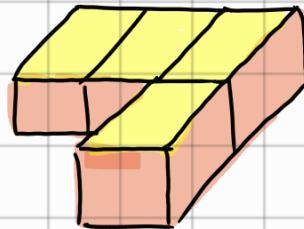


CURSO PROFESIONAL DE JAVASCRIPT

¿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!

PlatziMediaPlayer.js
An extensible media player.

@mdn HTMLMediaElement

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

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

#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>
```

¿CÓMO LLEGA UN SCRIPT AL NAVEGADOR?

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

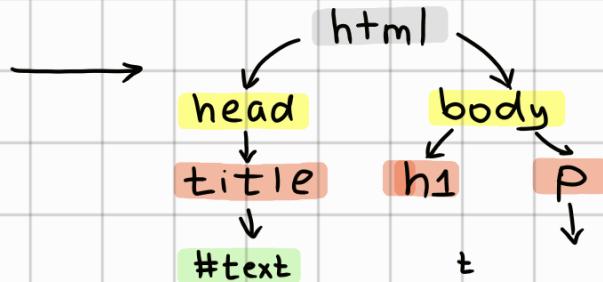


→ El navegador convierte el archivo a una estructura como de **árbol**

↓
Al terminar ocurre el evento

DOM Content Loaded

El documento se cargó al 100%.

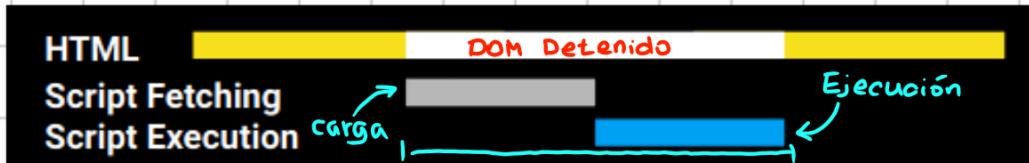


Cuando se tienen scripts embedidos
el navegador detiene el procesamiento
del DOM

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

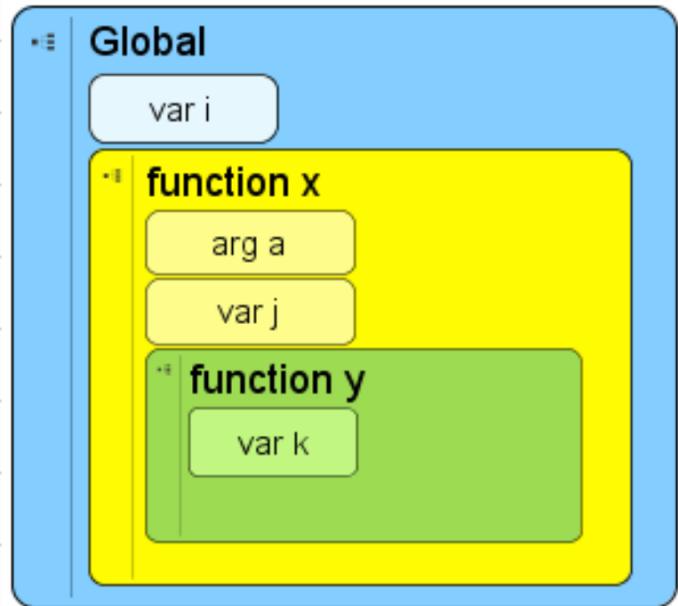
DOM Finalizado

Ejecución

SCOPE

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

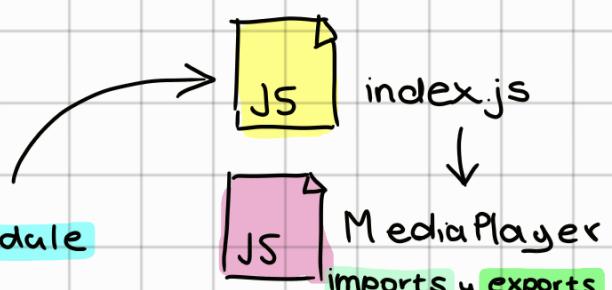
(a) Global: Están disponibles de forma global se usa la palabra **Var**, son accesibles por todos los scripts que se cargan en la página
⚠ Hay riesgo de sobreescritura



(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 invocadas 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()]});
```

↳ parámetros



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

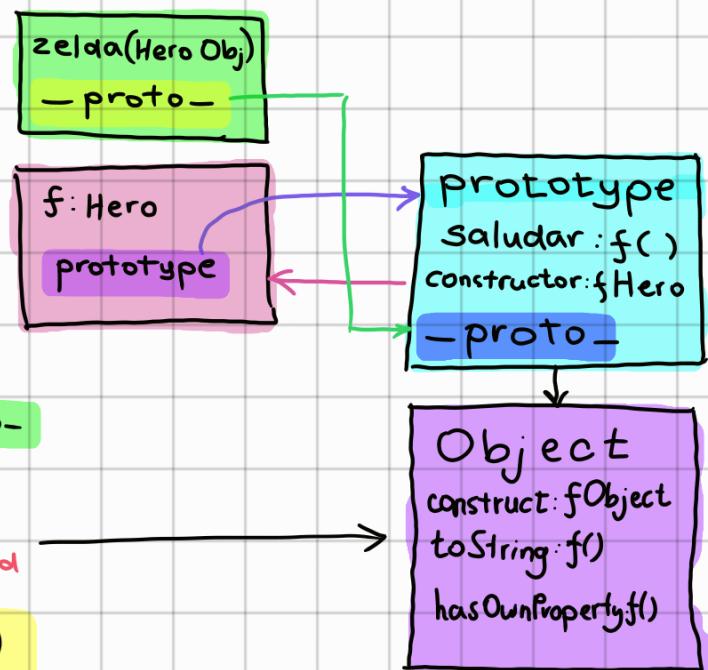
↳ toString()

↳ hasOwnProperty()

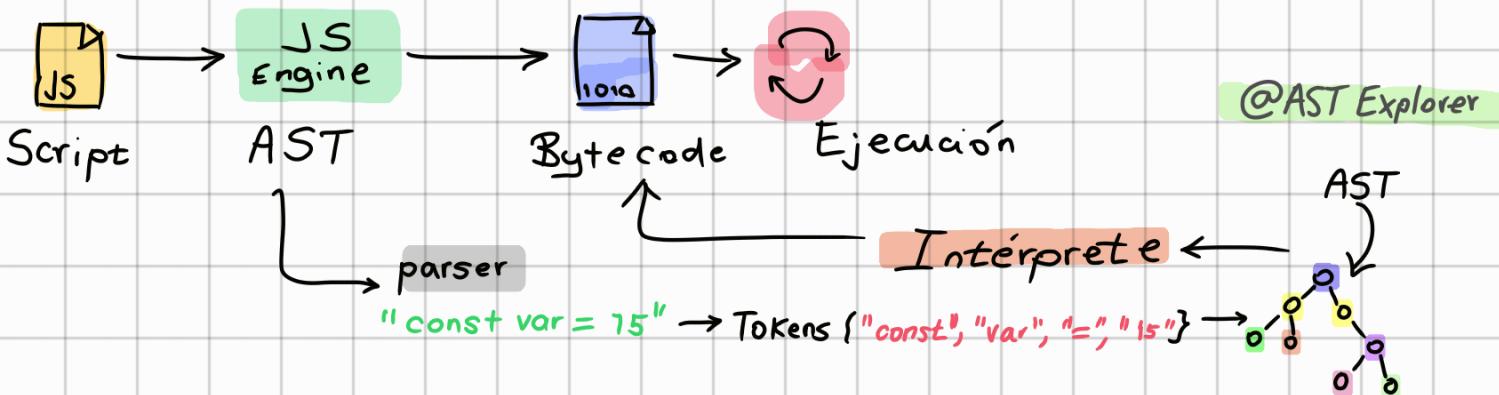
La herencia en cadena los protos. Para encontrar un método JS busca primero en el Objeto, si no lo encuentra, busca en -proto-. Si tampoco lo encuentra va al -proto- del -proto- y así sucesivamente.

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

Object.getPrototypeOf(<Obj>)



PARSERS Y ABSTRACT SYNTAX TREE



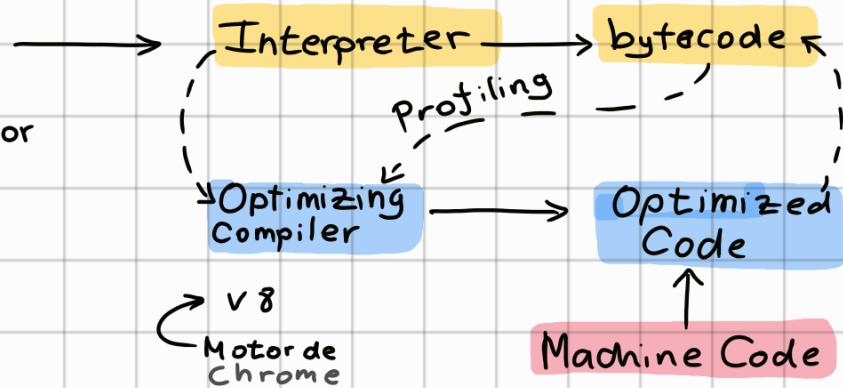
Bytecode vs Machine Code

- Assembly like

- Binario

- Portatil (VM)

- 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) { ← Convertir respuesta
    // Manejo del error a JSON
  }
}
```

Si la promesa retorna error