



JavaScript

Asincronía, Callbacks y Funciones de Orden Superior

< Ing := Carlos H. Rueda C++ />

1. ¿Qué es asincronía?



La **asincronía** se refiere a la ejecución de tareas o procesos de manera **no secuencial** o **en paralelo**, sin esperar a que una tarea se complete antes de comenzar otra. En el contexto de la programación y la informática, la **asincronía** se utiliza para realizar múltiples tareas de manera concurrente, lo que permite que un programa sea más eficiente y receptivo, especialmente cuando se trata de operaciones que pueden llevar tiempo.

* * * * *

La asincronía es importante en situaciones en las que se deben realizar tareas que pueden ser lentas o que involucren esperar a eventos externos, como la **lectura/escritura** de archivos, solicitudes a servidores, **entrada/salida** (E/S) de datos, interacción con bases de datos, y más. En lugar de bloquear la ejecución del programa hasta que se complete cada tarea, la asincronía permite que el programa continúe ejecutándose y maneje las tareas de manera concurrente o mediante programación por eventos.

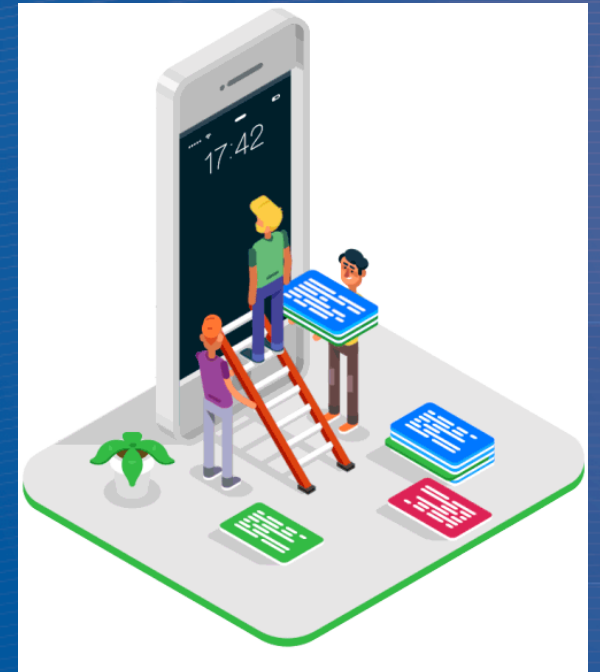
* * * * *

En lenguajes de programación como **JavaScript**, la asincronía se implementa a través de mecanismos como **callbacks**, **promesas**, **async/await**, hilos (**threads**) en lenguajes de programación como **Java** o **Python**, y otros métodos que permiten la ejecución de tareas en segundo plano sin bloquear el hilo principal de ejecución.



```
firstTask(data, function(err, result) {  
  secondTask(data, function(err, result) {  
    thirdTask(data, function(err, result) {  
      fourthTask(data, function(err, result) {  
        fifthTask(data, function(err, result) {  
          // Code  
        });  
      });  
    });  
  });  
});
```


La **asincronía** es fundamental en el desarrollo de aplicaciones modernas, especialmente en el contexto de aplicaciones web y móviles, donde la capacidad de responder a eventos de manera eficiente y de realizar operaciones en segundo plano es esencial para brindar una experiencia de usuario fluida.



1.2 Asincronía en Javascript

Es fundamental comprender que JavaScript, en su **hilo principal**, opera de manera **sincrónica**, lo que implica que las tareas se ejecutan en secuencia, siguiendo el orden de escritura en el código. En este contexto, **el código se procesa de manera lineal**, con cada tarea finalizando antes de que la siguiente comience, lo que se conoce como un "**modelo de ejecución de un solo hilo**" o "**modelo de ejecución secuencial**". A continuación se observa un ejemplo de como se ejecuta varias funciones en JavaScript de modo secuencial.

```
function tarea1() {  
    console.log("Tarea 1 iniciada");  
    console.log("Tarea 1 completada");  
}  
  
function tarea2() {  
    console.log("Tarea 2 iniciada");  
    console.log("Tarea 2 completada");  
}  
  
function tarea3() {  
    console.log("Tarea 3 iniciada");  
    console.log("Tarea 3 completada");  
}  
  
tarea1();  
tarea2();  
tarea3();  
  
console.log("Todas las tareas han finalizado.");
```


Consola

DevTools is now available in Spanish! [Always match Chrome's language](#) [Switch DevTools to S](#)

🔍

📄

Elements

Console

Sources

Network

Performance

Memory

Appl

▶

🚫

top ▼

👁

Filter

⚠️ Error with Permissions-Policy header: Unrecognized feature: 'ch-ua-form-factor'

⚠️ DevTools failed to load source map: Could not load content for https://ogs.googleusercontent.com/CGHIGw/d=1/ed=1/rs=AM-SdHu8ebGogWe-_T0ouClh0Sr24swq0Q/chrome.css.map: HTTP e

```
> function tarea1() {
  console.log('Tarea 1 iniciada');
  console.log('Tarea 1 completada');
}

function tarea2() {
  console.log('Tarea 2 iniciada');
  console.log('Tarea 2 completada');
}

function tarea3() {
  console.log('Tarea 3 iniciada');
  console.log('Tarea 3 completada');
}

tarea1();
tarea2();
tarea3();

console.log('Todas las tareas han finalizado.');
```

Tarea 1 iniciada

Tarea 1 completada

Tarea 2 iniciada

Tarea 2 completada

Tarea 3 iniciada

Tarea 3 completada

Todas las tareas han finalizado.

⏪

 undefined

* * * * *

En tareas de mayor complejidad, es necesario realizar operaciones **asincrónicas**, como solicitudes a **servidores**, **temporizadores**, **interacciones de usuario**, **lectura/escritura de archivos**. Estas operaciones **asincrónicas** permiten que el código continúe ejecutándose sin bloquearse mientras se espera la finalización de la tarea asincrónica.

* * * * *



Los mecanismos para lograr la **asincronía** en **JavaScript** incluyen **callbacks**, **promesas** y **async/await**, que permiten manejar tareas que pueden llevar tiempo sin bloquear el hilo principal, lo que es esencial para mantener la capacidad de respuesta en aplicaciones web y otros entornos. A continuación se menciona algunas situaciones donde se hace uso de la programación con asincronía.

* * * * *



👁️ Casos de Uso de la Asincronía:

- **Solicitudes a servidores:** Cuando una aplicación web necesita obtener datos de un servidor, como recuperar información de una base de datos, realizar llamadas a una API o cargar contenido externo, las solicitudes pueden llevar tiempo, y bloquear el hilo principal de ejecución podría hacer que la interfaz de usuario sea no receptiva.

* * * * *



👁️ Casos de Uso de la Asincronía:

- **Temporizadores:** Cuando se necesita programar tareas en el futuro, como animaciones, actualizaciones periódicas o recordatorios, la asincronía permite programar eventos para que ocurran en momentos específicos sin detener la ejecución del programa.

* * * * *



👁️ Casos de Uso de la Asincronía:

- **Interacciones de usuario:** Las aplicaciones web a menudo esperan la interacción del usuario, como hacer clic en un botón o completar un formulario. Las operaciones que se realizan en respuesta a estas interacciones suelen ser asincrónicas para no bloquear la aplicación mientras espera la acción del usuario.

* * * * *



👁️ Casos de Uso de la Asincronía:

- **Lectura/Escritura de archivos:** En entornos como Node.js, cuando se trabaja con archivos, la lectura y escritura de archivos pueden ser tareas lentas, por lo que es importante realizarlas de manera asincrónica para evitar bloqueos.

* * * * *



👁️ Casos de Uso de la Asincronía:

- **Eventos y notificaciones:** La asincronía es esencial cuando se necesita responder a eventos externos, como eventos del sistema operativo, eventos de hardware, notificaciones push o eventos generados por otros componentes del software.

* * * * *



👁️ Casos de Uso de la Asincronía:

- **Procesamiento de datos en lotes:** Cuando se deben realizar tareas de procesamiento intensivo de datos, como transformación, filtrado o análisis de grandes conjuntos de datos, es fundamental hacerlo de manera asincrónica para evitar bloqueos en la aplicación.

* * * * *



👁️ Casos de Uso de la Asincronía:

La **asincronía** es necesaria en situaciones en las que se deben realizar tareas que pueden ser **lentas**, **depender** de eventos externos o que pueden **bloquear** la ejecución del programa si se manejan de manera sincrónica. Permite que la aplicación siga siendo receptiva y eficiente al realizar múltiples tareas de manera concurrente.

* * * * *



👁 Ejemplo


A continuación se observa un ejemplo sencillo de como sería la ejecución el hilo principal del programa de forma **asíncrona**, a través del **setTimeout**. Recuerda es una simulación del comportamiento de asincronía.

```
console.log("Inicio de la tarea");

// Simular una operación asíncronica que se ejecutará después de 2 segundos
setTimeout(function () {
    console.log("Tarea asíncronica completada después de 2 segundos");
}, 2000);

console.log("Tarea principal continúa");
// Este mensaje se mostrará inmediatamente después del "Inicio de la tarea"
```


Consola

 DevTools is now available in Spanish! [Always match Chrome's language](#) [Switch DevTools to Spanish](#) [Don't show again](#)

Elements Console Sources Network Performance Memory Application Security Lighthouse

  top  Filter

```
> console.log('Inicio de la tarea');

// Simular una operación asíncronica que se ejecutará después de 2 segundos
setTimeout(function() {
  console.log('Tarea asíncronica completada después de 2 segundos');
}, 2000);

console.log('Tarea principal continúa');
Inicio de la tarea
Tarea principal continúa
< undefined
Tarea asíncronica completada después de 2 segundos
>
```

En este ejemplo, la función **setTimeout** se utiliza para simular una operación asíncrona que se ejecuta después de un período de tiempo (en este caso, 2 segundos). Mientras espera la finalización de la tarea asíncrona, el código continúa ejecutándose, lo que permite que "Tarea principal continúa" se muestre antes de que se complete la tarea asíncrona. Esto ilustra la asincronía en JavaScript, donde el flujo del programa no se **bloquea** mientras se espera la finalización de una tarea.

* * * * *

1.3 Lenguaje no bloqueante

Un lenguaje no bloqueante se refiere a un lenguaje de programación o a un enfoque de programación en el que las operaciones asíncronas se diseñan de tal manera que no bloquean el hilo de ejecución principal del programa. En otras palabras, en un lenguaje no bloqueante, es posible realizar múltiples tareas simultáneamente sin que una tarea espere a que otra se complete antes de continuar.

* * * * *

👁👁 Los lenguajes no bloqueantes son especialmente útiles en aplicaciones que requieren alta concurrencia, como aplicaciones web en tiempo real, servidores y sistemas distribuidos. Ejemplos de lenguajes y tecnologías no bloqueantes incluyen:

- **Node.js**
- **Erlang**
- **Go (Golang)**
- **Reactor Pattern**
- **Promesas y async/await:** Mecanismos en lenguajes como JavaScript para gestionar operaciones no bloqueantes de manera más legible y sencilla.

1.3.1 JavaScript es un lenguaje bloqueante o no bloqueante

JavaScript en sí mismo *no es intrínsecamente no bloqueante ni bloqueante*; su capacidad de ser no bloqueante o bloqueante depende del contexto en el que se ejecuta.



Node.js es un entorno de ejecución de JavaScript que se diseñó específicamente para *ser no bloqueante y asíncrono desde su inicio*.

En contraste, cuando **JavaScript** se ejecuta en un navegador web, su comportamiento **generalmente es más bloqueante en el hilo principal**, ya que las interacciones con el usuario, como eventos de clic y peticiones de red, pueden bloquear la ejecución del código en el hilo principal si no se manejan adecuadamente de forma **asincrónica**.

Entonces, **JavaScript** puede ser tanto **bloqueante** como no **bloqueante**, dependiendo del entorno de ejecución y de cómo se diseñe y se maneje el código en ese contexto. Node.js se destaca por su naturaleza no bloqueante, mientras que en el navegador web, se requiere un manejo cuidadoso de la asincronía para evitar bloqueos en el hilo principal.

1.4 Tareas asincronas

Tareas son asíncronas, asicronas por defecto:

Ejemplo de Tarea Asíncrona	Descripción de la Tarea
Fetch a una URL	Realiza una solicitud a una URL remota para recuperar un archivo en formato JSON,
Play de un .mp3 con new Audio()	Reproduce un archivo mp3 y la reproducción es asincrónica con respecto al flujo de ejecución del programa.

1.4 Tareas asincronas

Ejemplo de Tarea Asíncrona	Descripción de la Tarea
Tarea programada con setTimeout()	Programa una tarea para que se ejecute después de un cierto período de tiempo, permitiendo que el programa continúe ejecutándose.
Comunicación a la API de sintetizador de voz	Realiza una comunicación desde JavaScript a la API del sintetizador de voz del navegador para convertir texto en voz, y este proceso asincrónico puede llevar tiempo.

1.4 Tareas asincronas

Ejemplo de Tarea Asíncrona	Descripción de la Tarea
Comunicación a la API de un sensor del smartphone	Realiza una comunicación desde JavaScript a la API de un sensor del smartphone, lo que puede involucrar operaciones asincrónicas al interactuar con el hardware del dispositivo.

Estos ejemplos ilustran situaciones en las que la asincronía es necesaria para evitar bloqueos en la ejecución del programa, ya sea debido a la descarga de datos, la reproducción de multimedia, la programación de tareas futuras o la comunicación con APIs y sensores.

* * * * *



1.5 Gestionar la asincronia

Para gestionar la asincronía en JavaScript, puedes utilizar varios mecanismos y patrones de programación. Algunos de los principales métodos son:

👁️ Métodos para gestionar la asincronía:

- **Callbacks:** Los callbacks son funciones que se pasan como argumentos a otras funciones y se ejecutan una vez que se completa una operación asincrónica. Es uno de los métodos más antiguos para manejar la asincronía en JavaScript.

👁️ Métodos para gestionar la asincronía:

- **Promesas:** Las promesas son un mecanismo más moderno y más legible para manejar tareas asincrónicas. Permiten encadenar múltiples operaciones y manejar tanto el éxito como el fracaso de una tarea.

* * * * *



👁️ Métodos para gestionar la asincronía:

- **async/await**: La sintaxis **async/await** proporciona una forma más sencilla y legible de trabajar con promesas. Permite escribir código asíncrono de manera similar al código síncrono, lo que facilita la gestión de la asincronía.

* * * * *



2. Funciones callbacks

Un **callback** es simplemente una **función** que se **pasa como argumento a otra función** y se ejecuta después de que esta última ha completado su tarea. Los callbacks son ampliamente utilizados para manejar **operaciones asíncronas**, como solicitudes de red, temporizadores o interacciones de usuario.



```
firstTask(data, function(err, result) {  
  secondTask(data, function(err, result) {  
    thirdTask(data, function(err, result) {  
      fourthTask(data, function(err, result) {  
        fifthTask(data, function(err, result) {  
          // Code  
        });  
      });  
    });  
  });  
});
```


Aquí tienes un ejemplo sencillo de cómo funcionan las funciones de callback:

Ejemplo 1. Crear un callback

```
function myDisplayer(some) {  
    console.log(some);  
}  
  
function myCalculator(num1, num2, myCallback) {  
    let sum = num1 + num2;  
    myCallback(sum);  
}  
  
myCalculator(5, 5, myDisplayer);
```

Consola

 DevTools is now available in Spanish! [Always match Chrome's language](#)

Elements Console Sources Network Performance

  top  Filter

```
> function myDisplayer(some) {  
  console.log(some);  
}  
  
function myCalculator(num1, num2, myCallback) {  
  let sum = num1 + num2;  
  myCallback(sum);  
}  
  
myCalculator(5, 5, myDisplayer);  
10
```

* * * * *

Ejemplo 2. Eliminar de una lista los número negativos usando callback

```
const myNumbers = [4, 1, -20, -7, 5, 9, -6];

// Call removeNeg with a callback
const posNumbers = removeNeg(myNumbers, (x) => x ≥ 0);
console.log(posNumbers);

// Keep only positive numbers
function removeNeg(numbers, callback) {
  const myArray = [];
  for (const x of numbers)
    if (callback(x))
      myArray.push(x);
  return myArray;
}
```

DevTools is now available in Spanish! Always match Chrome's language

Elements Console Sources Network Performance

top Filter

```
> // Create an Array
const myNumbers = [4, 1, -20, -7, 5, 9, -6];

// Call removeNeg with a callback
const posNumbers = removeNeg(myNumbers, (x) => x >= 0);

// Display Result
console.log(posNumbers);

// Keep only positive numbers
function removeNeg(numbers, callback) {
  const myArray = [];
  for (const x of numbers) {
    if (callback(x)) {
      myArray.push(x);
    }
  }
  return myArray;
}
```

► (4) [4, 1, 5, 9]

* * * * *

Ejemplo 3

```
const list = ["A", "B", "C"];

function action(element, index) {
  console.log("i=", index, "list=", element);
}

list.forEach(action);
```

Esto se suele reescribir de la siguiente forma:

```
list.forEach((element, index) => {
  console.log("i=", index, "list=", element)
}))
```

Ejemplo 4

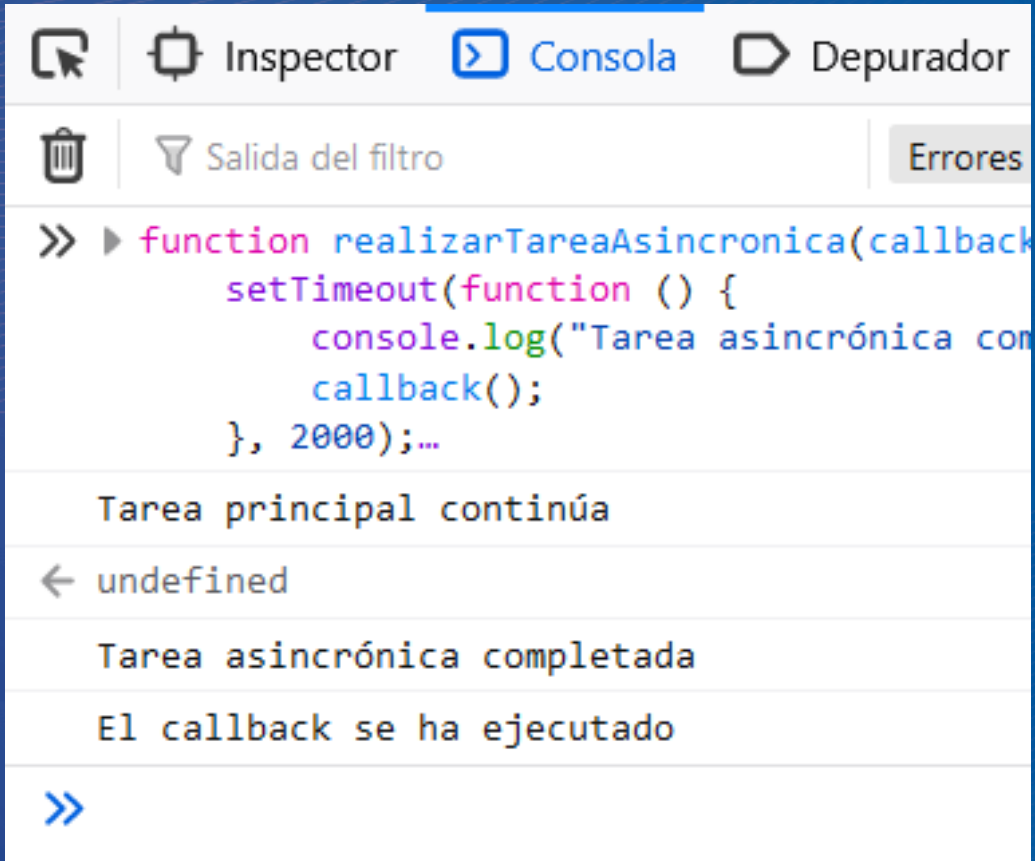
```
function action() {  
  console.log("He ejecutado la función");  
}  
setTimeout(action, 2000);;
```

Es mas legible asi:

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


2.1 Callbacks y asincronía

```
function realizarTareaAsincronica(callback) {  
    setTimeout(function () {  
        console.log("Tarea asincrónica completada");  
        callback();  
    }, 2000);  
}  
  
function miCallback() { // Función de callback  
    console.log("El callback se ha ejecutado");  
}  
  
// Llamando a la función asincrónica y pasando el callback  
realizarTareaAsincronica(miCallback);  
console.log("Tarea principal continúa");
```



The screenshot shows a web browser's developer console with the 'Consola' (Console) tab selected. At the top, there are icons for 'Inspector', 'Consola', and 'Depurador'. Below these, there's a 'Salida del filtro' (Filter output) section and an 'Errores' (Errors) button. The main console area displays the following:

```
>> ▶ function realizarTareaAsincronica(callback) {  
    setTimeout(function () {  
        console.log("Tarea asincrónica con  
        callback();  
    }, 2000);...
```

Below the code, the console shows a sequence of messages:

- Tarea principal continúa
- ← undefined
- Tarea asincrónica completada
- El callback se ha ejecutado

At the bottom, there's a blue double arrow icon (>>).

En este ejemplo, la función **realizarTareaAsincronica** toma una función de **callback** como argumento y la ejecuta después de completar su tarea asincrónica (simulada con **setTimeout**). El uso de callbacks permite que el código sea no bloqueante y siga ejecutándose mientras se espera

que se complete la tarea asincrónica.

2.2 Desventajas de los Callbacks

👁️ Desventajas de los Callbacks

Algunas desventajas que pueden complicar la estructura y la legibilidad del código. Aquí están algunas desventajas de los callbacks:

- **Dificultad de lectura y comprensión:**
- **Dificultad de depuración:**
- **Causas de errores:**
- **Problemas de rendimiento:**

* * * * *

Para evitar estas desventajas, es importante utilizar callbacks de forma responsable. A continuación, se presentan algunas recomendaciones:

- **Utilice callbacks solo cuando sea necesario:** Los callbacks no siempre son necesarios. Si es posible, evite su uso.
- **Escriba callbacks claras y concisas:** Las callbacks deben ser fáciles de entender y usar.
- **Use nombres de variables descriptivas:** Los nombres de las variables utilizadas en las callbacks deben ser descriptivos para que sea fácil entender lo que hacen.
- **Proporcione documentación adecuada:** La documentación de las callbacks debe ser clara y concisa.

3. Funciones de orden superior

Las funciones de orden superior (Higher-Order Functions) en JavaScript son funciones que pueden aceptar otras funciones como argumentos y/o devolver funciones como resultados. Esto permite un estilo de programación más flexible y funcional

* * * * *



👁️ Funciones que aceptan funciones como argumentos:

```
function operar(arr, operacion) {  
    let resultado = 0;  
    for (let num of arr) {  
        resultado = operacion(resultado, num);  
    }  
    return resultado;  
}  
  
function suma(a, b) {  
    return a + b;  
}  
  
function producto(a, b) {  
    return a * b;  
}  
  
const numeros = [1, 2, 3, 4, 5];  
const sumaTotal = operar(numeros, suma); // Resultado: 15  
const productoTotal = operar(numeros, producto); // Resultado: 120
```


Funciones que devuelven funciones:

```
function multiplicador(factor) {  
    return function (numero) {  
        return numero * factor;  
    };  
}
```

```
const duplicar = multiplicador(2);  
const triplicar = multiplicador(3);
```

```
console.log(duplicar(5)); // Resultado: 10  
console.log(triplicar(5)); // Resultado: 15
```

Otro ejemplo - lanzamiento de 10 dados:

```
const doTask = (iterations, callback) => {
  const numbers = [];

  for (let i = 0; i < iterations; i++) {
    const number = 1 + Math.floor(Math.random() * 6);
    numbers.push(number);
    if (number === 6) {
      callback({
        error: true,
        iter: i,
        message: "Se ha sacado un 6"
      });
      return;
    }
  }

  /* Termina bucle y no se ha sacado 6 */
  return callback(null, {
    error: false,
    value: numbers
  });
}
```


Llamada a la función **doTask()**, donde le pasamos esa función callback e implementamos el funcionamiento, que en nuestro caso serán dos simples `console.error()` y `console.log()`:

```
doTask(10, function(err, result) {  
  if (err) {  
    console.error(">>Error: ", err.message);  
    console.log(err);  
    return;  
  }  
  console.log("Tiradas correctas: ", result.value);  
});
```

Métodos de arreglo de orden superior:

JavaScript proporciona métodos de orden superior incorporados en arreglos, como `map()`, `filter()`, `reduce()`, y `forEach()`, que aceptan funciones como argumentos.

* * * * *



Métodos de arreglo de orden superior:

Por ejemplo:

```
const numeros = [1, 2, 3, 4, 5];

const duplicados = numeros.map(function (numero) {
  return numero * 2;
}); // Resultado: [2, 4, 6, 8, 10]

const pares = numeros.filter(function (numero) {
  return numero % 2 === 0;
}); // Resultado: [2, 4]

const sumaTotal = numeros.reduce(function (acumulador, numero) {
  return acumulador + numero;
}, 0); // Resultado: 15
```

3.1 ¿Callbacks son funciones de orden superior?

Las funciones de orden superior son un concepto más amplio que incluye a los callbacks. Los callbacks son un tipo de función de orden superior que se pasa como argumento a otra función y se invoca en un momento posterior, generalmente cuando se completa una operación asincrónica.

* * * * *



👁️ Las funciones de orden superior se refieren a cualquier función que cumple al menos uno de los siguientes criterios:

1. Acepta una o más funciones como argumentos.
2. Devuelve una función como resultado.

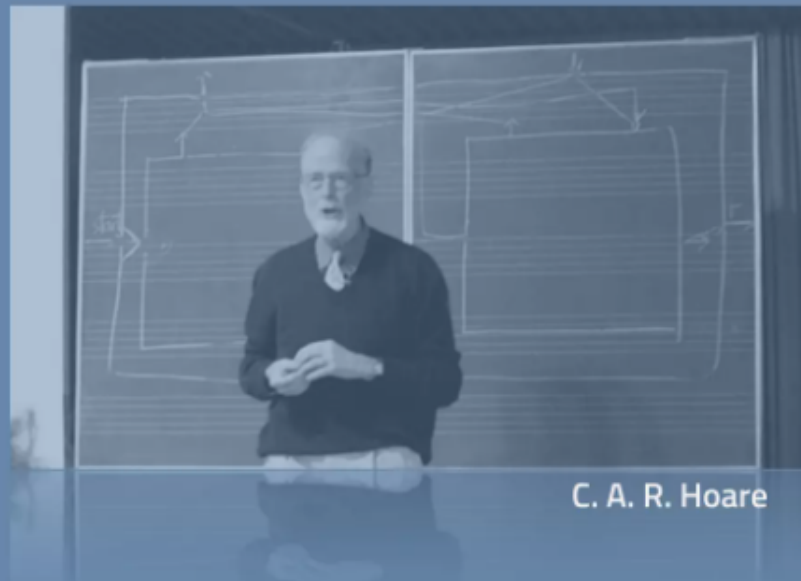
* * * * *



Los callbacks son un caso común de funciones de orden superior que cumplen el primer criterio, ya que se pasan como argumentos a otras funciones para especificar qué hacer después de que se complete una tarea asincrónica. Sin embargo, las funciones de orden superior pueden abarcar otros conceptos, como funciones que generan otras funciones o funciones que toman funciones como argumentos para personalizar su comportamiento.

Si bien los callbacks son un ejemplo específico de funciones de orden superior, las funciones de orden superior incluyen un conjunto más amplio de funciones que pueden aceptar o devolver funciones como parte de su funcionalidad.

"Hay dos formas de construir un diseño del software. Una es hacerlo tan simple que sea obvio que no hay deficiencias y la otra es hacerlo tan complicado que no haya deficiencias obvias. El primer método es mucho más difícil."



C. A. R. Hoare

Ejemplo

```
function mayorQue(n) {  
  return m  $\Rightarrow$  m > n;  
}  
let mayorQue10 = mayorQue(10);  
console.log(mayorQue10(11));
```

Que devuelve la ejecucion de este pequeño trozo de código?

* * * * *



Ejemplo

```
function ruidosa(funcion) {  
  return (...argumentos) => {  
    console.log("llamando con", argumentos);  
    let resultado = funcion(...argumentos);  
    console.log("llamada con", argumentos, ", retorno", resultado);  
    return resultado;  
  };  
}  
ruidosa(Math.min)(3, 2, 1);
```

Que devuelve la ejecucion de este pequeño trozo de código?



Ejercicio 1

Implementa una función que reciba una **función** como primer argumento, y un número **num** como segundo argumento. La función debe ejecutar la función num veces.

Argumentos

- **operation:** Una función, no recibe argumentos, no retorna ningún valor útil.
- **num:** el número de veces que queremos invocar operation

* * * * *

No lo des demasiadas vueltas, el código debe ser muy simple. No pasa nada si usas un loop.