Material descargable

egg

Teoría **Javascript II**

¡Hola! 👋 Te damos la bienvenida a Javascript II - Parte 3

Seguiremos abordando conceptos de Javascript, que nos ayudarán a entender el lenguaje en profundidad para luego aplicarlo a nuestros proyectos.

¡Que continúe el viaje! 🚀

Trabajando con Clases en El DOM

classList

En la teoría anterior introdujimos el tema de classList, poniendo algunos ejemplos sobre su uso, pero aquí queremos abordarlo en más profundidad.

classList es una propiedad en el **DOM (Document Object Model)** de JavaScript que se encuentra en **objetos Element**. **Esta propiedad se utiliza para acceder y manipular las clases CSS aplicadas a un elemento HTML específico.** La propiedad **classList** proporciona métodos para **agregar, eliminar, verificar y cambiar** clases en un elemento sin necesidad de acceder directamente al atributo class del elemento.

La propiedad **classList** es especialmente útil porque facilita la manipulación de clases sin necesidad de realizar operaciones complicadas de cadena en el atributo class.

Los métodos disponibles en classList incluyen:

- **add(className)**: Agrega una clase al elemento. Si la clase ya está presente, no se duplicará.
- **remove(className)**: Elimina una clase del elemento.
- **toggle(className)**: Agrega la clase si no está presente y la elimina si ya lo está, alternando el estado.
- **contains(className)**: Verifica si el elemento tiene una clase específica, devolviendo true o false.
- **replace(oldClass, newClass)**: Reemplaza una clase existente por una nueva.
- **item(index)**: Devuelve la clase en la posición index en la lista de clases del elemento.
- **length**: Proporciona el número total de clases en el elemento.

En resumen, classList facilita la manipulación de clases CSS en elementos HTML a través de métodos sencillos y claros, lo que mejora la legibilidad y mantenibilidad del código. Esto es especialmente útil cuando se trabaja con eventos interactivos, animaciones o cambios dinámicos en la interfaz de usuario.

Ejemplo 1: Agregar y eliminar clases

```
<!DOCTYPE html>
<html>
<head>
<style>
  .highlight {
    background-color: yellow;
  }
</style>
</head>
<body>

<button id="myButton">Haz clic</button>

<script>
const button = document.getElementById('myButton');

button.addEventListener('click', function() {
  // Agregar la clase 'highlight'
```

```
  button.classList.add('highlight');

  // Eliminar la clase 'highlight' después de 2 segundos
  setTimeout(function() {
    button.classList.remove('highlight');
  }, 2000);
});
</script>

</body>
</html>
```

En este ejemplo, **cuando se hace clic en el botón**, **se agrega la clase CSS "highlight"** al botón, lo que cambia su color de fondo a amarillo. Luego, después de 2 segundos, la clase se elimina y el color de fondo vuelve a la normalidad.

💡***Recuerden que les sugerimos probar estos códigos en Visual Studio para que puedan chequear el funcionamiento.***

Ejemplo 2: Alternar clases con "toggle"

```
<!DOCTYPE html>
<html>
<head>
<style>
  .active {
    font-weight: bold;
  }
</style>
</head>
<body>

<p id="myParagraph">Este es un párrafo.</p>

<script>
const paragraph = document.getElementById('myParagraph');
```

```
paragraph.addEventListener('click', function() {
  // Alternar la clase 'active'
  paragraph.classList.toggle('active');
});
</script>

</body>
</html>
```

En este ejemplo, cuando se hace clic en el párrafo, la clase CSS "active" se agrega o elimina según su estado actual. Esto cambia el peso de la fuente del párrafo a negrita y viceversa.

Ejemplo 3: Verificar si una clase está presente

```
<!DOCTYPE html>
<html>
<head>
<style>
  .error {
    color: red;
  }
</style>
</head>
<body>

<input type="text" id="myInput" placeholder="Escribe algo">

<script>
const input = document.getElementById('myInput');

input.addEventListener('input', function() {
  // Verificar si el valor del input está vacío
  if (input.value === '') {
    input.classList.add('error');
  } else {
    input.classList.remove('error');
  }
});
```

```
</script>

</body>
</html>
```

En este ejemplo, cuando el usuario escribe algo en el campo de entrada, se verifica si el valor está vacío. Si está vacío, se agrega la clase "error" para cambiar el color del texto a rojo. Si el usuario escribe algo, la clase se elimina y el color del texto vuelve a la normalidad.

---

addEventListener - keydown

Key down

Como ya vimos en clases anteriores el método **addEventListener** recibe dos argumentos: el tipo de evento a ser escuchado (el que vimos es click) y la función que se va a ejecutar cuando ocurra ese evento.

En este caso vamos a ver el evento **keydown,** que sucede cuando el usuario presiona una tecla.

El evento **keydown** ocurre cuando una tecla del teclado se presiona y se mantiene. Puede ser desencadenado por cualquier tecla del teclado, ya sea una letra, número, tecla de flecha, tecla de función, etc.

✏️ **Ejemplo de uso:**

Supongamos que queremos detectar cuándo el usuario presiona una tecla específica en el teclado y realizar una acción en respuesta. En este caso, usaremos addEventListener junto con el evento keydown.

```
<!DOCTYPE html>
<html>
<head>
</head>
<body>

<input type="text" id="myInput" placeholder="Escribe algo">

<script>
const input = document.getElementById('myInput');

input.addEventListener('keydown', function(event) {
  // Verificar si la tecla presionada es la tecla Enter (código 13)
  if (event.keyCode === 13) {
    alert('Presionaste la tecla Enter!');
  }
});
</script>

</body>
</html>
```

💡**Nota importante**:

El ejemplo anterior utiliza **event.keyCode**, pero es importante mencionar que este atributo ha sido considerado obsoleto en favor de **event.key** para identificar las teclas. Sin embargo, en navegadores antiguos, event.keyCode todavía puede ser necesario. En versiones más recientes, se recomienda usar event.key para obtener una representación más legible de la tecla presionada.

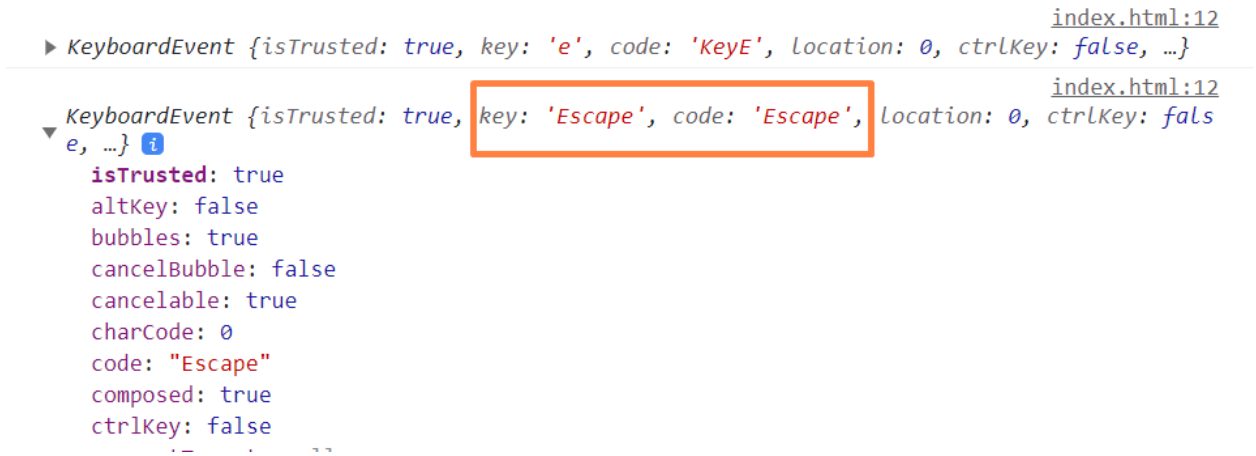
En resumen, addEventListener y el evento keydown son herramientas esenciales para interactuar con eventos del usuario en JavaScript y permiten crear interacciones dinámicas y receptivas en una página web.

✏️ Ejemplo de uso 2

Si quiero averiguar cuál fue la tecla presionada podemos hacer lo siguiente

```
<!DOCTYPE html>

<html>

  <head> </head>

  <body>

    <input type="text" id="myInput" placeholder="Escribe algo" />



    <script>

      const input = document.getElementById('myInput');



      input.addEventListener('keydown', function (event) {

        // Verificar cuál es la tecla presionada

        console.log(event);

      });

    </script>

  </body>

</html>
```

En la consola verán algo como esto:



Es decir, que dependiendo la tecla que se presione va a aparecer su código y esto nos ayudarpa a determinar cuál es la nomenclatura de la misma.

Entonces, si quisiéramos que suceda una alerta al presionar la tecla **'Escape' ,** podríamos primero identificar cómo se llama esa tecla (como vimos recién) y luego hacer el if

```
<!DOCTYPE html>

<html>

  <head> </head>

  <body>

    <input type="text" id="myInput" placeholder="Escribe algo" />



    <script>

      const input = document.getElementById('myInput');



      input.addEventListener('keydown', function (event) {

        // Verificar cuál es la tecla presionada

        console.log(event);

        if (event.key === 'Escape') {
```

```
          alert('Presionaste la tecla Escape!');

        }

      });

    </script>

  </body>

</html>
```

De esta manera pudimos ver dos cosas:

1. Identificar el código de cada tecla
2. Poder generar un condicional de acuerdo a la tecla presionada.

Objeto Window

El **objeto window** es uno de los objetos fundamentales proporcionados por los navegadores para interactuar con la ventana del navegador y controlar aspectos relacionados con la visualización y el comportamiento de una página web. Representa la ventana del navegador en la que se carga una página web y proporciona una serie de propiedades y métodos que permiten acceder y manipular diversas características de la ventana y del documento cargado en ella.

Algunas de las funcionalidades y propiedades más comunes proporcionadas por el objeto window incluyen:

- **Manipulación del DOM** (Document Object Model): A través del objeto window, se puede acceder al DOM del documento actual y manipular elementos HTML, cambiar su contenido, estilos y propiedades, añadir o eliminar elementos, etc.
- **Control de la ventana:** El objeto window permite ajustar el tamaño y la posición de la ventana del navegador, abrir y cerrar ventanas emergentes (pop-ups), y controlar la barra de desplazamiento.
- **Navegación**: Puede utilizarse para redireccionar a otras páginas web, recargar la página actual o volver a la página anterior en el historial de navegación.
- **Gestión de temporizadores**: El objeto window permite crear temporizadores para ejecutar funciones en intervalos específicos utilizando métodos como setTimeout y setInterval.
- **Gestión de eventos:** Se pueden agregar oyentes de eventos (event listeners) para manejar interacciones del usuario como clics, teclas presionadas, movimientos del mouse, entre otros.
- **Almacenamiento local**: El objeto window proporciona mecanismos para almacenar datos localmente en el navegador, como localStorage y sessionStorage.
- **Comunicación entre ventanas**: Puede utilizarse para comunicarse entre diferentes ventanas o pestañas abiertas del mismo sitio web utilizando técnicas como la API window.postMessage().
- **Manipulación de historial**: Permite acceder y manipular el historial de navegación del usuario, facilitando la navegación hacia adelante o hacia atrás en la historia del navegador.

Es importante tener en cuenta que aunque el objeto window es accesible globalmente en el ámbito del navegador, no es necesario utilizarlo de manera explícita en todos los casos, ya que muchos de sus métodos y propiedades son accesibles directamente sin la necesidad de referenciarlo. Por ejemplo, puedes utilizar setTimeout() en lugar de window.setTimeout().

👉 **Para ver más información sobre el objeto window ingresar a w3schools "The window Object"**

Aquí tienes una lista de algunos de los métodos más comunes del objeto window:

- **window.alert()**: Muestra un cuadro de alerta con un mensaje y un botón "Aceptar".
- **window.confirm():** Muestra un cuadro de confirmación con un mensaje y botones "Aceptar" y "Cancelar".
- **window.prompt():** Muestra un cuadro de diálogo que solicita al usuario que ingrese información.
- **window.open():** Abre una nueva ventana o pestaña del navegador con una URL específica y opciones de configuración.
- **window.close():** Cierra la ventana actual (nota: esto puede estar sujeto a restricciones de seguridad y a las políticas del navegador).
- **window.setTimeout(función, tiempo)** y **window.setInterval(función, intervalo):** Estos métodos crean temporizadores que ejecutan la función proporcionada después de un cierto tiempo o en intervalos regulares.
- **window.clearTimeout(id)** y **window.clearInterval(id)**: Estos métodos se utilizan para detener los temporizadores creados con setTimeout() o setInterval().
- **window.scrollTo(x, y)** y **window.scrollBy(x, y)**: Estos métodos se utilizan para desplazarse a una posición específica en la página o para realizar un desplazamiento relativo.
- **window.location.href** y métodos relacionados: Permite acceder y modificar la URL actual de la ventana, y se puede utilizar para redirigir a otra página.
- **window.history**: Proporciona métodos para navegar hacia adelante o hacia atrás en el historial de navegación y acceder a la información del historial.
- **window.document**: Proporciona acceso al DOM (Document Object Model) del documento actual, lo que permite manipular elementos HTML y contenido en la página.
- **window.addEventListener(evento, función)** y **window.removeEventListener(evento, función)**: Estos métodos permiten agregar oyentes de eventos a la ventana para manejar interacciones del usuario, como clics, teclas presionadas, etc.

- **window.postMessage(mensaje, origen)**: Permite la comunicación segura entre ventanas o pestañas abiertas en diferentes orígenes.

Estos son solo algunos ejemplos de los métodos proporcionados por el objeto window. Cada uno de estos métodos cumple una función específica y te permite interactuar con diferentes aspectos del navegador y el entorno de la página web

✏️ Veamos ejemplos de uso

Crear una Ventana Emergente:

```
function abrirVentana() {
    window.open('https://www.ejemplo.com', 'MiVentana',
'width=600,height=400');
}
```

Mostrar un Mensaje de Alerta:

```
function mostrarAlerta() {
    window.alert('¡Esto es una alerta!');
}
```

Solicitar Información al Usuario:

```
function pedirNombre() {
    const nombre = window.prompt('Por favor, ingresa tu nombre:');
    if (nombre) {
        window.alert(`Hola, ${nombre}!`);
    }
}
```

Temporizador para Cambiar el Color de Fondo:

```
function cambiarColor() {
    document.body.style.backgroundColor = 'red';
    setTimeout(function() {
        document.body.style.backgroundColor = 'white';
    }, 2000); // Cambia el color después de 2 segundos
}
```

Redireccionar a una Nueva Página:

```
function redireccionar() {
    window.location.href = 'https://www.otra-pagina.com';
}
```

Usar Confirmación para Eliminar un Elemento:

```
function eliminarElemento() {
    const confirmado = window.confirm('¿Estás seguro de que deseas eliminar este elemento?');
    if (confirmado) {
        // Código para eliminar el elemento
    }
}
```

Desplazamiento Suave en la Página:

```
function desplazamientoSuave() {
    window.scrollTo({
        top: 500,
        behavior: 'smooth' // Desplazamiento suave
    });
}
```

Manejar Eventos del Teclado:

```
window.addEventListener('keydown', function(event) {
    if (event.key === 'Enter') {
        window.alert('Presionaste la tecla Enter');
    }
});
```

Enviar Mensajes Entre Ventanas:

```
const otraVentana = window.open('https://www.otra-pagina.com',
'OtraVentana');

function enviarMensaje() {
    otraVentana.postMessage('¡Hola desde la primera ventana!', '*');
}
```

Navegación con Historial:

```
function retroceder() {
    window.history.back(); // Retrocede en el historial
}

function avanzar() {
    window.history.forward(); // Avanza en el historial
}
```

Estos ejemplos te ayudarán a comprender cómo utilizar los métodos del objeto **window** para realizar diversas acciones en una página web. Recuerda que algunos de estos ejemplos pueden requerir ciertos ajustes dependiendo del contexto de uso y de las políticas del navegador.

Asincronía

La asincronía se refiere a la capacidad del lenguaje para ejecutar tareas sin bloquear la ejecución del resto del código. En otras palabras, **permite que ciertas operaciones se realicen en segundo plano mientras el programa principal sigue funcionando.** Esto es especialmente útil cuando se trabaja con operaciones que pueden ser lentas, como la lectura de archivos, solicitudes a servidores, o cualquier tarea que involucre esperar resultados externos.

La asincronía en JavaScript se logra principalmente a través de dos mecanismos: **callbacks y Promesas (también Async/Await).**

Por lo que Javascript usa un modelo asíncrono y no bloqueante, con un loop de eventos implementado en un sólo hilo, (single thread) para operaciones de entrada y salida (input/output).

¿Pero qué significan todos estos conceptos? Ahora los vamos a explicar de manera más detallada.

SINGLE THREAD Y MULTI THREAD

Estos términos se refieren a la cantidad de hilos de ejecución que un programa o sistema operativo puede utilizar para llevar a cabo tareas concurrentemente.

Single Thread (Hilo Único):

Un programa de un solo hilo (single-threaded) tiene un único hilo de ejecución, lo que significa que solo puede realizar una tarea a la vez. Esto no significa que no pueda hacer cosas de manera rápida, sino que se ejecuta secuencialmente, una acción después de la otra. JavaScript, en su entorno de navegador, es un ejemplo clásico de lenguaje de programación de un solo hilo. Esto significa que si tienes código bloqueante, como una operación de larga duración, puede ralentizar o incluso congelar la interfaz de usuario hasta que se complete.

Multi Thread (Multi-Hilo):

Un programa de múltiples hilos (multi-threaded) tiene la capacidad de ejecutar múltiples hilos de ejecución en paralelo. Cada hilo puede realizar una tarea independiente. Esto permite un mejor rendimiento en situaciones donde hay tareas que pueden ejecutarse en paralelo, como en programas que manejan tareas pesadas, procesamiento de datos, o interacciones intensivas en tiempo real. Lenguajes de programación como Java y C++ son ejemplos de lenguajes que pueden aprovechar múltiples hilos.

La principal ventaja de los programas de múltiples hilos es que pueden aprovechar los procesadores multinúcleo, que son comunes en las computadoras modernas. Esto significa que incluso si el programa es un solo proceso, puede ejecutarse en varios núcleos simultáneamente, mejorando la eficiencia y el rendimiento.

Sin embargo, trabajar con múltiples hilos también puede ser complicado debido a problemas potenciales como condiciones de carrera (cuando varios hilos intentan acceder a un recurso compartido al mismo tiempo), bloqueo (un hilo bloquea el acceso a un recurso) y sincronización.

En resumen, la diferencia entre un programa de un solo hilo y uno de múltiples hilos radica en la cantidad de tareas simultáneas que pueden ejecutarse. Cada enfoque

tiene sus ventajas y desafíos, y la elección depende de las necesidades específicas del programa y el entorno en el que se ejecuta.

OPERACIONES DE CPU Y DE ENTRADA Y SALIDA

• Operaciones CPU: Aquellas que pasan el mayor tiempo consumiendo Procesos del CPU, por ejemplo, la escritura de ficheros.

• Operaciones de Entrada y Salida: Aquellas que pasan la mayor parte del tiempo esperando la respuesta de una petición o recurso, como la solicitud a una API o DB.

CONCURRENCIA Y PARALELISMO

• **Concurrencia**: cuando dos o más tareas progresan simultáneamente.

• **Paralelismo**: cuando dos o más tareas se ejecutan, al mismo tiempo.

BLOQUEANTE Y NO BLOQUEANTE

Se refiere a cómo la fase de espera de las operaciones afectan a nuestra aplicación:

• **Bloqueante**: Son operaciones que no devuelven el control a nuestra aplicación hasta que se ha completado. Por tanto el thread queda bloqueado en estado de espera.

• **No Bloqueante**: Son operaciones que devuelven inmediatamente el control a nuestra aplicación, independientemente del resultado de esta. En caso de que se haya completado, devolverá los datos solicitados. En caso contrario (si la operación no ha podido ser satisfecha) podría devolver un código de error.

SÍNCRONO Y ASÍNCRONO

Se refiere a ¿cuándo tendrá lugar la respuesta?:

• **Síncrono**: La respuesta sucede en el presente, una operación síncrona esperará el resultado.

• **Asíncrono**: La respuesta sucede a futuro, una operación asíncrona no esperará el resultado.

MECANISMOS ASÍNCRONOS EN JAVASCRIPT

Para controlar la asincronía, JavaScript cuenta con algunos mecanismos:

• Callbacks

• Promises

• Async / Await.

Callbacks

Los callbacks son funciones que se pasan como argumentos a otras funciones. Cuando una operación asíncrona se completa, se llama al **callback** para manejar el resultado.

✏️ **Aquí hay un ejemplo simple:**

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

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

console.log("Siguiente tarea");
```

En este caso, **setTimeout** es una función asincrónica que espera durante 1000 milisegundos antes de ejecutar el callback. Mientras tanto, el programa continúa ejecutando la siguiente tarea y luego, después de un segundo, el callback se ejecuta y muestra "Fin" en la consola.

Callbacks Encadenados (Callbacks Hell)

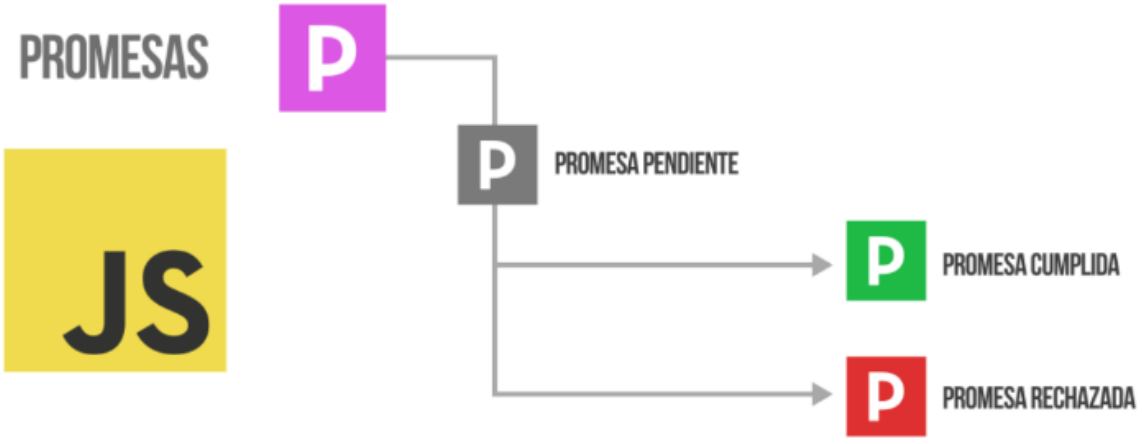
```
function obtenerDatos(callback) {
    setTimeout(function() {
        callback("Datos obtenidos");
    }, 1000);
}

function procesarDatos(datos, callback) {
    setTimeout(function() {
        callback("Datos procesados: " + datos);
    }, 1500);
}

console.log("Inicio");

obtenerDatos(function(datosObtenidos) {
    procesarDatos(datosObtenidos, function(datosProcesados) {
        console.log(datosProcesados);
    });
});

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

Este ejemplo muestra cómo los callbacks pueden anidarse, lo que a menudo lleva a una estructura llamada "callback hell" cuando se tienen muchas operaciones asíncronas anidadas. Esta estructura puede volverse difícil de leer y mantener.

Es importante destacar que mientras los callbacks son una forma tradicional de manejar la asincronía, las **Promesas** y **Async/Await** son enfoques más modernos y legibles que han sido introducidos en JavaScript para lidiar de manera más efectiva con las operaciones asíncronas.

Promises

Las Promesas son un concepto más moderno que proporciona una forma más estructurada de manejar la asincronía. Una Promesa representa un valor que puede estar disponible ahora, en el futuro o nunca. Las Promesas tienen tres estados: pendiente (**pending**), resuelta (**fulfilled**) y rechazada (**rejected**).



Aquí hay un ejemplo:

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

const miPromesa = new Promise(function(resolve, reject) {
    setTimeout(function() {
        resolve("Promesa resuelta");
    }, 1000);
});

miPromesa.then(function(resultado) {
    console.log(resultado);
});

console.log("Siguiente tarea");
```

En este caso, la Promesa se resuelve después de un segundo y el método .**then**() se llama para manejar el resultado. Esto proporciona un mejor control sobre el flujo de la asincronía y permite encadenar múltiples operaciones asíncronas.

Async/Await

Este enfoque, introducido en ECMAScript 2017 (ES8), ofrece una sintaxis más legible y fácil de entender para trabajar con Promesas. La palabra clave async se utiliza para declarar una función asíncrona, y dentro de ella, puedes usar await para esperar a que una Promesa se resuelva antes de continuar.

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

async function miFuncionAsincrona() {
    await new Promise(resolve => setTimeout(resolve, 1000));
    console.log("Tarea asíncrona completada");
}

miFuncionAsincrona();

console.log("Siguiente tarea");
```

En este ejemplo, la función miFuncionAsincrona se ejecutará de manera asíncrona y esperará un segundo antes de imprimir "Tarea asíncrona completada".

En resumen, la **asincronía en JavaScript** es esencial para manejar tareas que toman tiempo y para no bloquear la ejecución del resto del código. Ya sea a través de callbacks, Promesas o Async/Await, estas herramientas permiten un mejor control sobre el flujo de ejecución en programas que requieren operaciones asíncronas.

Cuando trabajemos con APIs, esto va a tener mucho más sentido. Pero por el momento queríamos que tuvieran un panorama completo de estos temas.

Hasta la próxima 🚀