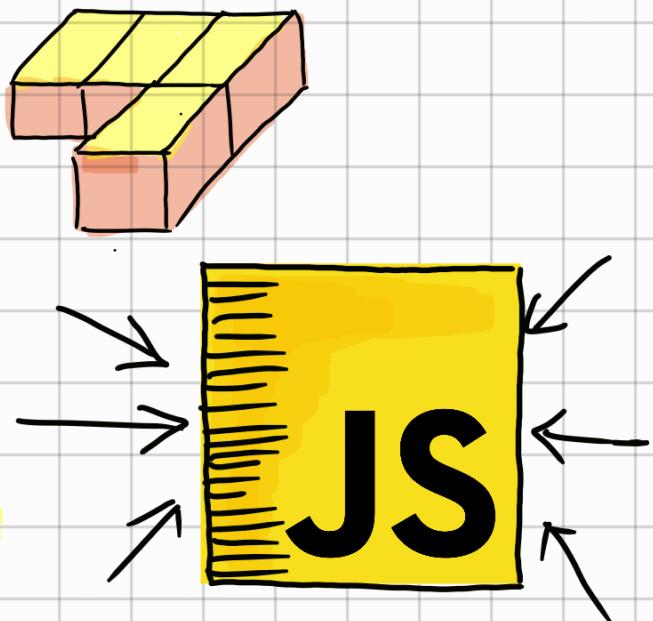


CURSO PROFESIONAL DE JAVASCRIPT

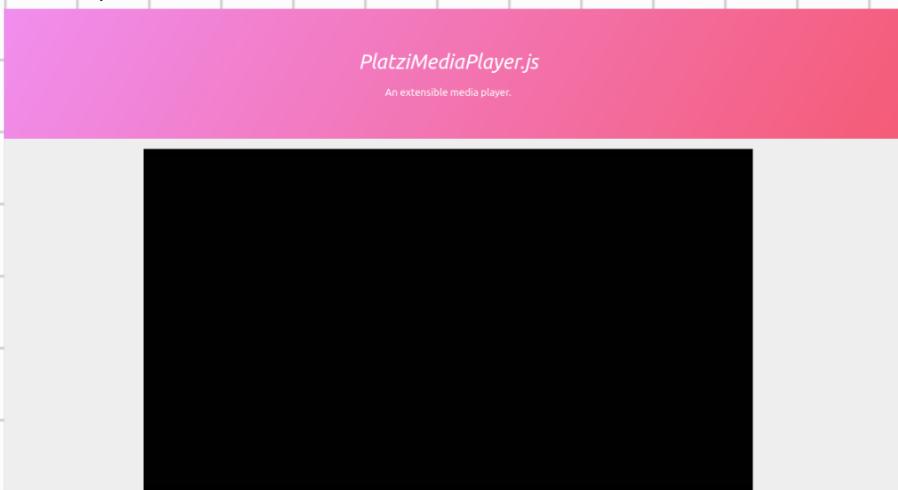
¿QUÉ SIGNIFICA SER PROFESIONAL DE JS?

- ⇒ Pasar de Jr a Master.
- ⇒ Fundamentos del lenguaje
- ⇒ Cómo funciona el lenguaje?
- ⇒ Promises, Getters/Setters, Proxies generadores.
- ⇒ Entornos de programación
- ⇒ Versado en código
- ⇒ Mejores prácticas
- ⇒ Ética → Ser un verdadero pro



INICIO DEL PROYECTO

- ⇒ Plataforma de videos
- ⇒ Plugins para el media Player
- ↑ modular!



@mdn HTMLMediaElement

El código que permite que el botón le de "play" ➡

#1. Crear el proyecto

npm init -y

#2. Instalar Live Server

npm i -D live-server

① Descargar la base del código HTML y CSS del repositorio de github.

#3. Correr Live Server

npm start

!!! En JS no existen clases, existen Objetos !!!

```
<script>
const video = document.querySelector("video")
const button = document.querySelector("button")

function MediaPlayer(config) { ← Objeto
  this.media = config.el
}

MediaPlayer.prototype.play = function() { ← Método
  this.media.play();
}
} Instanciarlo →
const player = new MediaPlayer({el:video}); // Pasar un archivo de configuración

button.onclick = () => player.play(); ← Llamar método
</script>
```

→ Existen la propiedad paused y la función pause() que nos permiten controlar el flujo del vídeo.

¿CÓMO LLEGA UN SCRIPT AL NAVEGADOR?

DOM: Representación de un documento HTML
<Document Object Model>

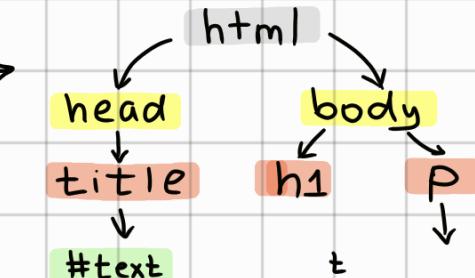


→ El navegador convierte el archivo a una estructura como de **árbol**

↓
Al terminar ocurre el evento

DOM Content Loaded

El documento se cargó al 100%.

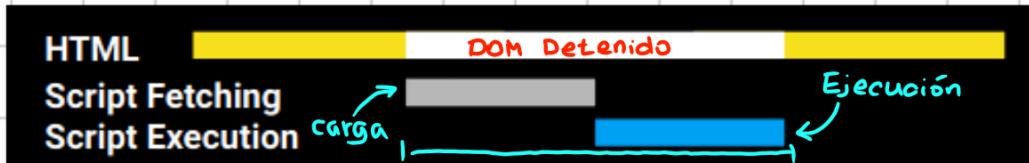


Cuando se tienen scripts embedidos
el navegador detiene el procesamiento
del DOM

Por esto, el mejor lugar para colocarlo es al final
justo antes del </body>



- En los scripts externos se debe "cargar el archivo" → **fetching**



ASYNC

- Permite realizar llamadas asíncronas
- Con async el **fetching** se realiza sin detener el DOM



La ejecución aún detiene el DOM

DEFER

- Esta asíncrona no detiene el procesamiento del DOM, y ejecuta /carga el script hasta el final

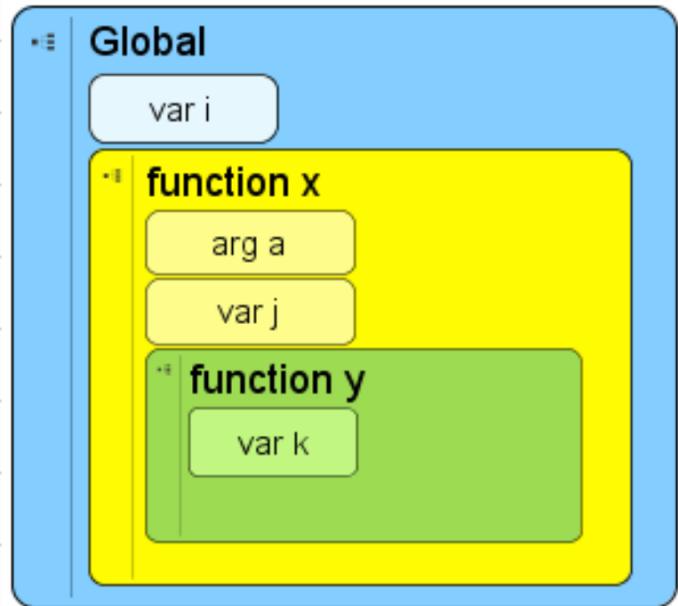


NO SE DETUVO

SCOPE

El scope es lo que define el tiempo de vida de una variable, también en qué partes del código pueden ser usadas.

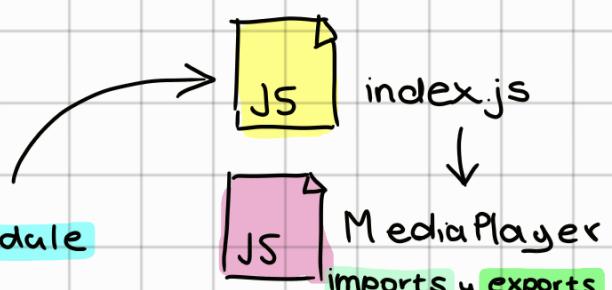
(a) Global: Están disponibles de forma global se usa la palabra **Var**, son accesibles por todos los scripts que se cargan en la página
⚠ Hay riesgo de sobreescritura



(b) Local: Son variables declaradas dentro de una función, solo son visibles dentro de la misma
(Incluyen los argumentos)

(c) Block: Variables definidas dentro de un bloque <>for,while<>
Se utilizan **Let y const** para declarar este tipo de variables.

(d) Módulo: Cuando se denota un script de tipo module con el atributo **type="module"** las variables son limitadas al archivo en el que están creadas.



CLOSURES

Un closure es una función interna que tiene acceso al scope de su función externa, ¡Incluso si la f(x) externa ejecuta un return!

```
function creaFunciones(){
    var nombre = 'Luis'
    // 'muestraNombre' es una función interna (un closure)
    function muestraNombre(){
        función interna
        console.log(nombre)
    }
    //Soy La referencia de una función 'muestraNombre'
    //y voy a recordar el estado de las variables
    //al momento que me invocuen :D
    return muestraNombre(también se puede hacer con objetos 🎯)
}
```

- Con closure podemos hacer **variables privadas**
- En resumen, los closures **recuerda el estado de las variables al momento de ser invocados y conserva el estado a través de las ejecuciones.**
- También tenemos funciones que **retornan funciones**.

THIS

En JS <this> representa algo distinto según el contexto

(a) En el global scope this es un objeto window.

(b) En un function scope this es un objeto window.

(c) Con strict mode this es undefined.

This en Objetos

Se refiere al objeto que está ejecutando un bloque de código.



① This en las "clases" se refiere a un objeto instanciado

LOS MÉTODOS CALL, APPLY & BIND

Estos métodos son parte del prototype de function.

- ✓ → Estos métodos permiten cambiar el contexto (this)
- de una función.
- ✓

CALL

<función>.call(<Obj>, <arg1>, <arg2>, ...)

↑ Ejecuta la f(x) ↑ parámetros adicionales.

APPLY

<función>.apply(<Obj>, [<arg1>, <arg2>, ...])

↑ Ejecuta la f(x) ↑ Array con los parámetros.

BIND

const func = <función>.bind(<Obj>, <arg1>, <arg2>, ...)

↑ Retorna una función con el contexto del Obj

↑ parámetros adicionales.

func() → Ejecuta la función.

Nota: (a) Currying → Guardar con bind solo algunos argumentos y dejar los otros para la f(x) resultante

PROTOTYPE

>> En Javascript TODO SON OBJETOS!

Los objetos {} ↴

```
function Hero(name) {  
  const hero = {  
    name: name  
  }  
  
  hero.saludar = function () {  
    console.log('Hola soy ${this.name}')  
  }  
  
  return hero  
}  
  
zelda = Hero('Zelda');  
zelda.saludar()  
  
link = Hero('Link');  
link.saludar()
```

Console

```
Hola soy Zelda  
Hola soy Link
```

La manera de
crear objetos

Object.create(prototypeObject, propertiesObject)

prototypeObject

Si solo colocamos este parámetro, me crea un nuevo objeto vacío, pero con un prototipo definido

```
const heroMethods = {  
  saludar: function () {  
    console.log('Hola soy ${this.name}')  
  }  
}
```

const hero = Object.create(heroMethods)
console.log(hero)

1 Si hacemos un console.log para ver que tiene el objeto

Hay una mejor forma de asignar funciones a un objeto: Prototype

Un prototipo es un objeto como una propiedad en c/f(x)
Es un objeto y se le pueden adjuntar propiedades y métodos

```
1  function MediaPlayer(config) {  
2    this.media = config.el; ← Un objeto  
3    this.plugins = config.plugins || [];  
4  
5    this._initPlugins();  
6  }  
7  
8  MediaPlayer.prototype._initPlugins = function() {  
9    this.plugins.forEach(plugin => {  
10      plugin.run(this);  
11    });  
12 }
```

↳ Agregar métodos

Para crear nuevos "objetos" del tipo se utiliza <<new>>

Es como un CONSTRUCTOR →

const player = new MediaPlayer({ el: video, plugins: [new AutoPlay()]});

new es un atajo, crea un this y no será necesario llamar a Object Create ni retornar nada, new lo hace solo!

HERENCIA PROTOTIPAL

-proto- → Encadenamiento de prototipos para la herencia.

→ Guarda la Herencia

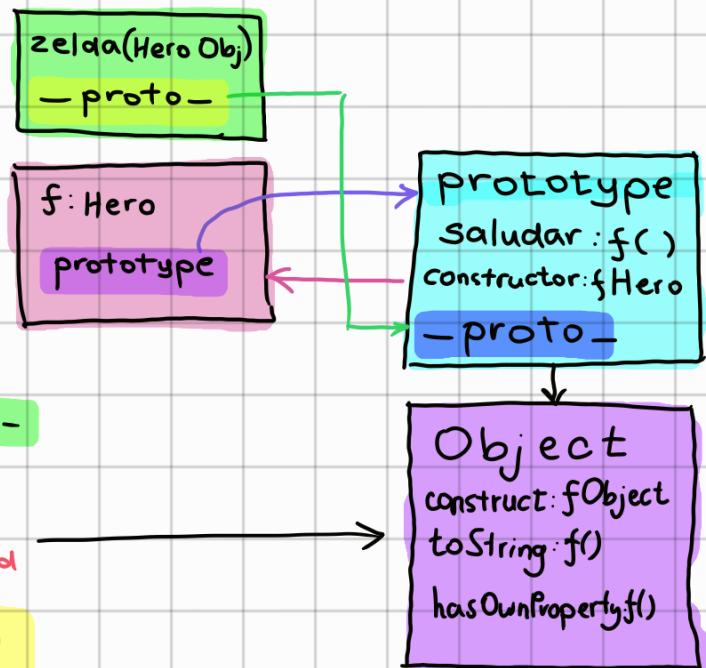
↳ toString()

↳ hasOwnProperty()

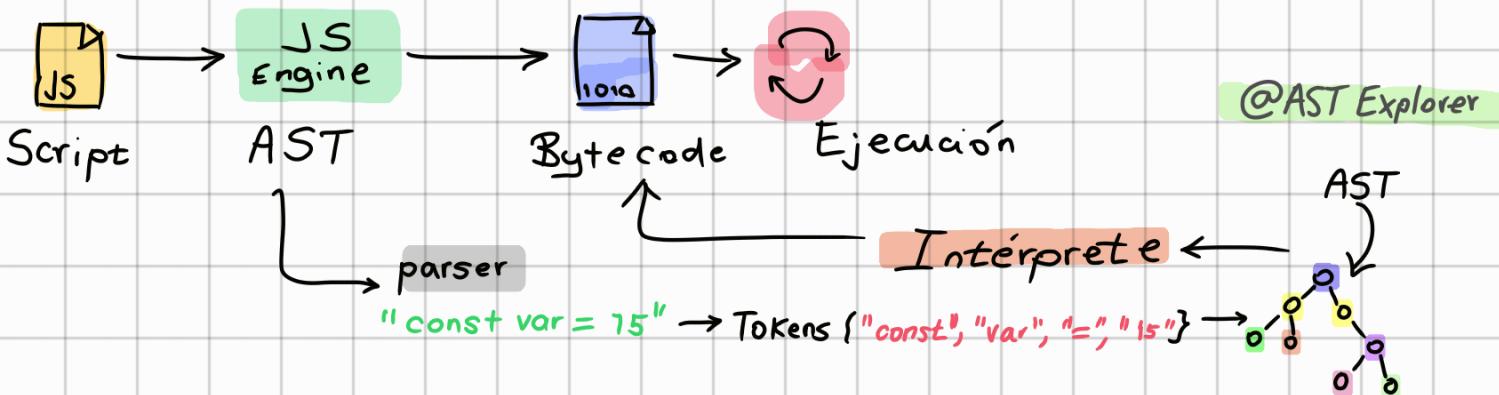
La herencia en cadena los protos. Para encontrar un método JS busca primero en el Objeto, si no lo encuentra, busca en -proto-. Si tampoco lo encuentra va al -proto- del -proto- y así sucesivamente.

Esto finaliza al llegar a Object
Si no existe, lanza un error de not found

Object.getPrototypeOf(<Obj>)



PARSERS Y ABSTRACT SYNTAX TREE



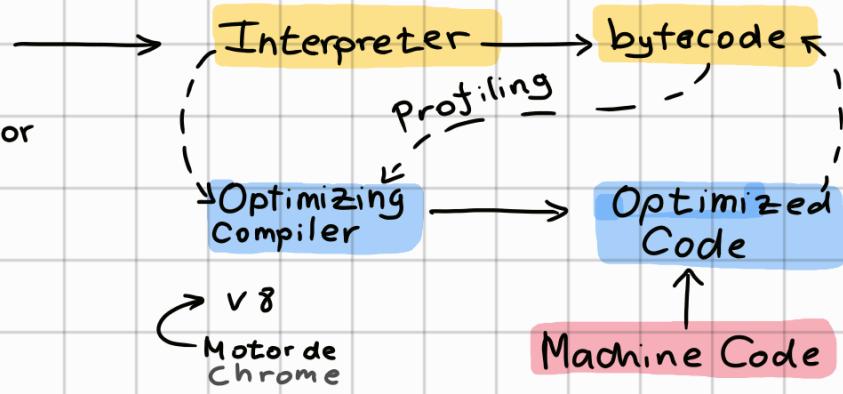
Bytecode vs Machine Code

- Assembly like

- Portatil (VM)

- Binario

- Procesador



EVENT LOOP

Es lo que hace que JS parezca multihilado cuando en realidad no lo es.

Stack

Heap



Schedule Task

Tareas que se van a realizar en el futuro

Task Queue

Tareas listas para Stack



El event loop controla el flujo de pasar los tasks al stack

PROMESAS

@MicroTasks Queue

Nos prometen que retornarán algo, pero que no oaremos el código

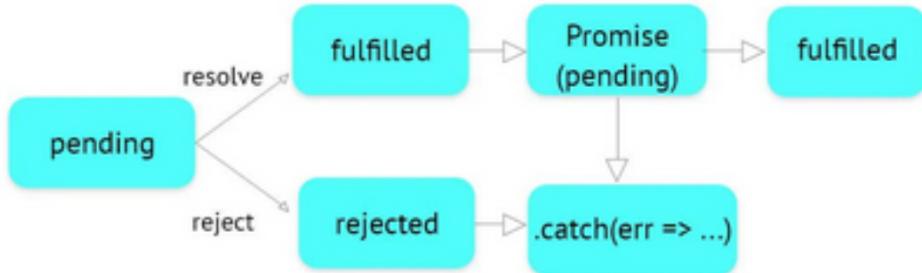
Cuando trabajamos con API's es mejor utilizar `async/await` que utilizar `then`

```
async function getTopMovies() { ← async permite usar await
  try {
    const response = await fetch(url); ← await retorna una
    const data = await response.json(); promesa
    return data;
  } catch (error) { ← Manejo del error
    // MAnejo del error
  }
}
```

Si la promesa retorna error

Convertir respuesta a JSON

El flujo de una promesa es:



¿Cuál es la diferencia entre `then` y `async/await`?

THEN

```
fetch(url)
  .then(response => response.json())
  .then(data => {
    console.log(data)
  })
```

AWAIT

```
const response = await fetch(url)
const data = await response.json()
console.log(data)
```

Las cosas son por aparte, y nosotros manejamos las promesas

Promise.all()

→ Recibe un array de promesas y las ejecuta todas al vez
(En paralelo)

→ Retorna un array con las respuestas

Promise.race()

→ Recibe un array de promesas
→ Solo retorna el valor de la promesa que se resuelva primero.

GETTERS & SETTERS

Son funciones que podemos usar en un **objeto** para tener propiedades virtuales

Se utilizan los **Keywords** **get** y **set** para crear estas propiedades

Retornan el valor

Asignan un valor

⇒ Los **getters** y **setter** se declaran como propiedades del objeto

```
const player = { ← Dentro de un
  play: () => this.play(),
  pause: () => this.pause(),
  media: this.media,
  get muted() {
    return this.media.muted; ↗ Get retorna algo
  },
  set muted(value) {
    this.media.muted = value;
  }
}
```

Set asigna un valor.

⇒ Se llaman con un objeto. propiedades

```
AutoPlay.prototype.run = function(player) {
  if(!player.muted) ← ¡No se utilizan (), solo
    player.muted = true; se llama como un atributo!
  player.play();   ↑ Este valor se pasa como si
  fuesen argumentos.
};
```

PROXY

Las intercepciones (traps) proveen acceso a las propiedades

Nos permite interceptar la lectura de las propiedades antes que la llamada llegue al Objeto podemos manipularla con lógica que nosotros definamos.

```
const Proxy = new Proxy(<b><target></b>, <b><handler></b>)
```

↓
Objeto a supervisar

↓
Lógica que define el comportamiento del proxy cuando una operación es realizada en él.

! Existe un operador **in** que indica si existe algo en un Objeto u arreglo.

GENERATORS

Kinda@Dynamic Programming

Son funciones especiales, pueden pausar su ejecución
y luego volver al punto donde quedaron

function* ← El * indica que es un generador

Características de los generadores

- Empiezan suspendidos y se tiene que llamar next para que se ejecuten
- Regresan un value y un boolean que nos indica si ya se acabo o no la ejecución del generador.
- yield es la instrucción que regresa un valor cada vez que llamamos a next y detiene la ejecución del generador .

↓
El hilo principal no se bloquea cuando el generador está pausado!

```
function* generador(){
  console.log('GENERATOR START')
  yield 1 ← regresa el valor
  yield 2 c/vez que se llama
  yield 3 next()
  console.log('GENERATOR END')
}

const gen = generador() ← Pausada
console.log( gen.next())
console.log( gen.next()) ← Se avanza por pasos
console.log( gen.next())
console.log( gen.next())
```

El generador tiene una propiedad done, que indica si la ejecución se ha completado.

FETCH... ¿CÓMO CANCELAR PETICIONES?

Fetch ofrece el Abort Controller que nos permite enviar una señal en plena ejecución para detenerla

Ejemplo:

```
const img = document.getElementById('huge-image')
var url = 'https://images.pexels.com/photos/974470/
  nature-stars-milky-way-galaxy-974470.jpeg?q=100'
var controller
async function loadRequest(){
  controller = new AbortController()
  try{
    const response = await fetch(url, { signal: controller.signal })
    const blob = await response.blob() ② blob es el binario que estamos pidiendo
    const imgUrl = URL.createObjectURL(blob)
    img.src = imgUrl
  } catch(err){
    console.log(err.message)
  }
}
function stopRequest(){
  controller.abort()
}
```

- La propiedad signal se puede usar para comunicarse o cancelar un request DOM
- El método abort() anula un request DOM antes de que se haya completado.
- La posibilidad de cancelar peticiones es muy útil en SPAs

.blob() ⇒ Binario de un request.

Ej: Una img en binario

INTERSECTION OBSERVER API

Proporciona una forma asíncrona de observar cambios en la intersección de un elemento con otro elemento ancestro o el **viewport** y así notificarlo para tomar alguna acción.

↑
@Parte visible del documento

→ Creando un Intersector

```
var options = {  
  root: document.querySelector('#scrollArea'),  
  rootMargin: '0px',  
  threshold: 1.0  
}  
  
var observer = new IntersectionObserver(callback, options);
```

La función se ejecuta cuando se cruce un umbral (THR) en una u otra dirección

Un threshold de 1.0 significa que cuando el 100% del elemento target esté visible dentro del elemento indicado en root, el callback es invocado.

→ Opciones

El objeto **options** controla las circunstancias bajo las que se deberá ejecutar callback

```
var options = {  
  root: document.querySelector('#scrollArea'),  
  rootMargin: '0px',  
  threshold: 1.0  
}  
  
var observer = new IntersectionObserver(callback, options);
```

→ Determinando un objeto para ser observado

```
var options = {  
  threshold: 0.25  
}  
  
var observer = new IntersectionObserver(callback, options)  
  
var target = document.querySelector('#elemento')  
observer.observe(target) // El contenedor en este ejemplo es toda  
// la pantalla del navegador  
  
1 Cuando el elemento target  
encuentra un threshold  
especificado por el  
IntersectionObserver, la  
función callback es invocada.  
  
function callback(entries, observer){  
  console.log( entries )  
}
```

0 pasamos la referencia de la función, no la ejecutamos

1 Cuando el elemento target encuentra un threshold especificado por el IntersectionObserver, la función callback es invocada.

2 La función callback recibe una lista(arreglo) de objetos IntersectionObserverEntry y el observer

-root: Elemento usado como viewport. Por default es el Viewport del browser

-rootMargin: Es el margen alrededor del elemento root. Genera un "cuadro delimitador" antes de calcular las intersecciones.

-Threshold: Número (o array) que indica qué porcentaje de visibilidad del elemento ejecuta el callback.

VISIBILITY CHANGE

Page visibility nos deja saber si un elemento es visible puede ser usado para ejecutar una acción cuando cambiamos de pestaña

```
document.addEventListener('visibilitychange', () => {  
  console.log(document.visibilityState);  
})
```

Callback

El objeto document controla la página web y todos sus elementos



- + Document.hidden → Indica si la página se encuentra oculta
- + Document.visibilityState → Indica la visibilidad actual del documento.
- + Document.onVisibilityChange

Es un Event Listener que proporciona el código para llamar cuando el evento visibilitychange se dispara.

¿CUÁNDO USARLO?

- Un sitio tiene un carrusel de imágenes que no debería avanzar a la siguiente diapositiva a no ser que el usuario esté viendo la página.
- Una aplicación que muestra un panel de información y no se quiere que se actualice la información del servidor cuando la página no está visible.
- Una página quiere detectar cuando se está pre cargando para poder mantener un recuento preciso de las visitas de página.
- Un sitio desea desactivar los sonidos cuando el dispositivo está en modo de espera (el usuario presiona el botón de encendido para apagar la pantalla).

document.visibilityState === 'visible'

El contenido de la página esta visible para el usuario. En la práctica, esto significa que la página es la pestaña de primer plano de una ventana no está minimizada.

document.visibilityState === 'hidden'

El contenido de la página no es visible para el usuario, ya sea porque la pestaña del documento está en el fondo o parte de una ventana que está minimizada, o porque la pantalla del dispositivo está apagada.

document.visibilityState === 'prerender'

El contenido de la página se está procesando previamente y no es visible para el usuario.

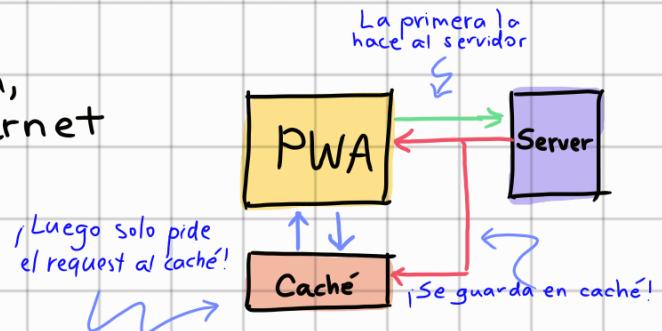
SERVICE WORKER

Es una capa que va a vivir entre el navegador y el internet.

Lo hacen similar a los proxies van a interceptar peticiones pero para guardar los resultados en caché

¿Para qué hacer esto?

→ La próxima vez que ocurre una petición, en lugar de pedirlo de nuevo a internet la busca en la caché, por lo tanto, funciona offline.



TYPESCRIPT



- Es un superset de JS que añade tipos a nuestras variables ayudando a la detección de errores de forma temprana y mejorando el autocompletado

≡ LOS NAVEGADORES NO ENTIENDEN TS ASÍ QUE SE OCUPA UN PARCEL

En TypeScript tenemos tipos explícitos para variables y funciones.

```
● ● ● ¿Cómo debemos implementar un Objeto
interface Product {← Interfaces → Type
  id: number;
  name: string; ← Tipado Explícito
  price: number;
  url: string;
  active: boolean;
  tags: string[];
  getRoundedPrice: (price: number) => number;
}
          ↑
          Tipode Retorno
```

Annotations in yellow:

- Three colored dots (red, yellow, green) above the interface definition.
- An arrow points from the question "¿Cómo debemos implementar un Objeto" to the interface definition.
- An arrow points from "Interfaces" to the opening brace of the interface.
- An arrow points from "Type" to the closing brace of the interface.
- An arrow points from "Tipado Explícito" to the type annotations on the properties.
- An arrow points from "Array de 1 solo tipo" to the array type annotation on the "tags" property.
- An arrow points from "Tipode Retorno" to the return type annotation on the "getRoundedPrice" method.

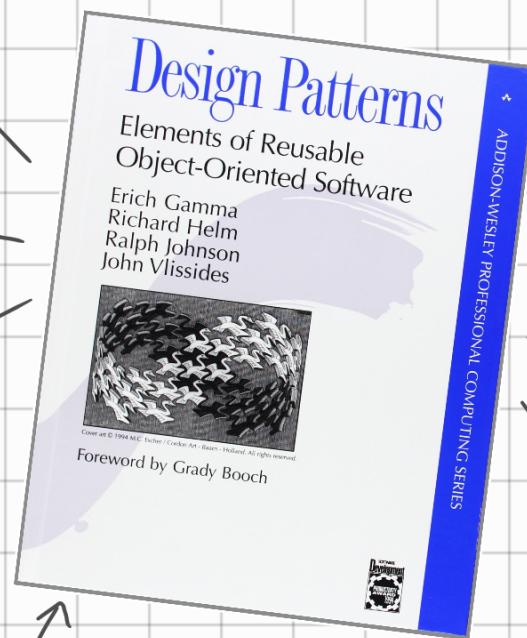
PATRONES DE DISEÑO

Son una solución para un problema en un contexto recurrente.

- Se refiere a la meta que se está tratando de alcanzar dentro de este contexto.
- El problema incluye todas las limitaciones que existen dentro de ese contexto.

DEBE SER UNA SOLUCIÓN
-GENÉRICA-

! Los patrones de diseño solo deben utilizarse si el problema aparece en el contexto específico
! No debemos buscar meterlos a la fuerza, introducen complejidad!

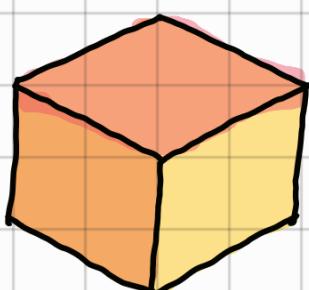


"En este libro aparecieron por primera vez"

CATEGORIAS

CREACIONALES

- Cómo crear Objetos?
- + Abstract Factory
- + Builder
- + Factory
- + Prototype
- + Singleton



ESTRUCTURALES

- Cómo formar nuevas estructuras flexibles y adaptables?
- + Adapter
- + Bridge
- + Composite
- + Decorator
- + Facade
- + Flyweight
- + Proxy

COMPORTAMIENTO

- Cómo se gestionan los algoritmos y responsabilidades entre objetos?
- + Chain Of Responsibility
- + Command
- + Interpreter
- + Iterator
- + Mediador
- + Memento
- + Observer
- + State
- + Strategy
- + Template Method
- + Visitor

SINGLETON

«Uno solo»

- Provee una única instancia global:

→ La propia clase es responsable de crear la única instancia.

→ Permite acceso global a la instancia mediante un método de la clase.

→ Tiene un constructor privado.

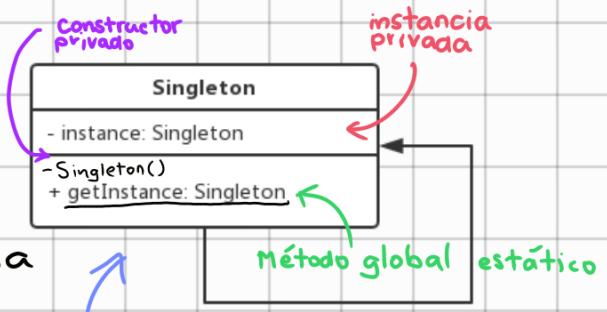


DIAGRAMA DE CLASES

You, seconds ago | 1 author (You)

```

class Singleton {
    private static instance: Singleton;
    private constructor() {}

    static getInstance(): Singleton {
        if (!Singleton.instance) {
            Singleton.instance = new Singleton();
        }
        return Singleton.instance;
    }
}
  
```

IMPLEMENTACIÓN EN TYPESCRIPT!

Redux implementa este patrón

OBSERVER

Es de los más usados: notificaciones, web sockets, listeners

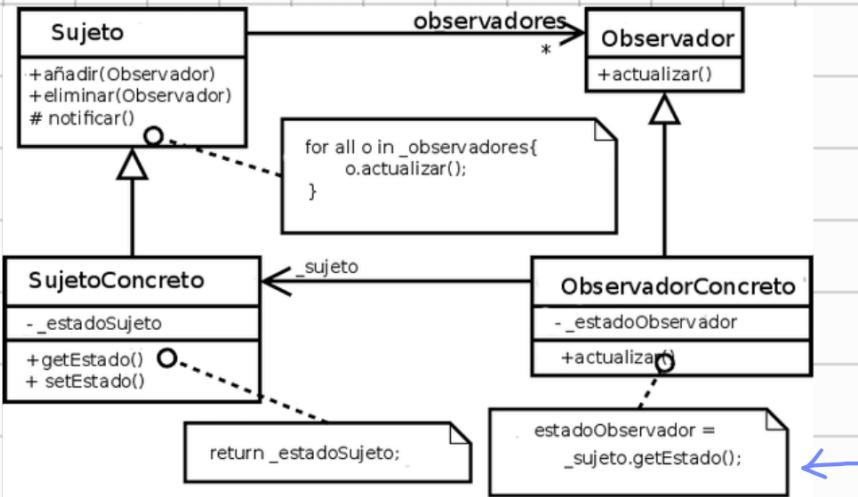


Diagrama UML del patrón Observer

#3 Clase Observador

```

class PriceDisplay implements Observer {
    update(data: any): void {
        console.log("New notification!", data);
    }
}
  
```

#1. Interfaces

```

interface Subject {
    subscribe: (observer: Observer) => void;
    unsubscribe: (observer: Observer) => void;
}

interface Observer {
    update: (data: any) => void;
}
  
```

#2. Clase del Sujeto

```

class BitCoinPrice implements Subject {
    private observers: Observer[] = [];

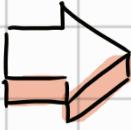
    subscribe(observer: Observer): void {
        this.observers.push(observer);
    }

    unsubscribe(observer: Observer): void {
        const index = this.observers.findIndex(obs => {
            return obs === observer
        });
        this.observers.splice(index, 1);
    }

    notify(data: any) {
        this.observers.forEach(obs => obs.update(data));
    }
}
  
```

DECORATOR

"Una entidad de Software debe quedar abierta para su extensión pero cerrada para su modificación"

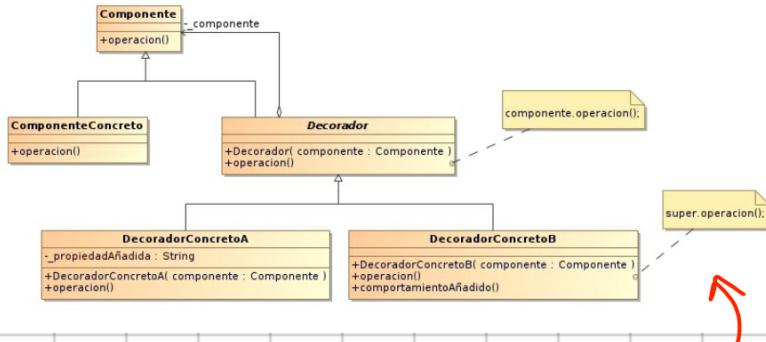


- Añade nuevas responsabilidades a un objeto de forma dinámica.

- Nos permite extender funcionalidad sin tener que usar subclases

Monkey Patching >>

Usar funciones para cambiar propiedades recibiendo los objetos



<<DECORA LAS INSTANCIAS SIN ALTERAR LA CLASE >>

@Memoización