22. Callbacks en programación asíncrona

Como vimos en el punto sobre las funciones, un **callback** (llamada de retorno) es una función que se pasa como **argumento** a otra función y se ejecuta después de que esta última haya completado una tarea específica. Los callbacks son fundamentales en JavaScript para manejar operaciones **asincrónicas** (como temporizadores, peticiones a APIs o eventos del usuario) y para personalizar comportamientos en funciones genéricas.

Definición simple:

- **Función que se ejecuta después de otra**.
- Permite definir qué hacer una vez que una operación ha finalizado.

Ejemplo Básico:

```
function procesarDatos(datos, retrollamada) {
   console.log("Procesando datos...");
   retrollamada(datos);
}

const funcion_a_enviar1 = function(entrada) {
      let entrada_mayus = entrada.toUpperCase()
      console.log(entrada_mayus);
};


const funcion_a_enviar2 = function(entrada) {
      let entrada_minus = entrada.toLowerCase()
      console.log(entrada_minus);
};

procesarDatos("Hola", funcion_a_enviar1);
procesarDatos("Hola", funcion_a_enviar2);
// Salida:
// HOLA
// hola
```

Salida:

```
¡Hola, Ana!
¡Adiós!
```

¿Por qué son importantes?

1. **Asincronía**:

- JavaScript es **monohilo** (una sola secuencia de ejecución). Los callbacks permiten ejecutar código después de operaciones que toman tiempo (ej: descargar datos).

2. **Flexibilidad**:
   - Personalizan el comportamiento de funciones genéricas. *Ejemplo*: `setTimeout` acepta un callback para definir qué hacer después del retraso.
3. **Eventos**:
   - Respuesta a acciones del usuario (ej: clicks, formularios).

Ejemplos de uso que hemos visto

1. Métodos de Array

```
const animales = ["perro", "gato", "pájaro"];

// Callback en forEach
animales.forEach(animal => {
  console.log(animal.toUpperCase());
});
```

2. Eventos del DOM

```
// Callback para un evento de click
document.getElementById("boton").addEventListener("click", function() {
  console.log("¡Botón presionado!");
});
```

3. Temporizadores (`setTimeout` y `setInterval`)

```
// Callback en setTimeout
setTimeout(() => {
  console.log("Este mensaje se muestra después de 2 segundos");
}, 2000);

// Callback en setInterval
setInterval(() => {
  console.log("Este mensaje se repite cada segundo");
}, 1000);
```

Características de los Callbacks

1. **Pueden ser Anónimos o Nombrados**:

```
// Función nombrada
function mostrarMensaje() {
```

```
    console.log("Mensaje mostrado");
}

setTimeout(mostrarMensaje, 1000);

// Función anónima
setTimeout(() => {
    console.log("Otro mensaje");
}, 1000);
```

2. **Se Ejecutan en el Contexto del Llamador**:
   - Quien recibe el callback decide **cuándo** y **cómo** ejecutarlo.

Callbacks Asincrónicos

```
function descargarDatos(callback) {
    setTimeout(() => {
        callback("Datos descargados");
    }, 2000);
}

console.log("Iniciando descarga...");
descargarDatos((datos) => {
    console.log(datos); // Se ejecuta después de 2 segundos
});
console.log("Descarga en progreso...");

// Salida:
// Iniciando descarga...
// Descarga en progreso...
// (Después de 2 segundos): "Datos descargados"
```

Callback Hell (Infierno de Callbacks)

El **callback hell** se refiere a una situación en la que tienes múltiples funciones asíncronas anidadas unas dentro de otras usando *callbacks*, lo que lleva a un código difícil de leer, mantener y depurar. También se le conoce como la **"pirámide de la perdición"** por su estructura en forma de escalera o pirámide.

Ejemplo clásico de callback hell:

```
loginUser('usuario', 'contraseña', function(error, user) {
  if (!error) {
    getUserData(user.id, function(error, data) {
      if (!error) {
```

```
      getUserPosts(data, function(error, posts) {
        if (!error) {
          sendNotification(posts, function(error, result) {
            if (!error) {
              console.log('Notificación enviada');
            }
          });
        }
      });
    }
  });
}
});
```

Este tipo de estructura:

- Es difícil de leer.
- Es propensa a errores.
- Complica el manejo de errores y la lógica del programa.

¿Cómo se soluciona?

A lo largo del tiempo, JavaScript ha evolucionado con nuevas formas de manejar código asíncrono para evitar el callback hell:

✅ Promesas (Promises)

```
loginUser('usuario', 'contraseña')
  .then(user => getUserData(user.id))
  .then(data => getUserPosts(data))
  .then(posts => sendNotification(posts))
  .then(() => console.log('Notificación enviada'))
  .catch(error => console.error(error));
```

✅ Async/Await

```
async function enviarNotificacion() {
  try {
    const user = await loginUser('usuario', 'contraseña');
    const data = await getUserData(user.id);
    const posts = await getUserPosts(data);
    await sendNotification(posts);
    console.log('Notificación enviada');
  } catch (error) {
    console.error(error);
  }
}
```