Curso Profesional Javascript

# ¿Qué significa ser un profesional de JavaScript?

El camino para llegar a ser profesional es largo y duro, no es fácil. Todos necesitamos que nos guíen para saber qué hacer y qué no. Este camino es conocido como la ruta de pasar de **Junior** a **Senior**, *este es un duro camino lleno de experiencia*.

**¿Qué forma a un profesional?**

Te presento una lista de estas cosas que lo forman:

1. Conocimiento del lenguaje.
2. Conocimiento de entornos de programación.
3. Mejores prácticas.
4. Versado en código.
5. Herramientas.
6. Ética / Profesionalismo.
7. Experiencia.

**El lenguaje: JavaScript**

Debemos tener muy claro cuáles son los **fundamentos** de JavaScript antes de comenzar con esto. Existen *features* muy raros y hay que estudiarlos. Tenemos que saber cómo **funcionan las cosas** en JavaScript.

**No fundamentos**

Los **no fundamentos**" representan las siguientes características del lenguaje:

1. **Promesas (nivel pro)**.
2. **Getters, setters**: son formas de obtener valor de una variable sin tener que poner this.name.
3. **Proxies**: es un *feature* muy raro, pero que más adelante veremos a profundidad. Sirve para interceptar a una función antes de que se ejecute.
4. **Generadores**: esto es raro, pero vamos a ver que sí es eficiente.

**¿Cómo funciona?**

Este lenguaje corre sobre un motor. JavaScript no contiene clases como otros lenguajes de programación, esto es algo que vuela mucho la cabeza, es muy difícil de entender. Otro *feature* muy *cool* que vamos a aprender es event loop, es lo que permite que pueda correr muchos procesos a la vez.

**Entornos de programación**

Cuando estamos desarrollando lo hacemos para la **WEB**, para un celular, para seguidores. Existen diferentes entornos que nos ofrecen **APIS**, tenemos que conocer todo esto. V

**Versado en código**

Esto quiere decir que tenemos que leer mucho código, un lugar hermoso para ponernos a leer código es **GitHub**. Debemos leer mucho y hacerlo de forma muy constante.

**Mejores prácticas**

No vamos a reinventar la rueda, hay muchas personas que ya han solucionado los problemas más comunes, tenemos que usar estas soluciones, a estas soluciones se les llama: **patrones de diseño**.

‌

**Ética**

Esta es la parte más importante de ser un profesional. Un buen profesional cumple con los siguientes valores:

1. **Es responsable**.
2. **Entrega a tiempo sus trabajos**.
3. **Sabe decir que no**.
4. **No hace daño**.

‌

**Experiencia**

La experiencia no es algo que se pueda enseñar, tenemos que encontrarla nosotros mismos en el camino a ser profesionales. Todo está en nosotros, tenemos que estudiar y practicar mucho.

Te compartimos este recordatorio para que seas un profesional en Javascript y en otros lenguajes de programación o tecnologías.



# Inicio del proyecto

En este curso vamos a estar desarrollando una aplicación llamada: Platzi Video. En toda plataforma de video hay un componente especial en el desarrollo, tenemos que saber implementar el ***MediPlayer***, en este curso vamos a estar desarrollando este feature de forma modular, esto quiere decir que vamos a desarrollar plugins que vamos a implementar a nuestro reproductor, extendiéndole sus funcionalidades. Vamos a comenzar con un poco de **CSS** y **HTML** ya escrito.

‌

## Primer paso

‌

Crearemos nuestros primeros archivos usando npm init -y, donde -y es una bandera que le dicta a **npm** que le diga sí a todas las preguntas que haga.

npm init -y

‌

Esto nos creará un archivo package.json que lo sustituiremos por el siguiente:

{

"name": "platzi-media-player",

"version": "1.0.0",

"description": "Proyecto del Curso Profesional de JavaScript de la Escuela de JavaScript de Platzi.",

"license": "MIT",

"author": "César Augusto Barco <augustopayza@gmail.com>",

"keywords": [

"platzi"

],

"scripts": {

"start": "live-server"

},

"devDependencies": {

"live-server": "^1.2.1"

}

}

‌

Una vez tengamos todo esto listo vamos a proceder a instalar nuestro live-server para empezar a trabajar. Para instalar esto vamos a usar el siguiente comando npm i -D live-server donde i significa install y la bandera -D develoment, esto quiere decir que no lo vamos a usar en producción.

‌

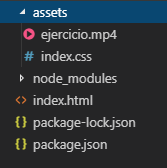
Una vez instalado ya lo podremos usar con el package.json que dejé arriba. Lo usaremos con el comando start que llamará a su vez a live-server.

‌

Antes de ejecutar este vamos a implementar varios archivos. Estos serán los siguientes:

[HTML](https://firebasestorage.googleapis.com/v0/b/gitbook-28427.appspot.com/o/assets%2F-LlTyKe9xd6RJ6x5f2-z%2F-Ln9ZjFBlm-s-f1q7F63%2F-Ln9_0AO_RTk9HHBf8tM%2Findex.html?alt=media&token=090c9439-1f81-49e1-9087-a2175ef1fbde), [CSS](https://firebasestorage.googleapis.com/v0/b/gitbook-28427.appspot.com/o/assets%2F-LlTyKe9xd6RJ6x5f2-z%2F-Ln9ZjFBlm-s-f1q7F63%2F-Ln9_73eBUNHnXNiJ6eJ%2Findex.css?alt=media&token=d91cbcaa-be06-47ea-8f51-538e5bd1f2ed)

También cualquier video que tengamos en nuestra PC. Nuestras carpetas tienen que quedar de la siguiente forma:

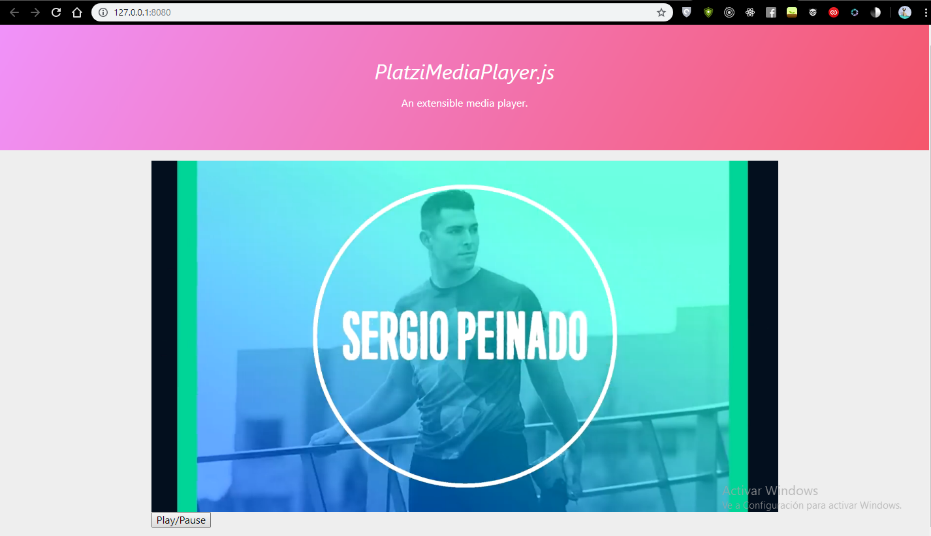


Ahora sí vamos a ejecutar nuestro pequeña aplicación.

npm start

‌

Nuestra pequeña aplicación andará en la **IP** que nos muestre la terminal.



‌

## ¿Qué sigue?

‌

Tenemos un botón que no funciona, lo vamos a poner a funcionar con un media query. Abrimos nuestras etiquetas de script.

‌

Tenemos un vídeo que debemos manipular, lo vamos a hacer con querySelector(""), a este tenemos que pasarlo un elemento, en este caso será **video**, es el único elemento **video** en nuestro **HTML**. Tambien debemos traer nuestro botón con `querySelector``.

**const** video = document.querySelector("video")

**const** button = document.querySelector("button")

‌

Cuando le demos **click** a nuestro botón queremos que el vídeo se reproduzca. Lo hacemos de la siguiente manera:

**button**.onclick = ()=> **video**.play()

‌

El video.play() se saca de la API que trae el navegador, todos los elementos del DOM traen un API. Para saber cuales son las opciones de esta API podemos ir a [MDN](https://developer.mozilla.org/es/docs/Web/API/HTMLMediaElement) a ver toda la documentación. No podemos darle play de una vez a penas se entre en la página, esto pasa por que los navegadores tienen una seguridad que no permite que esto pase, solo se puede dar play si el usuario tiene la libertad de hacerlo.

‌

Ahora nuestro código no es muy extensible, vamos a lograr esto usando **prototipado**. Para hacerlo extensible se pueden usar clases, pero en este caso usaremos protitype, usaremos el siguiente código para lograrlo.

**const** video = document.querySelector("video");

**const** button = document.querySelector("button");

**function** **MediaPlayer**(){}

MediaPlayer.prototype.play = **function**() {

video.play()

}

**const** player = **new** MediaPlayer()

button.onclick = () => player.play();

‌

Explicación:

‌

1. Creamos una función llamada mediaPlayer que nos servirá como prototipo.
2. A mediaPlayer le asignamos una función llamada play usando prototype. Esta función le dará inicio al video.
3. Luego con el botón se acciona una función llamada player que es una instancia del prototipo mediaPlayer que creamos. La instancia se crea usando la palabra new.

‌

## Hagámoslo más reutilizable

‌

Para que nuestro código sea más reutilizable debemos hacerlo de esta manera:

**const** video = document.querySelector("video");

**const** button = document.querySelector("button");

**function** **MediaPlayer**(config) {

**this**.media = config.el;

}

MediaPlayer.prototype.play = **function**() {

**this**.media.play();

};

**const** player = **new** MediaPlayer({ el: video });

button.onclick = () => player.play();

‌

Explicación:

‌

1. A nuestra función madre o prototipo le pasamos una configuración. Esta configuración lo que va a tener es el elemento video original. Le asignamos a this.media el elemento video.
2. A la función extendida le asignamos play() a this.media para que se ejecute cuando presionemos el botón.
3. En nuestra función especial player es una instancia del prototipo le asignaremos el valor de video para que lo reciba en configuración. Esto lo haremos con **destructuración de objetos**.

‌

Acá no podemos usar ***arrow function*** por que el valor de this es global. Más adelante se verá con más detalle.

‌

Para agregarle la funcionalidad de pausa y play con el mismo botón, debemos condicionar la función play de MediaPlayer de la siguiente manera:

MediaPlayer.prototype.play = **function**() {

**if**(**this**.media.paused){

**this**.media.play();

} **else** {

**this**.media.pause()

}

// o podemos usar lo siguiente:

// this.media.paused ? this.media.play() : this.media.pause()

};

**Repaso de Conceptos Fundamentales**

# Cómo llega un script al navegador

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

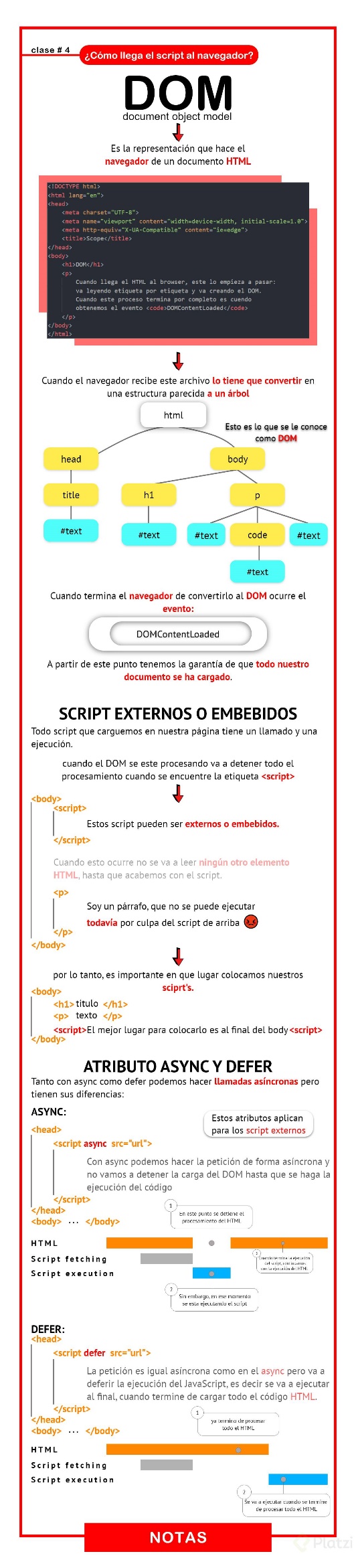
El navegador interpreta el archivo HTML y cuando termina de transformarlo al DOM se dispara el evento **DOMContentLoaded** lo que significa que todo el documento está disponible para ser manipulado.

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.

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.



# 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

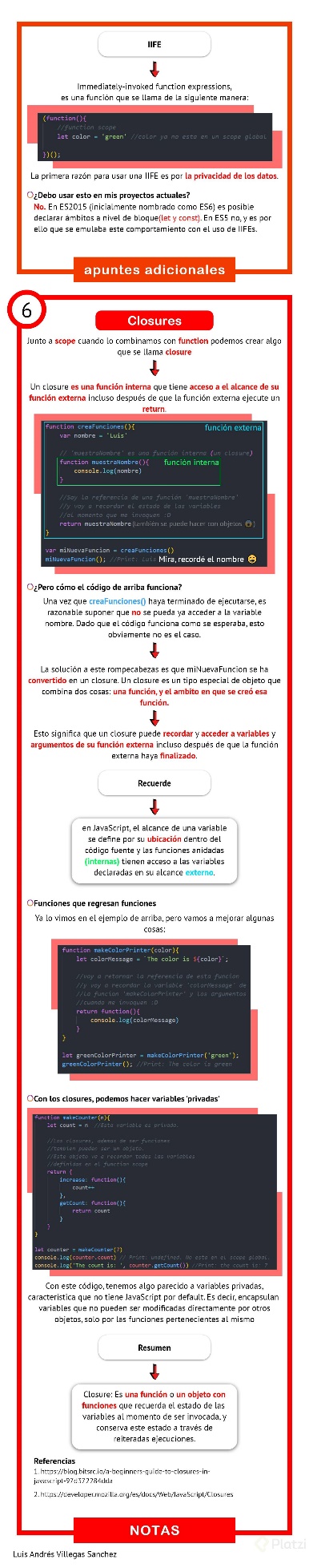
Cuando se denota un script de tipo module con el atributo type="module las variables son limitadas al archivo en el que están declaradas.



# Closures

Son funciones que regresan una función o un objeto con funciones que mantienen las variables que fueron declaradas fuera de su scope.

Los **closures** nos sirven para tener algo parecido a variables privadas, característica que no tiene JavaScript por *default*. Es decir, encapsulan variables que no pueden ser modificadas directamente por otros objetos, sólo por funciones pertenecientes al mismo.



# PLUGINS

Antes de comenzar a diseñar nuestro **plugin** vamos a ver una estrategia para poder adjuntar muchos otros **plugins**.

‌

## Vamos al código

Nos dirigiremos a la función que ya hicimos llamada MediaPlayer, en ella agregaremos los plugins por el objeto de configuración.

function MediaPlayer(config) {

this.media = config.el;

this.plugins = config.plugins;

}

‌

Debería poder funcionar este código incluso cuando no hay plugin, No está de más dar un valor inicial.

this.plugins = config.plugins || [];

‌

Para colocar las dos plecas verticales || usamos la combinación Alt+124.

‌

Si ya añadimos los plugins en la configuración entones ya lo podemos usar en index.js. Ya podemos parle los **plugins** por la instancia a la función que creamos.

const player = new MediaPlayer({

el: video,

plugins: []

});

‌

El primer **plugin** que vamos a agregar va a ser el que nos va a solucionar el problema del autoplay. Este **plugin** no existe, pero lo vamos a crear.

import AutoPlay from "./plugins/AutoPlay.js";

const player = new MediaPlayer({

el: video,

plugins: [new AutoPlay()]

});

‌

No estamos seguros si nuestro plugins va a recibir parámetros pero luego vemos. Vamos a crear la carpeta y el archivo inexistente que instanciamos. Cuando usamos **script** con **type movil** tenemos que ser específicos y usar la extensión .js.



‌

En el archivo nuevo escribiremos el siguiente código:

function AutoPlay() { }

export default AutoPlay;

‌

Con esto nuestro código no tiene errores, pero tampoco tiene funcionalidades nueva. Ya lo tenemos preparado para empezar a integrar un nuevo **plugin**.

‌

En MediaPlayer vamos a necesito un tipo de inicialización.

function MediaPlayer(config) {

this.media = config.el;

this.plugins = config.plugins || [];

this.\_initPlugins();

}

MediaPlayer.prototype.\_initPlugins = function() {

this.plugins.forEach(element => {

element.run()

});

}

‌

De esta forma iteramos en cada **plugin** y lo **inicializamos** con una función llamada run(). Esta función tenemos que declararla en nuestro **plugin**.

function AutoPlay() { }

AutoPlay.prototype.run = function() {

}

export default AutoPlay;

‌

Necesitamos que esta función le de Play al video, pero tenemos que darle acceso. Para esto le pasamos una instancia del MediaPlayer, en el cual ejecutaremos las siguientes funciones.

AutoPlay.prototype.run = function (player) {

player.mute()

player.play()

}

‌

Para que la función **run** reciba **player** tenemos que pasársela en MediaPlayer usando this que representará **MediaPlayer**.

MediaPlayer.prototype.\_initPlugins = function() {

this.plugins.forEach(element => {

element.run(this)

});

}

‌

No tenemos la función mute de MediaPlayer, por ende no funcionará. Vamos a crearla.

MediaPlayer.prototype.mute = function () {

this.media.muted = true;

}

‌

Ahora crearemos un botón para que el usuario pueda mutear y desmutear cuando quiera. Para esto tenemos que crear un segundo botón en el cual llamemos por una ID ya que no será el único botón. El botón anterior también le pondremos una ID.

<button id="playPause">Play/Pause</button>

<button id="unmuteMute">Mute/Unmute</button>

‌

Llamamos correctamente a estos dos botones.

const button = document.querySelector("#playPause");

const muteUnmute = document.querySelector('#unmuteMute')

‌

Cuando le demos **click** llamará a la nueva función que crearemos para que haga **mute** y unmute.

muteUnmute.onclick = () => player.unmuteMute();

‌

El objeto player es una instancia de MediaPlayer, allí crearemos la función unmuteMute.

MediaPlayer.prototype.unmuteMute = function () {

this.media.muted ? this.media.muted = false : this.media.muted = true;

};

‌

De esta forma ya tendremos la funcionalidad de **mutear** y **desmutear**.

# this

*this* se refiere a un objeto, ese objeto es el que actualmente está ejecutando un pedazo de código.

No se puede asignar un valor a *this* directamente y este depende de en que scope nos encontramos:

* Cuando llamamos a *this* en el **Global Scope o Function Scope**, se hace referencia al objeto *window*. A excepción de cuando estamos en **strict mode** que nos regresará *undefined*.
* Cuando llamamos a *this* desde **una función** que está contenida en un objeto, *this* se hace referencia a ese objeto.
* Cuando llamamos a *this* desde una **“clase”**, se hace referencia a la instancia generada por el constructor.

# Los métodos call, apply y bind

Estas funciones nos sirven para establecer el valor de *this*, es decir cambiar el contexto que se va usar cuando la función sea llamada.

Las funciones **call, apply y bind** son parte del prototipo Function. Toda función usa este prototipo y por lo tanto tiene estas tres funciones.

* **functionName.call()**. Ejecuta la función recibiendo como primer argumento el *this* y los siguientes son los argumentos que recibe la función que llamó a call.
* **functionName.apply()**. Ejecuta la función recibiendo como primer argumento el *this* y como segundo un arreglo con los argumentos que recibe la función que llamó a apply.
* **functionName.bind()**. Recibe como primer y único argumento el *this*. No ejecuta la función, sólo regresa otra función con el nuevo this integrado.

# Prototype

En Javascript todo son objetos, no tenemos clases, no tenemos ese plano para crear objetos.

Todos los objetos “heredan” de un prototipo que a su vez hereda de otro prototipo y así sucesivamente creando lo que se llama la **prototype chain**.

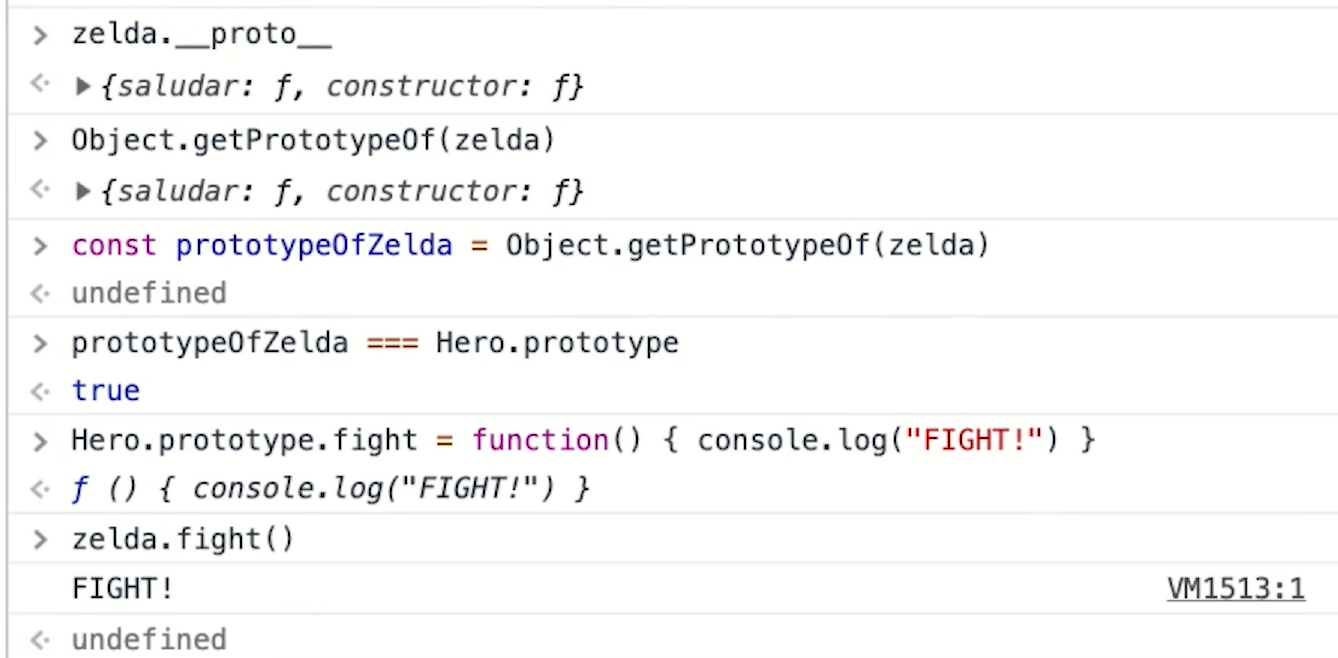
La keyword new crea un nuevo objeto que “hereda” todas las propiedades del prototype de otro objeto. No confundir prototype con **proto** que es sólo una propiedad en cada instancía que apunta al prototipo del que hereda.

# 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*.

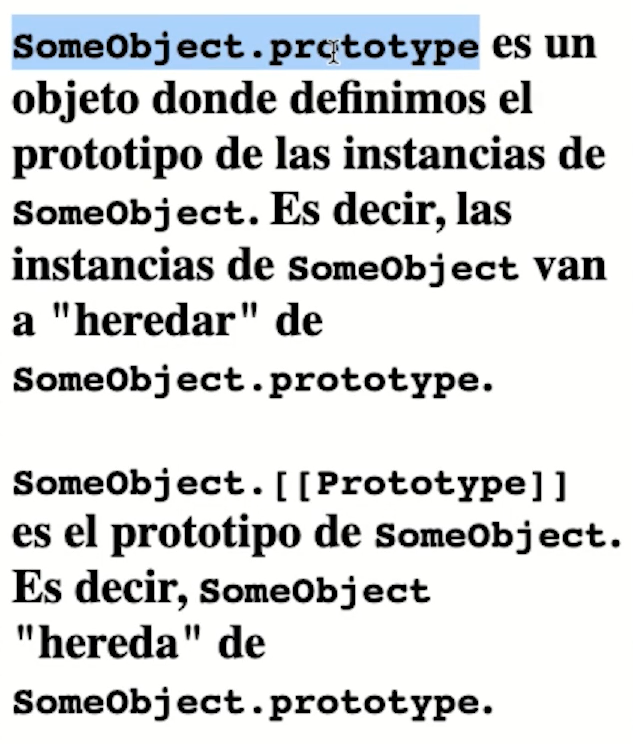
Cuando se llama a una función o variable que no se encuentra en el mismo objeto que la llamó, se busca en toda la prototype chain hasta encontrarla o regresar *undefined*.

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









**Repaso de Conceptos Fundamentales**

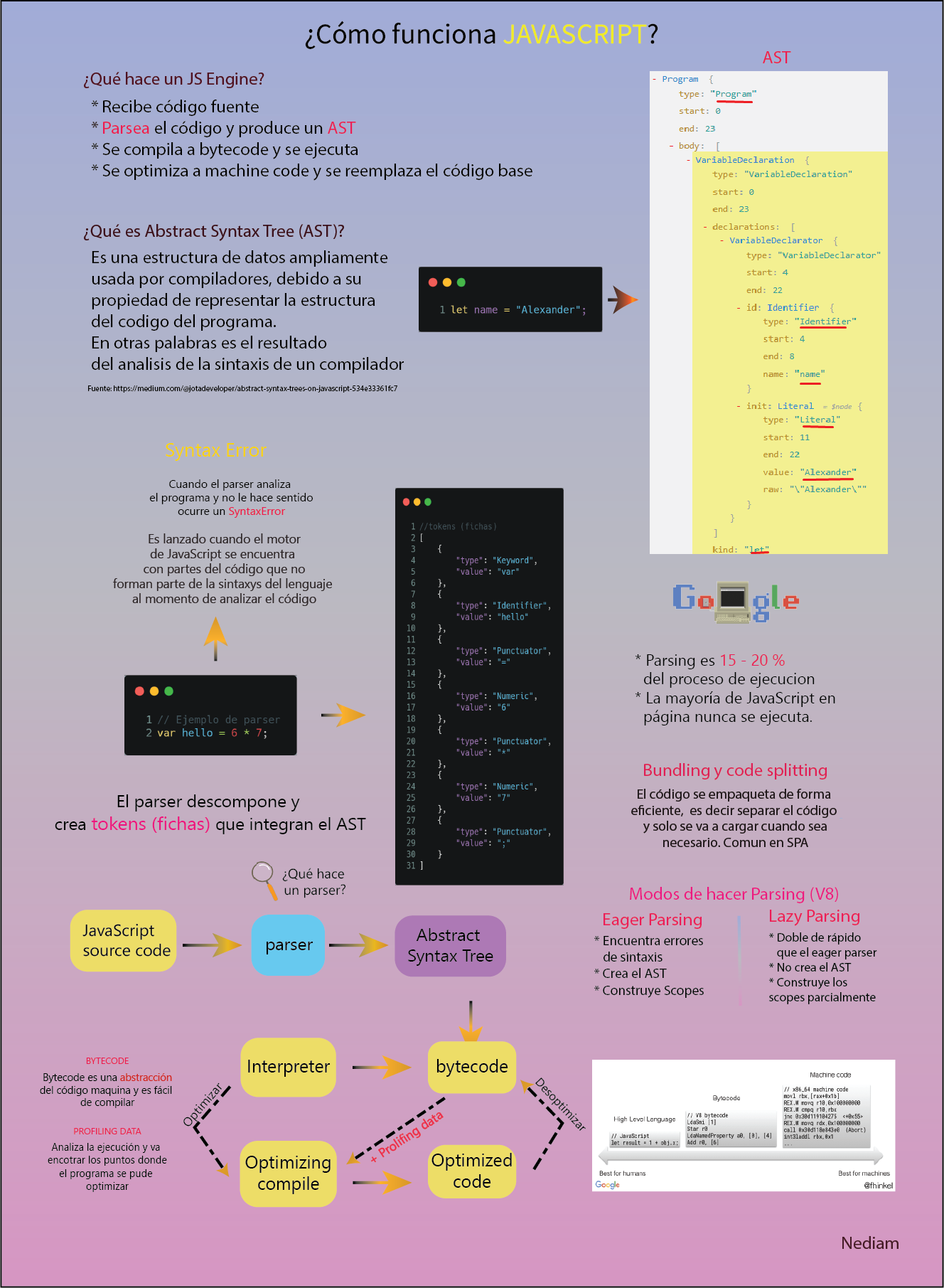
# Parsers y el Abstract Syntax Tree

El JS Engine recibe el código fuente y lo procesa de la siguiente manera:

* El **parser** descompone y crea tokens que integran el **AST**.
* Se compila a **bytecode** y se ejecuta.
* Lo que se pueda se **optimiza a machine code** y se reemplaza el código base.

Un **SyntaxError** es lanzado cuando el motor JavaScript encuentra partes que no forman parte de la sintaxis del lenguaje y esto lo logra gracias a que se tiene un AST generado por el parser.

El *parser* es del 15% al 20% del proceso de ejecución por lo que hay que usar parser del código justo en el momento que lo necesitamos y no antes de saber si se va a usar o no.



# Abstract Syntax Tree en Práctica

## Abstract Syntax Tree en Práctica

‌

Vamos a usar el **AST** para crear una regla de **eslint**, este analizará estéticamente nuestro código a ver si hay que levantar un warning por violar la sintaxis. Muchas de estas reglas ya viene con e eslint, pero podemos agregar nuestras propias reglas. Vamos a usar la herramienta [AST | Explorer](https://astexplorer.net/#/gist/16fc27fc420f705455f2b42b6c804aa1/d9cc7988c2c743d7edfbb3c3b1abed866c975ee4) para experimentar. Usaremos la configuración por defecto, veremos en la parte superior izquierda el código que vamos a ingresar, a la derecha el tree creado, en la parte inferior izquierda las funciones de las reglas y a la derecha de eso la salida de nuestro código.

‌

## Test

‌

En el **link** de **AST Explorer** ya tenemos un código escrito. Donde el la primera entrada tenemos las tareas que debe cumplir nuestro **fixer**.

const pi = 3.1415;

const half\_pi = 1.57075;

// variable constantes

// variables que guarden un numero

// El nombre de la variable tiene que estar en UPPERCASE

‌

A la derecha tenemos el árbol completo de todas estas declaraciones y gracias a el podemos manipular, detectar errores o interpretar lo que escribamos. Luego implementamos una función que recibe la declaración de la variable y accedemos a los datos que nos ofrece el AST para lograr cumplir con los requerimientos de nuestro solucionador.

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 numero

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())

}

})

}

}

}

}

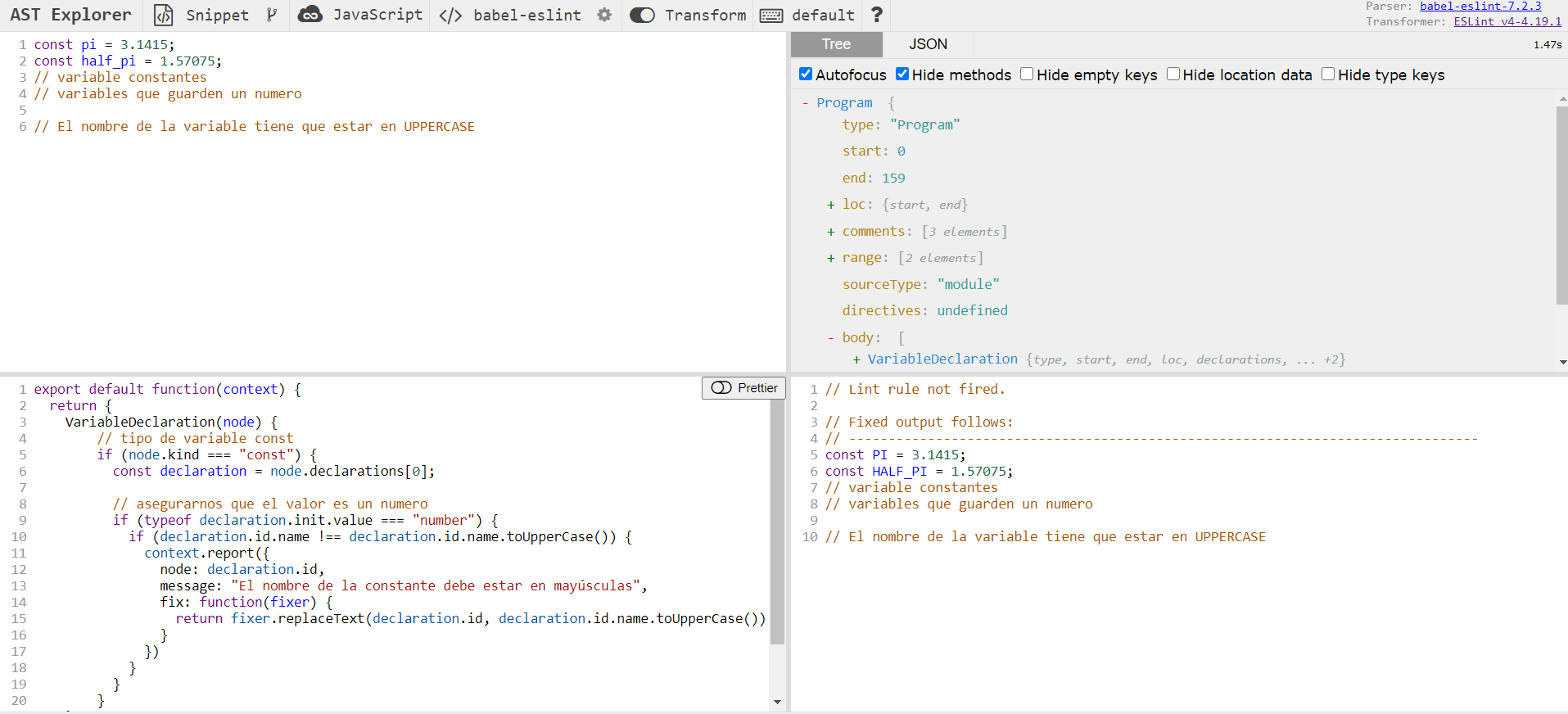
};

};

‌

Con context.report() podemos mandar un warning y además podemos solucionar el problema que se haya presentado.

https://astexplorer.net/#/gist/16fc27fc420f705455f2b42b6c804aa1/d9cc7988c2c743d7edfbb3c3b1abed866c975ee4



# Cómo funciona el JavaScript Engine

Una vez tenemos el **AST** ahora hay que convertirlo a Bytecode.

**Bytecode** es como el código assembler pero en lugar de operar en el procesador opera en la máquina virtual **V8** del navegador.

**Machine code** es el más bajo nivel, es código binario que va directo al procesador.

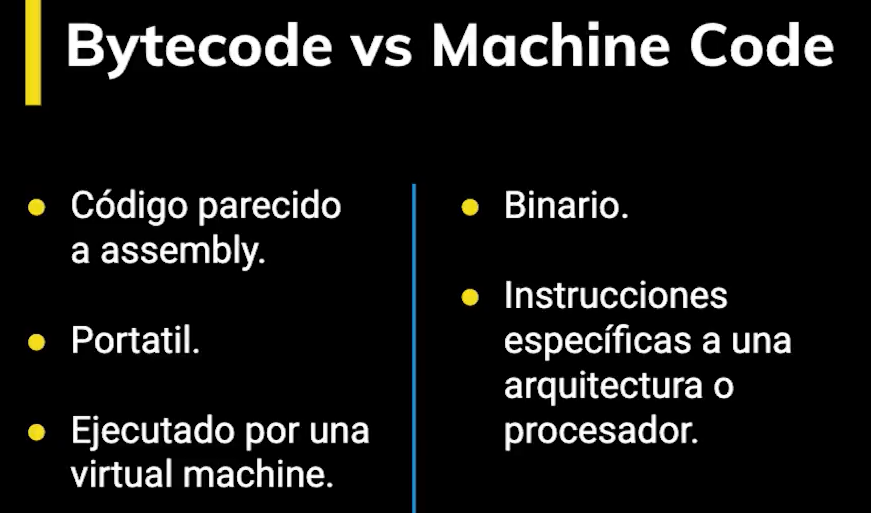
**El profiler** se sitúa en medio del bytecode y el optimizador

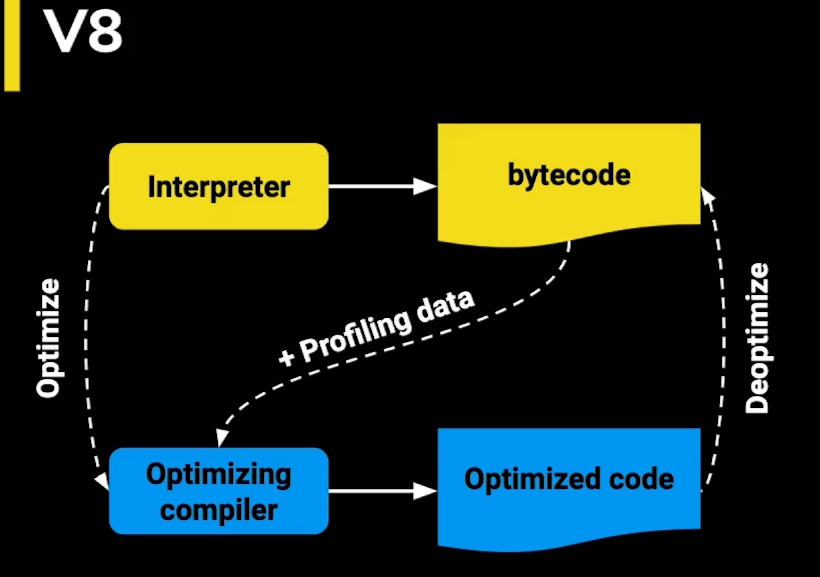
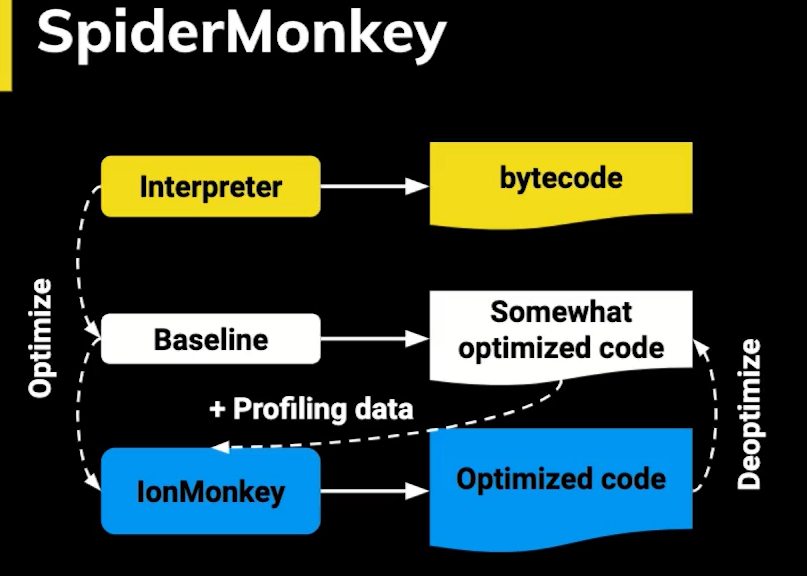
Cada máquina virtual tiene sus particularidades, por ejemplo V8 tiene algo llamado **Hot Functions**.

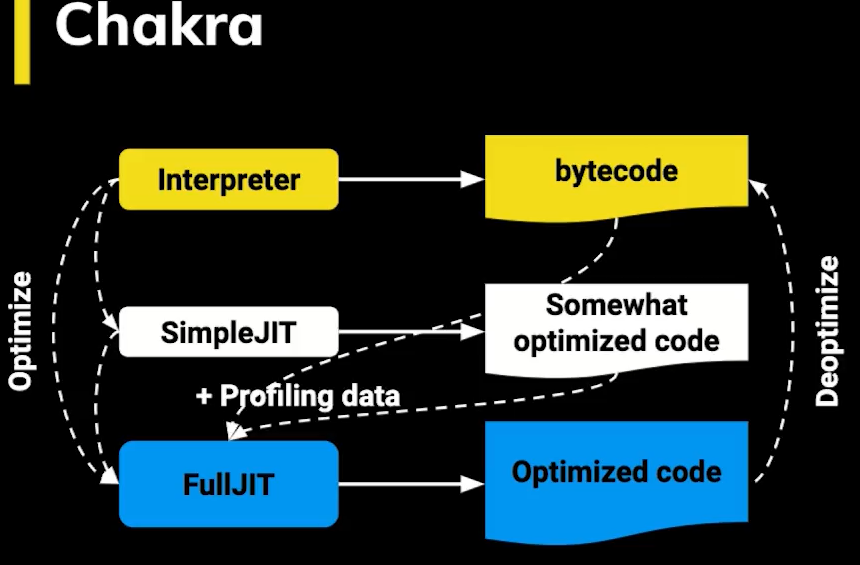
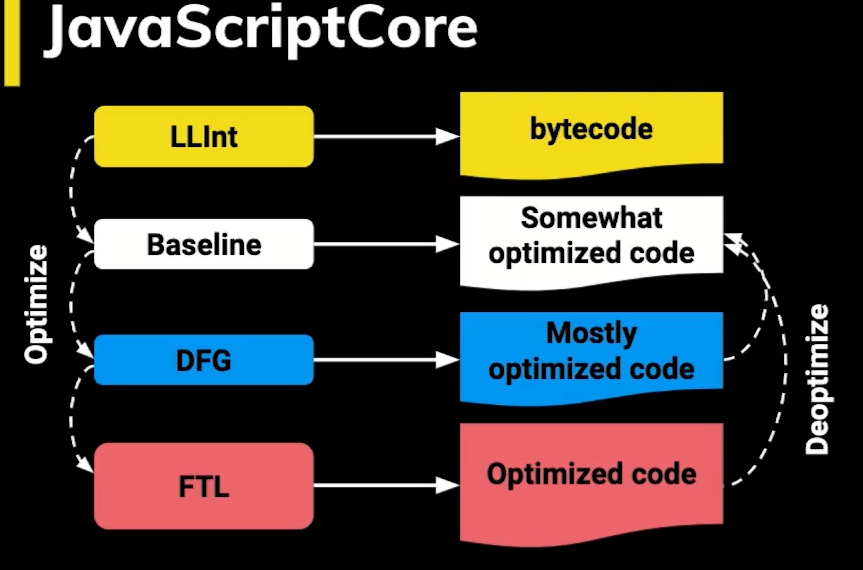
Cuando una sentencia función es ejecutada muy frecuentemente, V8 la denomina como una *hot function* y hace una optimización que consiste en convertirla a *machine code* para no tener que interpretarla de nuevo y agilizar su ejecución.

Cada navegador tiene su implementación de JavaScript Engine:

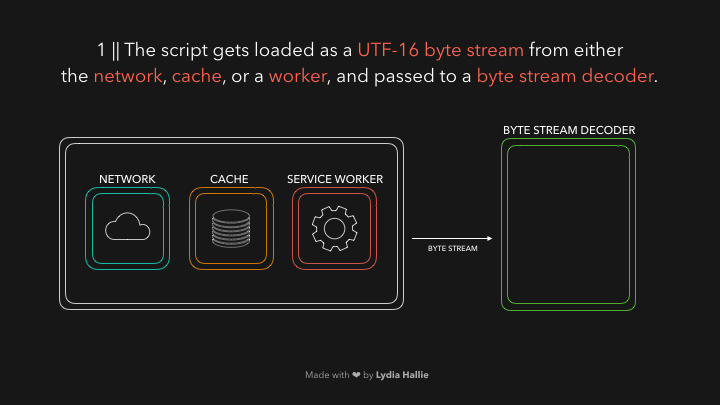
* SpiderMonkey - Firefox
* Chackra - Edge
* JavaScriptCore - Safari
* V8 – Chrome



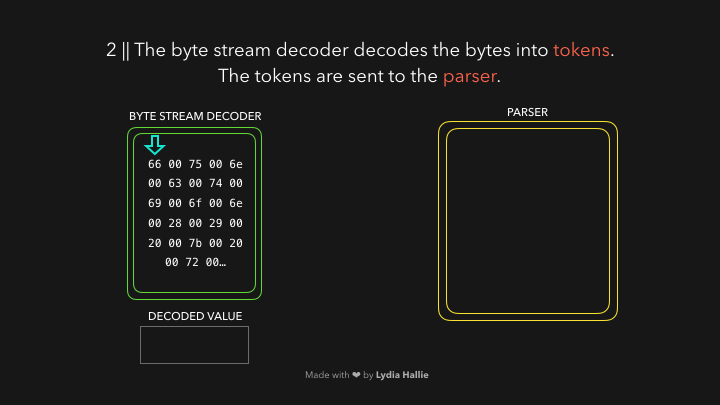
 

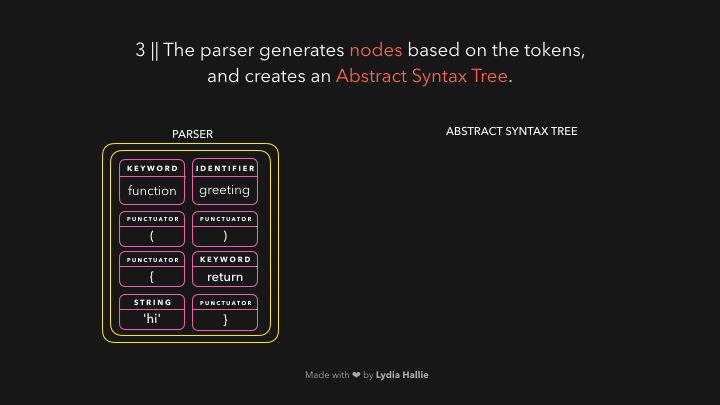
El script se carga como un **flujo de bytes UTF-16**, ya sea para la **red**, la **caché** o un\*\* trabajador\*\*, y se pasa a un **decodificador de flujo de bytes**.



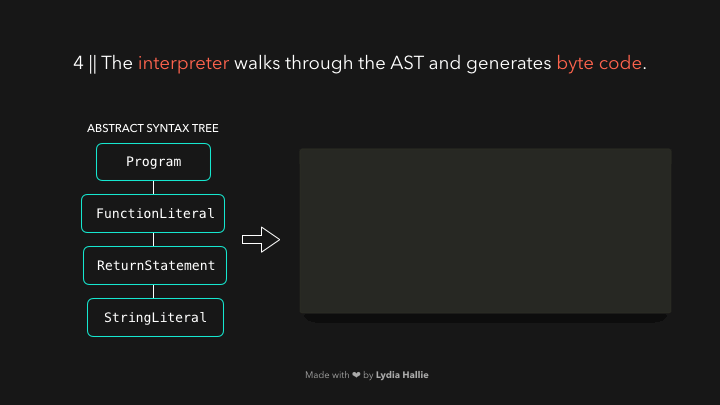
El decodificador de flujo de bytes decodifica los bytes en **tokens**. Los tokens se envían al **analizador**.



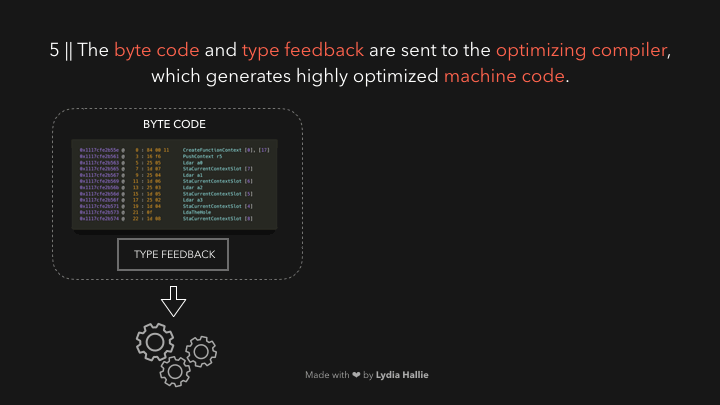
El analizador genera \*\*nodos \*\*basados en los tokens, y crea un **Abstract Syntax Tree**.



El intérprete camina a través de la **AST** y genera **byte code**.



El **byte code** y el **feedback type** se envían al **optimizing compiler**, que genera **código de máquina** altamente optimizado.



# Event Loop

El **Event Loop** hace que Javascript parezca ser multihilo a pesar de que corre en un solo proceso.

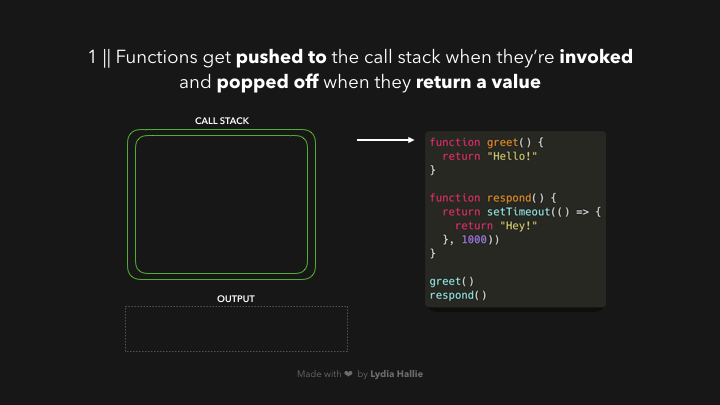
Javascript se organiza usando las siguientes estructuras de datos:

* **Stack**. Va apilando de forma organizada las diferentes instrucciones que se llaman. Lleva así un rastro de dónde está el programa, en que punto de ejecución nos encontramos.
* **Memory Heap**. De forma desorganizada se guarda información de las variables y del scope.
* **Schedule Tasks**. Aquí se agregan a la cola, las tareas programadas para su ejecución.
* **Task Queue**. Aquí se agregan las tares que ya están listas para pasar al stack y ser ejecutadas. El stack debe estar vacío para que esto suceda.
* **MicroTask Queue**. Aquí se agregan las promesas. Esta Queue es la que tiene mayor prioridad.

El Event Loop es un loop que está ejecutando todo el tiempo y pasa periódicamente revisando las queues y el stack moviendo tareas entre estas dos estructuras.

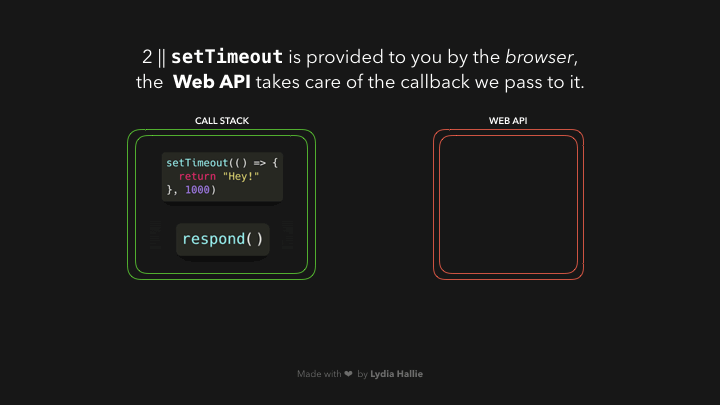
La explicaión del profesor es asombrosa. Explicar cómo funciona el EventLoop de una forma tan sencilla es asombroso.  
.  
Aquí dejo un resumen de todo el funcionamiento de lo que vimos en esta clase.  
.

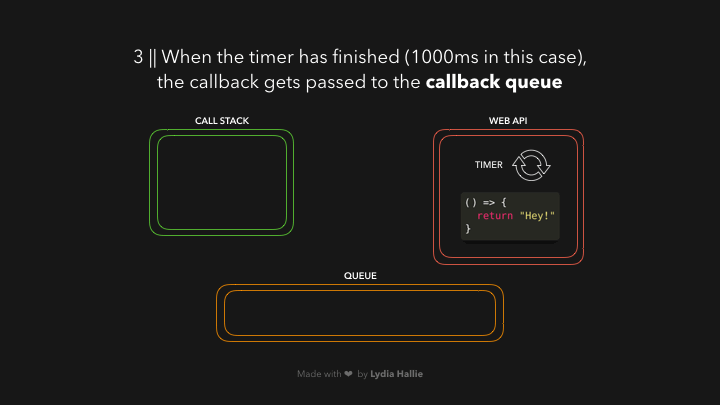
Las funciones son empujadas al **call stack** cuando son invocadas y se sacan cuando **devuelven un valor**

.  


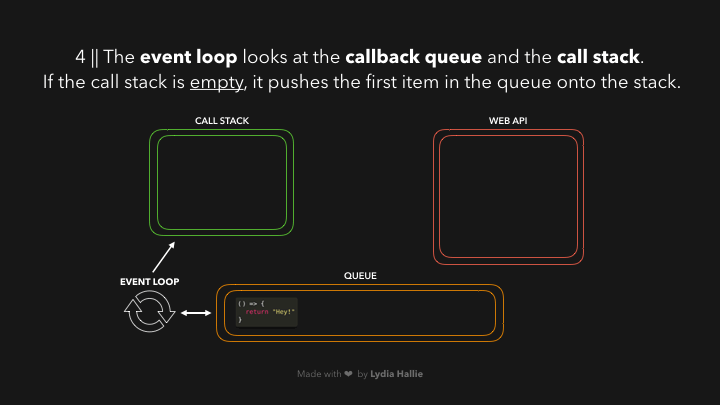
.

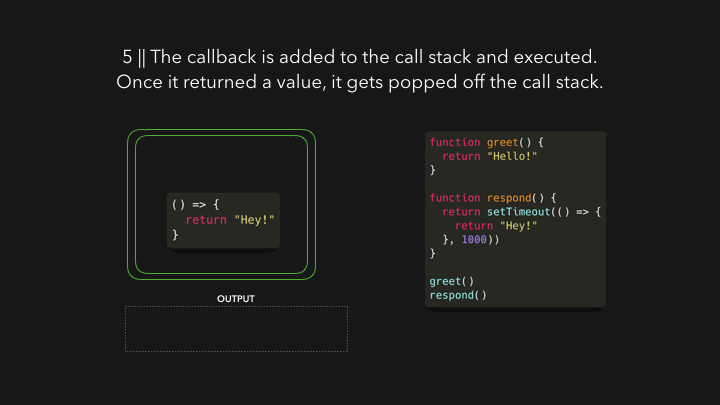
**setTimeOut** es proveído por el navegador, la **Web API** se encarga del **callback** que le pasemos.

.  


.  
Cuando el timer ha terminado (1000ms en este caso), el **callback** se pasa al **callback queue**  
.  


.  
El **Event Loop** mira hacia el **callback queue** y al **call stack**. Si el call stack está vacío, este empuja el primer elemento de la cola en el stack.

.  
  
.  
El callback es añadido al call stack para luego ser ejecutado. Una vez retorna un valor, este es sacado de call stack.

.  
.  
.  
Este pequeño resumen es sacado del post de \*\*[@lydiahallie](https://twitter.com/lydiahallie) \*\* [**✨♻️ JavaScript Visualized: Event Loop**](https://dev.to/lydiahallie/javascript-visualized-event-loop-3dif) donde también se explica muy bien como funciona el Event Loop

12

**Fundamentos Intermedios**

# Promesas

Para crear las promesas usamos la clase Promise. El constructor de Promise recibe un sólo argumento, un callback con dos parámetros, **resolve** y **reject**. resolve es la función a ejecutar cuando se resuelve y reject cuando se rechaza.

El async/await es sólo syntax sugar de una promesa, por debajo es exactamente lo mismo.

La clase Promise tiene algunos métodos estáticos bastante útiles:

* **Promise.all**. Da error si una de las promesas es rechazada.
* **Promise.race**. Regresa sólo la promesa que se resuelva primero.

## Funciones asíncronas

‌

Cuando queremos tener una función que se ejecute eventualmente podemos usar las promesas, también se usan para esperar datos que van a tardar en llegar. Para esperar una información hay que usar el keyword await, pero para usarlo hay que colocar async antes de la función.

function resolveAfter2Seconds() {

return new Promise(resolve => {

setTimeout(() => {

resolve('resolved');

}, 2000);

});

}

async function asyncCall() {

console.log('calling');

var result = await resolveAfter2Seconds();

console.log(result);

// expected output: 'resolved'

}

asyncCall();

‌

Para esperar información de alguna API necesitamos usar este tipo de funciones para esperar la información en un tiempo indeterminado.

‌

## Promesas

‌

El objeto **Promise** (Promesa) es usado para computaciones asíncronas. Una promesa representa un valor que puede estar disponible ahora, en el futuro, o nunca.

new Promise( /\* ejecutor \*/ function(resolver, rechazar) { ... } );

‌

Una **Promesa** es un proxy para un valor no necesariamente conocido en el momento que es creada la promesa. Permite asociar manejadores que actuarán asincrónicamente sobre un eventual valor en caso de éxito, o la razón de falla en caso de una falla. Esto permite que métodos asíncronos devuelvan valores como si fueran síncronos: en vez de inmediatamente retornar el valor final, el método asíncrono devuelve una promesa de suministrar el valor en algún momento en el futuro.

‌

Una Promesa se encuentra en uno de los siguientes estados:

‌

* pendiente (pending): estado inicial, no cumplida o rechazada.
* cumplida (fulfilled): significa que la operación se completó satisfactoriamente.
* rechazada (rejected): significa que la operación falló.

‌

[Promise.all(iterable)](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Promise/all)

‌

Devuelve una de dos promesas: una que se cumple cuando todas las promesas en el argumento iterable han sido cumplidas, o una que se rechaza tan pronto como una de las promesas del argumento iterable es rechazada. Si la promesa retornada es cumplida, lo hace con un arreglo de los valores de las promesas cumplidas en el mismo orden definido en el iterable. Si la promesa retornada es rechazada, es rechazada con la razón de la primera promesa rechazada en el iterable. Este método puede ser útil para agregar resultados de múltiples promesas

‌

[Promise.race(iterable)](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Promise/race)

‌

Devuelve una promesa que se cumple o rechaza tan pronto como una de las promesas del iterable se cumple o rechaza, con el valor o razón de esa promesa.

# Getters y setters

Los getters y setters son funciones que podemos usar en un objeto para tener propiedades virtuales. Se usan los keywords *set* y *get* para crear estas propiedades.

Estas propiedades al ser funciones pueden llevar una validación de por medio y ser usadas con el operador de asignación como si fueran una variable más dentro del objeto.

Me parece que aunque se aborda la funcionalidad de los getters y setters, en el contexto en el cual se explica, no facilita su comprensión. Este seria un resumen:

**Getters**

**let** person = {

name:'Miguel',

last\_name:'Soler',

age:28,

languages:['js','css','react'],

get skills(){

**return** **this**.languages

}

}

console.log(person.skills)

En este caso estoy retornando el valor de languages atraves de un getter llamado skills, pero para ver mejor su uso, mira este ejemplo:

**let** person = {

name:'Miguel',

last\_name:'Soler',

age:28,

languages:['js','css','react'],

get fullName(){

**return** `${this.name} ${this.last\_name}`

}

}

console.log(person.fullName)

En este caso estoy retornando el nombre completo (fullName), como una propiedad del objeto person. El fullName también lo podría traer a través de un método de la siguiente forma:

**let** person = {

name:'Miguel',

last\_name:'Soler',

age:28,

languages:['js','css','react'],

fullName: **function** (){

**return** `${this.name} ${this.last\_name}`

}

}

console.log(person.fullName())

Sin embargo, a traves del getter la semantica es mucho màs transparente y mantiene mejor la integridad de la data.

**Setter**

**let** person = {

name:'Miguel',

last\_name:'Soler',

age:28,

languages:[],

set skills(skills){

**this**.languages = skills

}

}

person.skills = ['ruby','rails','nodejs']

console.log(person.languages)

En suma los getters y setters nos permiten tener el control sobre las propiedades que podemos almacenar o recuperar.