// String

let nombre: string = "Jasan";

let saludo: string = `Me llamo ${nombre}`;

// Arreglos

let people: string[] = [];

people = ["Isabel", "Nicole", "Raúl"];

// people.push(34);

// Arreglos de Strings and numbers:

let peopleAndNumbers: Array<string | number> = [];

peopleAndNumbers.push("Ricardo");

peopleAndNumbers.push(345);

// Enum

enum Color {

Rojo = "Rojo",

Verde = "Verde",

Azul = "Amarillo",

}

let colorFavorito: Color = Color.Azul;

console.log(`Mi color favorito es: ${colorFavorito}`);

// Any

let comodin: any = "Joker";

comodin = { type: "WildCard" }

// Object

let someObject: object = { type: "WildCard" };

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes" \l "tabla-de-contenido)

**Funciones en Typescript**

En Javascript las funciones toman argumentos y pudieran regresar algun valor. **En Typescript** podemos ser explicitos de como deben ser esos argumentos y también podemos proveer información de cual es valor que debe regresar la función.

Typescript va a evitar que cometamos erroes que le puedan pasar a cualquiera, typescript es programación pareja, escribes y tu pareja te va diciendo ahi tienes un error.

// Funciones

function add(a: number, b: number): number {

return a + b;

}

const sum = add(4, 25)

function createAdder(a: number): (number) => number {

return function (b: number) {

return a + b;

}

}

const addFour = createAdder(4);

const fourPlus6 = addFour(6);

function fullName(firtsName: string, lastName?: string): string {

return `${firtsName} ${lastName}`;

}

const jasan = fullName('Jasan');

function fullValue(firtsName: string = "Pepe", lastName: string = "Smith"): string {

return `${firtsName} ${lastName}`;

}

const person = fullValue();

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes" \l "tabla-de-contenido)

**Interfaces en Typescript**

Las interfaces nos permiten declarar la forma que tiene un objeto, esto puede ser útil, porque nos ayuda en autocompletado y evitar cometer algunos erroes.

enum Color {

Rojo = "Rojo",

Verde = "Verde"

};

interface Rectangulo {

height: number,

width: number

color?: Color

}

let rect: Rectangulo = {

height: 4,

width: 3,

// color: Color.Verde

}

function area(r: Rectangulo) {

return r.height \* r.width;

}

const areaReact = area(rect);

rect.toString = function () {

return this.color ? `Un rectangulo ${this.color}` : `Un rectangulo`;

}

console.log(rect.toString());

Las interfaces definen la forma exacta que debe tener un objeto, no podemos añadir propiedades de más, ni de menos, en caso de que una propiedad sea opcional, la tenemos que marcar como opcional

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Clases en Typescript**

JavaScript tradicional utiliza funciones y herencia basada en prototipos para construir componentes reutilizables, pero esto puede resultar un poco incómodo para los programadores más cómodos con un enfoque orientado a objetos, donde las clases heredan la funcionalidad y los objetos se crean a partir de estas clases. A partir de ECMAScript 2015, también conocido como ECMAScript 6, los programadores de JavaScript podrán construir sus aplicaciones utilizando este enfoque basado en clases orientado a objetos. En TypeScript, permitimos que los desarrolladores usen estas técnicas ahora y las compilen en JavaScript que funcione en todos los principales navegadores y plataformas, sin tener que esperar a la próxima versión de JavaScript.

Echemos un vistazo a un ejemplo simple basado en clases:

class Greeter {

greeting: string;

constructor(message: string) {

this.greeting = message;

}

greet() {

return"Hello, " + this.greeting;

}

}

let greeter = new Greeter("world");

La sintaxis debería resultarle familiar si ha usado C # o Java anteriormente. Declaramos una nueva clase Greeter. Esta clase tiene tres miembros: una propiedad llamada greeting, un constructor y un método greet.

Notarás que en la clase cuando nos referimos a uno de los miembros de la clase que anteponemos this.. Esto denota que es un acceso de miembro.

En la última línea construimos una instancia de la Greeterclase usando new. Esto llama al constructor que definimos anteriormente, creando un nuevo objeto con la Greeterforma y ejecutando el constructor para inicializarlo.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Herencia Typescript**

En TypeScript, podemos usar patrones comunes orientados a objetos. Uno de los patrones más fundamentales en la programación basada en clases es poder extender las clases existentes para crear otras nuevas usando la herencia.

Echemos un vistazo a un ejemplo:

class Animal {

move(distanceInMeters: number = 0) {

console.log(`Animal moved ${distanceInMeters}m.`);

}

}

class Dog extends Animal {

bark() {

console.log('Woof! Woof!');

}

}

const dog = new Dog();

dog.bark();

dog.move(10);

dog.bark();

Este ejemplo muestra la característica de herencia más básica: las clases heredan propiedades y métodos de las clases base. Aquí, Doghay una clase derivada que deriva de la clase Animalbase usando la extendspalabra clave. Las clases derivadas a menudo se denominan subclases , y las clases base a menudo se denominan superclases .

Debido a que Dogextiende la funcionalidad desde Animal, pudimos crear una instancia de Dogque podría ambos bark()y move().

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Modificadores de Acceso en Typescript**

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Público por defecto**

En nuestros ejemplos, hemos podido acceder libremente a los miembros que declaramos en todos nuestros programas. Si está familiarizado con las clases en otros idiomas, puede haber notado en los ejemplos anteriores que no hemos tenido que usar la palabrapublic para lograr esto; por ejemplo, C # requiere que cada miembro esté explícitamente etiquetado publiccomo visible. En TypeScript, cada miembro es publicpor defecto.

Aún puede marcar un miembro publicexplícitamente. Podríamos haber escrito la Animalclase de la sección anterior de la siguiente manera:

class Animal {

public name: string;

publicconstructor(theName: string) { this.name = theName; }

public move(distanceInMeters: number) {

console.log(`${this.name} moved ${distanceInMeters}m.`);

}

}

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Private**

Cuando se marca un miembro private, no se puede acceder desde fuera de su clase que lo contiene. Por ejemplo:

class Animal {

private name: string;

constructor(theName: string) { this.name = theName; }

}

new Animal("Cat").name; // Error: 'name' is private;

TypeScript es un sistema de tipo estructural. Cuando comparamos dos tipos diferentes, independientemente de su procedencia, si los tipos de todos los miembros son compatibles, entonces decimos que los tipos mismos son compatibles.

Sin embargo, al comparar tipos que tienen privatey protectedmiembros, tratamos estos tipos de manera diferente. Para que dos tipos se consideren compatibles, si uno de ellos tiene un privatemiembro, el otro debe tener un privatemiembro que se originó en la misma declaración. Lo mismo se aplica a los protectedmiembros.

Veamos un ejemplo para ver mejor cómo se desarrolla esto en la práctica:

class Animal {

private name: string;

constructor(theName: string) { this.name = theName; }

}

class Rhino extends Animal {

constructor() { super("Rhino"); }

}

class Employee {

private name: string;

constructor(theName: string) { this.name = theName; }

}

let animal = new Animal("Goat");

let rhino = new Rhino();

let employee = new Employee("Bob");

animal = rhino;

animal = employee; // Error: 'Animal' and 'Employee' are not compatible

En este ejemplo, tenemos una Animaly una Rhino, con Rhinoser una subclase de Animal. También tenemos una nueva clase Employeeque se ve idéntica Animalen términos de forma. Creamos algunas instancias de estas clases y luego tratamos de asignarlas entre sí para ver qué sucederá. Porque Animaly Rhinocomparten el privatelado de su forma desde la misma declaración de private name: stringin Animal, son compatibles. Sin embargo, este no es el caso Employee. Cuando intentamos asignar de a Employeea Animal, obtenemos un error de que estos tipos no son compatibles. Aunque Employeetambién tiene un privatemiembro llamado name, no es el que declaramos enAnimal .

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Protected**

El protectedmodificador actúa de manera muy similar al privatemodificador con la excepción de que los miembros declarados protectedtambién pueden accederse dentro de las clases derivadas. Por ejemplo:

class Person {

protected name: string;

constructor(name: string) { this.name = name; }

}

class Employee extends Person {

private department: string;

constructor(name: string, department: string) {

super(name);

this.department = department;

}

public getElevatorPitch() {

return`Hello, my name is ${this.name} and I work in ${this.department}.`;

}

}

let howard = new Employee("Howard", "Sales");

console.log(howard.getElevatorPitch());

console.log(howard.name); // error

Tenga en cuenta que si bien no podemos usarlo namedesde fuera Person, aún podemos usarlo desde un método de instancia de Employeeporque Employeederiva dePerson .

Un constructor también puede estar marcado protected. Esto significa que la clase no se puede instanciar fuera de su clase que contiene, sino que se puede extender. Por ejemplo:

class Person {

protected name: string;

protected constructor(theName: string) { this.name = theName; }

}

// Employee can extend Person

class Employee extends Person {

private department: string;

constructor(name: string, department: string) {

super(name);

this.department = department;

}

public getElevatorPitch() {

return`Hello, my name is ${this.name} and I work in ${this.department}.`;

}

}

let howard = new Employee("Howard", "Sales");

let john = new Person("John"); // Error: The 'Person' constructor is protected

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes" \l "tabla-de-contenido)

**Convertir Proyecto a Typescript**

Quick fix es algo que vas a poder usar si usas typescript y visual studio code, vsc ya trae un plugin que habilita todas estas funcionalidades de a gratis, esa es una buenas convinaciones en el desarrollo de js. vscode y typescript.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Refactorización**

La [refactorización del código fuente](https://en.wikipedia.org/wiki/Code_refactoring) puede mejorar la calidad y la facilidad de mantenimiento de su proyecto al reestructurar su código sin modificar el comportamiento del tiempo de ejecución. Visual Studio Code admite operaciones de refactorización (refactorizaciones) como el Método deextracción y la [Variable de extracción](https://refactoring.com/catalog/extractVariable.html) para mejorar su base de código desde su editor.

**La refactorización de código** es el proceso de reestructurar el código de computadora existente, cambiar la [factorización](https://en.wikipedia.org/wiki/Decomposition_(computer_science)), sin cambiar su comportamiento externo. La refactorización está destinada a mejorar los atributos [no funcionales](https://en.wikipedia.org/wiki/Non-functional_requirement) del [software](https://en.wikipedia.org/wiki/Software) . Las ventajas incluyen [legibilidad](https://en.wikipedia.org/wiki/Readability) mejorada del código y [complejidad] reducida ; Estos pueden mejorar el [mantenimiento del código fuente](https://en.wikipedia.org/wiki/Maintainability) y crear una [arquitectura] interna más expresiva o [un modelo de objeto] para mejorar la [extensibilidad](https://en.wikipedia.org/wiki/Extensibility).

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Patrones de diseño**

Los patrones de diseño son unas técnicas para resolver problemas comunes en el [desarrollo de software](https://es.wikipedia.org/wiki/Ingenier%C3%ADa_de_software) y otros ámbitos referentes al diseño de interacción o interfaces.

Un patrón de diseño resulta ser una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.

**Sumary**: Los patrones de diseño son como recetas que resuelven problemas que nos enfrentamos frecuentemente en el diseño de software, en especifico, son una solución de un problema dentro de un contexto, y esté contexto no se vale que ocurra una sola vez, tiene que pasar muchas veces. Porque si ocurriera una sola vez, fuera una solución pero no lo podemos categorizar como un patrón de diseño, los patrones de diseño van a resolver problemas dentro de un contexto recurrente.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Objetivos de los patrones de diseño**

Los patrones de diseño pretenden:

* Proporcionar catálogos de elementos reusables en el diseño de sistemas software.
* Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
* Formalizar un vocabulario común entre diseñadores.
* Estandarizar el modo en que se realiza el diseño.
* Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente.

Asimismo, no pretenden:

* Imponer ciertas alternativas de diseño frente a otras.
* Eliminar la creatividad inherente al proceso de diseño.

No es obligatorio utilizar los patrones, solo es aconsejable en el caso de tener el mismo problema o similar que soluciona el patrón, siempre teniendo en cuenta que en un caso particular puede no ser aplicable. "Abusar o forzar el uso de los patrones puede ser un error".

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Historia sobre Patrones de diseño**

No los inventaron los ingenieros de software, programadores o desarrolladores, los patrones de diseño vienen de la arquitectura de estos 2 libros: [The timeless way of building](https://en.wikipedia.org/wiki/The_Timeless_Way_of_Building) y [A Pattern Language](https://en.wikipedia.org/wiki/A_Pattern_Language) escritos por [Christopher Alexander](https://en.wikipedia.org/wiki/Christopher_Alexander).

Lo que buscaba este author era reconocer ciertos patrones que el veía en la arquitectura, no solo en la moderna sino en la arquitectura de toda la historia y así poder construir un lenguje. Cuando tienes un lenguaje sobre todos estos conceptos y entre arquitectos se hablan estas palabras, saben que se estan referiendo a algo exactamente, algo muy particular, una idea exacta, y cuando los recopilas estas creando un recetario para futuras generaciones de arquitectos que vallan a hacer nuevos edificios, nadie quiere reinventar la rueda cuando ya la inventamos y admeás funciona bien.

De aquí 4 personas que conocemos como el Gang of Four (GoF) o Ganga de cuatro compuesto por **Erich Gamma**, **Richard Helm**, **Ralph Johnson** y **John Vlissides**, en el que se recogían 23 patrones de diseño comunes. Son 4 authores que escribieron un libro famosisimo que se llamá [Design Patterns](https://en.wikipedia.org/wiki/Design_Patterns).

Es de aquí que el sofware comienza a tomar estás ideas, es un libro de los 90s pero su importancia y validez aún se mantiene.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Beneficios de utilizar patrones de diseño**

* Son una caja bien probadas a problemas comunes en diseño de software.
* Te proveen un lenguaje común que te permiten comunicarte de una forma especifica y eficiente.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Desventajas de utilizar patrones de diseño**

* Introducen un nivel de complejidad
* Son soluciones a las limitaciones de un lenguaje de programación. ejemplo: Java o C# no tienen funciones de alto nivel.
* "Cuando lo que tienes es un martillo, todo te parece un clavo".

No es algo que constamente vamos a estar buscando usar, siempre debemos tratar de resolver los problemas de una forma simple natural, pero cuando ya vemos que no hay una forma fácil, podemos ir a los patrones de diseño y decir, vemos si ya una solución detallada que podamos aplicar.

Los patrones, como todas las formas de complejidad, deben evitarse hasta que sean absolutamente necesarios.

Recuerda no siempre estar pensando en patrones de diseño deja que naturalmente surgan las necesidades para usarlos, porque siempre preferimos tener programas que sean simples y recuerda que los patrones de diseño van a incluir un nivel de complejidad.

saber más [codigohorror](https://blog.codinghorror.com/head-first-design-patterns/)

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Categorias de patrones de diseño**

El libro design patterns de GoF definio la lita inicial de patrones de diseño de software, los dividio en 3 categorias:

* Patrones creacionales
* Patrones estructurales
* Patrones de comportamiento

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Patrones Creacionales**

Proveen diferentes mecanismos para crear objetos. Nos ayudan a encapsular y abstraer dicha creación:

[Object Pool](https://es.wikipedia.org/wiki/Object_pool_(patr%C3%B3n_de_dise%C3%B1o)) (Pisicina de objetos): no pertenece a los patrones especificados por GoF): se obtienen objetos nuevos a través de la clonación. Utilizado cuando el costo de crear una clase es mayor que el de clonarla. Especialmente con objetos muy complejos. Se especifica un tipo de objeto a crear y se utiliza una interfaz del prototipo para crear un nuevo objeto por clonación. El proceso de clonación se inicia instanciando un tipo de objeto de la clase que queremos clonar.

[Abstract Factory](https://es.wikipedia.org/wiki/Abstract_Factory) (fábrica abstracta): permite trabajar con objetos de distintas familias de manera que las familias no se mezclen entre sí y haciendo transparente el tipo de familia concreta que se esté usando. El problema a solucionar por este patrón es el de crear diferentes familias de objetos, como por ejemplo, la creación de interfaces gráficas de distintos tipos (ventana, menú, botón, etc.).

[Builder](https://es.wikipedia.org/wiki/Builder_(patr%C3%B3n_de_dise%C3%B1o)) (constructor virtual): abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto.

Es usado para permitir la creación de una variedad de objetos complejos desde un objeto fuente. Es como si tenemos un punto de partida que es esté objeto y vamos a poder tomar varios caminos dependiendo de cuales funciones o métodos llamemos.

Separa la creación de objeto complejo de su estructura, de tal forma que el mismo proceso de construcción puede servir para crear representaciones diferentes.

[Factory Method](https://es.wikipedia.org/wiki/Factory_Method_(patr%C3%B3n_de_dise%C3%B1o)) (método de fabricación): centraliza en una clase constructora la creación de objetos de un subtipo de un tipo determinado, ocultando al usuario la casuística, es decir, la diversidad de casos particulares que se pueden prever, para elegir el subtipo que crear. Parte del principio de que las subclases determinan la clase a implementar. A continuación se muestra un ejemplo de este patrón:

class ConcreteCreator extends Creator{

protected Product factoryMethod(){

return new ConcreteProduct();

}

}

interface Product{...}

class ConcreteProduct implements Product{...}

public class Client{

public static void main(String args[])

{

Creator unCreator = new ConcreteCreator();

unCreator.factoryMethod();

}

}

[Prototype](https://es.wikipedia.org/wiki/Prototipo_(patr%C3%B3n_de_dise%C3%B1o)) (prototipo): crea nuevos objetos clonándolos de una instancia ya existente.

[Singleton](https://es.wikipedia.org/wiki/Singleton) (instancia única): garantiza la existencia de una única instancia para una clase y la creación de un mecanismo de acceso global a dicha instancia. Restringe la instanciación de una clase o valor de un tipo a un solo objeto. A continuación se muestra un ejemplo de este patrón:

ejemplo en Java:

public sealed class Singleton

{

private static volatile Singleton instance;

private static object syncRoot = new Object();

private Singleton()

{

System.Windows.Forms.MessageBox.Show("Nuevo Singleton");

}

public static Singleton GetInstance

{

get

{

if (instance == null)

{

lock(syncRoot)

{

if (instance == null)

instance = new Singleton();

}

}

return instance;

}

}

}

[Model View Controller](https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador) (MVC) ♙En español: Modelo Vista Controlador. Es un patrón de arquitectura de software que separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones. Este patrón plantea la separación del problema en tres capas: la capa model, que representa la realidad; la capa controller , que conoce los métodos y atributos del modelo, recibe y realiza lo que el usuario quiere hacer; y la capa vista, que muestra un aspecto del modelo y es utilizada por la capa anterior para interactuar con el usuario.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Patrones estructurales**

Describen formas de componer objetos para formar nuevas estructuras flexibles y eficientes.

* [Adapter o Wrapper](https://es.wikipedia.org/wiki/Adaptador_(patr%C3%B3n_de_dise%C3%B1o)) (Adaptador o Envoltorio): Adapta una interfaz para que pueda ser utilizada por una clase que de otro modo no podría utilizarla.
* [Bridge](https://es.wikipedia.org/wiki/Bridge_(patr%C3%B3n_de_dise%C3%B1o)) (Puente): Desacopla una abstracción de su implementación.
* [Composite](https://es.wikipedia.org/wiki/Composite_(patr%C3%B3n_de_dise%C3%B1o)) (Objeto compuesto): Permite tratar objetos compuestos como si de uno simple se tratase.
* [Decorator](https://es.wikipedia.org/wiki/Decorator_(patr%C3%B3n_de_dise%C3%B1o)) (Decorador): Añade funcionalidad a una clase dinámicamente.
* [Facade](https://es.wikipedia.org/wiki/Facade_(patr%C3%B3n_de_dise%C3%B1o)) (Fachada): Provee de una interfaz unificada simple para acceder a una interfaz o grupo de interfaces de un subsistema.
* [Flyweight](https://es.wikipedia.org/wiki/Flyweight_(patr%C3%B3n_de_dise%C3%B1o)) (Peso ligero): Reduce la redundancia cuando gran cantidad de objetos poseen idéntica información.
* [Proxy](https://es.wikipedia.org/wiki/Proxy_(patr%C3%B3n_de_dise%C3%B1o)): Proporciona un intermediario de un objeto para controlar su acceso.
* [Module](https://es.wikipedia.org/wiki/M%C3%B3dulo_(patr%C3%B3n_de_dise%C3%B1o)): Agrupa varios elementos relacionados, como clases, singletons, y métodos, utilizados globalmente, en una entidad única.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Patrones de comportamiento**

Gestionan algoritmos y responsabilidades. Estos patrones lo que ofrecen son formas de poder manejar diferentes algoritmos, funcionalidades que pueden llegar a tener tus objetos o las relaciones entre ellos.

Se definen como patrones de diseño software que ofrecen soluciones respecto a la interacción y responsabilidades entre clases y objetos, así como los algoritmos que encapsulan:

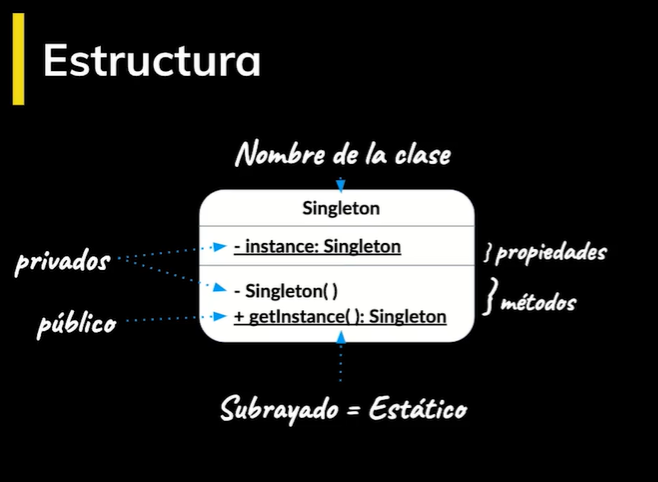
[Chain of Responsibility](https://es.wikipedia.org/wiki/Cadena_de_responsabilidad) (Cadena de responsabilidad): Permite establecer la línea que deben llevar los mensajes para que los objetos realicen la tarea indicada. [Command](https://es.wikipedia.org/wiki/Command_(patr%C3%B3n_de_dise%C3%B1o)) (Orden): Encapsula una operación en un objeto, permitiendo ejecutar dicha operación sin necesidad de conocer el contenido de la misma. [Interpreter](https://es.wikipedia.org/wiki/Interpreter_(patr%C3%B3n_de_dise%C3%B1o)) (Intérprete): Dado un lenguaje, define una gramática para dicho lenguaje, así como las herramientas necesarias para interpretarlo. [Iterator](https://es.wikipedia.org/wiki/Iterador_(patr%C3%B3n_de_dise%C3%B1o)) (Iterador): Permite realizar recorridos sobre objetos compuestos independientemente de la implementación de estos. [Mediator](https://es.wikipedia.org/wiki/Mediator_(patr%C3%B3n_de_dise%C3%B1o)) (Mediador): Define un objeto que coordine la comunicación entre objetos de distintas clases, pero que funcionan como un conjunto. [Memento](https://es.wikipedia.org/wiki/Memento_(patr%C3%B3n_de_dise%C3%B1o)) (Recuerdo): Permite volver a estados anteriores del sistema. [Observer](https://es.wikipedia.org/wiki/Observer_(patr%C3%B3n_de_dise%C3%B1o)) (Observador): Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y actualicen automáticamente todos los objetos que dependen de él. [State](https://es.wikipedia.org/wiki/State_(patr%C3%B3n_de_dise%C3%B1o)) (Estado): Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. [Strategy](https://es.wikipedia.org/wiki/Strategy_(patr%C3%B3n_de_dise%C3%B1o)) (Estrategia): Permite disponer de varios métodos para resolver un problema y elegir cuál utilizar en tiempo de ejecución. [Template Method](https://es.wikipedia.org/wiki/Patr%C3%B3n_de_m%C3%A9todo_de_la_plantilla) (Método plantilla): Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos, esto permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura. [Visitor](https://es.wikipedia.org/wiki/Visitor_(patr%C3%B3n_de_dise%C3%B1o)) (Visitante): Permite definir nuevas operaciones sobre una jerarquía de clases sin modificar las clases sobre las que opera.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Patrón Singleton y casos de uso**

Singleton es un patron creacional, el singleton te va ayudar a que una clase solo sea capaz de crear una sola instancia de si misma, esa instancia siempre va a ser la misma, para describir patrones de diseño, siempre usamos una notación que se llamá UML.

Diagrama UML de una clase que implementa el patrón singleton.

[](https://github.com/JasanHdz/javascript-professional/blob/master/assets/umlsingleton.png)

En ingeniería de software, singleton o instancia única es un patrón de diseño que permite restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto.

Su intención consiste en garantizar que una clase solo tenga una instancia y proporcionar un punto de acceso global a ella.

El patrón singleton se implementa creando en nuestra clase un método que crea una instancia del objeto solo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor (con modificadores de acceso como protegido o privado).

El patrón singleton se implementa creando en nuestra clase un método que crea una instancia del objeto solo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor (con modificadores de acceso como protegido o privado).

La instrumentación del patrón puede ser delicada en programas con múltiples hilos de ejecución. Si dos hilos de ejecución intentan crear la instancia al mismo tiempo y esta no existe todavía, solo uno de ellos debe lograr crear el objeto. La solución clásica para este problema es utilizar exclusión mutua en el método de creación de la clase que implementa el patrón.

Las situaciones más habituales de aplicación de este patrón son aquellas en las que dicha clase controla el acceso a un recurso físico único (como puede ser el ratón o un archivo abierto en modo exclusivo) o cuando cierto tipo de datos debe estar disponible para todos los demás objetos de la aplicación.

El patrón singleton provee una única instancia global gracias a que:

La propia clase es responsable de crear la única instancia. Permite el acceso global a dicha instancia mediante un método de clase. Declara el constructor de clase como privado para que no sea instanciable directamente. Al estar internamente autoreferenciada, en lenguajes como Java, el recolector de basura no actúa.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Implementación del patrón Singleton con Typescript**

Uno de los patrones de diseño de creación más populares es el patrón Singleton que restringe la creación de instancias de una clase a un objeto.

En esta publicación, le mostraré cómo usar el patrón junto con TypeScript.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Es genial con Typescript**

La biblia de los patrones de diseño, a saber, el libro de Gang of Four (GoF), presenta la aplicación de patrones utilizando el lenguaje C ++, un lenguaje estáticamente tipado.

TypeScript permite implementar el patrón Singleton gracias a las siguientes características:

* soporte para modificadores de acceso (privado, protegido, público),
* soporte para métodos estáticos,
* siendo un lenguaje estáticamente escrito.

class Singleton {

private static instance: Singleton;

private constructor() {

// initialition

}

static getInstance() {

if (!Singleton.instance) {

Singleton.instance = new Singleton();

}

return Singleton.instance;

}

}

export default Singleton;

Luego podemos crear instancias de Singleton que harán referencia al mismo objeto en memoria.

import Singleton from './Singleton';

const a = Singleton.getInstance();

const b = Singleton.getInstance();

console.log("¿A es igual a B?", a === b);

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Observer (patrón de diseño)**

Observador (en inglés:[Observer](https://en.wikipedia.org/wiki/Observer_pattern)) es un [patrón de diseño](https://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o) de software que define una dependencia del tipo uno a muchos entre objetos, de manera que cuando uno de los objetos cambia su estado, notifica este cambio a todos los dependientes. Se trata de un patrón de comportamiento (existen de tres tipos: creación, estructurales y de comportamiento), por lo que está relacionado con algoritmos de funcionamiento y asignación de responsabilidades a [clases](https://es.wikipedia.org/wiki/Clase_(inform%C3%A1tica)) y [objetos](https://es.wikipedia.org/wiki/Objeto_(programaci%C3%B3n)).

Los patrones de comportamiento describen no solamente estructuras de relación entre objetos o clases sino también esquemas de comunicación entre ellos y se pueden clasificar en función de que trabajen con clases (método plantilla) u objetos (cadena de responsabilidad, comando, iterador, recuerdo, observador, estado, estrategia, visitante).

La variación de la encapsulación es la base de muchos patrones de comportamiento, por lo que cuando un aspecto de un programa cambia frecuentemente, estos patrones definen un objeto que encapsula dicho aspecto. Los patrones definen una clase abstracta que describe la encapsulación del objeto.

Este patrón también se conoce como el patrón de publicación-inscripción o modelo-patrón. Estos nombres sugieren las ideas básicas del patrón, que son: el objeto de datos, que se le puede llamar Sujeto a partir de ahora, contiene atributos mediante los cuales cualquier objeto observador o vista se puede suscribir a él pasándole una referencia a sí mismo. El Sujeto mantiene así una lista de las referencias a sus observadores. Los observadores a su vez están obligados a implementar unos métodos determinados mediante los cuales el Sujeto es capaz de notificar a sus observadores suscritos los cambios que sufre para que todos ellos tengan la oportunidad de refrescar el contenido representado. De manera que cuando se produce un cambio en el Sujeto, ejecutado, por ejemplo, por alguno de los observadores, el objeto de datos puede recorrer la lista de observadores avisando a cada uno. Este patrón suele utilizarse en los [entornos de trabajo](https://es.wikipedia.org/wiki/Framework) de interfaces gráficas orientados a objetos, en los que la forma de capturar los eventos es suscribir listeners a los objetos que pueden disparar eventos.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

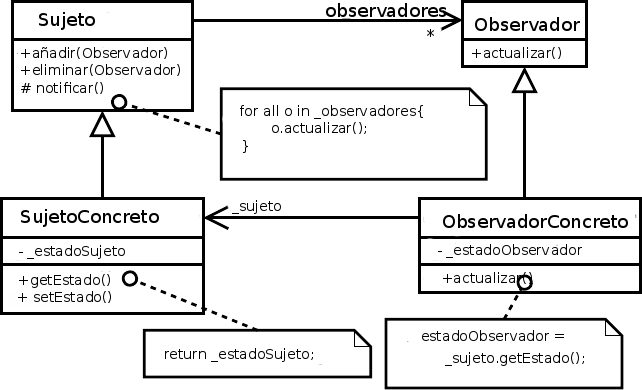
**Objetivo**

Definir una dependencia uno a muchos entre objetos, de tal forma que cuando el objeto cambie de estado, todos sus objetos dependientes sean notificados automáticamente. Se trata de desacoplar la clase de los objetos clientes del objeto, aumentando la modularidad del lenguaje, creando las mínimas dependencias y evitando bucles de actualización (espera activa o sondeo). En definitiva, normalmente, se usará el patrón observador cuando un elemento quiere estar pendiente de otro, sin tener que estar comprobando de forma continua si ha cambiado o no.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Motivación**

Si se necesita consistencia entre clases relacionadas, pero con independencia, es decir, con un bajo [acoplamiento](https://es.wikipedia.org/wiki/Acoplamiento_(inform%C3%A1tica)).

[](https://github.com/JasanHdz/javascript-professional/blob/master/assets/observer.png)

El patrón observador es la clave del patrón de arquitectura Modelo Vista Controlador (MVC).1​ De hecho el patrón fue implementado por primera vez en el MVC de Smalltalk basado en un entorno de trabajo de interfaz.2​ Este patrón está implementado en numerosos bibliotecas y sistemas, incluyendo todos los toolkits de GUI.

Patrones relacionados: publicador-subscriptor, mediador, singleton.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Participantes**

Habrá sujetos concretos cuyos cambios pueden resultar interesantes a otros y observadores a los que al menos les interesa estar pendientes de un elemento y en un momento dado, reaccionar ante sus notificaciones de cambio. Todos los sujetos tienen en común que un conjunto de objetos quieren estar pendientes de ellos. Cualquier elemento que quiera ser observado tiene que permitir indicar:

1. "Estoy interesado en tus cambios".
2. "Ya no estoy interesado en tus cambios".

El observable tiene que tener, además, un mecanismo de aviso a los interesados. A continuación se detallan a los participantes de forma desglosada:

**Sujeto (subject):** El sujeto proporciona una interfaz para agregar (attach) y eliminar (detach) observadores. El Sujeto conoce a todos sus observadores.

**Observador (observer):** Define el método que usa el sujeto para notificar cambios en su estado (update/notify). **Sujeto concreto (concrete subject):** Mantiene el estado de interés para los observadores concretos y los notifica cuando cambia su estado. No tienen porque ser elementos de la misma jerarquía.

**Observador concreto (concrete observer):** Mantiene una referencia al sujeto concreto e implementa la interfaz de actualización, es decir, guardan la referencia del objeto que observan, así en caso de ser notificados de algún cambio, pueden preguntar sobre este cambio.

Es uno de los patrones más utilizados, algunos ejemplos típicos son

* Newsletter
* Sockets
* Listeners en páginas web

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Implicaciones sobre Observer**

**Problema 1:**

Para evitar que el observador concreto tenga una asociación con el sujeto concreto, se podría hacer que la relación entre sujeto y observador fuese bidireccional, evitando así asociaciones concretas, el problema es que dejaría de ser una interfaz. El que deje de ser una interfaz puede producir problemas si el lenguaje de programación no soporta la [herencia múltiple](https://es.wikipedia.org/wiki/Herencia_m%C3%BAltiple).

Se podría eliminar la bidireccionalidad de la asociación pasando el sujeto como parámetro al método actualizar y ya no se tendría que referenciar el objeto observado. Esto podría causar problemas si se observan varios objetos, tanto de la misma clase como de distintas, ya que no elimina dependencias, y para hacer operaciones específicas sobre el objeto actualizado obliga a hacer en la implementación.

**Problema 2:** Si hay muchos sujetos sin observador, la estructura de los observadores está desaprovechada, para solucionarlo se puede tener un intermediario que centralice el almacenamiento de la asociación de cada sujeto con sus observadores. Para esta solución se crea ese gestor de observadores usando el patrón [singleton](https://es.wikipedia.org/wiki/Singleton) (instancia única), ya que proporciona una única referencia y no una por cada sujeto. El gestor aunque mejora el aprovechamiento del espacio, hace que se reduzca el rendimiento y se pierde eficiencia en el método notificar.

**Problema 3:** El responsable de iniciar la comunicación es el sujeto concreto, pero se puede dar un problema cuando el objeto concreto está siendo actualizado de forma continua ya que debido a esto se tendría que realizar muchas actualizaciones en muy poco tiempo. La solución sería suspender temporalmente las llamadas al método de actualización/notificación; por ejemplo, haciendo que el cliente pueda activar o desactivar las notificaciones y notificar todos los cambios cuando las vuelva a habilitar. El patrón Estado sería una posible solución para diseñar esta variante de no notificar si no se han dado cambios o hacerlo en caso de que si.

**Problema 4 (referencias inválidas):** A la hora de implementar este patrón se debe tener cuidado cuando un elemento observable desaparece. En ciertos lenguajes será el gestor de memoria el que cada cierto tiempo debe de limpiar las referencias liberadas, pero si un observable que sigue siendo observado puede no liberarse nunca. Para solucionar este problema puede crearse una función destruir que notifique al gestor que el elemento observable va a desaparecer y si no se está usando la variante del gestor el observable directamente desregistrará a sus observadores. Antes de esto hay que eliminar las referencias a este elemento, por tanto, hay que eliminar a los observadores antes de eliminar al observable, ya que así se evitará tanto que aparezcan referencias inválidas al objeto una vez este haya sido eliminado, como que se produzcan operaciones inválidas intentando invocarlo.

Se puede avisar a los observadores creando un método actualizar especial, en el que se tendrían dos opciones:

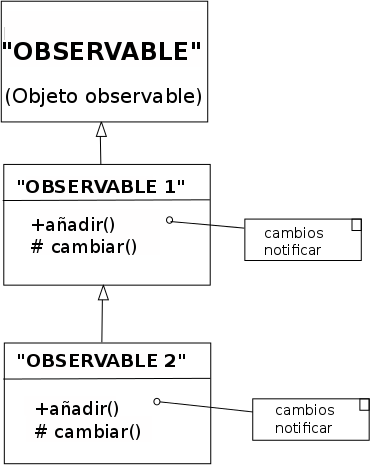
1. El observador también muere.
2. El observador sigue vivo, pero apunta a nulo.

**Problema 5:** Ya que se debe asegurar la consistencia del estado del sujeto antes de iniciar una notificación, siempre se notificará al final, ya que aunque en entorno multihilo se notifica antes de hacer los cambios, puede que los observadores soliciten información al observable cuando aún se van a hacer más cambios y se darían problemas de consistencia si se accede a un estado que aún no es el definitivo. De esta forma, los observadores ya no accederán a sujetos en estado inconsistente.

Por ejemplo:

Secuencia incorrecta: a b c notificar() d e f Secuencia correcta: a b c d e f notificar()

Jerarquía con varios tipos des observadores: en este caso el hilo redefine cambios, no los notifica.

[](https://github.com/JasanHdz/javascript-professional/blob/master/assets/JerarquiasObservador.png)

Jerarquía de varios observadores

**Problema 6:** En mecanismos de notificación tradicionalmente hay dos opciones: pull que es la que propone el patrón observador; y push que es la que se tendría si se incluye información como parámetros en el mecanismo de actualización. El problema de hacer esto es que la interfaz del observador se vuelve más específica y por tanto menos genérica y reutilizable.

PULL: los objetos avisan de que han cambiado y el observador pregunta cuál ha sido el cambio.

PUSH: minimiza (eficiencia) que cuando algo cambia y se informará a todos los interesados, se realicen el menor número de llamadas posibles.

Dependiendo del problema que haya que resolver, se habrá de valorar que implementación se ajusta mejor para resolverlo de la forma más eficiente y efectiva o si las variantes anteriores pueden combinarse entre sí dependiendo de las características de escenario concreto. Por ejemplo, la opción 2 podría aplicarse cuando interese aplicar en un sujeto concreto n métodos seguidos y no se quiere notificar hasta que todos finalicen su ejecución.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Implementación del patrón Observer con Typescript**

Docuento HTML:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Observer</title>

</head>

<body>

<h1>Precio del Bitcoin</h1>

<p><em id="price">$0.00</em></p>

<input type="text" id="value">

<script src="index.ts"></script>

</body>

</html>

Lógica de Typescript

// Definiendo algunas interfaces

interface Observer {

updated: (data: any) => void;

}

interface Subject {

subscribe: (observer: Observer) => void;

unsubscribe: (observer: Observer) => void;

}

class BitcoinPrice implements Subject {

observers: Observer[] = [];

constructor() {

const el: HTMLInputElement = document.querySelector("#value");

el.addEventListener('input', () => {

this.notify(el.value);

})

}

subscribe(observer: Observer) {

this.observers.push(observer)

}

unsubscribe(observer: Observer) {

const index = this.observers.findIndex(obs => {

return obs === observer;

})

this.observers.splice(index, 1);

}

notify(data: any) {

this.observers.forEach(observer => observer.updated(data))

}

}

class PriceDisplay implements Observer {

private el: HTMLElement;

constructor() {

this.el = document.querySelector("#price");

}

updated(data: any) {

this.el.innerText = data;

}

}

const value = new BitcoinPrice();

const display = new PriceDisplay();

// Subscribimos el Display al Value

value.subscribe(display);

// Simulamos unsubscribe usando un setTimeout de 5 segundos

setTimeout(

() => value.unsubscribe(display),

5000

)

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Casos de uso del patrón Observer: Redux**

[](https://github.com/JasanHdz/javascript-professional/blob/master/assets/redux.png)

Te ayuda a escribir aplicaciones que se comportan de manera consistente, corren en distintos ambientes (cliente, servidor y nativo), y son fáciles de probar. Además de eso, provee una gran experiencia de desarrollo, gracias a [edición en vivo combinado con un depurador sobre una línea de tiempo](https://github.com/reduxjs/redux-devtools).

Puedes usar Redux combinado con React, o cual cualquier otra librería de vistas. Es muy pequeño (2kB) y no tiene dependencias.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Conceptos básicos**

Redux de por si es muy simple.

Imagine que el estado de su aplicación se describe como un simble objeto. Por ejemplo, el estado de una aplicación de tareas (TODO List) puede tener el siguiente aspecto:

{

todos: [{

text: 'Comer',

completed: true

}, {

text: 'Hacer ejercicio',

completed: false

}],

visibilityFilter: 'SHOW\_COMPLETED'

}

Este objeto es como un “modelo” excepto que no hay setters. Esto es así para que diferentes partes del código no puedan cambiar el estado arbitrariamente, causando errores difíciles de reproducir.

Para cambiar algo en el estado, es necesario enviar una acción. Una acción es un simple objeto en JavaScript (observe cómo no introducimos ninguna magia) que describe lo que sucedió. A continuación mostramos algunos ejemplos de acciones:

{ type: 'ADD\_TODO', text: 'Ir a nadar a la piscina' }

{ type: 'TOGGLE\_TODO', index: 1 }

{ type: 'SET\_VISIBILITY\_FILTER', filter: 'SHOW\_ALL' }

Hacer valer que cada cambio sea descrito como una acción nos permite tener una claro entendimiento de lo que está pasando en la aplicación. Si algo cambió, sabemos por qué cambió. Las acciones son como migas de pan (el rastro) de lo que ha sucedido. Finalmente, para juntar el estado y las acciones entre si, escribimos una función llamada reductor (reducer). Una vez más, nada de magia sobre él asunto, es sólo una función que toma el estado y la acción como argumentos y devuelve el siguiente estado de la aplicación. Sería difícil escribir tal función para una aplicación grande, por lo que escribimos funciones más pequeñas que gestionan partes del estado:

function visibilityFilter(state = 'SHOW\_ALL', action) {

if (action.type === 'SET\_VISIBILITY\_FILTER') {

return action.filter;

} else {

return state;

}

}

function todos(state = [], action) {

switch (action.type) {

case 'ADD\_TODO':

return state.concat([{ text: action.text, completed: false }]);

case 'TOGGLE\_TODO':

return state.map((todo, index) =>

action.index === index ?

{ text: todo.text, completed: !todo.completed } :

todo

)

default:

return state;

}

}

Y escribimos otro reductor que gestiona el estado completo de nuestra aplicación llamando a esos dos reductores por sus respectivas state keys:

function todoApp(state = {}, action) {

return {

todos: todos(state.todos, action),

visibilityFilter: visibilityFilter(state.visibilityFilter, action)

};

}

Esto es básicamente toda la idea de Redux. Tenga en cuenta que no hemos utilizado ninguna API de Redux. Ya se incluyen algunas utilidades para facilitar este patrón, pero la idea principal es que usted describe cómo su estado se actualiza con el tiempo en respuesta a los objetos de acción, y el 90% del código que se escribe es simplemente JavaScript, sin uso de Redux en si mismo, sus APIs, o cualquier magia.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Patrón Decoratory casos de uso**

Decorator (patrón de diseño)

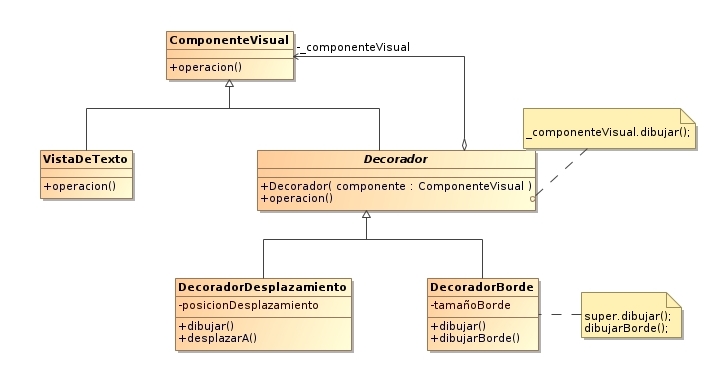
El patrón [Decorator](https://es.wikipedia.org/wiki/Decorator_(patr%C3%B3n_de_dise%C3%B1o)) responde a la necesidad de añadir dinámicamente funcionalidad a un Objeto. Esto nos permite no tener que crear sucesivas clases que hereden de la primera incorporando la nueva funcionalidad, sino otras que la implementan y se asocian a la primera.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

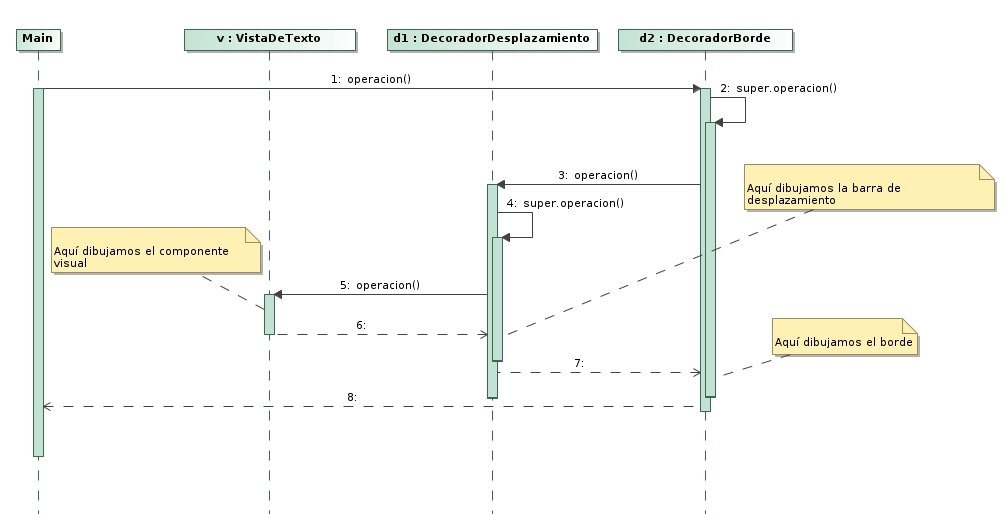
**Decorator motivation**

Un ejemplo para poder ver la aplicabilidad del patrón decorador podría ser el siguiente:

Disponemos de una herramienta para crear interfaces gráﬁcas, que permite añadir funcionalidades como bordes o barras de desplazamiento a cualquier componente de la interfaz. Una posible solución sería utilizar la herencia para extender las responsabilidades de la clase. Si optamos por esta solución, estaríamos haciendo un diseño inflexible (estático), ya que el cliente no puede controlar cuándo y cómo decorar el componente con esa propiedad. La solución está en encapsular dentro de otro objeto, llamado Decorador, las nuevas responsabilidades. El decorador redirige las peticiones al componente y, además, puede realizar acciones adicionales antes y después de la redirección. De este modo, se pueden añadir decoradores con cualidades añadidas recursivamente.

[](https://github.com/JasanHdz/javascript-professional/blob/master/assets/decorator.png)

En este diagrama de clases, podemos ver que la interfaz decorador implementa la interfaz del componente, redirigiendo todos los métodos al componente visual que encapsula. Las subclases decoradoras refinan los métodos del componente, añadiendo responsabilidades. También se puede ver que el cliente no necesita hacer distinción entre los componentes visuales decorados y los sin decorar.

[](https://github.com/JasanHdz/javascript-professional/blob/master/assets/secuencia.png)

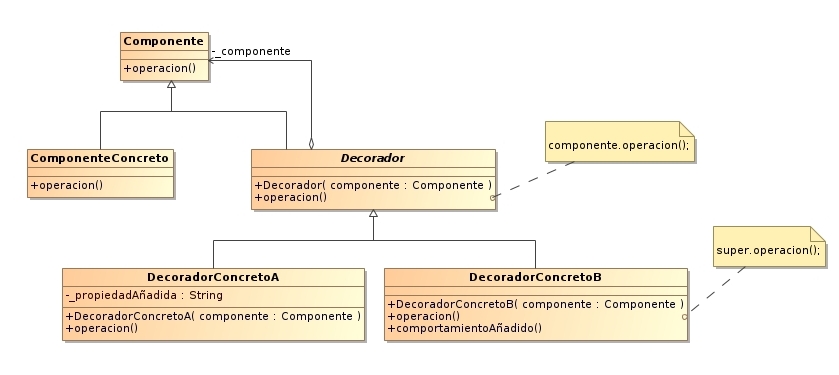
[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Decorator Aplicabilidad**

Añadir responsabilidades a objetos individuales de forma dinámica y transparente Responsabilidades de un objeto pueden ser retiradas Cuando la extensión mediante la herencia no es viable Hay una necesidad de extender la funcionalidad de una clase, pero no hay razones para extenderlo a través de la herencia. Existe la necesidad de extender dinámicamente la funcionalidad de un objeto y quizás quitar la funcionalidad extendida.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Decorator: Estructura**

[](https://github.com/JasanHdz/javascript-professional/blob/master/assets/decoratorgenerico.png)

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Decorator: Participantes**

* Componente Deﬁne la interfaz para los objetos que pueden tener responsabilidades añadidas.
* Componente Concreto Deﬁne un objeto al cual se le pueden agregar responsabilidades adicionales.
* Decorador Mantiene una referencia al componente asociado. Implementa la interfaz de la superclase Componente delegando en el componente asociado.
* Decorador Concreto Añade responsabilidades al componente.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Decorator Colaboraciones:**

El decorador redirige las peticiones al componente asociado. Opcionalmente puede realizar tareas adicionales antes y después de redirigir la petición

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Decorator Consecuencias**

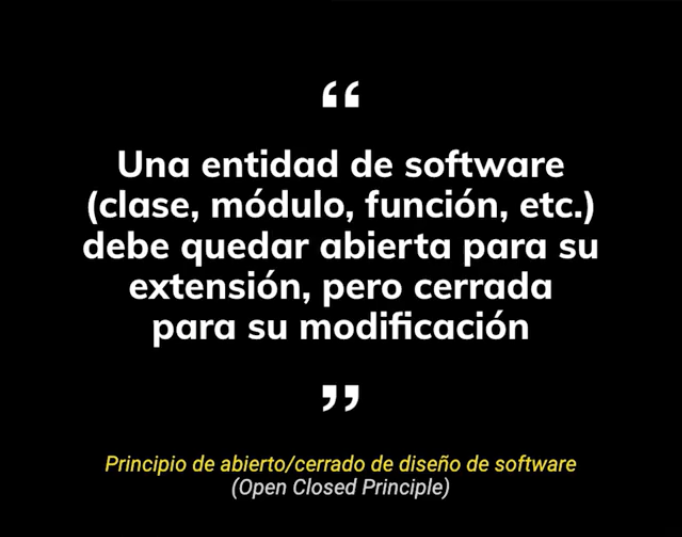
Más flexible que la herencia. Al utilizar este patrón, se pueden añadir y eliminar responsabilidades en tiempo de ejecución. Además, evita la utilización de la herencia con muchas clases y también, en algunos casos, la herencia múltiple. Evita la aparición de clases con muchas responsabilidades en las clases superiores de la jerarquía. Este patrón nos permite ir incorporando de manera incremental responsabilidades. Genera gran cantidad de objetos pequeños. El uso de decoradores da como resultado sistemas formados por muchos objetos pequeños y parecidos. Puede haber problemas con la identidad de los objetos. Un decorador se comporta como un envoltorio transparente. Pero desde el punto de vista de la identidad de objetos, estos no son idénticos, por lo tanto no deberíamos apoyarnos en la identidad cuando estamos usando decoradores.

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

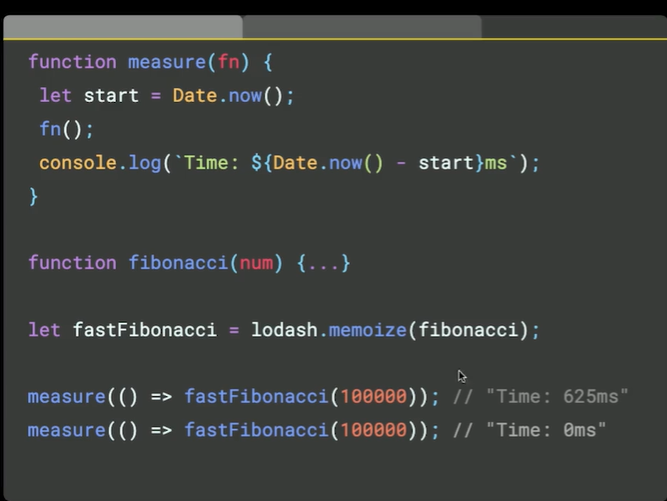
**Decorator Implementación**

El patrón Decorator soluciona este problema de una manera mucho más sencilla y extensible.

Se crea a partir de Ventana la subclase abstracta VentanaDecorator y, heredando de ella, BordeDecorator y BotonDeAyudaDecorator. VentanaDecorator encapsula el comportamiento de Ventana y utiliza composición recursiva para que sea posible añadir tantas “capas” de Decorators como se desee. Podemos crear tantos Decorators como queramos heredando de VentanaDecorator.

[](https://github.com/JasanHdz/javascript-professional/blob/master/assets/openclosed.png)

[](https://github.com/JasanHdz/javascript-professional/blob/master/assets/monkeypatching.png)

[](https://github.com/JasanHdz/javascript-professional/blob/master/assets/timeexecute.png)

[🡡 volver al inicio](https://github.com/JasanHdz/javascript-professional/tree/master/notes#tabla-de-contenido)

**Implementación del patrón Decorator**

Documento HTML:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Decorator Patterns</title>

</head>

<body>

<div>

<h1>Decorator Desgin Patterns</h1>

<label for="email">Email</label>

<input type="text" id="email">

</div>

</body>

</html>

class Field {

errors: string[];

input: HTMLInputElement;

constructor(input: HTMLInputElement) {

this.input = input;

this.errors = [];

let errorMessage = document.createElement('p');

errorMessage.className = 'text-danger';

this.input.parentNode.insertBefore(errorMessage, this.input.nextSibling);

this.input.addEventListener('input', () => {

this.errors = [];

this.validate();

errorMessage.innerText = this.errors[0] || ' ';

})

}

validate() {}

}

function RequireFieldDecorator(field: Field): Field {

let validate = field.validate;

field.validate = function () {

validate()

let value = field.input.value;

if (!value) {

field.errors.push("Requisito");

}

};

return field;

}

function EmailFieldDecorator(field: Field): Field {

let validate = field.validate;

field.validate = function () {

validate()

let value = field.input.value;

if (value.indexOf("@") === -1) {

field.errors.push("Debe ser un email");

}

};

return field;

}

let field = new Field(document.querySelector("#email"));

RequireFieldDecorator(field);

EmailFieldDecorator(RequireFieldDecorator(field));

**Publicar en NPM**

Para publicar en npm hay un requisito: necesitas una cuenta en npm, es gratis no cuesta nada, es muy fácil de hacer y esto es todo lo que vas a necesitar