Curso AulaScript

Sección 4: Avanzado

1. Asincronismo en JavaScript

El asincronismo permite ejecutar otras tareas mientras se espera la finalización de un proceso, como la solicitud de datos a un servidor.

- **Síncrono**: El código se ejecuta en orden y bloquea el flujo.
- **Asíncrono**: No bloquea el flujo. JavaScript sigue ejecutando otras tareas mientras espera el resultado de una operación asíncrona (como una petición HTTP).

Ejemplo:

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

setTimeout(() => {

  console.log("Proceso asíncrono");

}, 2000);
```

console.log("Fin");

Resultado: "Inicio", "Fin", y después de 2 segundos, "Proceso asíncrono".

2. Callbacks

Un **callback** es una función que se pasa como argumento a otra función y se ejecuta una vez que se completa una tarea.
Ventaja: Facilita la ejecución de código una vez completado un proceso asíncrono.
Desventaja: El "callback hell", cuando hay muchas funciones anidadas.

Ejemplo:

```
function proceso(callback) {

  console.log("Proceso iniciado");
```

```
  setTimeout(() => {
    console.log("Proceso completado");
    callback();
  }, 3000);
}

proceso(() => {
  console.log("Callback ejecutado");
```

});

3. Promesas

Una **promesa** es un objeto que representa la eventual finalización (o falla) de una operación asíncrona.

- Estados de una promesa: `pendiente`, `resuelta`, o `rechazada`.
- Se gestionan con `.then()` para manejar el éxito y `.catch()` para errores.

Ejemplo:

```
let promesa = new Promise((resolve, reject) => {
  setTimeout(() => resolve("Promesa cumplida"), 2000);
});

promesa.then((resultado) => {
  console.log(resultado);
}).catch((error) => {
  console.log(error);
```

});

Resultado: Después de 2 segundos, "Promesa cumplida".

4. Async/Await

Async/Await es una forma más clara y sencilla de manejar promesas. `async` convierte una función en asíncrona, y `await` pausa la ejecución hasta que una promesa se resuelva.

Ejemplo:

```
async function obtenerDatos() {
  try {
    let respuesta = await fetch("https://api.example.com/datos");
    let datos = await respuesta.json();
    console.log(datos);
  } catch (error) {
    console.log("Error:", error);
  }
}

obtenerDatos();
```

Aquí, `await` detiene la ejecución hasta que la promesa se cumple.

5. Web Workers

Los **Web Workers** permiten ejecutar scripts en segundo plano sin bloquear la interfaz principal de la página. Son útiles para realizar tareas complejas que consumen tiempo, como cálculos grandes o procesamiento intensivo.

Ejemplo:

```
// main.js
```

```
let worker = new Worker("worker.js");
worker.postMessage("Iniciar trabajo");


worker.onmessage = function(event) {
  console.log("Resultado del worker:", event.data);
};


// worker.js
onmessage = function(event) {
  let resultado = event.data + " completado";
  postMessage(resultado);
};
```

El código principal sigue funcionando sin bloqueo mientras el `Web Worker` ejecuta su tarea en segundo plano.