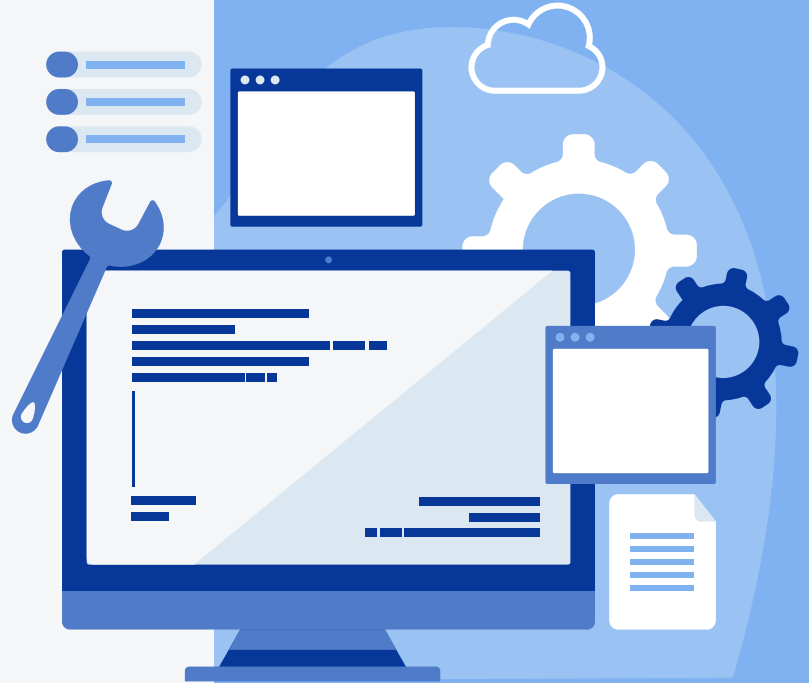



Programación Asíncrona

Cordero Hernández, Marco R.



En sesiones pasadas...

- Introducción a JS
- Funciones
- JSON
- Arreglos

A yellow square containing the letters 'JS' in a bold, black, sans-serif font, representing the JavaScript logo.

JS

CONTENIDOS

01

Conceptos

02

Low-level

03

Callbacks

04

Promesas

05

Async/Await



01

Conceptos

Programación Asíncrona

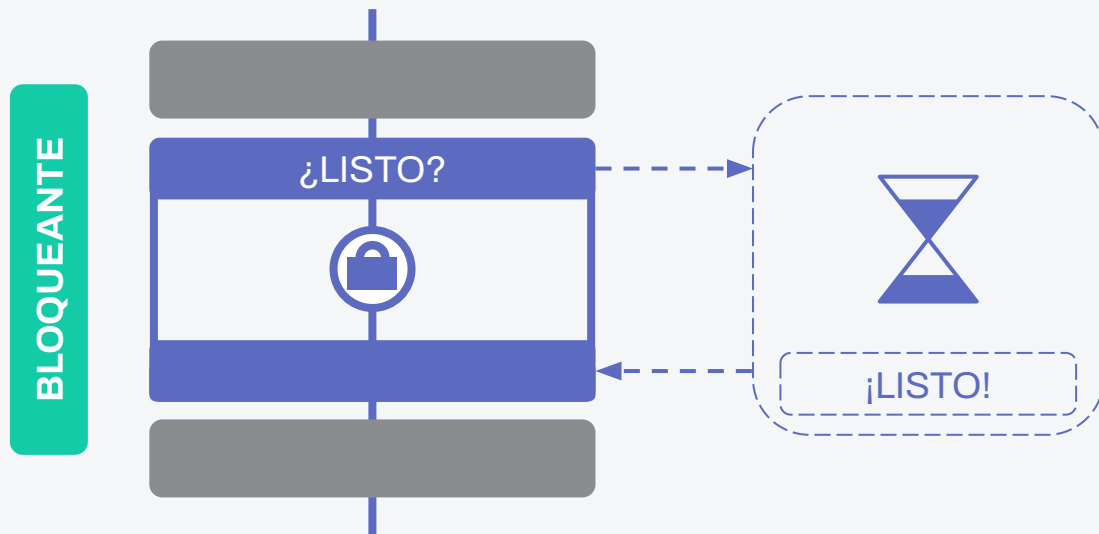
JavaScript fue diseñado para ejecutarse en los navegadores, realizar peticiones sobre la red y procesar la interacción del usuario, todo esto manteniendo la interfaz fluida.

JavaScript *NO ES* multithreaded, por lo que en un solo **thread** debe procesar todas estas actividades de manera asíncrona.



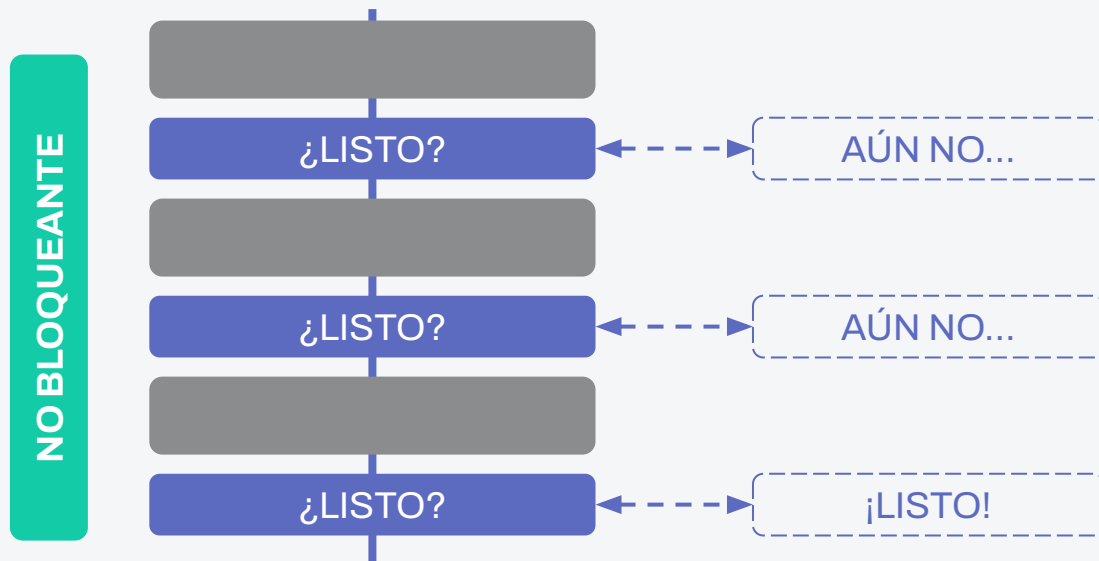
Programación Asíncrona

Entrada/Salida bloqueante: No devuelve el control de la aplicación hasta que se completa. El *thread* queda bloqueado en estado de espera.



Programación Asíncrona

Entrada/Salida No bloqueante: Sí devuelve el control. Una vez completada, regresa los datos solicitados. Si la operación no pudo realizarse, se genera un error.



Síncrono VS Asíncrono

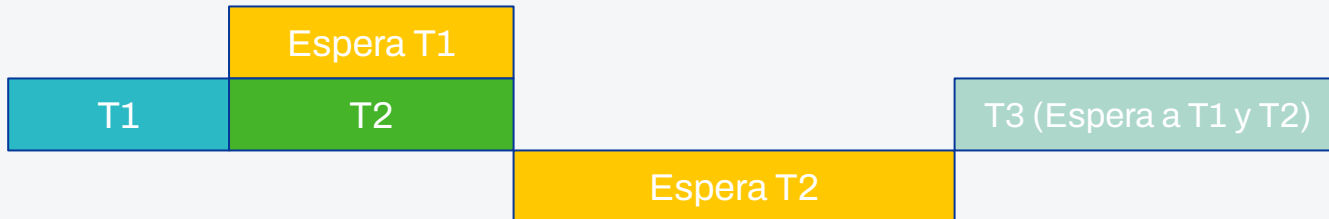
Síncrono: Ejecución secuencial

Asíncrono: Al finalizar la operación, se notifica al programa principal

Síncrono



Asíncrono



¿Cuál es la salida del siguiente código?



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

setTimeout(function() {console.log("A");}, 7000);

setTimeout(function() {console.log("B");}, 0);

setTimeout(function() {console.log("C");}, 2000);

setTimeout(function() {console.log("D");}, 1000);

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

Primero haz una inferencia del resultado, luego ejecuta el código ¿Coincide tu resultado?



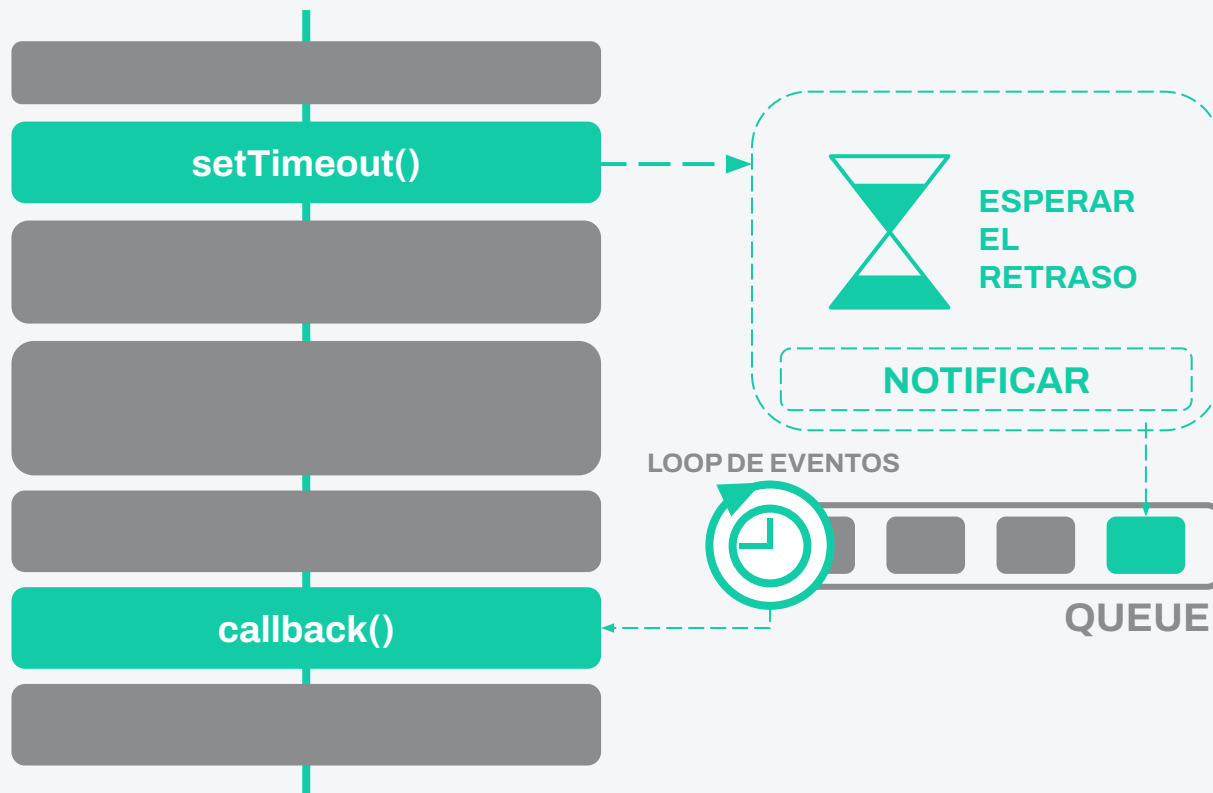
02

Low-level

El modelo de JavaScript

- JS emplea un **modelo asíncrono *no bloqueante*** con un loop de eventos de *único thread* para las interfaces de entrada y salida
- Al finalizar una operación de forma asíncrona, se “enfila” la notificación en espera de ser procesados por el entorno de JS
- El **loop de eventos** se encarga de procesar mensaje a mensaje cada uno esperando su turno
- Una vez procesado el mensaje, se ejecuta su función asociada (**callback**) como respuesta a la operación de entrada/salida

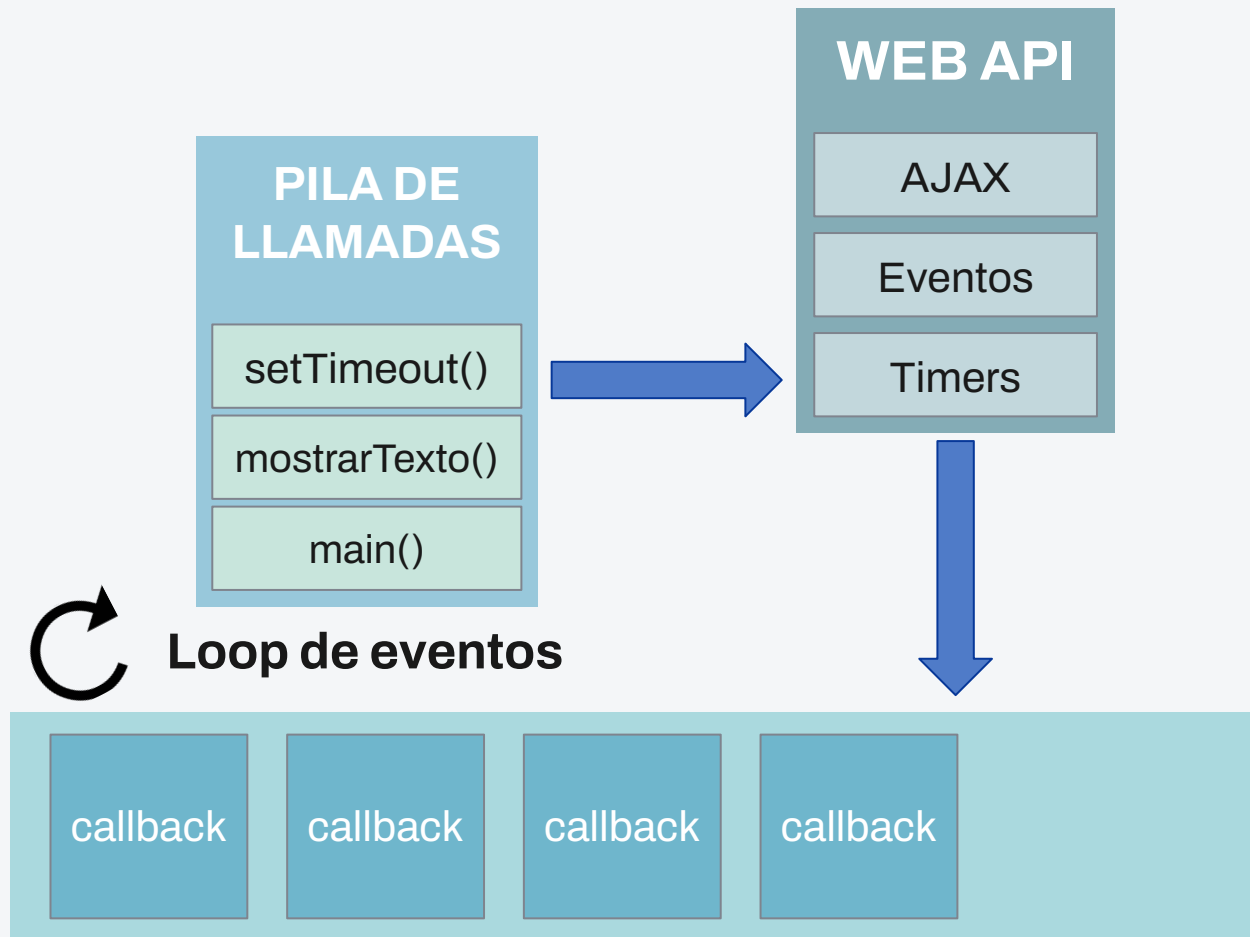
El modelo de JavaScript





Entendiendo el modelo

```
console.log('Iniciando main() ...');  
  
function mostrarTexto(input) {  
    setTimeout(() => console.log(input), 3000);  
}  
  
mostrarTexto("Hola mundo");  
console.log("...Terminando main()");
```





03

Callbacks

Callbacks


Los **callbacks** son las *funciones que se pueden ejecutar una vez que ocurra el evento esperado*.

- **Ejemplo:** Al dar click en algún botón, es posible ejecutar la función **callback** que determinará qué acciones se ejecutarán

Callback Hell/Pyramid of Doom

- Si se requiere que se completen varias fases y se usa una estrategia de **callbacks** es muy probable que se introduzcan errores al codificar debido al anidamiento de **callbacks**

Callbacks



```
pan.pourWater(function() {  
  range.bringToBoil(function() {  
    range.lowerHeat(function() {  
      pan.addRice(function() {  
        setTimeout(function() {  
          range.turnOff();  
          serve();  
        }, 15 * 60 * 1000);  
      });  
    });  
  });  
});
```

pyramid of doom

mozilla



Callbacks

```
fase1(function(result) {  
    fase2(result, function(newResult) {  
        fase3(newResult, function(otherResult) {  
            fase4(otherResult, function(finalResult) {  
                console.log('Así hasta llegar al infierno... -> ' + finalResult);  
            }, failureCallback);  
        }, failureCallback);  
    }, failureCallback);  
}, failureCallback);
```

Ejercicio

Mostrar un mensaje de Hola**N** donde **N** corresponde a lo siguiente:

- Mostrar el mensaje después de **N** = 1 segundo
- Mostrar el mensaje después de **N** = 2 segundos
- ...
- Mostrar el mensaje después de **N** = 5 segundos

En “paralelo” mostrar la palabra Mundo**N** cada segundo hasta mostrar 4 veces la palabra mundo



04

Promesas



Promises

Las **promesas** permiten organizar de mejor manera el código cuando hay dependencias asíncronas.

Una promesa (**promise**) es un **objeto de JS** que representan un valor que *podría* estar disponible *ahora, en un futuro o nunca*.

```
// Instanciación  
let p = new Promise(/* Ejecutor */ function(resolve, reject) {});
```



Promises

Una promesa **o** se cumple (**resolve(value)**) **o** se rechaza (**reject(error)**)

```
Promise.then(cbExito, cbError);
```

Una vez resuelta o rechazada la promesa, el resto del código se ignora.

Para crear una promesa “resuelta” se usa

```
Promise.resolve();
```

Para esperar a que todas las promesas se cumplan, se usa

```
Promise.all([pr1, pr2, pr3]).then();
```



Ejemplo

```
let promise = new Promise(function (resolve, reject) {  
  // Después de 1 segundo, indicar estatus de la promesa  
  setTimeout(() => {  
    console.log('Procesando la promesa...');  
    if (Math.random() < 0.5) {  
      resolve("Promesa correcta");  
    } else {  
      reject(new Error("Promesa errónea"))  
    }  
  }, 1000);  
});
```



Ejemplo

```
promise.then(function success(result) {  
    console.log(result);  
}, function failure(error) {  
    console.log('Promesa completada con error ->', error);  
});
```




Promesas como retorno de funciones

```
function loadScript(src) {  
    return new Promise(function(resolve, reject) {  
        let script = document.createElement('script');  
        script.src = src;  
  
        script.onload = () => resolve(script);  
        script.onerror = () => reject(  
            new Error("Script load error: " + src));  
  
        document.head.append(script);  
    });  
}
```



Promesas como retorno de funciones

```
let promise = loadScript('https://algunaLibreria.js');

promise.then(
  script => alert(`${script.src} cargado exitosamente`),
  error => alert(`Error -> ${error.message}`)
);

promise.then(script => alert('Algo extra por resolver...'));
```

Resolución de Callback Hell con promesas



```
// Formato con promesas
fase1().then(function(result) {
    return fase2(result);
}).then(function(newResult) {
    return fase3(newResult);
}).then(function(otherResult) {
    return fase4(otherResult);
}).then(function(finalResult) {
    console.log(`Resultado final promises ${finalResult}`);
}).catch(failureCallback);
```

Resolución de Callback Hell con promesas



```
// Otra sintaxis  
fase1().then(result => fase2(result))  
    .then(newResult => fase3(newResult))  
    .then(otherResult => fase4(otherResult))  
    .then(finalResult => console.log(`Resultado final  
promises ${finalResult}`))  
    .catch(failureCallback);
```



Ejercicio

Existe la tarea 1 que se encarga de mostrar el mensaje Hola 1, Hola 2, ..., Hola 10 (un mensaje por cada segundo/1000ms)

Existe la tarea 2 que se encarga de mostrar el mensaje Mundo 1, Mundo 2, ..., Mundo 5 (por cada dos segundos/2000ms)

Existe la tarea 3 que imprime el mensaje ***fin*** cuando terminan las dos tareas anteriores

1. Resuelve el ejercicio usando callbacks
2. Resuelve el ejercicio usando promises



05

Async/Await

Async/Await

Usar la palabra **async** antes de una función significa que regresará una promesa implícita (esta será añadida automáticamente por el motor de **JS** que se esté utilizando)

await espera a que se cumpla la promesa desde dentro de la función. **Solo puede ser usada dentro de funciones definidas como async.**



Async/Await

```
// Función sin async/await
function loadJson(url) {
    return fetch(url).then(response => {
        if (response.status == 200) {
            return response.json();
        } else {
            throw new Error(response.status);
        }
    })
}

loadJson('user.json').catch(alert);
```




Async/Await

```
// Función con async/await  
async function loadJson(url) {  
    let response = await fetch(url);  
  
    if (response.status == 200) {  
        let json = await response.json();  
        return json;  
    }  
  
    throw new Error(response.status);  
}  
  
loadJson('user.json').catch(alert);
```



Solución del ejercicio anterior con Async/Await

```
function delay(ms) {  
  return new Promise(function (resolve) {  
    setTimeout(() => resolve(), ms);  
  });  
}
```

```
async function aaPrintHola() {  
  for (let i = 0; i < 10; i++) {  
    await delay(1000);  
    console.log(`Hola${i+1}`);  
  }  
  
  return Promise.resolve();  
}
```

```
async function aaPrintMundo() {  
  for (let i = 0; i < 5; i++) {  
    await delay(2000);  
    console.log(`Mundo${i+1}`);  
  }  
  
  return Promise.resolve();  
}
```

```
(async () => {  
  await Promise.all([aaPrintHola(), aaPrintMundo()]);  
  console.log("FIN");  
})(); // IIFE
```