sustantiva

CON•SENTIDO

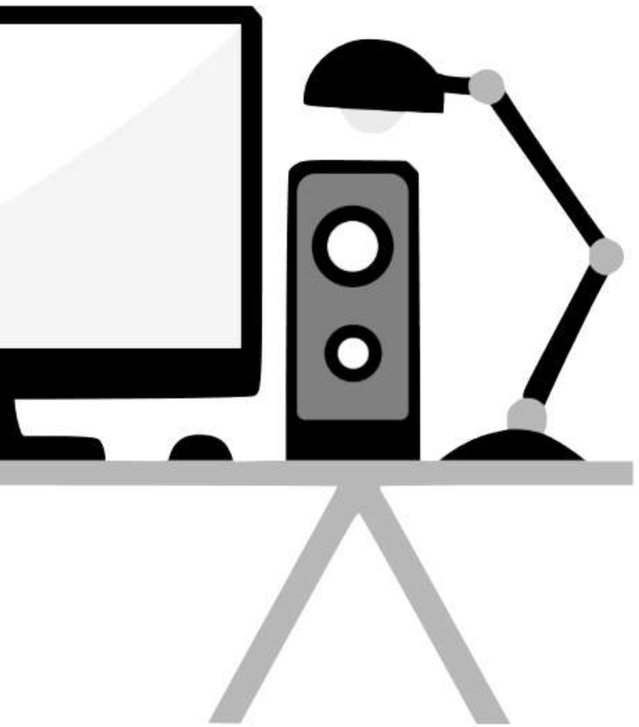
JAVASCRIPT ASÍNCRONO

Lección 04

Competencia General

Utilizar elementos de programación asíncrona para resolver un problema simple distinguiendo los diversos mecanismos para su implementación acorde al lenguaje JavaScript.

Contenidos



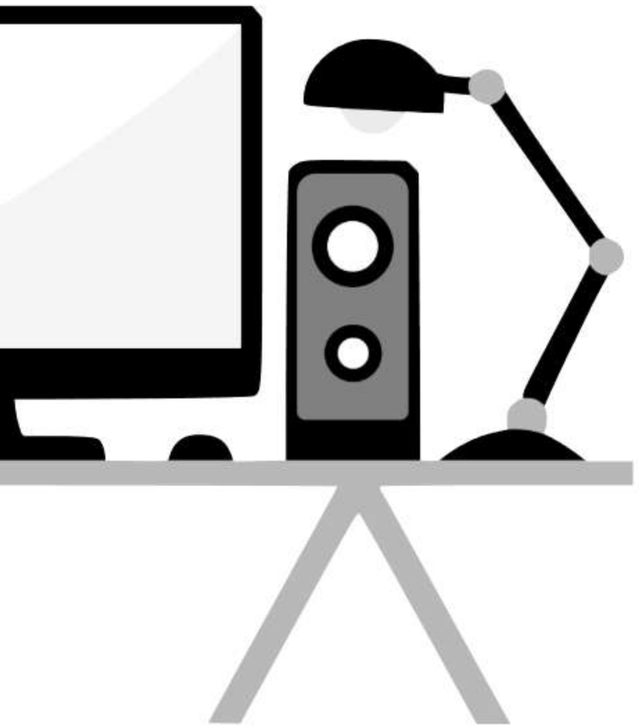
JavaScript asíncrono

1. Qué es la programación asíncrona
2. Que es un Thread
3. Código asíncrono vs código bloqueante

Callbacks asíncronos

1. Qué es una función asíncrona.
2. Qué es un callback
3. Creando una función callback
4. Ejemplos con SetTimeout
5. Paso de parámetros en funciones callback
6. Invocando una llamada callback

Contenidos



Promises

1. Qué es una promesa
2. Ventajas de las promesas sobre los callback
3. Bloques .then() y .catch()
4. Construyendo una promesa
5. Resolviendo una promesa
6. Rechazando una promesa

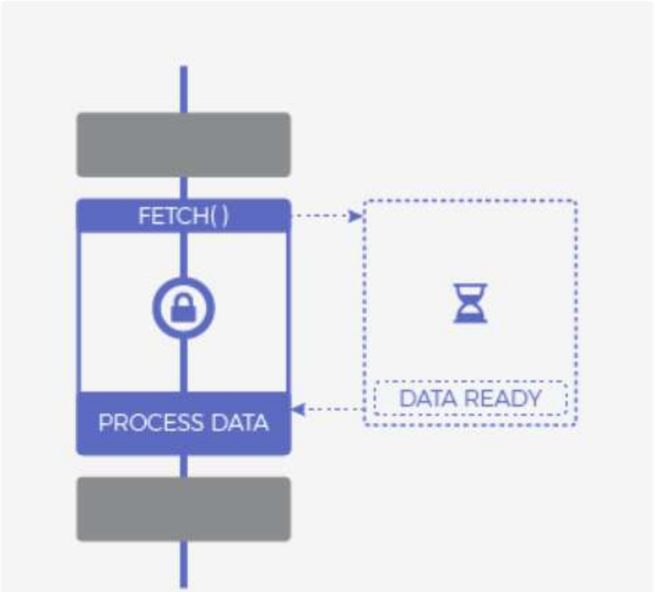
Async/Await

1. Creando funciones asíncronas con async/await
2. Capturando un error con catch
3. Ventajas y desventajas de async/await

¿Qué es la programación asíncrona?

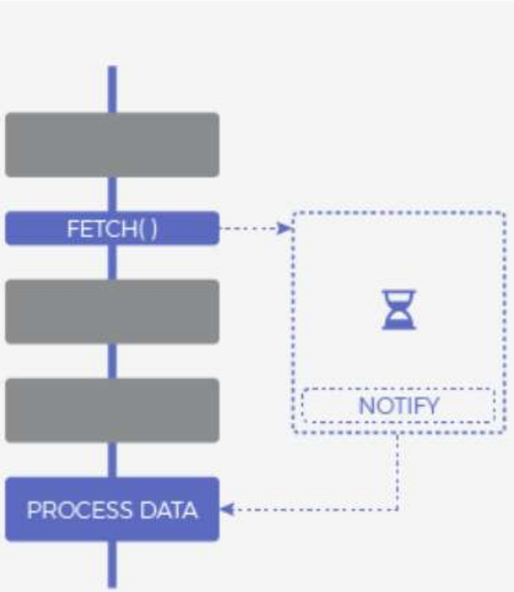
La programación asíncrona es la capacidad de “diferir” la ejecución de una función a la espera de que se complete una operación.

Tener un código asíncrono nos permite tener una mejor respuesta en las aplicaciones y reduce el tiempo de espera del cliente



SÍNCRONO

Ejecución secuencial. Retorna cuando la operación ha sido completada en su totalidad.



ASÍNCRONO

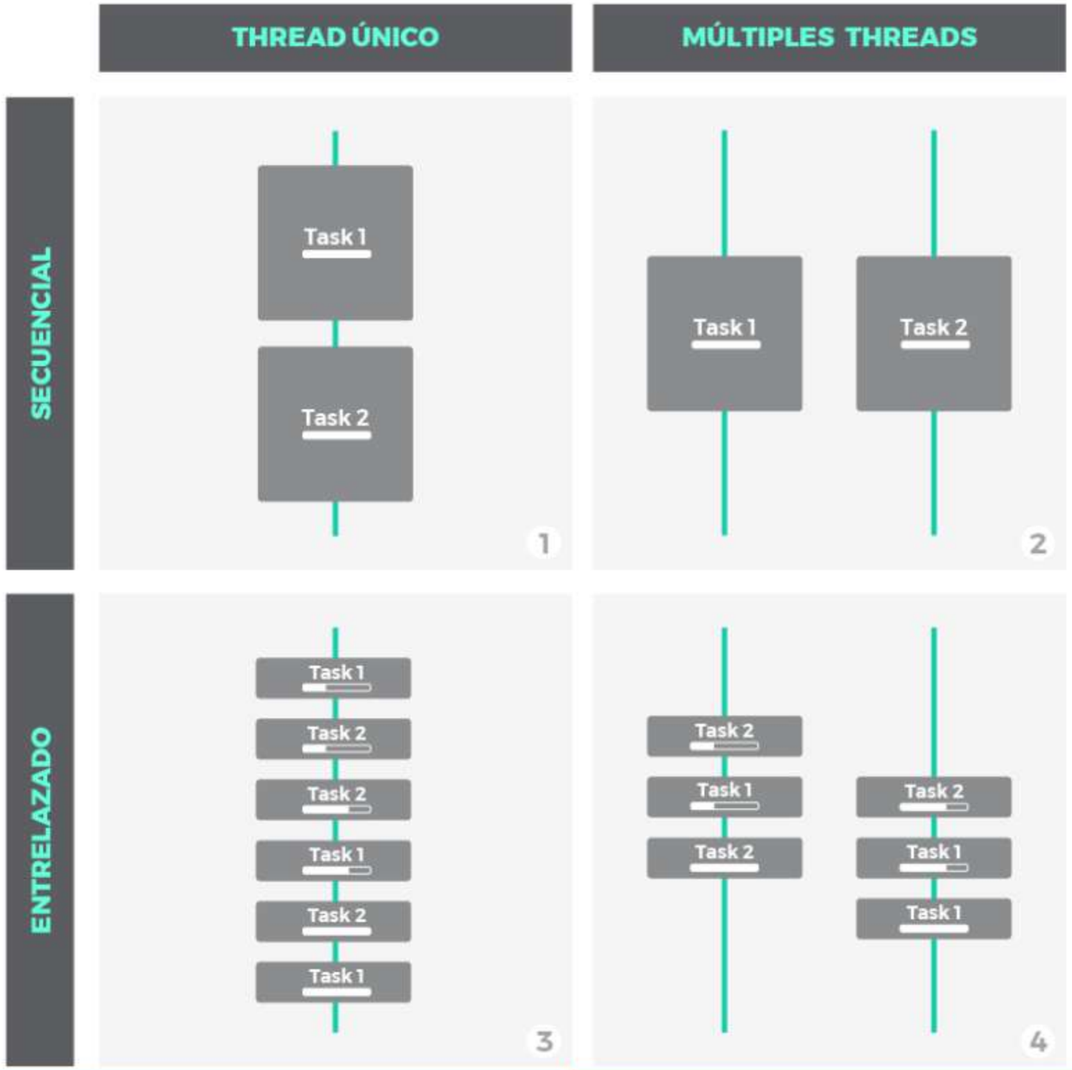
La finalización de la operación es notificada al programa principal. El procesado de la respuesta se hará en algún momento futuro.

Qué es un thread

Un thread es una característica que permite a una aplicación realizar múltiples tareas a la vez (concurrencia).

Un thread tiene control sobre otros threads siempre que pertenezcan al mismo proceso, pudiendo terminarlos si es preciso. También puede sincronizarse con otros thread esperando que alguno termine su ejecución o dormir hasta que el usuario lo vuelva a activar.

Nota: no confundir concurrencia con multitask (multitarea)

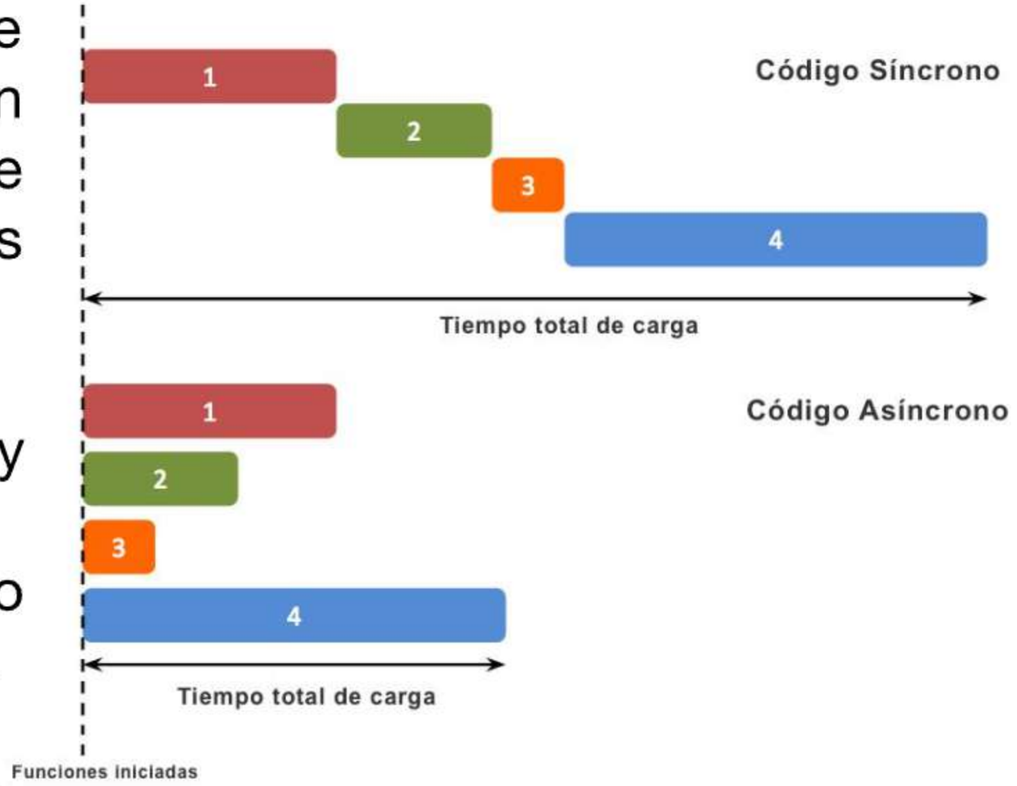


Código asíncrono vs código bloqueante

Las tareas que se realizan en un programa usualmente llevan un tiempo de carga asociado. Por ejemplo un acceso a un disco duro o base de datos. Durante ese procesamiento las tareas en proceso tienen dos opciones:

- Quedarse “suspendidas” hasta que termine la tarea y retorne un resultado
- Liberar el flujo de ejecución, de modo que el proceso que inició la acción puede atender otras necesidades.

Siempre debemos decantarnos por métodos asíncronos, ya que son capaces de aprovechar todas las características del equipo.



Función asíncrona

En JavaScript una función asíncrona se crea como un objeto ***AsyncFunction,*** ésta hereda atributos y propiedades desde AsyncFunction.prototype.

Cada operación se carga al call stack y cuando termine envia una notificación de que a terminado, en ese momento se procesa la respuesta.


```
function sayWord(w){
   console.log(w);
}

function sayGreeting(g){
    sayWord(g);
}

sayGreeting("Hello!");
```

```
function dos() {
  setTimeout(function () {
    console.log("Dos");
  }, 1000);
}

function uno() {
  setTimeout(function () {
    console.log("Uno");
  }, 0);
  dos();
  console.log("Tres");
}

uno();
```

Callback

Un callback es una función que se ejecutará después de que otra función haya terminado de ejecutarse.

JavaScript es un lenguaje orientado a eventos. Esto significa que en lugar de esperar una respuesta para avanzar, JavaScript seguirá ejecutándose mientras escucha otros eventos.

```
function doHomework(subject, callback) {
  alert(`Starting my ${subject} homework.`);
  callback();
}

doHomework('math', function() {
  alert('Finished my homework');
});
```

SetTimeout

El método setTimeout ( ) permite ejecutar un fragmento de código, una vez transcurrido un tiempo determinado

Para un mejor entendimiento puedes pensar en este método como un temporizador para ejecutar código.

El tiempo va en milisegundos

```
setTimeout(function(){
    console.log("Esto va segundo en 2
segundos después");
}, 2000);

console.log("Esto va primero");


/*
  Output:
    Esto va primero
    Esto va segundo en 2 segundos
después
*/
```

Creando una función callback

Supongamos que tenemos un sistema que registra usuarios, y después de registrarlo queremos enviarle un mensaje de bienvenida. Para no acoplar el sistema a una acción específica, usamos un **callback**:

```
function registrarUsuario(nombre, callback) {
  console.log("Registrando al usuario: " + nombre);
 // Simulamos un proceso asincrónico (por ejemplo, guardar en una base de datos)
  setTimeout(function() {
    console.log("Usuario registrado exitosamente.");
    // Ejecutamos el callback luego del proceso
    callback(nombre);
  }, 2000);
}
// Función que se usará como callback
function enviarBienvenida(nombre) {
  console.log("¡Bienvenido/a, " + nombre + "! Gracias por registrarte.");
}
// Llamamos a la función principal pasando el callback como argumento
registrarUsuario("Pedro", enviarBienvenida);
```

Creando una función callback

Cómo funciona

1. Se llama a **registrarUsuario** con el nombre del usuario y la función **enviarBienvenida** como **callback**.
2. Dentro de **registrarUsuario**, se simula un proceso con setTimeout.
3. Una vez terminado, se llama a la función callback(nombre), que es **enviarBienvenida**.

```
function registrarUsuario(nombre, callback) {
  console.log("Registrando al usuario: " + nombre);
 // Simulamos un proceso asincrónico (por ejemplo, guardar en una base de datos)
  setTimeout(function() {
    console.log("Usuario registrado exitosamente.");
    // Ejecutamos el callback luego del proceso
    callback(nombre);
  }, 2000);
}
// Función que se usará como callback
function enviarBienvenida(nombre) {
  console.log("¡Bienvenido/a, " + nombre + "! Gracias por registrarte.");
}
// Llamamos a la función principal pasando el callback como argumento
registrarUsuario("Pedro", enviarBienvenida);
```

Promesas

Una promesa representa la terminación o fracaso de una operación asíncrona. Esta operación para dar respuesta puede tener un valor o estar en proceso de obtención. Puede que una promesa nunca obtenga resultado.

Una promesa pasa por 3 estados:

- Pendiente
- Cumplida
- Rechazada

```
let miPrimeraPromise = new Promise((resolve, reject) => {
  // Llamamos a resolve(...) cuando lo que estabamos haciendo finali
za con éxito, y reject(...) cuando falla.
  // En este ejemplo, usamos setTimeout(...) para simular código así
ncrono.
  setTimeout(function(){
    resolve("¡Éxito!"); // ¡Todo salió bien!
  }, 250);
});

miPrimeraPromise.then((successMessage) => {
  // succesMessage es lo que sea que pasamos en la función resolve(.
..) de arriba.
  console.log("¡Sí! " + successMessage);
});
```

sustantiva
CON•SENTIDO

Ventajas de usar promesas sobre callbacks

1. Las promesas no requieren un callback para resolver la asincronía.
2. El envío de respuesta en las promesas se realiza a través de resolve y reject.
3. Los errores en las promesas se controlan por medio de catch.
4. La promesa se vuelve un código más limpio, fácil de entender y de mantener.

Bloques then y catch

Para manejar el resultado de una promesa, una vez que ha sido resuelta o rechazada, se utilizan los métodos .then() y .catch().

- Promise.then(success, error) permite definir una función para el caso exitoso (success) y otra para el caso de error (error) en una sola llamada.

- Promise.then(success).catch(error) separa el manejo del éxito y el error en dos bloques. Esta forma es más flexible y recomendada, ya que captura errores tanto de la promesa original como de cualquier error dentro del bloque .then().

```
Promise.resolve("Hi!").then(success, error);
// Logs 'Resolved: Hi!'

Promise.resolve("Hi!").then(success).catch(er
ror);
// Logs 'Resolved: Hi!'

function success(value) {
console.log("Resolved: ", value);
}

function error(err) {
console.log("Error: ", err);
}
```

Trabajando con promesas

Vamos a crear una promesa que se resuelva o se rechace según un número generado de forma pseudoaleatoria.
Si el número es mayor o igual a 0,5, la promesa se resolverá; de lo contrario, será rechazada.

```
let promesa = new Promise((resolver, rechazar) => {
  setTimeout(() => {
    let numero = Math.random();
    if (numero >= 0.5) {
      resolver("Éxito");
    } else {
      rechazar("Error");
    }
  }, 2000);
});

promesa
  .then((data) => console.log(data))
  .catch((data) => console.log(data));
```

Async/Await

Función *async*

Una función declarada con la palabra clave `async` siempre devuelve una **Promesa**. Esto significa que su resultado podrá ser manejado de forma asíncrona, permitiendo integrar lógica que dependa del tiempo de respuesta de otras operaciones como llamadas a APIs, lectura de archivos o temporizadores.

Palabra clave *`await`*

`await` solo puede usarse dentro de funciones `async` y **pausa la ejecución** de esa función hasta que la Promesa que está esperando sea resuelta o rechazada.
Esto permite escribir código asíncrono de forma secuencial y más fácil de leer, sin necesidad de usar múltiples funciones `.then()` anidadas.

Async/Await

Vamos a transformar la función anterior a una función usando async / await

En lugar de utilizar los métodos .then() y .catch(), podemos manejar promesas de forma más legible usando async y await. Esto nos permite trabajar con código asíncrono con una sintaxis similar al código secuencial.

```
function promesa() {
  return new Promise((resolver, rechazar) => {
    setTimeout(() => {
      let numero = Math.random();
      if (numero >= 0.5) {
        resolver("Éxito");
      } else {
        rechazar("Error");
      }
    }, 2000);
  });
}

async function ejecutarPromesa() {
  try {
    const resultado = await promesa();
    console.log(resultado);
  } catch (error) {
    console.log(error);
  }
}

ejecutarPromesa();
```

Async/Await

```
let promesa = new Promise((resolver, rechazar) => {
  setTimeout(() => {
    let numero = Math.random();
    if (numero >= 0.5) {
      resolver("Éxito");
    } else {
      rechazar("Error");
    }
  }, 2000);
});

promesa
  .then((data) => console.log(data))
  .catch((data) => console.log(data));
```

```
function promesa() {
  return new Promise((resolver, rechazar) => {
    setTimeout(() => {
      let numero = Math.random();
      if (numero >= 0.5) {
        resolver("Éxito");
      } else {
        rechazar("Error");
      }
    }, 2000);
  });
}

async function ejecutarPromesa() {
  try {
    const resultado = await promesa();
    console.log(resultado);
  } catch (error) {
    console.log(error);
  }
}

ejecutarPromesa();
```

Capturar un error con Catch

Cuando trabajamos con funciones asíncronas, podemos capturar errores utilizando el bloque try...catch. Este mecanismo permite manejar fallos sin necesidad de encadenar métodos .catch() como en las promesas tradicionales.

```
async function ejecutarPromesa() {
  try {
    const resultado = await promesa();
    console.log(resultado);
  } catch (error) {
    console.log(error);
  }
}

ejecutarPromesa();
```

Cuadro comparativo de ventajas y desventajas

	Ventajas	Desventajas
Callback	• Implementación directa • Alta compatibilidad	• Flujo poco intuitivo • Manejo de errores complejo
Promesas	• Flujo más legible • Manejo estructurado de errores	• Puede volverse complejo en casos muy anidados
Async await	• Sintaxis clara y secuencial • Manejo fácil de errores con try/catch	• Requiere entornos modernos (no soportado en IE11)


sustantiva
CON•SENTIDO