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.

‌

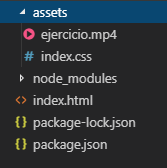
Una vez instalado ya lo podremos usar con el package.json que dejé arriba. Lo usaremos con el comando start que llamará a su vez a live-server.

‌

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:

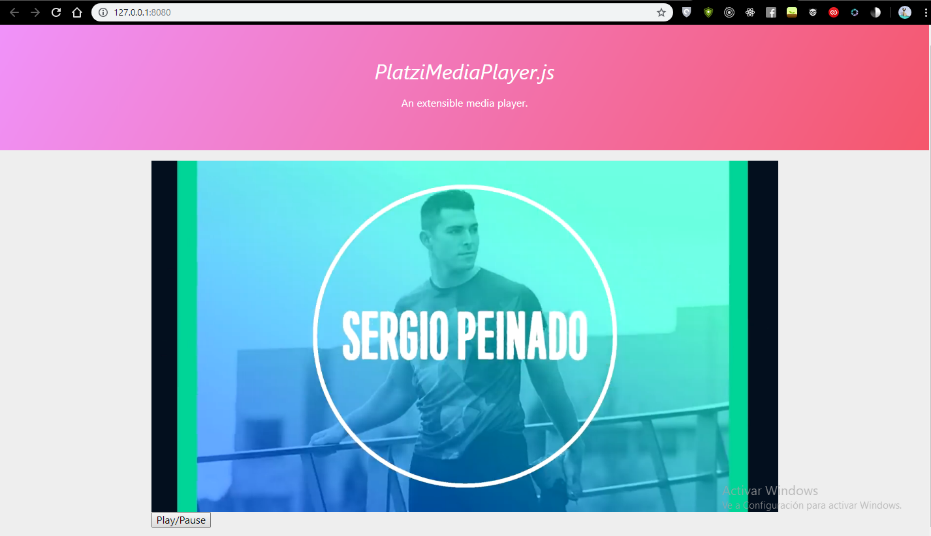


Ahora sí vamos a ejecutar nuestro pequeña aplicación.

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.

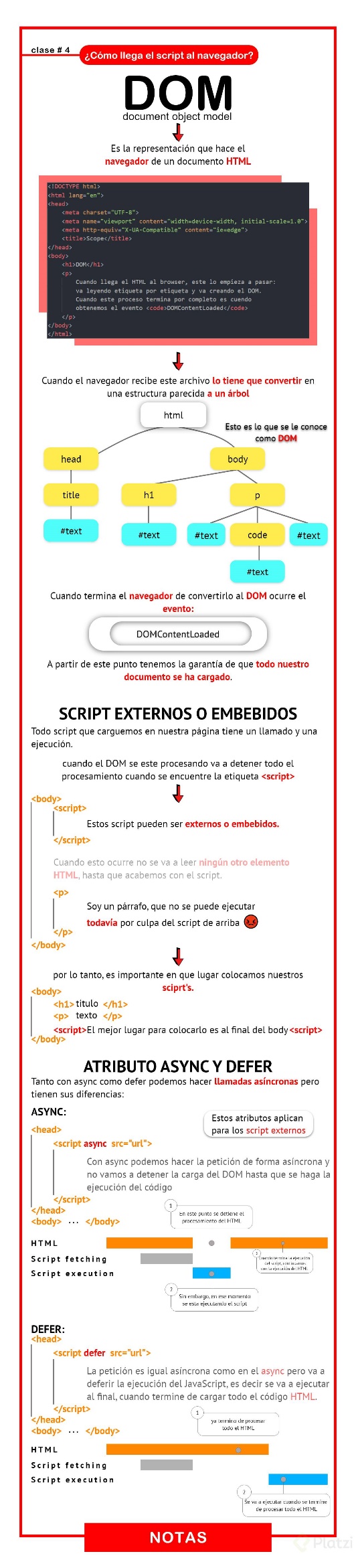
El navegador interpreta el archivo HTML y cuando termina de transformarlo al DOM se dispara el evento **DOMContentLoaded** lo que significa que todo el documento está disponible para ser manipulado.

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

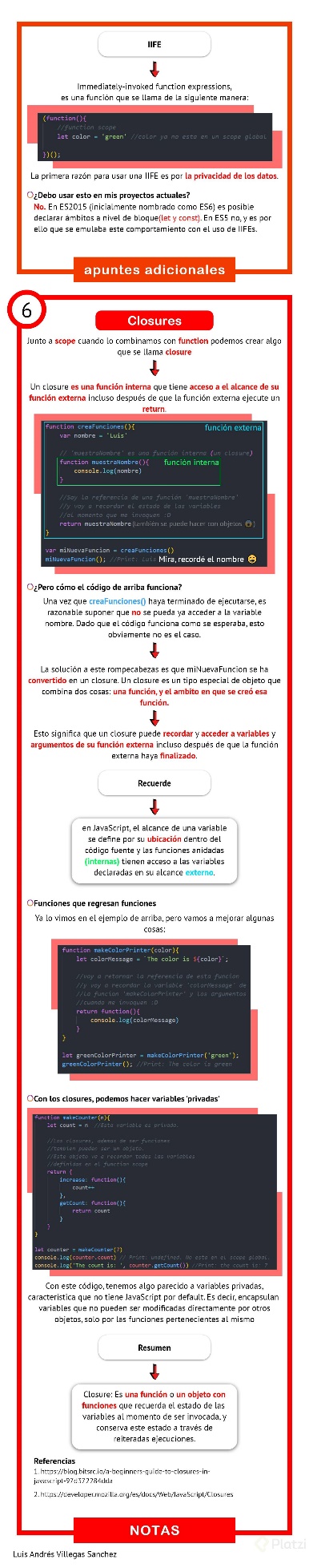
Cuando se denota un script de tipo module con el atributo type="module las variables son limitadas al archivo en el que están declaradas.



# 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**.

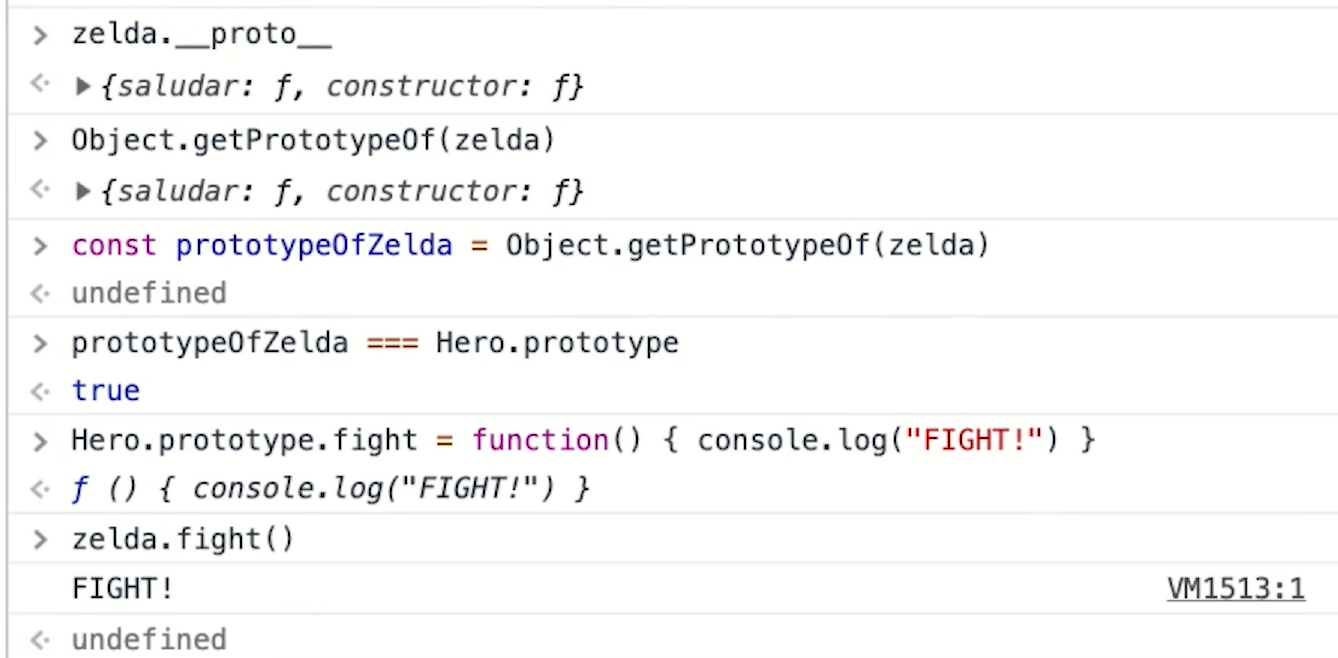
La keyword new crea un nuevo objeto que “hereda” todas las propiedades del prototype de otro objeto. No confundir prototype con **proto** que es sólo una propiedad en cada instancía que apunta al prototipo del que hereda.

# 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*.

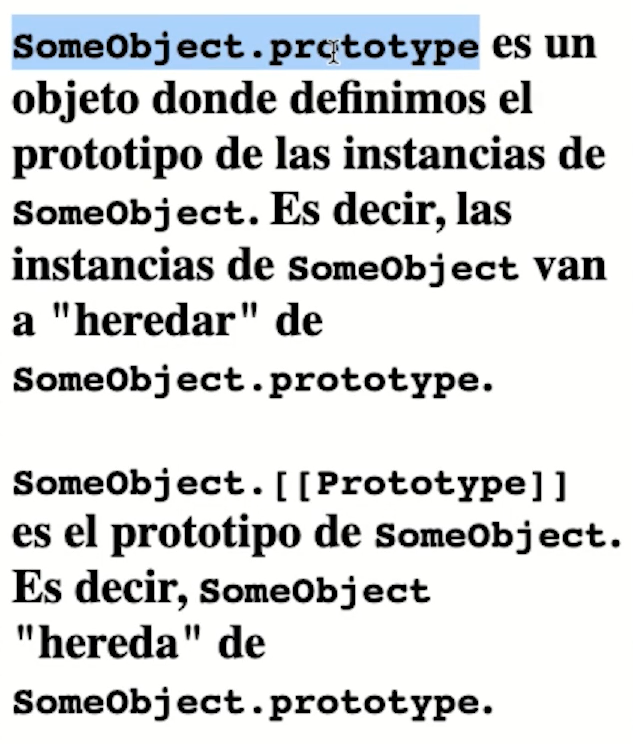
Cuando se llama a una función o variable que no se encuentra en el mismo objeto que la llamó, se busca en toda la prototype chain hasta encontrarla o regresar *undefined*.

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**

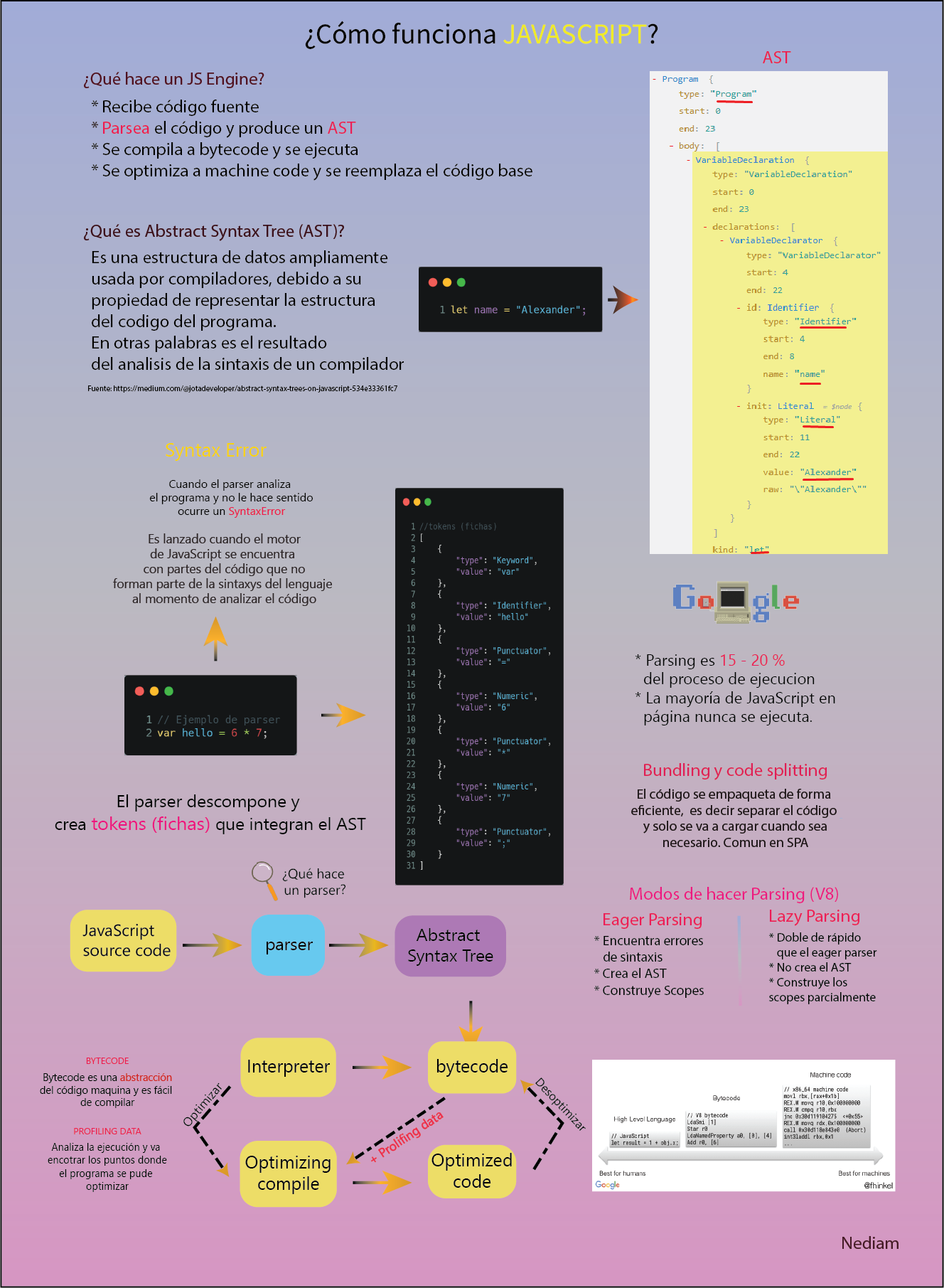
# Parsers y el Abstract Syntax Tree

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())

}

})

}

}

}

}

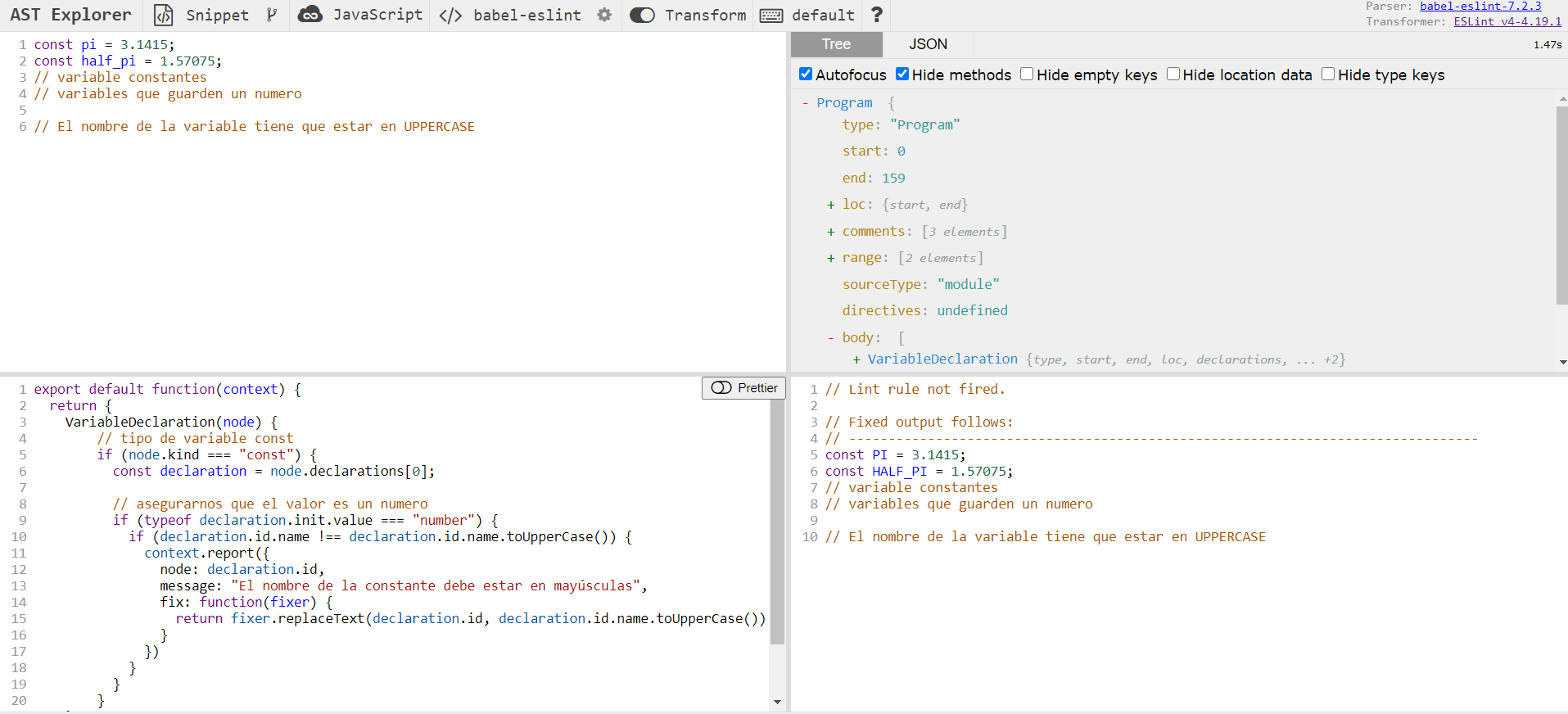
};

};

‌

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.

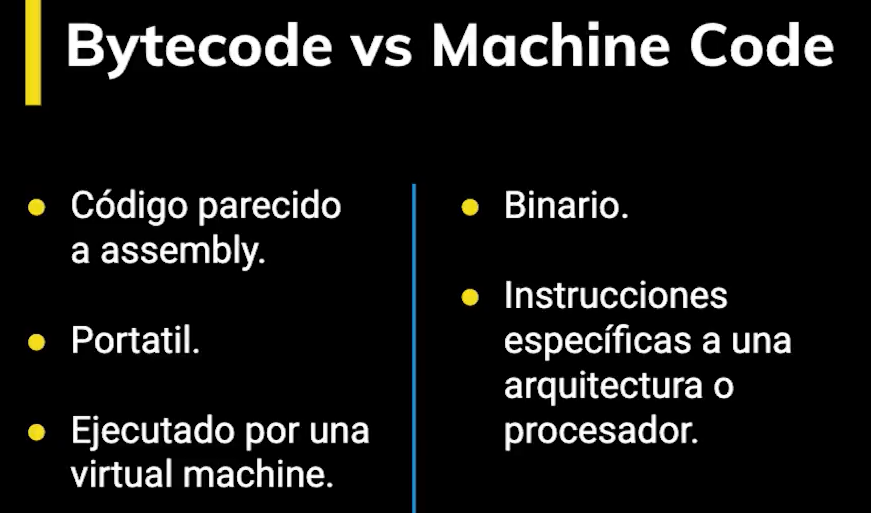
**El profiler** se sitúa en medio del bytecode y el optimizador

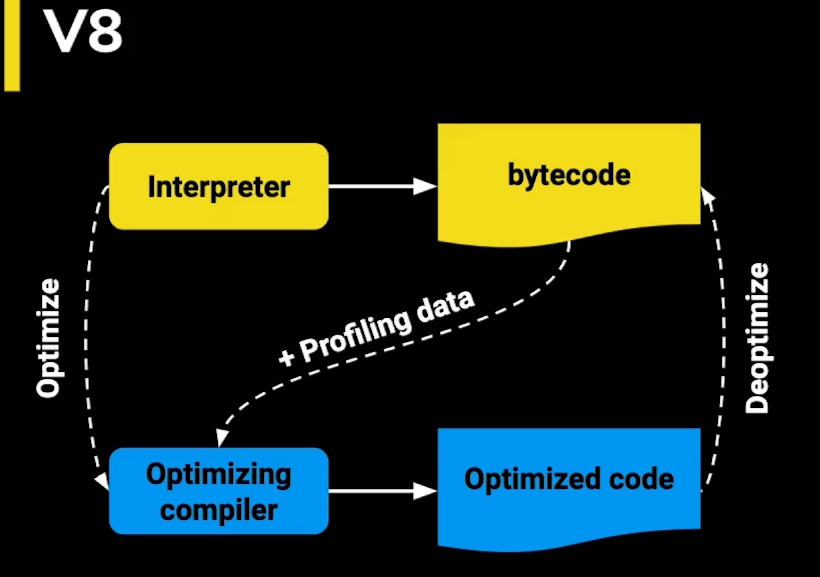
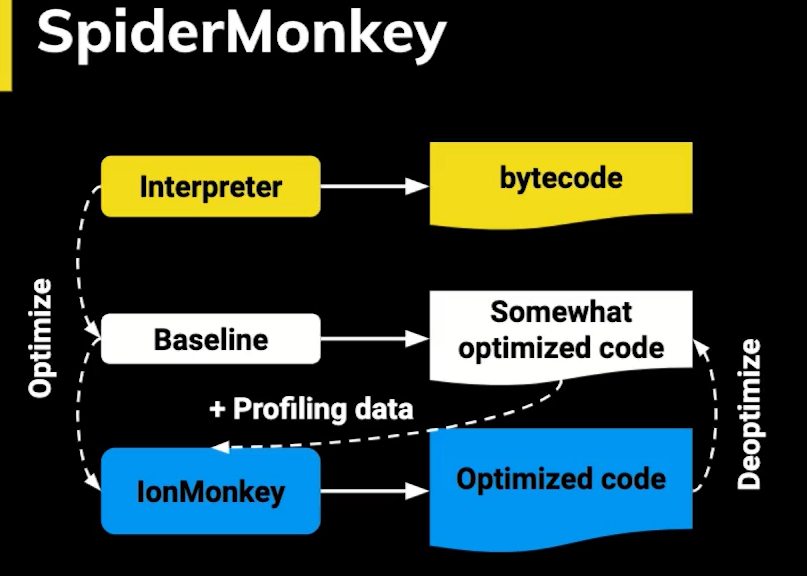
Cada máquina virtual tiene sus particularidades, por ejemplo V8 tiene algo llamado **Hot Functions**.

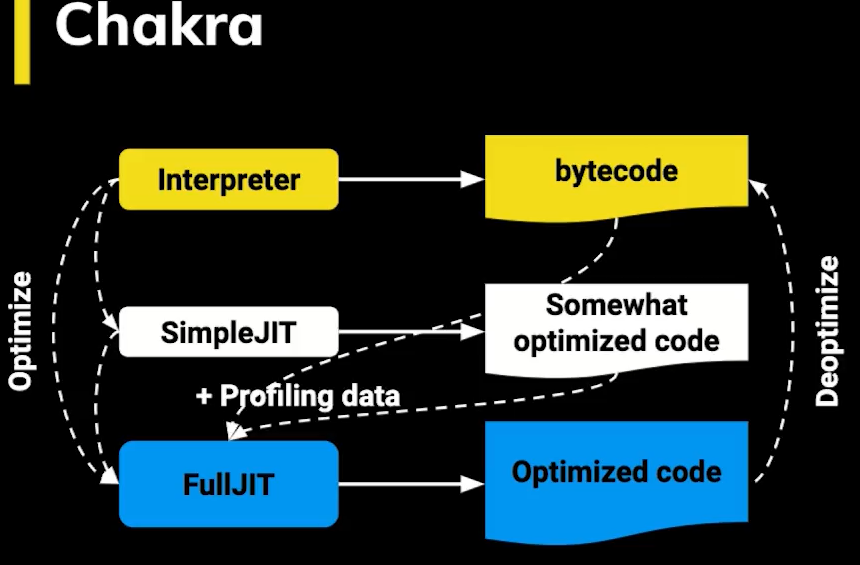
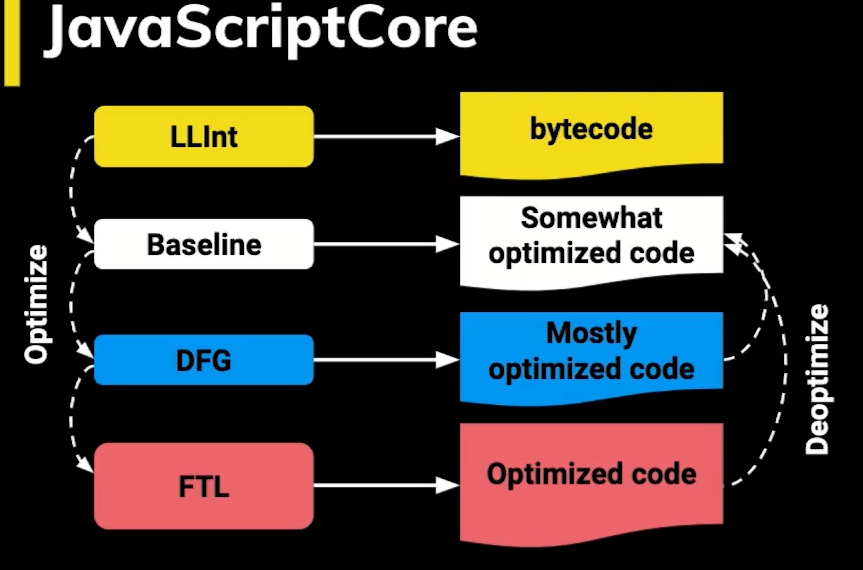
Cuando una sentencia función es ejecutada muy frecuentemente, V8 la denomina como una *hot function* y hace una optimización que consiste en convertirla a *machine code* para no tener que interpretarla de nuevo y agilizar su ejecución.

Cada navegador tiene su implementación de JavaScript Engine:

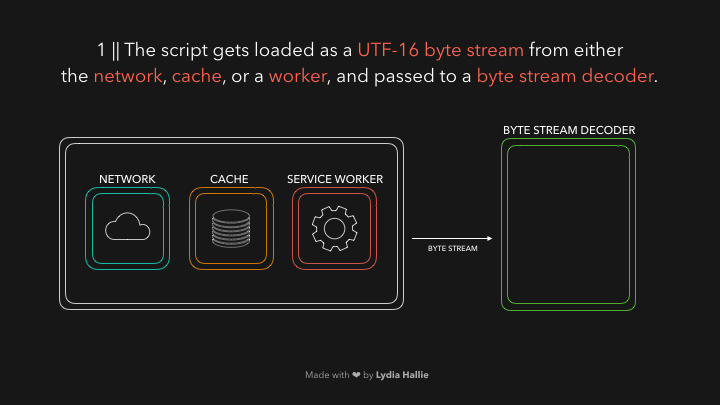
* SpiderMonkey - Firefox
* Chackra - Edge
* JavaScriptCore - Safari
* V8 – Chrome



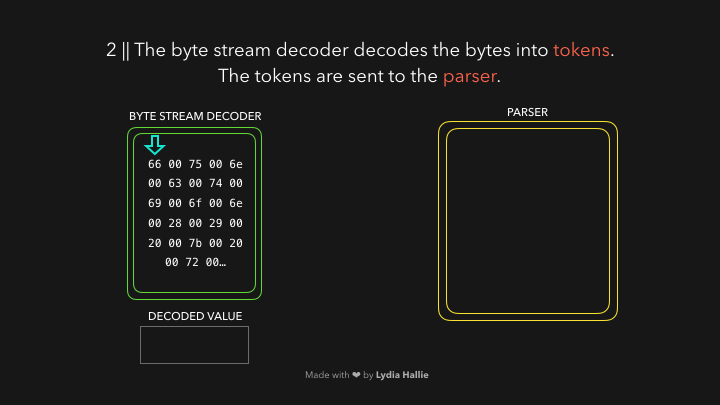
 

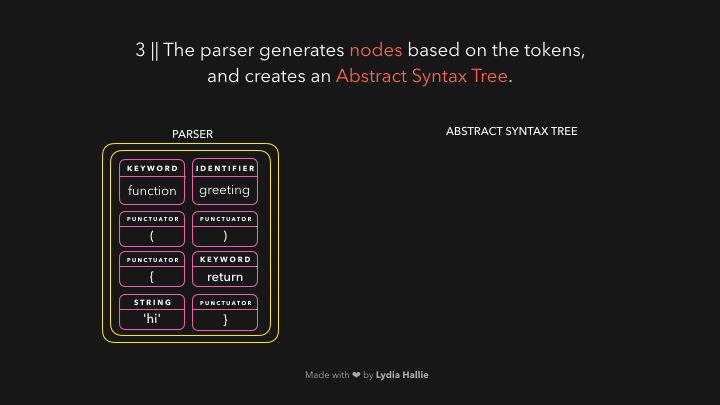
El script se carga como un **flujo de bytes UTF-16**, ya sea para la **red**, la **caché** o un\*\* trabajador\*\*, y se pasa a un **decodificador de flujo de bytes**.



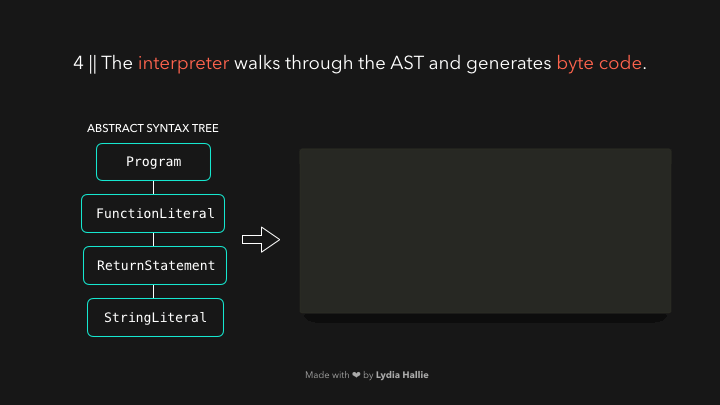
El decodificador de flujo de bytes decodifica los bytes en **tokens**. Los tokens se envían al **analizador**.



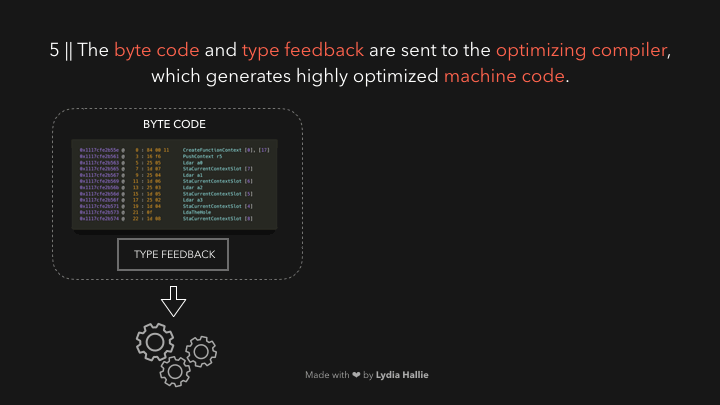
El analizador genera \*\*nodos \*\*basados en los tokens, y crea un **Abstract Syntax Tree**.



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.

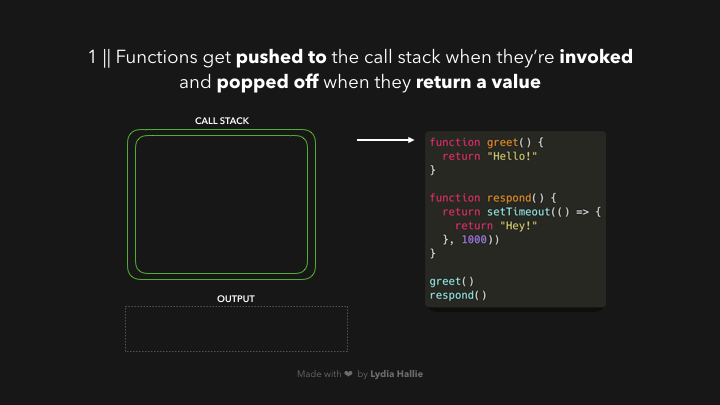
Javascript se organiza usando las siguientes estructuras de datos:

* **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.

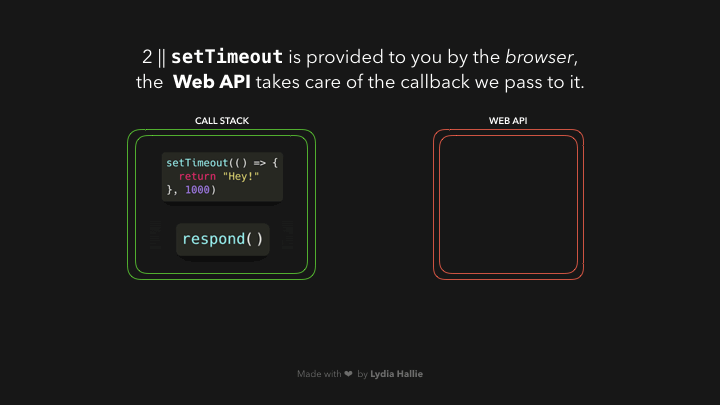
La explicaión del profesor es asombrosa. Explicar cómo funciona el EventLoop de una forma tan sencilla es asombroso.  
.  
Aquí dejo un resumen de todo el funcionamiento de lo que vimos en esta clase.  
.

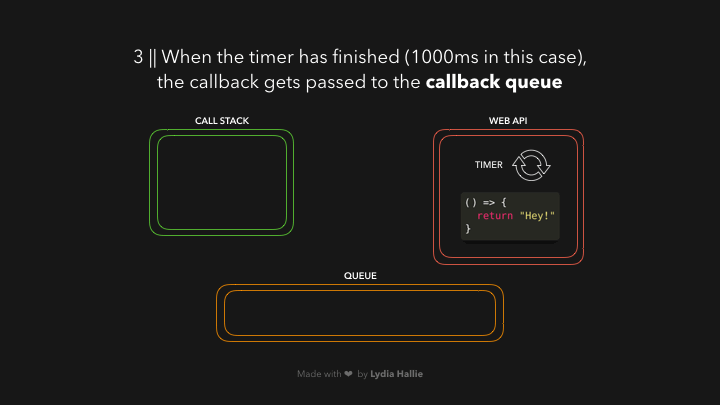
Las funciones son empujadas al **call stack** cuando son invocadas y se sacan cuando **devuelven un valor**

.  


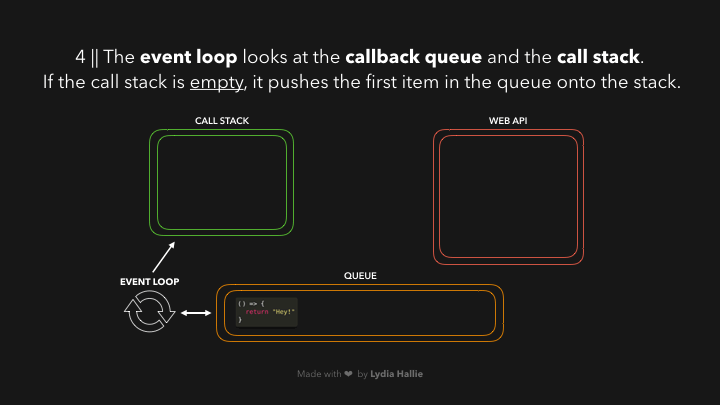
.

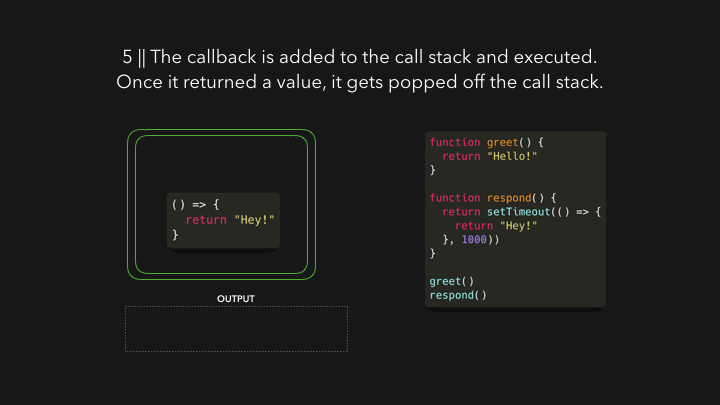
**setTimeOut** es proveído por el navegador, la **Web API** se encarga del **callback** que le pasemos.

.  


.  
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.

# Proxy

El proxy sirve para interceptar la lectura de propiedades de un objeto (los get, y set) entre muchas otras funciones. Así, antes de que la llamada llegue al objeto podemos manipularla con una lógica que nosotros definamos.

## Proxy

El objeto **Proxy** se usa para definir un comportamiento personalizado para operaciones fundamentales (por ejemplo, para observar propiedades, cuando se asignan, enumeración, invocación de funciones, etc).

## Métodos del objeto handler[Sección](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Proxy#M%C3%A9todos_del_objeto_handler)

El objeto controlador es un objeto marcador de posición que contiene trampas para Proxy.

Todas las trampas son opcionales. Si no se ha definido una trampa, el comportamiento predeterminado es reenviar la operación al objetivo.

[handler.get()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy/handler/get)

Una trampa para obtener valores de propiedad.

## Sintaxis[Sección](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy/handler/get#Syntax)

var p = new Proxy(target, {

get: function(target, property, receiver) {

}

});

Los siguientes parámetros se pasan al getmétodo. thisestá vinculado al controlador.

target

El objeto de destino.

property

El nombre o [Symbol](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol%22) de la propiedad a obtener.

receiver

El proxy o un objeto que hereda del proxy.

El getmétodo puede devolver cualquier valor.

## Ejemplo

const monster1 = {

secret: 'easily scared',

eyeCount: 4

};

const handler1 = {

get: function(target, prop, receiver) {

if (prop === 'secret') {

return `${target.secret.substr(0, 4)} ... shhhh!`;

} else {

return Reflect.get(...arguments);

}

}

};

const proxy1 = new Proxy(monster1, handler1);

console.log(proxy1.eyeCount);

// expected output: 4

console.log(proxy1.secret);

// expected output: "easi ... shhhh!"

## Object.keys()

El método Object.keys() devuelve un array de las propiedades names de un objeto, en el mismo orden como se obtienen en un loop normal.

Object.keys devuelve un array cuyos elementos son strings correspondientes a las propiedades enumerables que se encuentran directamente en el object. El orden de las propiedades es el mismo que se proporciona al iterar manualmente sobre las propiedades del objeto.

## Array.prototype.find()

El método \*\*find()\*\* devuelve el **valor** del **primer elemento** del array que cumple la función de prueba proporcionada. En cualquier otro caso se devuelve [undefined](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/undefined" \o "La propiedad global undefined representa el valor primitivo undefined. Es uno de los valores primitivos de JavaScript." \t "_blank).

## Distancia de Levenshtein

[Ir a la navegación](https://es.wikipedia.org/wiki/Distancia_de_Levenshtein#mw-head)[Ir a la búsqueda](https://es.wikipedia.org/wiki/Distancia_de_Levenshtein#p-search)

La **distancia de Levenshtein**, **distancia de edición** o **distancia entre palabras** es el número mínimo de operaciones requeridas para transformar una [cadena de caracteres](https://es.wikipedia.org/wiki/Cadena_de_caracteres) en otra, se usa ampliamente en [teoría de la información](https://es.wikipedia.org/wiki/Teor%C3%ADa_de_la_informaci%C3%B3n) y [ciencias de la computación](https://es.wikipedia.org/wiki/Ciencias_de_la_computaci%C3%B3n). Se entiende por operación, bien una inserción, eliminación o la sustitución de un carácter. Esta distancia recibe ese nombre en honor al científico ruso [Vladimir Levenshtein](https://es.wikipedia.org/wiki/Vladimir_Levenshtein), quien se ocupó de esta distancia en [1965](https://es.wikipedia.org/wiki/1965). Es útil en programas que determinan cuán similares son dos cadenas de caracteres, como es el caso de los [correctores ortográficos](https://es.wikipedia.org/wiki/Corrector_ortogr%C3%A1fico).

Otra gran utilidad de los Proxy son la validación de datos, podemos usarlos como piezas intermedias para controlar ingeniosamente las posibles entradas del usuario.

<https://es.wikipedia.org/wiki/Distancia_de_Levenshtein>  
<https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Proxy>

# Generators

Los generadores son funciones especiales, pueden pausar su ejecución y luego volver al punto donde se quedaron recordando su *scope*.

Algunas de sus características:

* Los generadores regresan una función.
* Empiezan suspendidos y se tiene que llamar *next* para que ejecuten.
* Regresan un *value* y un *boolean done* que define si ya terminaron.
* *yield* es la instrucción que regresa un valor cada vez que llamamos a *next* y detiene la ejecución del generador.

## Generador

El objecto Generador es retornado por [generator function](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Statements/function*" \o "La documentación acerca de este tema no ha sido escrita todavía . ¡Por favor  considera contribuir !" \t "_blank) y conforma tanto un protocolo iterable como un protocolo iterador.

## [Sintaxis](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Generador#Sintaxis)

function\* gen() {

yield 1;

yield 2;

yield 3;

}

var g = gen(); // "Generator { }"

## [Métodos](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Generador#M%C3%A9todos)

[Generator.prototype.next()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Generator/next)

Retorna el valor ofecido por la expresión [yield](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Operators/yield" \o "La documentación acerca de este tema no ha sido escrita todavía . ¡Por favor  considera contribuir !" \t "_blank)

[Generator.prototype.return()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Generator/return)

Retorna el valor dado y finaliza el generador.

[Generator.prototype.throw()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Generator/throw)

Lanzá un error al generador

## [Ejemplo](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Generador#Ejemplo)

[Un iterador infinito](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Generador#Un_iterador_infinito)

**function**\* **idMaker**() {

**var** index = 0;

**while**(**true**)

**yield** index++;

}

**var** gen = idMaker(); // "Generator { }"

console.log(gen.next().value); // 0

console.log(gen.next().value); // 1

console.log(gen.next().value); // 2

// ...

**APIs del dom**

# Fetch - Cómo cancelar peticiones

Las peticiones AJAX 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.

El método [fetch](https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/fetch" \t "_blank) abrirá un canal de comunicación con un recurso externo (en red) devolviendo una promesa, por lo tanto se puede acceder a la respuesta mediante un then.

fetchResponsePromise = fetch(resource, init);

Como todas las promesas se puede acceder a la respuesta dentro de funciones async, mediante al el uso de la palabra reservada await y de esta manera utilizar de manera más sencilla la respuesta.

fetch(myRequest).**then**(**function**(response) {

// code

});

fetch historicamente proviene del XMLHttpReques (o XHR para los cuates) con un funcionamiento similar al JQUERY.ajax (**A**syncronous **J**avaScript **A**nd **X**ML) con un mejor manejo de los errores (ya que fetch sólo fallará cuando hayan problemas de red, manejando los errores mediante Response.status, Response.statusText y Response.ok).  
.  
Una de las ventajas adicionales que se manejan con fetch vs XHR, es que fetch permite la cancelación de una señal (signal) mediante objetos de tipo AbortController.  
.  
**Para conseguir la señal interrumpida se debe de configurar un objeto de tipo AbortController, mismo que tiene una objeto de tipo AbortSignal, que es el que especificará si la señal se ha interrumpido o no (mediante AbortSignal.aborted)**

**let** controller = **new** AbortController()

**En el objeto de configuración del await existe un atributo llamado signal que recibe una señal de tipo AbortSignal (para este caso se guardó en el controller).**

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

**Para parar la señal, sólo debemos acceder al método abort() del AbortController que genera el AbortSignal**

controller.abort();

# IntersectionObserver

Sirve para observar elementos y si cruzan un umbral que nosotros definimos nos lo va a notificar para tomar acción.

El umbral se define por el porcentaje que tiene intersección con el viewport, con la parte visible de nuestra página.

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](https://developer.mozilla.org/es/docs/Glossary/viewport" \o "viewport: Un viewport representa la región poligonal (normalmente rectangular) en gráficas de computación que está siendo visualizada en ese instante. En términos de navegadores web, se refiere a la parte del documento que usted está viendo, la cual es actualmente visible en su ventana (o la pantalla, si el documento está siendo visto en modo pantalla completa). El contenido fuera del viewport no es visible en la pantalla hasta que sea desplazado dentro de él." \t "_blank) 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.

## [Creando un intersection observer](https://developer.mozilla.org/es/docs/Web/API/Intersection_Observer_API#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.

## Opciones de Intersection observer

El objeto options pasado al constructor [IntersectionObserver()](https://developer.mozilla.org/es/docs/Web/API/IntersectionObserver/IntersectionObserver" \o "La documentación acerca de este tema no ha sido escrita todavía . ¡Por favor  considera contribuir !" \t "_blank) 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" \o "La propiedad CSS margin establece el margen para los cuatro lados. Es una abreviación para evitar tener que establecer cada lado por separado con las otras propiedades de margen:  margin-top, margin-right, margin-bottom y margin-left." \t "_blank) 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.

## 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" \o "La documentación acerca de este tema no ha sido escrita todavía . ¡Por favor  considera contribuir !" \t "_blank) 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" \o "El método window.requestIdleCallback() encola la función que será ejecutada en periodos de inactividad del navegador permitiendo a los desarrolladores ejecutar en segundo plano tareas de baja prioridad del bucle de eventos, sin perjudicar la latencia de eventos principales como animaciones o respuestas a entradas. La funciones son ejecutadas normalmente en orden FIFO (primero en entrar primero en salir) salvo que se alcance el timeout definido de la función antes de que el navegador la ejecute." \t "_blank).

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

# VisibilityChange

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.

## Document: visibilitychange

E levento visibilitychange se dispara al documento cuando el contenido de su pestaña se ha hecho visible o se ha ocultado.

## Ejemplo

Este ejemplo comienza a reproducir una pista de música cuando el documento se hace visible, y pausa la música cuando el documento ya no es visible.

document.addEventListener("visibilitychange", **function**() {

**if** (document.visibilityState === 'visible') {

backgroundMusic.play();

} **else** {

backgroundMusic.pause();

}

});

## [Casos de uso](https://developer.mozilla.org/es/docs/Web/API/Page_Visibility_API#Casos_de_uso)

Consideremos algunos casos de uso para la API de Visibilidad de Página.

* 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 precargando para poder mantener un recuento preciso de las vistat 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).

# Service Workers

Sirven para hacer que nuestras aplicaciones funcionen Offline.

Muy usados en las **Progressive Web Apps** (PWA) los ServiceWorkers son una capa que vive entre el navegador y el Internet.

Parecido a como lo hacen los proxys van a interceptar peticiones para guardar el resultado en cache y la próxima vez que se haga la petición tomar del cache ese resultado.

Los Service workers actúan esencialmente como proxy servers asentados entre las aplicaciones web, el navegador y la red (cuando está accesible). Están destinados, entre otras cosas, a permitir la creación de experiencias offline efectivas, interceptando peticiones de red y realizando la acción apropiada si la conexión de red está disponible y hay disponibles contenidos actualizados en el servidor. También permitirán el acceso a notificaciones tipo push y APIs de sincronización en segundo plano.

## [Descarga, instalación y activación](https://developer.mozilla.org/es/docs/Web/API/Service_Worker_API#Descarga_instalaci%C3%B3n_y_activaci%C3%B3n)

En este punto, su service worker observará el siguiente ciclo de vida:

1. Descarga
2. Instalación
3. Activación

El service worker se descaga inmediatamente cuando un usuario accede por primera vez a un sitio controlado por el mismo.

Después de esto se descarga cada 24 horas aproximadamente. Se puede descargar con más frecuencia, pero **debe** ser descargado cada 24 horas para prevenir que una mala programación sea molesta durante mucho tiempo.

## ServiceWorkerContainer.register()

[Parámetros](https://developer.mozilla.org/es/docs/Web/API/ServiceWorkerContainer/register#Par%C3%A1metros)

ServiceWorkerContainer.register(scriptURL, options)

.then(**function**(ServiceWorkerRegistration) { ... });

scriptURL

La URL del script de trabajador de servicio.

options Optional

Un objeto que contiene opciones de registro. Las opciones disponibles actualmente son:

* alcance: [USVString](https://developer.mozilla.org/es/docs/Web/API/USVString" \o "La documentación acerca de este tema no ha sido escrita todavía . ¡Por favor  considera contribuir !" \t "_blank) representa una URL que define el alcance de registro de un “service worker”; es decir, qué rango de URL puede controlar un “service worker”. Esto es generalmente una URL relativa. El valor predeterminado es la URL que obtendría si resolviera ‘./’ utilizando la ubicación de la página web como base. No es, como se cree comúnmente, relativo a la ubicación del “service worker”. Vea la sección de Ejemplos para más información sobre cómo funciona.

## Ejemplo

**if** ('serviceWorker' **in** navigator) {

// Register a service worker hosted at the root of the

// site using the default scope.

navigator.serviceWorker.register('/sw.js').then(**function**(registration) {

console.log('Service worker registration succeeded:', registration);

}).catch(**function**(error) {

console.log('Service worker registration failed:', error);

});

} **else** {

console.log('Service workers are not supported.');

}

## CacheStorage.open()

The **open()** method of the [CacheStorage](https://developer.mozilla.org/en-US/docs/Web/API/CacheStorage" \o "The CacheStorage interface represents the storage for Cache objects." \t "_blank) interface returns a [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) that resolves to the [Cache](https://developer.mozilla.org/en-US/docs/Web/API/Cache) object matching the cacheName.

## Ejemplo

**var** cachedResponse = caches.match(event.request).catch(**function**() {

**return** fetch(event.request);

}).then(**function**(response) {

caches.open('v1').then(**function**(cache) {

cache.put(event.request, response);

});

**return** response.clone();

}).catch(**function**() {

**return** caches.match('/sw-test/gallery/myLittleVader.jpg');

});

## Cache

La \*\*Cache\*\*interfaz proporciona un mecanismo de almacenamiento para [Request](http://fetch.spec.whatwg.org/#request)/ [Response](http://fetch.spec.whatwg.org/#response)pares de objetos que se almacenan en caché, por ejemplo, como parte del [ServiceWorker](https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorker" \o "La interfaz ServiceWorker de la API ServiceWorker proporciona una referencia a un trabajador de servicio.  Se pueden asociar múltiples contextos de navegación (por ejemplo, páginas, trabajadores, etc.) con el mismo trabajador de servicio, cada uno a través de un objeto único de ServiceWorker." \t "_blank)ciclo de vida. Tenga en cuenta que la Cacheinterfaz está expuesta a ámbitos con ventanas, así como a los trabajadores. No tiene que usarlo junto con los trabajadores del servicio, aunque esté definido en la especificación del trabajador del servicio.

## Métodos[Sección](https://developer.mozilla.org/en-US/docs/Web/API/Cache#Methods)

[Cache.match(request, options)](https://developer.mozilla.org/en-US/docs/Web/API/Cache/match)

Devuelve un [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise" \o "El objeto Promise representa la finalización (o falla) eventual de una operación asincrónica y su valor resultante." \t "_blank)que resuelve la respuesta asociada con la primera solicitud coincidente en el Cacheobjeto.

[Cache.matchAll(request, options)](https://developer.mozilla.org/en-US/docs/Web/API/Cache/matchAll)

Devuelve un [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise" \o "El objeto Promise representa la finalización (o falla) eventual de una operación asincrónica y su valor resultante." \t "_blank)que resuelve una matriz de todas las solicitudes coincidentes en el Cacheobjeto.

[Cache.add(request)](https://developer.mozilla.org/en-US/docs/Web/API/Cache/add)

Toma una URL, la recupera y agrega el objeto de respuesta resultante a la caché dada. Esto es funcionalmente equivalente a llamar fetch(), luego usar put()para agregar los resultados al caché.

[Cache.addAll(requests)](https://developer.mozilla.org/en-US/docs/Web/API/Cache/addAll)

Toma una matriz de URL, las recupera y agrega los objetos de respuesta resultantes a la caché dada.

[Cache.put(request, response)](https://developer.mozilla.org/en-US/docs/Web/API/Cache/put)

Toma tanto una solicitud como su respuesta y la agrega al caché dado.

[Cache.delete(request, options)](https://developer.mozilla.org/en-US/docs/Web/API/Cache/delete)

Encuentra la [Cache](https://developer.mozilla.org/en-US/docs/Web/API/Cache) entrada cuya clave es la solicitud, devolviendo una [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise" \o "El objeto Promise representa la finalización (o falla) eventual de una operación asincrónica y su valor resultante." \t "_blank)que se resuelve truesi Cachese encuentra y elimina una entrada coincidente . Si no Cachese encuentra ninguna entrada, la promesa se resuelve false.

[Cache.keys(request, options)](https://developer.mozilla.org/en-US/docs/Web/API/Cache/keys)

Devuelve un [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise" \o "El objeto Promise representa la finalización (o falla) eventual de una operación asincrónica y su valor resultante." \t "_blank)que se resuelve en una matriz de Cacheclaves.

**TypeScript**

# Introducción

**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**.

**TypeScript** es un lenguaje de programación libre y de [código abierto](https://es.wikipedia.org/wiki/C%C3%B3digo_abierto) desarrollado y mantenido por [Microsoft](https://es.wikipedia.org/wiki/Microsoft). Es un superconjunto de [JavaScript](https://es.wikipedia.org/wiki/JavaScript), que esencialmente añade tipos estáticos y objetos basados en clases. [Anders Hejlsberg](https://es.wikipedia.org/wiki/Anders_Hejlsberg" \o "Anders Hejlsberg" \t "_blank), diseñador de [C#](https://es.wikipedia.org/wiki/C) y creador de [Delphi](https://es.wikipedia.org/wiki/Embarcadero_Delphi) y [Turbo Pascal](https://es.wikipedia.org/wiki/Turbo_Pascal), ha trabajado en el desarrollo de TypeScript.[1](https://es.wikipedia.org/wiki/TypeScript#cite_note-1)​ TypeScript puede ser usado para desarrollar aplicaciones JavaScript que se ejecutarán en el lado del cliente o del servidor ([Node.js](https://es.wikipedia.org/wiki/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.

## [Introducción](https://www.typescriptlang.org/docs/handbook/basic-types.html#introduction)

Para que los programas sean útiles, necesitamos poder trabajar con algunas de las unidades de datos más simples: números, cadenas, estructuras, valores booleanos y similares. En TypeScript, admitimos casi los mismos tipos que esperaría en JavaScript, con un tipo de enumeración conveniente para ayudar a las cosas.

## TypeScript en 5 minutos

Comencemos creando una aplicación web simple con TypeScript.

## [Instalación de TypeScript](https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html#installing-typescript)

Hay dos formas principales de obtener las herramientas TypeScript:

* Vía npm (el administrador de paquetes Node.js)
* Al instalar los complementos de Visual Studio de TypeScript

Visual Studio 2017 y Visual Studio 2015 Update 3 incluyen TypeScript de forma predeterminada. Si no instaló TypeScript con Visual Studio, aún puede [descargarlo](https://www.typescriptlang.org/#download-links) .

Para usuarios de NPM:

> npm **install** -g typescript

## [Construyendo su primer archivo TypeScript](https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html#building-your-first-typescript-file)

En su editor, escriba el siguiente código JavaScript en greeter.ts:

**function** **greeter**(person) {

**return** "Hello, " + person;

}

**let** user = "Jane User";

document.body.textContent = greeter(user);

## [Compilando su código](https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html#compiling-your-code)

Utilizamos una .tsextensión, pero este código es solo JavaScript. Podría haber copiado / pegado esto directamente desde una aplicación JavaScript existente.

En la línea de comando, ejecute el compilador TypeScript:

tsc greeter.ts

El resultado será un archivo greeter.jsque contiene el mismo JavaScript que usted introdujo. ¡Estamos funcionando con TypeScript en nuestra aplicación JavaScript!

Ahora podemos comenzar a aprovechar algunas de las nuevas herramientas que ofrece TypeScript. Agregue una : stringanotación de tipo al argumento de la función ‘persona’ como se muestra aquí:

**function** **greeter**(person: string) {

**return** "Hello, " + person;

}

**let** user = "Jane User";

document.body.textContent = greeter(user);

## [Numero](https://www.typescriptlang.org/docs/handbook/basic-types.html#number)

Como en JavaScript, todos los números en TypeScript son valores de coma flotante. Estos números de coma flotante obtienen el tipo number. Además de los literales hexadecimales y decimales, TypeScript también admite literales binarios y octales introducidos en ECMAScript 2015.

**let** decimal: number = 6;

**let** hex: number = 0xf00d;

**let** binary: number = 0b1010;

**let** octal: number = 0o744;

# Tipos básicos

Tipos:

* **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.

## [Boolean](https://www.typescriptlang.org/docs/handbook/basic-types.html#boolean)

El tipo de datos más básico es el valor verdadero / falso simple, que JavaScript y TypeScript llaman un booleanvalor.

**let** isDone: boolean = **false**;

## [Numero](https://www.typescriptlang.org/docs/handbook/basic-types.html#number)

Como en JavaScript, todos los números en TypeScript son valores de coma flotante. Estos números de coma flotante obtienen el tipo number. Además de los literales hexadecimales y decimales, TypeScript también admite literales binarios y octales introducidos en ECMAScript 2015.

**let** decimal: number = 6;

**let** hex: number = 0xf00d;

**let** binary: number = 0b1010;

**let** octal: number = 0o744;

## [String](https://www.typescriptlang.org/docs/handbook/basic-types.html#string)

Otra parte fundamental de la creación de programas en JavaScript para páginas web y servidores es trabajar con datos textuales. Como en otros idiomas, usamos el tipo stringpara referirnos a estos tipos de datos textuales. Al igual que JavaScript, TypeScript también utiliza comillas dobles ( ") o comillas simples ( ') para rodear los datos de cadena.

**let** color: string = "blue";

color = 'red';

También puede usar cadenas de plantillas , que pueden abarcar varias líneas y tener expresiones incrustadas. Estas cadenas están rodeadas por el carácter backtick / backquote ( `), y las expresiones incrustadas son de la forma ${ expr }.

**let** fullName: string = `Bob Bobbington`;

**let** age: number = 37;

**let** sentence: string = `Hello, my name is ${ fullName }.

I'll be ${ age + 1 } years old next month.`;

Esto es equivalente a declarar sentenceasí:

**let** sentence: string = "Hello, my name is " + fullName + ".\n\n" +

"I'll be " + (age + 1) + " years old next month.";

## [Array](https://www.typescriptlang.org/docs/handbook/basic-types.html#array)

TypeScript, como JavaScript, le permite trabajar con matrices de valores. Los tipos de matriz se pueden escribir de una de dos maneras. En el primero, usa el tipo de elementos seguido de []para denotar una matriz de ese tipo de elemento:

**let** list: number[] = [1, 2, 3];

La segunda forma usa un tipo de matriz genérico Array<elemType>:

**let** list: Array<number> = [1, 2, 3];

## [Tuple](https://www.typescriptlang.org/docs/handbook/basic-types.html#tuple)

Los tipos de tupla le permiten expresar una matriz con un número fijo de elementos cuyos tipos son conocidos, pero no necesitan ser los mismos. Por ejemplo, es posible que desee representar un valor como un par de ay stringa number:

// Declare a tuple type

**let** x: [string, number];

// Initialize it

x = ["hello", 10]; // OK

// Initialize it incorrectly

x = [10, "hello"]; // Error

Al acceder a un elemento con un índice conocido, se recupera el tipo correcto:

console.log(x[0].substring(1)); // OK

console.log(x[1].substring(1)); // Error, 'number' does not have 'substring'

El acceso a un elemento fuera del conjunto de índices conocidos falla con un error:

x[3] = "world"; // Error, Property '3' does not exist on type '[string, number]'.

console.log(x[5].toString()); // Error, Property '5' does not exist on type '[string, number]'.

## [Enum](https://www.typescriptlang.org/docs/handbook/basic-types.html#enum)

Una adición útil al conjunto estándar de tipos de datos de JavaScript es enum. Como en lenguajes como C #, una enumeración es una forma de dar nombres más amigables a conjuntos de valores numéricos.

**enum** Color {Red, Green, Blue}

**let** c: Color = Color.Green;

Por defecto, las enumeraciones comienzan a numerar a sus miembros a partir de 0. Puede cambiar esto configurando manualmente el valor de uno de sus miembros. Por ejemplo, podemos comenzar el ejemplo anterior en 1lugar de 0:

**enum** Color {Red = 1, Green, Blue}

**let** c: Color = Color.Green;

O, incluso establezca manualmente todos los valores en la enumeración:

**enum** Color {Red = 1, Green = 2, Blue = 4}

**let** c: Color = Color.Green;

Una característica útil de las enumeraciones es que también puede pasar de un valor numérico al nombre de ese valor en la enumeración. Por ejemplo, si tuviéramos el valor 2pero no estuviéramos seguros de a qué se asignó en la Colorenumeración anterior, podríamos buscar el nombre correspondiente:

**enum** Color {Red = 1, Green, Blue}

**let** colorName: string = Color[2];

console.log(colorName); // Displays 'Green' as its value is 2 above

## [Any](https://www.typescriptlang.org/docs/handbook/basic-types.html#any)

Es posible que necesitemos describir el tipo de variables que no sabemos cuando estamos escribiendo una solicitud. Estos valores pueden provenir de contenido dinámico, por ejemplo, del usuario o de una biblioteca de terceros. En estos casos, queremos inhabilitar la verificación de tipos y dejar que los valores pasen por las comprobaciones en tiempo de compilación. Para hacerlo, los etiquetamos con el anytipo:

**let** notSure: any = 4;

notSure = "maybe a string instead";

notSure = **false**; // okay, definitely a boolean

El anytipo es una forma poderosa de trabajar con JavaScript existente, lo que le permite activar y desactivar gradualmente la verificación de tipos durante la compilación. Puede esperar Objectjugar un papel similar, como lo hace en otros idiomas. Sin embargo, las variables de tipo Objectsolo le permiten asignarles cualquier valor. No puede invocar métodos arbitrarios en ellos, incluso los que realmente existen:

**let** notSure: any = 4;

notSure.ifItExists(); // okay, ifItExists might exist at runtime

notSure.toFixed(); // okay, toFixed exists (but the compiler doesn't check)

**let** prettySure: Object = 4;

prettySure.toFixed(); // Error: Property 'toFixed' doesn't exist on type 'Object'.

El anytipo también es útil si conoce alguna parte del tipo, pero quizás no toda. Por ejemplo, puede tener una matriz, pero la matriz tiene una combinación de diferentes tipos:

**let** list: any[] = [1, **true**, "free"];

list[1] = 100;

## [Object](https://www.typescriptlang.org/docs/handbook/basic-types.html#object)

objectes un tipo que representa el tipo no primitivo, es decir, todo lo que no es number, string, boolean, symbol, null, o undefined.

Con el objecttipo, las API como Object.createse pueden representar mejor. Por ejemplo:

**declare** **function** **create**(o: object | **null**): **void**;

create({ prop: 0 }); // OK

create(**null**); // OK

create(42); // Error

create("string"); // Error

create(**false**); // Error

create(**undefined**); // Error

# Funciones

En Typescript podemos ser explícitos con el tipo de los argumentos y el tipo de retorno de una función.

TypeScript aporta varias características que JavaScript no dispone hasta el momento cuando tenemos que plantear funciones y métodos.

## Parámetros tipados y funciones que retornan un valor.

Podemos indicar a cada parámetro el tipo de dato que puede recibir y también el tipo de dato que retorna la función o método en caso que estemos en una clase:

**function** **sumar**(valor1:number, valor2:number): **number** {

**return** valor1+valor2;

}

console.log(sumar(10, 5));

La función sumar recibe dos parámetros de tipo number y retorna un valor de tipo number. Luego si llamamos a esta función enviando un valor distinto a number el compilador nos avisará del error:

console.log(sumar('juan', 'carlos'));

Se genera un error: "Argument of type '"juan"' is not assignable to parameter of type 'number'.

Inclusive editores de texto moderno como Visual Studio Code pueden antes de compilarse avisar del error.

El tipado estático favorece a identificar este tipo de errores antes de ejecutar la aplicación. Lo mismo cuando una función retorna un dato debemos indicar al final de la misma dicho tipo:

**function** **sumar**(valor1:number, valor2:number): **number** {}

La función sumar retorna un valor de tipo number.

Luego si la función retorna un tipo distinto a number se genera un error:

**function** **sumar**(valor1:number, valor2:number): **number** {

**return** 'Hola mundo';

}

Como estamos retornando un string se genera el error: Type '"Hola mundo"' is not assignable to type 'number'.

## [Parámetros opcionales y predeterminados](https://www.typescriptlang.org/docs/handbook/functions.html#optional-and-default-parameters)

En TypeScript, se supone que cada parámetro es requerido por la función. Esto no significa que no se pueda dar nullo undefined, sino que, cuando se llama a la función, el compilador verificará que el usuario haya proporcionado un valor para cada parámetro. El compilador también supone que estos parámetros son los únicos parámetros que se pasarán a la función. En resumen, el número de argumentos dados a una función tiene que coincidir con el número de parámetros que la función espera.

**function** **buildName**(firstName: string, lastName: string) {

**return** firstName + " " + lastName;

}

**let** result1 = buildName("Bob"); // error, too few parameters

**let** result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters

**let** result3 = buildName("Bob", "Adams"); // ah, just right

En JavaScript, cada parámetro es opcional, y los usuarios pueden dejarlos como mejor les parezca. Cuando lo hacen, su valor es undefined. Podemos obtener esta funcionalidad en TypeScript agregando un ?al final de los parámetros que queremos que sean opcionales. Por ejemplo, supongamos que queremos que el parámetro de apellido anterior sea opcional:

**function** **buildName**(firstName: string, lastName?: string) {

**if** (lastName)

**return** firstName + " " + lastName;

**else**

**return** firstName;

}

**let** result1 = buildName("Bob"); // works correctly now

**let** result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters

**let** result3 = buildName("Bob", "Adams"); // ah, just right

Cualquier parámetro opcional debe seguir los parámetros requeridos. Si quisiéramos hacer que el primer nombre fuera opcional, en lugar del apellido, tendríamos que cambiar el orden de los parámetros en la función, colocando el primer nombre en la lista.

En TypeScript, también podemos establecer un valor que se asignará a un parámetro si el usuario no proporciona uno, o si el usuario pasa undefineden su lugar. Estos se denominan parámetros inicializados por defecto. Tomemos el ejemplo anterior y establezcamos el apellido por defecto "Smith".

function buildName(firstName: string, lastName = "Smith") {

return firstName + " " + lastName;

}

let result1 = buildName("Bob"); // works correctly now, returns "Bob Smith"

let result2 = buildName("Bob", undefined); // still works, also returns "Bob Smith"

let result3 = buildName("Bob", "Adams", "Sr."); // error, too many parameters

let result4 = buildName("Bob", "Adams"); // ah, just right

Los parámetros inicializados por defecto que vienen después de que todos los parámetros requeridos se tratan como opcionales, y al igual que los parámetros opcionales, se pueden omitir al llamar a su función respectiva. Esto significa que los parámetros opcionales y los parámetros predeterminados finales compartirán elementos comunes en sus tipos, por lo que ambos

**function** **buildName**(firstName: string, lastName?: string) {

// ...

}

y

function buildName(firstName: string, lastName = "Smith") {

// ...

}

compartir el mismo tipo (firstName: string, lastName?: string) => string. El valor predeterminado de lastNamedesaparece en el tipo, solo dejando atrás el hecho de que el parámetro es opcional.

A diferencia de los parámetros opcionales simples, los parámetros inicializados por defecto no necesitan ocurrir después de los parámetros requeridos. Si un parámetro inicializado predeterminado viene antes que un parámetro requerido, los usuarios deben pasar explícitamente undefinedpara obtener el valor inicializado predeterminado. Por ejemplo, podríamos escribir nuestro último ejemplo con solo un inicializador predeterminado en firstName:

function buildName(firstName = "Will", lastName: string) {

return firstName + " " + lastName;

}

let result1 = buildName("Bob"); // error, too few parameters

let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters

let result3 = buildName("Bob", "Adams"); // okay and returns "Bob Adams"

let result4 = buildName(undefined, "Adams"); // okay and returns "Will Adams"

# Interfaces

Nos permiten declarar la forma exacta de un objeto, definiendo los tipos de sus propiedades y si son opcionales o no.

## [Introducción](https://www.typescriptlang.org/docs/handbook/interfaces.html#introduction)

Uno de los principios centrales de TypeScript es que la verificación de tipos se centra en la forma que tienen los valores. Esto a veces se llama “tipificación de pato” o “subtipo estructural”. En TypeScript, las interfaces cumplen la función de nombrar estos tipos y son una forma poderosa de definir contratos dentro de su código, así como contratos con código fuera de su proyecto.

## [Nuestra primera interfaz](https://www.typescriptlang.org/docs/handbook/interfaces.html#our-first-interface)

La forma más fácil de ver cómo funcionan las interfaces es comenzar con un ejemplo simple:

**function** **printLabel**(labeledObj: { label: string }) {

console.log(labeledObj.label);

}

**let** myObj = {size: 10, label: "Size 10 Object"};

printLabel(myObj);

El verificador de tipos verifica la llamada a printLabel. La printLabelfunción tiene un único parámetro que requiere que el objeto pasado tenga una propiedad llamada labelde tipo string. Tenga en cuenta que nuestro objeto en realidad tiene más propiedades que esto, pero el compilador solo verifica que al menos las requeridas estén presentes y coincidan con los tipos requeridos. Hay algunos casos en que TypeScript no es tan indulgente, que cubriremos en un momento.

Podemos escribir el mismo ejemplo nuevamente, esta vez usando una interfaz para describir el requisito de tener la labelpropiedad que es una cadena:

**interface** LabeledValue {

label: string;

}

**function** **printLabel**(labeledObj: LabeledValue) {

console.log(labeledObj.label);

}

**let** myObj = {size: 10, label: "Size 10 Object"};

printLabel(myObj);

La interfaz LabeledValuees un nombre que ahora podemos usar para describir el requisito en el ejemplo anterior. Todavía representa tener una sola propiedad llamada labelque es de tipo string. Tenga en cuenta que no tuvimos que decir explícitamente que el objeto que pasamos printLabelimplementa esta interfaz como podríamos tener que hacerlo en otros idiomas. Aquí, lo único que importa es la forma. Si el objeto que pasamos a la función cumple con los requisitos enumerados, entonces está permitido.

Vale la pena señalar que el verificador de tipo no requiere que estas propiedades vengan en ningún tipo de orden, solo que las propiedades que requiere la interfaz están presentes y tienen el tipo requerido.

## [Propiedades opcionales](https://www.typescriptlang.org/docs/handbook/interfaces.html#optional-properties)

No todas las propiedades de una interfaz pueden ser necesarias. Algunos existen bajo ciertas condiciones o pueden no estar allí en absoluto. Estas propiedades opcionales son populares cuando se crean patrones como “bolsas de opciones” donde se pasa un objeto a una función que solo tiene un par de propiedades rellenadas.

Aquí hay un ejemplo de este patrón:

**interface** SquareConfig {

color?: string;

width?: number;

}

**function** **createSquare**(config: SquareConfig): {color: string; area: number} {

**let** newSquare = {color: "white", area: 100};

**if** (config.color) {

newSquare.color = config.color;

}

**if** (config.width) {

newSquare.area = config.width \* config.width;

}

**return** newSquare;

}

**let** mySquare = createSquare({color: "black"});

Las interfaces con propiedades opcionales se escriben de manera similar a otras interfaces, con cada propiedad opcional indicada por un ?al final del nombre de la propiedad en la declaración.

La ventaja de las propiedades opcionales es que puede describir estas propiedades posiblemente disponibles y al mismo tiempo evitar el uso de propiedades que no forman parte de la interfaz. Por ejemplo, si hubiéramos escrito mal el nombre de la colorpropiedad createSquare, obtendríamos un mensaje de error informándonos:

**interface** SquareConfig {

color?: string;

width?: number;

}

**function** **createSquare**(config: SquareConfig): { color: string; area: number } {

**let** newSquare = {color: "white", area: 100};

**if** (config.clor) {

// Error: Property 'clor' does not exist on type 'SquareConfig'

newSquare.color = config.clor;

}

**if** (config.width) {

newSquare.area = config.width \* config.width;

}

**return** newSquare;

}

**let** mySquare = createSquare({color: "black"});

# Clases

En las clases en TypeScript sí existen las propiedades privadas.

## [Introducción](https://www.typescriptlang.org/docs/handbook/classes.html#introduction)

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.

## [Clases](https://www.typescriptlang.org/docs/handbook/classes.html#classes)

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.

## [Herencia](https://www.typescriptlang.org/docs/handbook/classes.html#inheritance)

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 Animal base 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().

## [Modificadores públicos, privados y protegidos.](https://www.typescriptlang.org/docs/handbook/classes.html#public-private-and-protected-modifiers)

## [Público por defecto](https://www.typescriptlang.org/docs/handbook/classes.html#public-by-default)

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;

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

**public** move(distanceInMeters: number) {

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

}

}

## [Comprensión private](https://www.typescriptlang.org/docs/handbook/classes.html#understanding-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 .

## [Comprensiónprotected](https://www.typescriptlang.org/docs/handbook/classes.html#understanding-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

Con experiencia en desarrollo móvil e interesado y especializado en tecnologías web

Pensar en ingles

Agregar 10 reclutadores al dia

Mover

Hagámosle