

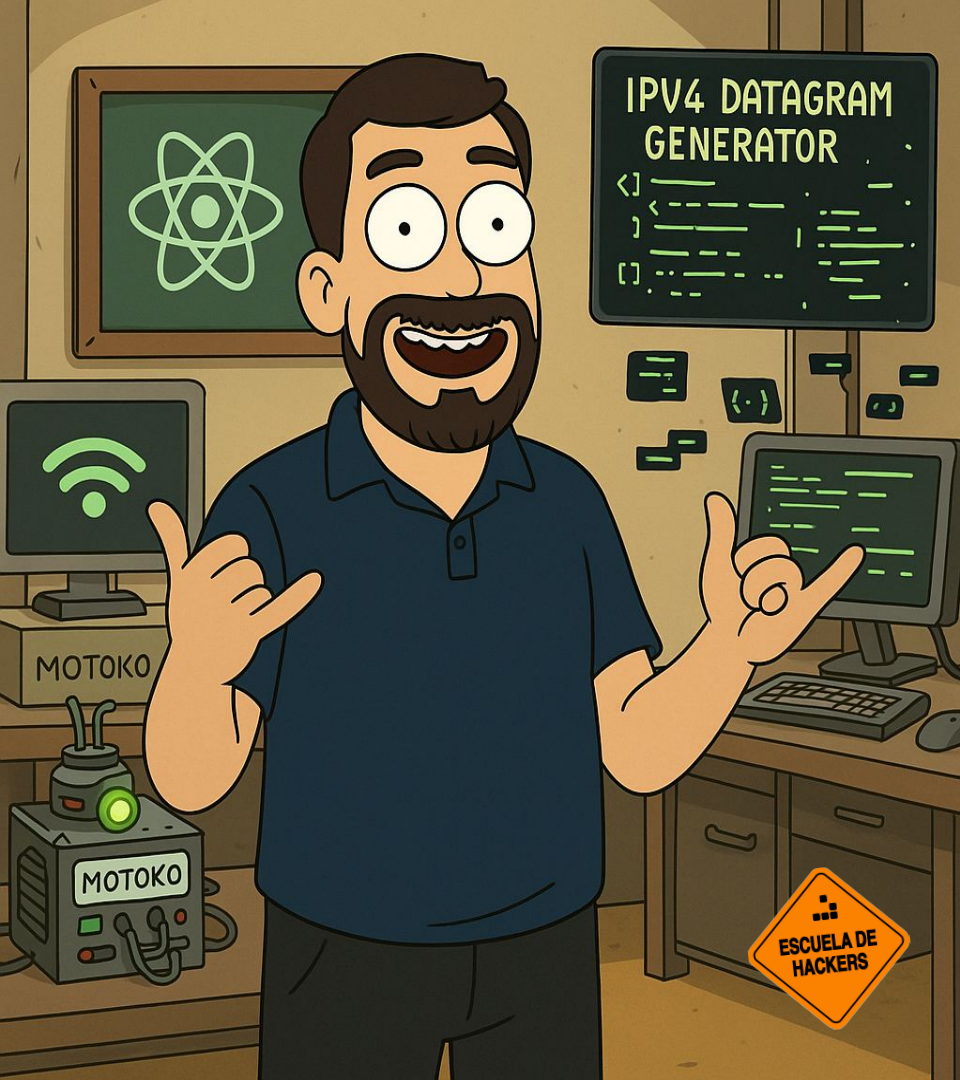
Asincronismo en Javascript

DEV.F.
DESARROLLAMOS(PERSONAS);

Elaborado por: Jesua Luján

 [Jesua Hadai Luján](#)

dev



Objetivos de la sesión: 🔥

- Entender la **lógica de los hilos de ejecución**. 🤔
- Comprender **la concurrencia y paralelismo en la ejecución de procesos** y su rendimiento. 🧑💻
- Aprender cómo JS se ejecuta con ayuda del **Call Stack** y el **Callback queue**. 😈
- Entender qué son los **Callbacks**.





Guía Práctica de Asincronía y Concurrencia en JavaScript

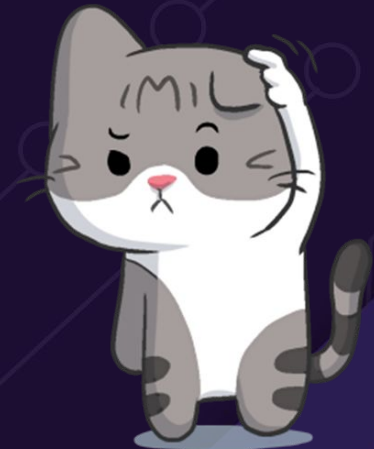
¿Por qué importa esto?

Cuando escribes código, normalmente lo haces de forma **sincrónica**: línea por línea, paso a paso. Pero el mundo real (¡y la web!) no funciona así. Las APIs tardan, los archivos se leen lento, y si tu código espera por todo eso... se congela más que un Windows 98.



HILOS DE EJECUCIÓN

DEV.F.
DESARROLLAMOS(PERSONAS);



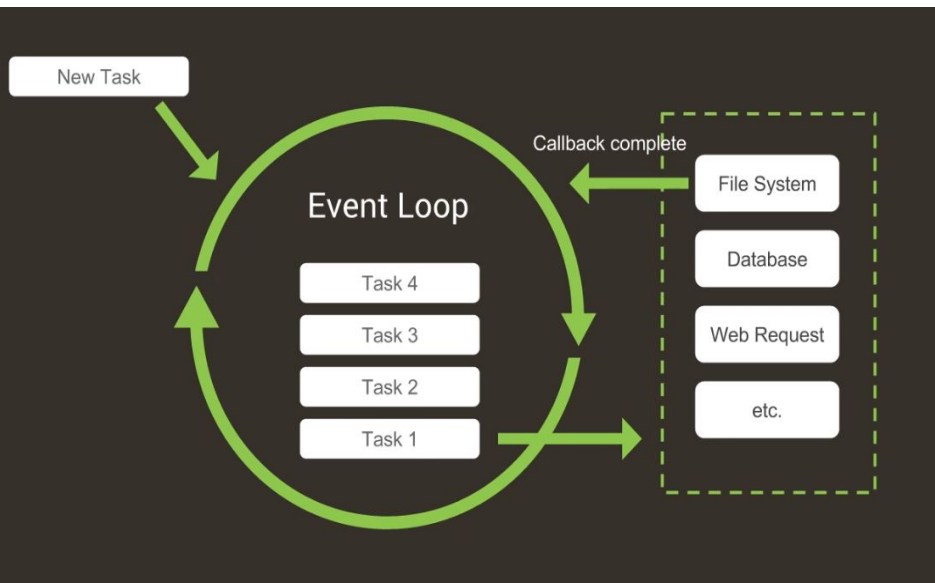


Hilos de Ejecución

JavaScript es un **lenguaje de un solo hilo (single-threaded)**. Esto significa que solo puede hacer **una cosa a la vez**. Todo el código se ejecuta en el **hilo principal**.

Pero, ¿cómo hace entonces para no quedarse esperando que termine una petición de red o una lectura de archivo?

Respuesta mágica: **asincronía + event loop**.





Hilos de Ejecución

Un hilo de ejecución en JavaScript es una **instancia de un objeto que permite a la aplicación ejecutar varias tareas al mismo tiempo**. Cada hilo de ejecución tiene su propia pila de ejecución, lo que significa que cada hilo puede ejecutar código de manera independiente y simultánea.

```
// Definir una función llamada saludar que recibe el nombre y la edad
const saludar = (nombre, edad) => {
  console.log(`Hola ${nombre} tu edad es de ${edad}`);
};
// Llamar a saludar y pasarle argumentos
setTimeout(saludar, 1000, "Luis", 20);
```

Nombre de función

Milisegundos de espera

Argumentos que se pasan a función. Podrían ser infinitos





Asincronía en JavaScript

La **asincronía** permite que el programa siga corriendo sin tener que esperar que algo termine.

```
console.log("Inicio");

setTimeout(() => {
  console.log("Tarea asíncrona");
}, 1000);

console.log("Fin");
```

Resultado:

```
Inicio
Fin
Tarea asíncrona
```



```
const tarea1 = () => {  
  console.log('Tarea 1 iniciada');  
  setTimeout(() => {  
    console.log('Tarea 1 completada 🚀');  
  }, 2000);  
};
```

Codeium: Refactor | Explain | Generate JSDoc | X

```
const tarea2 = () => {  
  console.log('Tarea 2 iniciada' );  
  setTimeout(() => {  
    console.log('Tarea 2 completada 🚀');  
  }, 1000);  
};
```

```
tarea1();  
tarea2();
```



Hilos de Ejecución

En JavaScript, los hilos de ejecución se implementan **mediante el uso de eventos y callbacks**. Cuando se ejecuta un código asíncrono, como una promesa o una función de callback, el motor de JavaScript **crea un hilo de ejecución para ejecutar esa tarea**.

Hilos de ejecución

En este ejemplo, se crean dos tareas (tarea1 y tarea2) que se ejecutan al mismo tiempo. Cada tarea imprime un mensaje en la consola y luego espera 1 segundo o 2 segundos antes de imprimir otro mensaje. Aunque las tareas se ejecutan al mismo tiempo, el orden en que se imprimen los mensajes puede variar, ya que cada tarea tiene su propio hilo de ejecución.

```
const tarea1 = () => {  
  console.log('Tarea 1 iniciada');  
  setTimeout(() => {  
    console.log('Tarea 1 completada 🚀');  
  }, 2000);  
};
```

Codeium: Refactor | Explain | Generate JSDoc | X

```
const tarea2 = () => {  
  console.log('Tarea 2 iniciada ');  
  setTimeout(() => {  
    console.log('Tarea 2 completada 🚀');  
  }, 1000);  
};
```

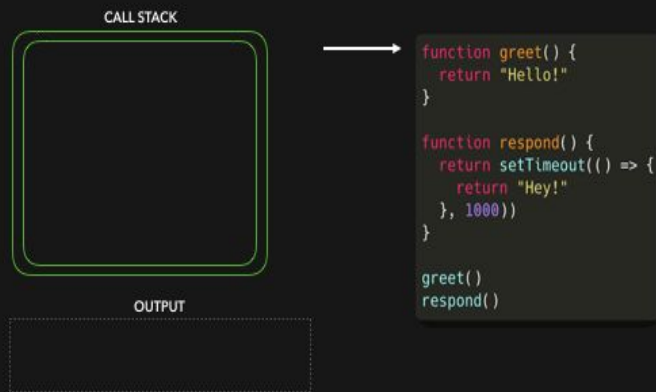
```
tarea1();  
tarea2();
```



Hilos de ejecución

En resumen, un hilo de ejecución en JavaScript es una instancia de un objeto que permite a la aplicación ejecutar varias tareas al mismo tiempo y **cada tarea se ejecuta en su propio hilo de ejecución.**

1 || Functions get **pushed to** the call stack when they're **invoked** and **popped off** when they **return a value**



Made with ♥ by Lydia Hallie

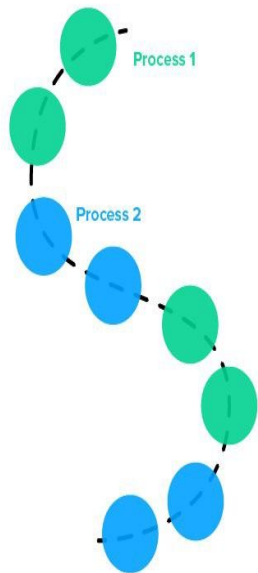


Concurrencia y Paralelismo

DEV.FX
DESARROLLAMOS(PERSONAS);

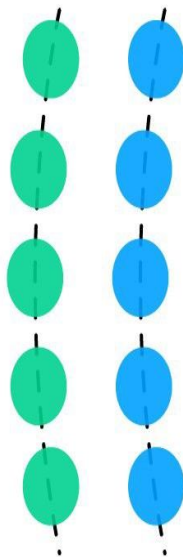


Concurrency



VS

Parallelism



Concurrencia vs Paralelismo

- **Concurrencia:** Capacidad de manejar múltiples tareas, alternando entre ellas.
- **Paralelismo:** Ejecutar varias tareas *al mismo tiempo* (literalmente).





Concurrencia vs Paralelismo

- **Concurrencia:** Capacidad del CPU para ejecutar más de un proceso al mismo tiempo (ayudándose del número de núcleos).
- **Paralelismo:** Toma un único problema y mediante concurrencia llega a la solución más rápido (divide y venceras).



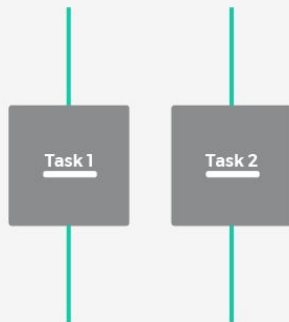
THREAD ÚNICO

MÚLTIPLES THREADS

SECUENCIAL



1

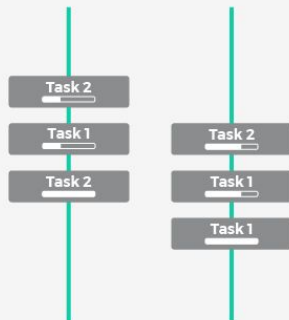


2

ENTRELAZADO



3



4





Concurrencia vs Paralelismo

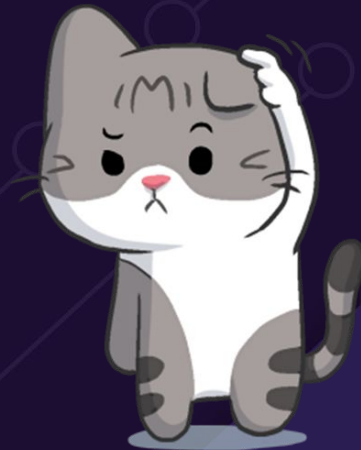
Concepto	Qué es	JavaScript lo hace
Concurrencia	Capacidad de manejar múltiples tareas, alternando entre ellas.	✅ Sí (gracias al event loop)
Paralelismo	Ejecutar varias tareas <i>al mismo tiempo</i> (literalmente).	❌ No (a menos que uses Web Workers)

JavaScript no hace **paralelismo verdadero** porque solo tiene un hilo, pero puede simular **concurrencia** muy eficientemente.



Callbacks

DEV.F
DESARROLLAMOS(PERSONAS);



Callbacks

```
function saludar(nombre) {  
  alert('Hola ' + nombre)  
}
```

```
function procesarEntradaUsuario(callback) {  
  var nombre = prompt('Por favor ingresa tu nombre')  
  callback(nombre)  
}
```

```
procesarEntradaUsuario(saludar)
```

Promises

```
function saludar(nombre) {  
  alert('Hola ' + nombre)  
}
```

```
function procesarEntradaUsuario() {  
  let nombre = prompt('Por favor ingresa tu nombre')  
  return Promise.resolve(nombre)  
}
```

```
procesarEntradaUsuario()  
  .then(nombre => saludar(nombre))
```



JS

¿Qué es un Callback?

- Se le conoce como **llamada de vuelta**.
- Es una función que recibe como parámetro **otra función y la ejecuta**.
- La función “callback” por lo regular **va a realizar algo con los resultados de la función que la está ejecutando**.

```
function operation (num1, num2, callback) {  
  return setTimeout (() => callback(num1, num2), 2000)  
}
```

```
operation (1, 3, (a , b) => { console.log(a + b) })  
operation (1, 3, (a , b) => { console.log(a * b) })  
operation (1, 3, (a , b) => { console.log(a - b) })
```





Callbacks

- Es una forma de ejecutar código de forma “asíncrona” ya que **una función va a llamar a otra.**
- Cuando pasamos un callback solo pasamos la definición de la función y no la ejecutamos en el parámetro, **así la función contenedora elige cuándo ejecutar el callback.**

```
function operation (num1, num2, callback) {  
  return setTimeout (() => callback(num1, num2), 2000)  
}
```

```
operation (1, 3, (a , b) => { console.log(a + b) })  
operation (1, 3, (a , b) => { console.log(a * b) })  
operation (1, 3, (a , b) => { console.log(a - b) })
```




Ejemplo de callback

```
/* Tenemos 2 funciones que devuelven un valor */  
function soyCien() { return 100; }  
function SoyDoscientos() { return 200; }  
  
/* Esta función recibe como parametro 2 funciones y las ejecuta */  
function sumaDosFunciones(functionOne, functionTwo) {  
    const suma = functionOne() + functionTwo();  
    return suma; // retornando un nuevo valor, en este caso su suma  
}  
  
/* Invocamos a sumaDosFunciones y le pasamos 2 funciones como parámetros */  
console.log(sumaDosFunciones(soyCien, SoyDoscientos));  
// Resultado → 300
```



Ejemplo de setTimeout



```
setTimeout(function() {  
  console.log("He ejecutado la función");  
}, 2000);
```

Le decimos a `setTimeout()` que ejecute la función callback que le hemos pasado por primera parámetro, cuando transcurran 2000 milisegundos (es decir, 2 segundos, que le indicamos como segundo parámetro)



Ventajas y Desventajas

Ventajas

- Son fáciles de usar.
- Pueden solucionar problemas de flujo de una aplicación.
- Ayudan a manejar excepciones.
- Son útiles cuando quieres hacer consultas a una BD o servicio web.

Desventajas

- En ocasiones el concepto es confuso.
- Si se usa demasiado se puede caer en algo llamado “callback hell”
- El uso excesivo puede afectar el performance (rendimiento).
- Para programadores juniors no es fácil de comprender y entender la ejecución de los callbacks.



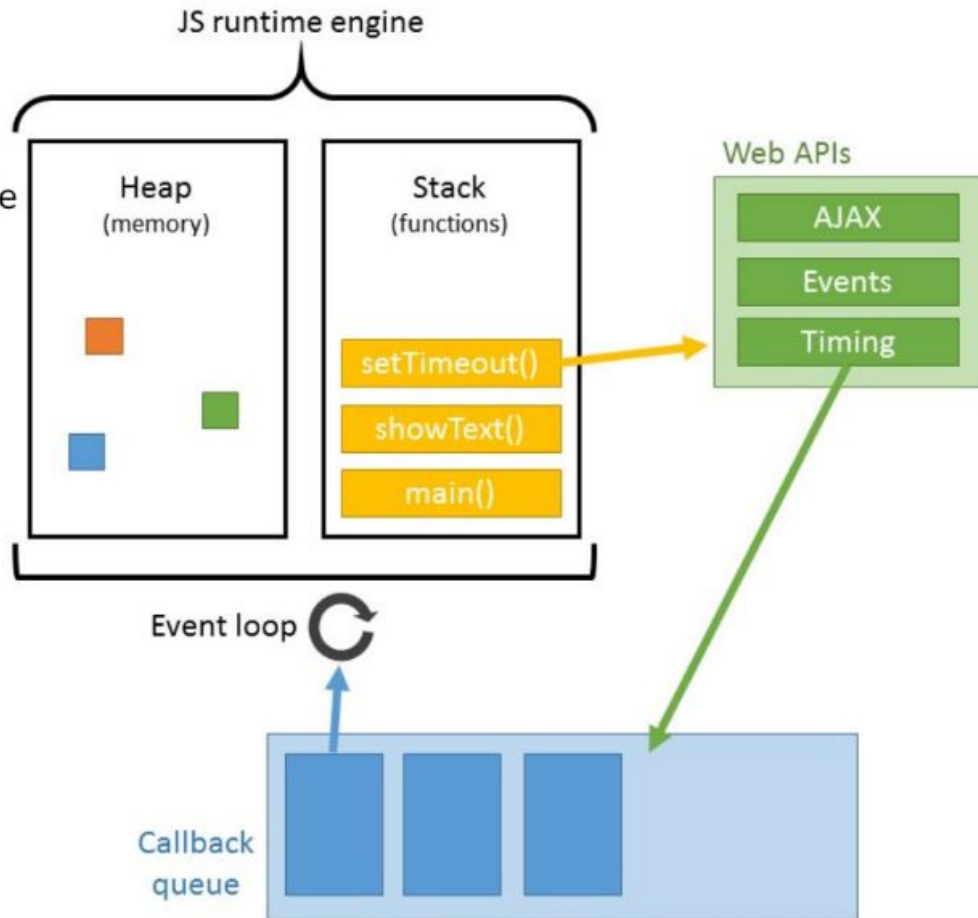
CALLBACK HELL

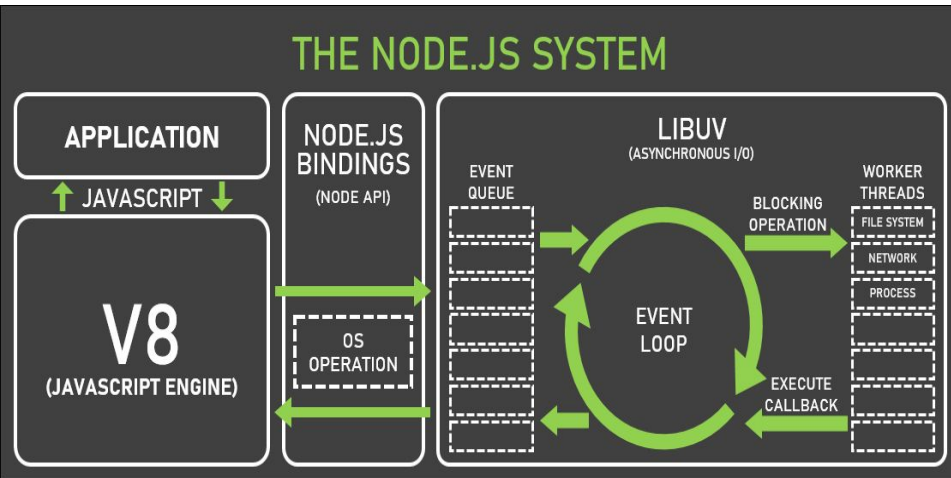
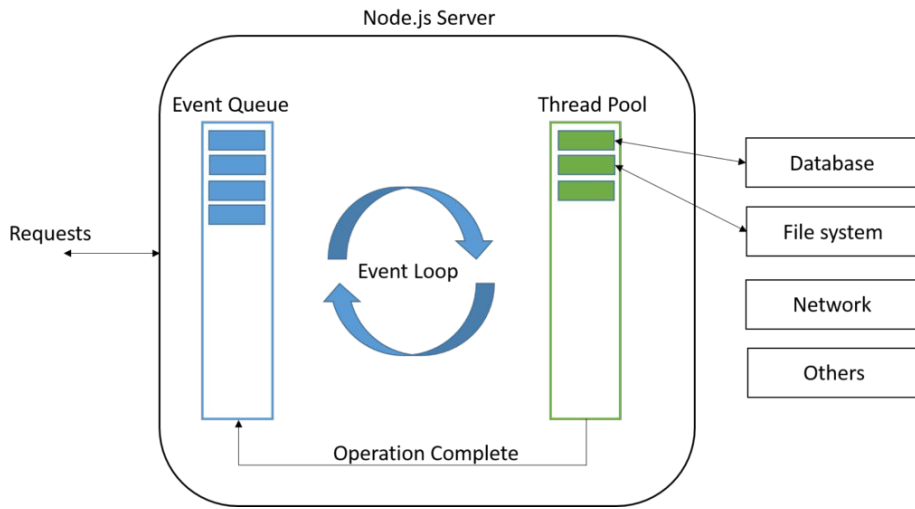
```
console.log("Pyramid of dooooooooooom! 😱 😓")

step1(function (value1) {
  step2(function (value2) {
    step3(function (value3) {
      step4(function (value4) {
        step5(function (value5) {
          step6(function (value6) {
            step7(function (value7) {
              //Do something with value 4
              console.log(value7)
            });
          });
        });
      });
    });
  });
});
```



Javascript se considera un lenguaje single thread, asíncrono y no bloqueante.



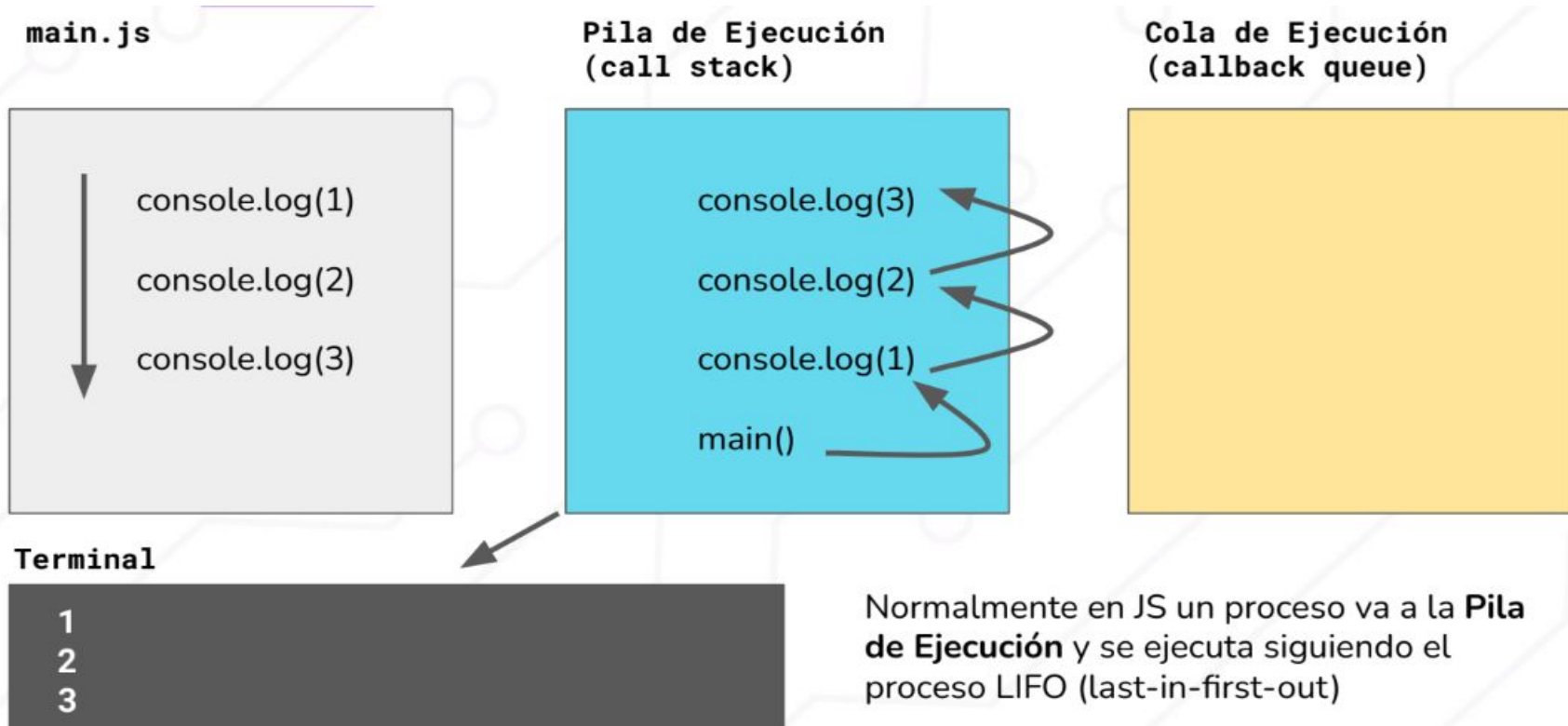


EVENT LOOP

- Javascript posee un modelo de concurrencias basado en un **“loop de eventos”**.
- Es el motor, quién está al pendiente de que elementos se pasan a la cola o a la pila de ejecución.
- Es el encargado de entender el orden de ejecución.

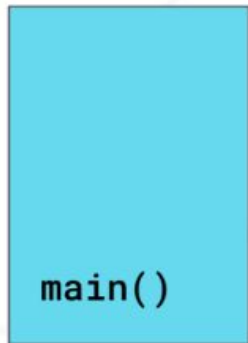


Flujo normal síncrono



Ejecución Call Stack

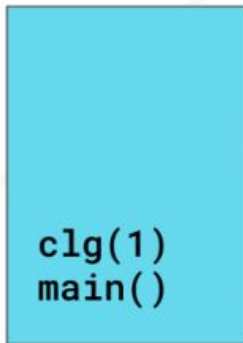
Pila de
Ejecución
(call stack)



Terminal



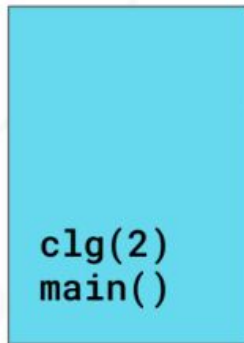
Pila de
Ejecución
(call stack)



Terminal



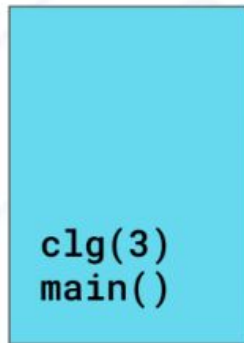
Pila de
Ejecución
(call stack)



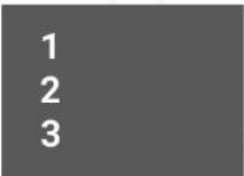
Terminal



Pila de
Ejecución
(call stack)



Terminal



Pila de
Ejecución
(call stack)



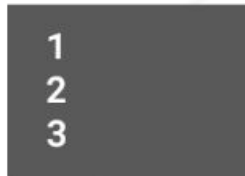
Terminal



Pila de
Ejecución
(call stack)

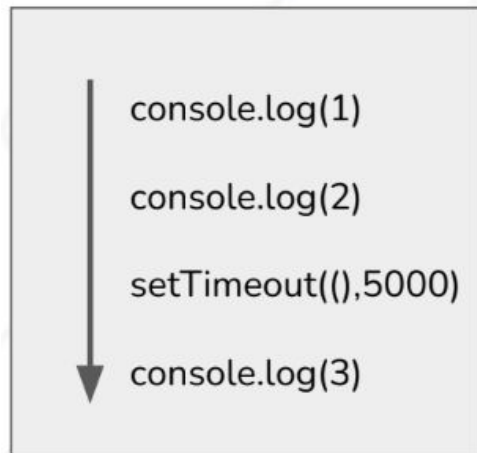


Terminal

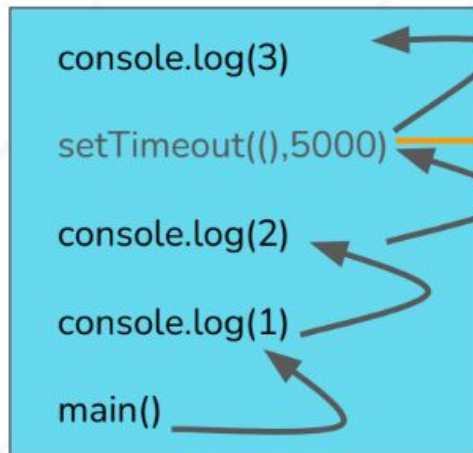


Flujo asíncrono

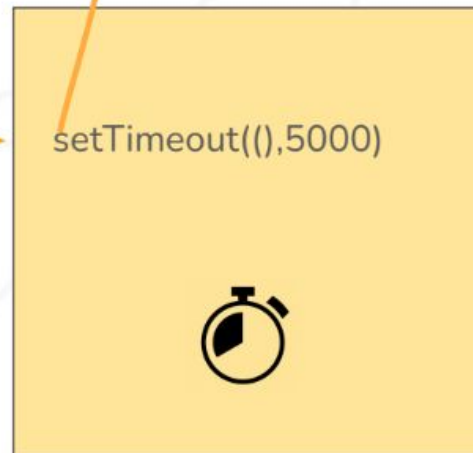
main.js



Pila de Ejecución
(call stack)



Cola de Ejecución
(callback queue)



Terminal



Los procesos considerados asíncronos se van a la cola de ejecución, terminan su ejecución y tienen que esperar a que la pila esta vacia para poder regresar con el resultado y continuar su ejecución.




Ejemplo #1



```
console.log("---Todo en Pila de Ejecución---");  
console.log(1);  
console.log(2);  
console.log(3);
```



Ejemplo #2



```
console.log("---El 2 y 3 van a la Cola de Ejecución---");
console.log(1);
// SetTimeout Espera N segundos para ejecutar un CALLBACK.
// Recibe 2 parametros: setTimeout(callback, milisegundos)
setTimeout(()⇒{ //Simular Ir a Base de Datos con un callback;
    return console.log(2)
},3000);
setTimeout(()⇒{
    return console.log(3)
},2000);
console.log(4);
```



Ejemplo #3



```
console.log("---Simulación de Cuello de Botella---");  
console.log(1);  
setTimeout(()⇒{  
    return console.log(2);  
},2000);  
for (let index = 0; index < 9999999999; index++);  
console.log(3);
```



DEV.F

IMPORTANTE:

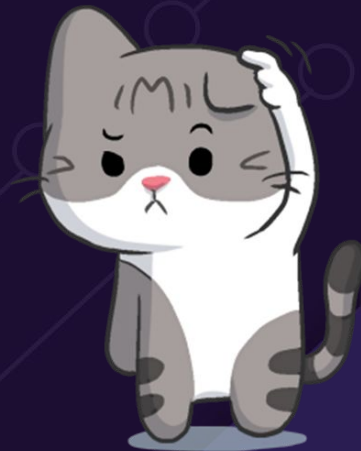
Muchos de los procesos que involucran pedir información de forma externa suelen ser asíncronos.

*Por ejemplo, **las consultas a bases de datos son por naturaleza asíncronas.***



Promesas

DEV.F
DESARROLLAMOS(PERSONAS);





{ api }

Promesas

Las promesas son una forma de garantizar una respuesta de un callback (que recordemos es asíncrono), **ya que estas no te puedan dar una respuesta al momento, si no en un futuro.**

Dicha respuesta de la promesa puede ser cumplida satisfactoriamente o rechazada por algún motivo **(surgió un error).**



JS

DEV.FL



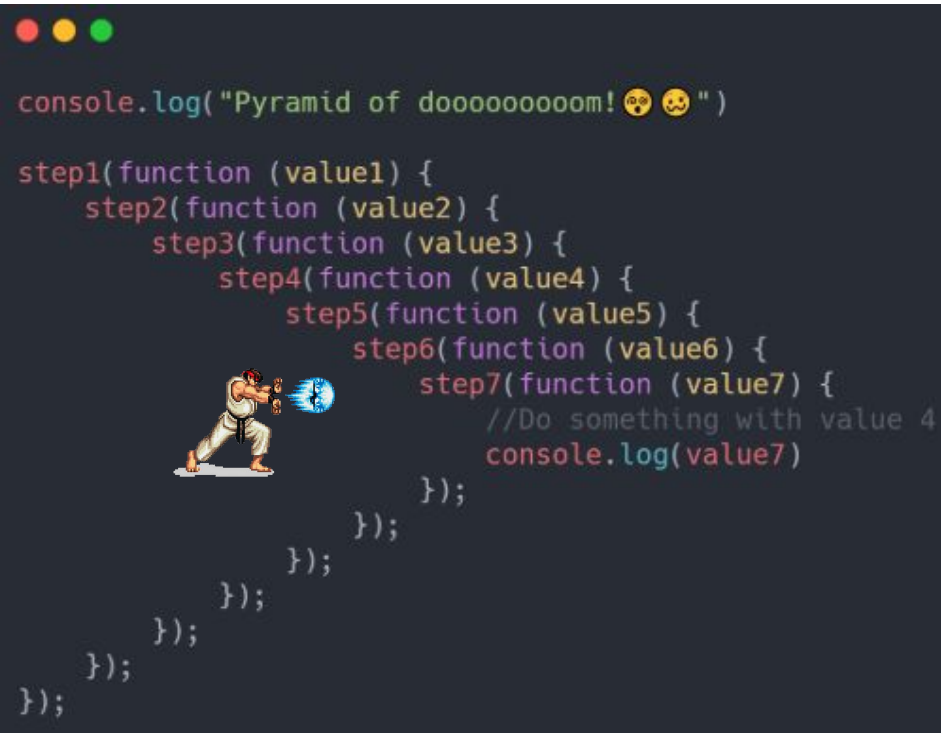
La Problemática

Uno de los desafíos que encontramos al comenzar a desarrollar código asíncrono es que normalmente ocupamos hacer varias operaciones en una sola instrucción, **que involucran usar el resultado de una invocación asíncrona previa para continuar con nuestro código y así sucesivamente.**



Callback Hell 🔥

El "Callback Hell" (Infierno de Callbacks) **es un problema en programación asíncrona** (especialmente en JavaScript) que se produce cuando se anidan demasiados callbacks uno dentro de otro. Esto resulta en un código difícil de leer, entender y mantener, que a menudo se asemeja a una pirámide (de ahí el término "Pirámide de la Perdición").

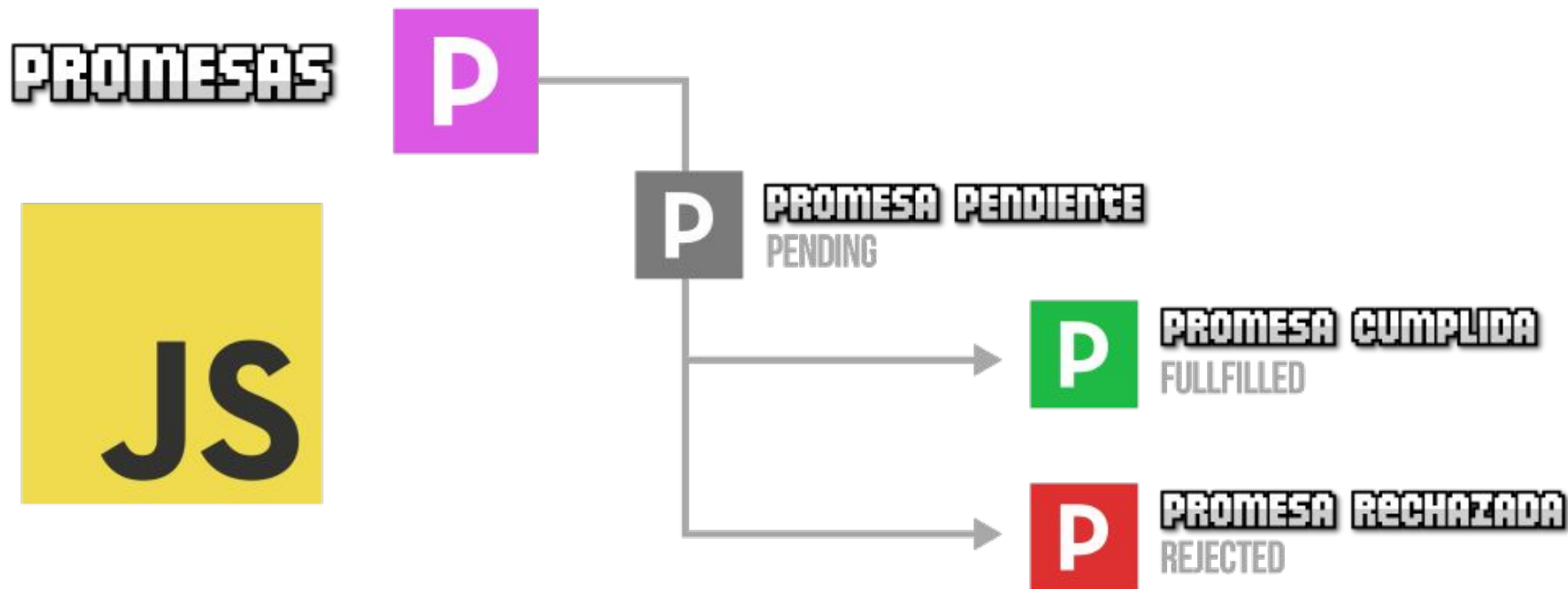


Estructura de una Promesa

DEV.F.
DESARROLLAMOS(PERSONAS);



Estados de una promesa



En las promesas se tienen siempre 3 estados:

- Pendiente: estado inicial de la promesa.
- Resuelta: Todo se ejecutó correctamente.
- Rechazada: Hubo un problema.



Sintaxis de una promesa

El primer paso para usar una promesa es declararla, utilizamos un constructor `Promise` y 2 parámetros(`resolve`, `reject`)

```
//Declaración de una Promesa
const Promesa = new Promise((resolve, reject) => {
  //Código Asíncrono a Resolver
  // Ejemplo: Llamadas a APIs, Bases de Datos,
  // Leer Archivos, etc.
  resolve(cosaQueDevuelvoSiSeEjecutaBien);
  reject(cosaQueDevuelvoSiSeEjecutaMal);
})
```

resolve: se ejecuta cuando el objetivo de la promesa se efectuó de manera correcta

reject: se ejecuta cuando el objetivo de la promesa ocasionó un error o no se llegó a cumplir.





```
//Declaración de una Promesa
const Promesa = new Promise((resolve, reject) => {
  //Código Asíncrono a Resolver
  // Ejemplo: Llamadas a APIs, Bases de Datos,
  // Leer Archivos, etc.
  resolve(cosaQueDevuelvoSiSeEjecutaBien);
  reject(cosaQueDevuelvoSiSeEjecutaMal);
})

//Ejecución de la promesa
Promesa
  .then( (resultado) => {
    //logica con el resultado
  }).catch (error) => {
    //manejo el error, ejemplo:
    console.log(error);
  }
```

Ejecutando una promesa

Podemos ejecutar una promesa por medio de los **métodos .then & .catch**

Funciona muy similar a la lógica de una estructura de control (if - else)

.then: es el código que se ejecuta cuando se resuelve la promesa de forma satisfactoria (fulfilled).

.catch: si la promesa no se resuelve (rejected) ejecutará esta parte del código, devolviendo comúnmente un error.



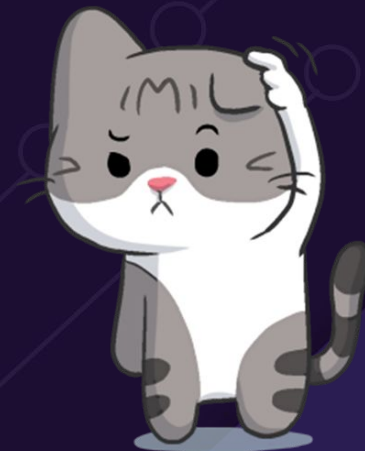
Ejemplo

```
const promise = new Promise((resolve, reject) => {  
  const number = Math.floor(Math.random() * 10);  
  
  setTimeout(  
    () => number > 5  
      ? resolve(number)  
      : reject(new Error('Menor a 5')),  
    1000  
  );  
});  
  
promise  
  .then(number => console.log(number))  
  .catch(error => console.error(error));
```

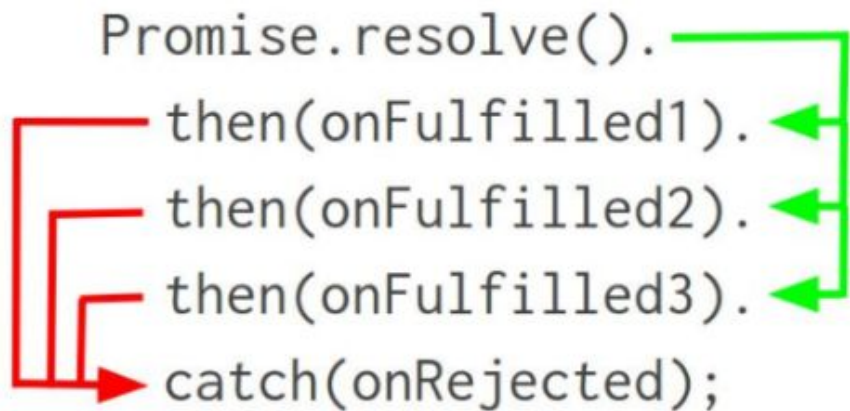


Encadenamiento de Promesas

DEV.FX
DESARROLLAMOS(PERSONAS);



```
Promise.resolve().  
  then(onFulfilled1).  
  then(onFulfilled2).  
  then(onFulfilled3).  
  catch(onRejected);
```



Encadenamiento de promesas

Una necesidad común es el ejecutar dos o más operaciones asíncronas seguidas, donde **cada operación posterior se inicia cuando la operación previa tiene éxito, con el resultado del paso previo.**

Logramos esto creando una cadena de objetos promises.



Ejemplo



```
hazAlgo().then(function(resultado) {  
    return hazAlgoMas(resultado);  
})  
.then(function(nuevoResultado) {  
    return hazLaTerceraCosa(nuevoResultado);  
})  
.then(function(resultadoFinal) {  
    console.log('Obtenido el resultado final: ' + resultadoFinal);  
})  
.catch(falloCallback);
```



Resumen de promesas

Es usado para interacciones asíncronas.

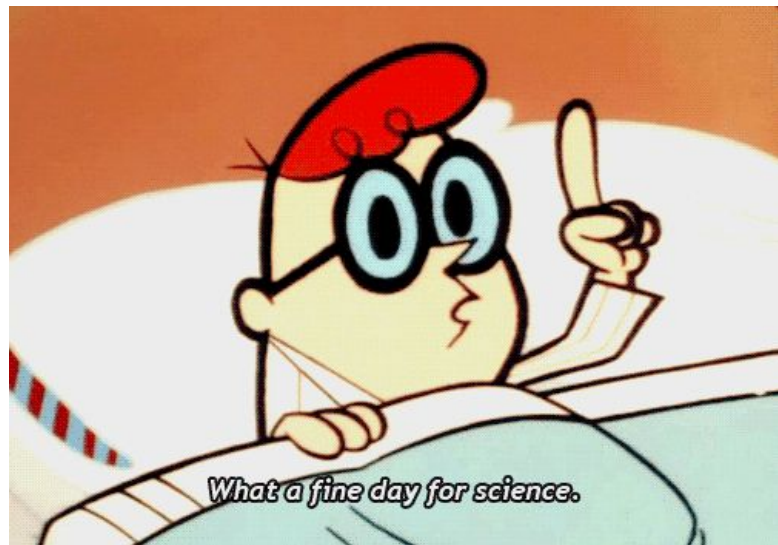
Se compone de dos aspectos:

- **resolve:** se ejecuta cuando el objetivo de la promesa se efectuó de manera correcta.
- **reject:** se ejecuta cuando el objetivo de la promesa ocasionó un error o no se llegó a cumplir.

Se utiliza la palabra reservada Promise.

En las promesas se tienen siempre 3 estados:

- Pendiente: estado inicial de la promesa.
- Resuelta: Todo se ejecutó correctamente.
- Rechazada: Hubo un problema.



I'm fulfilling this request

something gone wrong

```
var x = new Promise((resolve, reject) => {  
  if(success) resolve('data ..')  
  if(err) reject('error')  
})
```

```
x.then((data)=>{...})  
  .catch((err)=>{...})
```

```
async function add(x, y){  
  let promise = sum(x, y)
```

```
  let result = await promise  
  return result  
}
```

Works only inside
async functions

```
function sum(x, y) {  
  let promise = new Promise((resolve, reject) => {  
    setTimeout(() => resolve(x + y), 100)  
  })  
  
  return promise  
}
```

It can be an API call!

```
add(5, 10).then(z => console.log(z)) // 15
```

Wrapped in a resolved
promise automatically

```
async function add(x, y){
```

```
  return x + y
```

same under the hood

```
  return Promise.resolve(x + y)
```

```
add(5, 10)  
  .then(z => console.log(z)) // 15
```



PROMISE

ASYNC

AWAIT

✨ Async / Await + Try-Catch

`async` y `await` son azúcar sintáctico sobre las promesas, pero hacen el código mucho más limpio.

```
async function leerArchivo() {  
  try {  
    const contenido = await new Promise((resolve) => {  
      setTimeout(() => resolve("📄 Contenido del archivo"), 1000);  
    });  
    console.log(contenido);  
  } catch (error) {  
    console.error("Error leyendo el archivo", error);  
  }  
}
```

`leerArchivo();`





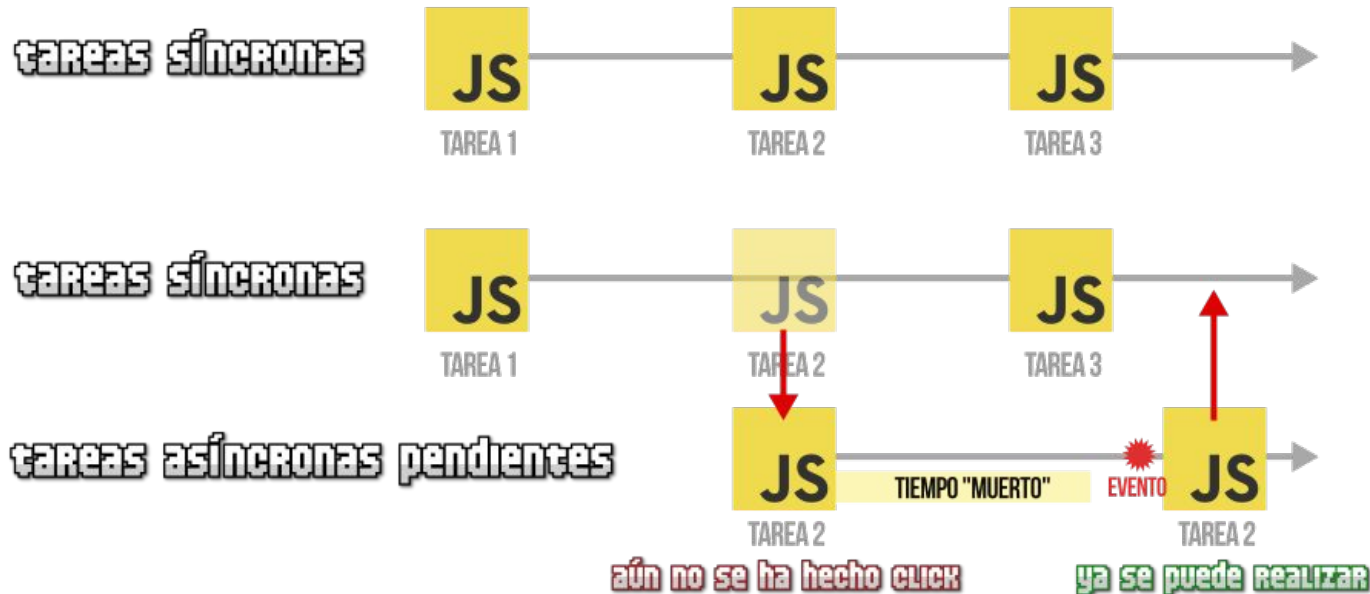
Ejemplo Asíncrono Completo

```
async function obtenerDatosDeUsuario(id) {  
  try {  
    console.log("Buscando datos del usuario...");  
    const usuario = await fetch(`https://api.example.com/users/${id}`);  
    const data = await usuario.json();  
    console.log("Usuario encontrado:", data);  
  } catch (error) {  
    console.error("Hubo un error al obtener el usuario:", error);  
  }  
}  
  
obtenerDatosDeUsuario(1);
```

(Nota: `fetch` está disponible en el navegador o con `node-fetch` en Node.js)



Síncrono y Asíncrono (Async)



Síncrono: toda operación o tarea se ejecuta de forma secuencial y por lo tanto debemos esperar para procesar el resultado.

Asíncrono: la finalización de la operación es notificada al programa principal. El procesamiento de la respuesta se hará en algún momento futuro.





Comparación: Callbacks vs Promesas vs Async/Await

Característica	Callbacks	Promesas	Async/Await
Legibilidad	✗ Difícil	● Mejor	✓ Excelente
Manejo de errores	✗ Complicado	● <code>.catch()</code>	✓ <code>try/catch</code>
Composición	✗ Callback hell	● <code>.then()</code>	✓ Secuencial





Conclusión rápida

- **Callbacks:** sirven, pero se complican si encadenas muchos.
- **Promesas:** más limpias, mejores para componer.
- **Async/Await:** estilo más legible, fácil de debuggear, ideal para flujos secuenciales.

