**programación asíncrona**

La programación asíncrona es una técnica que permite a tu programa iniciar una tarea de larga duración y seguir respondiendo a otros eventos mientras esa tarea se ejecuta, en lugar de tener que esperar hasta que esa tarea haya terminado. Una vez que dicha tarea ha finalizado, tu programa presenta el resultado.

**Set time out**

**¿Cómo funciona setTimeout en JavaScript?**

Esta es la sintaxis de setTimeout():

* **Argumento 1:** función a ejecutar al terminar el temporizador
* **Argumento 2:** tiempo de espera en milisegundos

setTimeout(function callback a ejecutar, temporizador en milisegundos);

Si el temporizador en milisegundos es omitido, entonces la función callback será ejecutada inmediatamente sin esperar nada.

function greetings(){ console.log("Hola, Mundo!"); }setTimeout(greetings, 1500);

Seguramente, necesitarás pasarle argumentos a la función callback. Por ello, la sintaxis de setTimeout expande sus argumentos para enviar tantos parámetros como necesite la función callback.

setTimeout(function, milisegundos, parametro, parametro, ...); // sintaxis

**// Definimos una función que queremos ejecutar después de un retraso**

function saludo() {

console.log('¡Hola! Este mensaje se muestra después de 3 segundos.');

}

**// Usamos setTimeout para llamar a la función saludo después de 3000 milisegundos (3 segundos)**

setTimeout(saludo, 3000);

**set Interval**

setInterval es una función en JavaScript que se utiliza para ejecutar repetidamente una función o un bloque de código, con un intervalo fijo de tiempo entre cada ejecución. A diferencia de setTimeout, que ejecuta el código solo una vez después de un retraso, setInterval continúa ejecutando el código indefinidamente a intervalos regulares hasta que se detiene explícitamente utilizando clearInterval.

**// Definimos una función que queremos ejecutar en intervalos regulares**

function mostrarHoraActual() {

let fecha = new Date();

console.log('La hora actual es: ' + fecha.toLocaleTimeString());

}

**// Usamos setInterval para llamar a la función mostrarHoraActual cada 1000 milisegundos (1 segundo)**

let intervalo = setInterval(mostrarHoraActual, 1000);

**// Si en algún momento queremos detener este intervalo, podemos usar clearInterval**

**// Por ejemplo, después de 10 segundos, detenemos el intervalo**

setTimeout(() => {

clearInterval(intervalo);

console.log('Intervalo detenido.');

}, 10000);

En este ejemplo, **mostrarHoraActual** se ejecuta cada segundo. Este comportamiento continúa hasta que **clearInterval** se llama con el identificador del intervalo (en este caso, **intervalo**) como argumento, deteniendo las ejecuciones futuras.

**asincronía y el even loop**

**Asincronía**

La asincronía permite que ciertas operaciones se realicen en "segundo plano" y que sus resultados se manejen solo cuando estén listos, mientras el hilo principal continúa ejecutando el resto del código sin interrupciones. Esto se logra mediante callbacks, promesas (Promises), y la sintaxis async/await, permitiendo a los desarrolladores escribir código que espera por ciertas operaciones sin detener la ejecución de otras partes del programa.

**Event Loop**

El Event Loop (bucle de eventos) es el mecanismo que facilita la ejecución de código, la recolección de eventos y la ejecución de sub-tareas dentro de un ambiente asincrónico. Funciona siguiendo un modelo simple pero poderoso:

1. **Pila de Ejecución (Call Stack):** Donde se apilan las funciones que están siendo ejecutadas. Cuando se llama a una función, se coloca en la pila; cuando termina, se saca de la pila.
2. **Heap:** Un área de memoria desestructurada para la asignación dinámica de objetos.
3. **Cola de Mensajes/Tareas (Message/Task Queue):** Aquí se encolan los mensajes o eventos cuando una operación asincrónica se completa y su callback está listo para ser ejecutado.

El Event Loop revisa la pila de ejecución; si está vacía, toma el primer evento de la cola de mensajes y lo ejecuta, lo que puede involucrar más llamadas a funciones que se apilan y ejecutan inmediatamente. Este proceso se repite, permitiendo que JavaScript ejecute tareas asincrónicas, maneje eventos y ejecute callbacks de manera eficiente.

**¿Para qué sirven?**

La combinación de asincronía y el Event Loop permite a JavaScript ser altamente eficiente y no bloqueante en la ejecución de código. Esto significa que operaciones intensivas o que toman mucho tiempo, como cargar un archivo o hacer una solicitud de red, no detienen el flujo general del programa. Esto es crucial en el desarrollo web para crear aplicaciones responsivas y de alto rendimiento, donde múltiples tareas (como animaciones, cargas de datos, etc.) necesitan ocurrir al mismo tiempo sin interrumpir la experiencia del usuario.

**Ejemplo**

console.log('Primero');

setTimeout(() => {

console.log('Segundo');

}, 0);

console.log('Tercero');

setTimeout(() => {

console.log('Cuarto');

}, 0);

console.log('Quinto');

**callback**

Un callback es una función que se pasa como argumento a otra función con la intención de que sea ejecutada más tarde. Los callbacks son muy utilizados en JavaScript para operaciones asincrónicas, permitiendo que el código continúe ejecutándose sin bloqueos mientras espera que algo ocurra, como la respuesta de una solicitud HTTP, la lectura de un archivo, o simplemente el transcurso de tiempo con setTimeout. Una vez que el evento para el cual el callback fue diseñado ocurre, el callback es llamado para manejar el resultado.

**¿Para qué sirve?**

Los callbacks sirven principalmente para:

1. **Manejar la asincronía:** Permiten ejecutar código después de que una tarea asincrónica ha sido completada.
2. **Encapsulación de comportamiento:** Permiten definir el código que se ejecutará después de completar una determinada tarea, sin necesidad de escribir el código directamente en la función que realiza la tarea.
3. **Control de flujo:** En aplicaciones complejas, facilitan la gestión del orden en que se ejecutan las tareas.

**Ejemplo con setTimeout**

ejemplo básico que muestra cómo se utiliza un callback con **setTimeout** para ejecutar un bloque de código después de un cierto retardo:

function mostrarMensaje() {

console.log('Este mensaje se muestra después de 3 segundos');

}

/**/ Usamos setTimeout para retrasar la ejecución de la función mostrarMensaje**

setTimeout(mostrarMensaje, 3000);

**Ejemplo con Operaciones Asincrónicas**

Vamos a ver un ejemplo más complejo que simula una operación de lectura de base de datos asincrónica:

function obtenerUsuarioDesdeBD(idUsuario, callback) {

**// Simula la obtención de un usuario de una base de datos de manera asincrónica**

setTimeout(() => {

**// Aquí es donde se "obtiene" el usuario**

const usuario = {

id: idUsuario,

nombre: 'Juan'

};

**// Una vez obtenido el usuario, llamamos al callback pasándole el usuario**

callback(usuario);

}, 2000);

}

obtenerUsuarioDesdeBD(1, function(usuario) {

console.log('Usuario obtenido:', usuario.nombre);

});

**Promesas**

Las **Promesas** en JavaScript son objetos que representan el resultado eventual (ya sea éxito o fallo) de una operación asincrónica. Ofrecen una alternativa más potente y flexible a los callbacks para manejar asincronía. Con las promesas, puedes encadenar operaciones asincrónicas de manera más limpia y manejar errores de manera más coherente.

**¿Para qué sirve?**

Las promesas se utilizan para:

1. **Manejar resultados de operaciones asincrónicas:** Permiten escribir código que espera de forma explícita por el éxito o fallo de una tarea asincrónica.
2. **Encadenamiento de operaciones asincrónicas:** Facilitan la realización de varias tareas asincrónicas de forma secuencial sin anidar callbacks, mejorando la legibilidad del código.
3. **Manejo de errores:** Proporcionan una forma estructurada de capturar errores que puedan ocurrir en cualquiera de las operaciones asincrónicas encadenadas.

**Ejemplo**

ejemplo simple que muestra cómo crear y utilizar una promesa para realizar una tarea asincrónica:

let promesaDeCafe = new Promise((resolve, reject) => {

**// Simulando una tarea asincrónica, como obtener café**

const hayCafe = true; // Cambia esto a false para simular un fallo

setTimeout(() => {

if (hayCafe) {

resolve('Café listo');

} else {

reject('No hay café');

}

}, 1000);

});

promesaDeCafe.then((resultado) => {

console.log(resultado); // Maneja el éxito

}).catch((error) => {

console.error(error); // Maneja el error

});

**el Async y el await**

* **async**: Imagina que tienes una lista de tareas para hacer durante el día, pero hay una tarea que depende de que la tienda esté abierta, y no sabes exactamente cuándo abrirá. En lugar de esperar frente a la tienda y no hacer nada más, decides seguir con tus otras tareas y volver a la tienda más tarde. En programación, cuando marcas una función con **async**, estás diciendo: "Esta tarea puede tomar algo de tiempo, así que voy a continuar con otras cosas mientras tanto".
* **await**: Ahora, cuando finalmente vuelves a la tienda y resulta que está abierta, decides esperar a que te entreguen lo que necesitas. En este momento, estás pausando tus otras tareas para enfocarte en esta única tarea. Usar **await** en una función **async** es como decir: "Voy a esperar aquí hasta que termine esta tarea específica antes de seguir con las siguientes".

Así que, básicamente, **async** y **await** trabajan juntos para ayudar a manejar tareas que pueden tomar un poco más de tiempo, como pedir datos de internet, sin tener que detener todo lo demás que tu programa esté haciendo.

**// Función que simula la espera de datos de, por ejemplo, una API**

function obtenerDatos() {

return new Promise(resolve => {

setTimeout(() => {

resolve("Aquí están tus datos");

}, 2000); **// Espera 2 segundos antes de completar la promesa**

});

}

**// Función asíncrona que usa await para esperar los datos**

async function mostrarDatos() {

console.log("Esperando por los datos...");

const datos = await obtenerDatos(); **// Esperamos a que obtenerDatos() se resuelva**

console.log(datos); **// Mostramos los datos obtenidos**

}

mostrarDatos();

**¿Qué es el procesamiento de un solo hilo?**

El procesamiento de un solo hilo en JavaScript se refiere al hecho de que el motor de JavaScript ejecuta una sola tarea a la vez en un solo hilo de ejecución. Esto significa que las instrucciones de JavaScript se ejecutan secuencialmente, una después de la otra, sin la capacidad de realizar múltiples operaciones al mismo tiempo.

**¿Para qué sirve el procesamiento de un solo hilo en JavaScript?**

El procesamiento de un solo hilo en JavaScript es útil porque simplifica el manejo de la concurrencia y evita problemas como las condiciones de carrera y las inconsistencias de datos. JavaScript está diseñado para ser asíncrono, lo que significa que puede realizar operaciones como solicitudes de red o temporizadores sin bloquear el hilo principal de ejecución. Esto se logra a través de mecanismos como devoluciones de llamada (callbacks), promesas y async/await.

Sin embargo, aunque JavaScript es de un solo hilo, existen formas de lograr cierto grado de concurrencia y procesamiento paralelo, como el uso de Web Workers para ejecutar scripts en segundo plano, así como el aprovechamiento de la capacidad de procesamiento de los navegadores modernos para realizar operaciones intensivas en la CPU de manera eficiente.

// Función que simula una tarea asíncrona

function tareaAsincrona(callback) {

// Simulamos un retraso de 2 segundos

setTimeout(function() {

// Después del retraso, ejecutamos la devolución de llamada con un mensaje

callback("Tarea asíncrona completada");

}, 2000);

}

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

// Llamamos a la función tareaAsincrona y pasamos una función de devolución de llamada como argumento

tareaAsincrona(function(mensaje) {

// Esta función se ejecutará después de que la tarea asíncrona haya terminado

console.log(mensaje);

});

console.log("Fin de la tarea");

**procesamiento de multihilo**

**Procesamiento de multihilo en JavaScript**

En JavaScript, el modelo de concurrencia es basado en un solo hilo, lo que significa que normalmente solo se puede ejecutar una tarea a la vez. Sin embargo, hay formas de lograr el procesamiento de multihilo a través de técnicas como Web Workers.

**¿Qué son los Web Workers?**

Los Web Workers son scripts que se ejecutan en segundo plano, en paralelo al hilo principal de JavaScript en un navegador web. Permiten ejecutar tareas en segundo plano sin bloquear la interfaz de usuario (UI), lo que puede mejorar la capacidad de respuesta de una aplicación web y permitir el procesamiento de multihilo.

**¿Para qué sirven los Web Workers en JavaScript?**

Los Web Workers son útiles cuando se necesita realizar tareas intensivas en el navegador, como procesamiento de imágenes, cálculos complejos o peticiones de red, sin afectar la capacidad de respuesta de la interfaz de usuario. Permiten dividir el trabajo en múltiples hilos y aprovechar los múltiples núcleos de CPU disponibles en la computadora del usuario.

**Ejemplo de uso de Web Workers en JavaScript:**

**// Crear un nuevo Web Worker desde un archivo externo**

const myWorker = new Worker('worker.js');

**// Escuchar mensajes del Web Worker**

myWorker.onmessage = function(event) {

console.log('Mensaje del Web Worker:', event.data);

};

**// Enviar un mensaje al Web Worker**

myWorker.postMessage('Hola desde el hilo principal');

**// Contenido del archivo "worker.js"**

**// Escuchar mensajes del hilo principal**

self.onmessage = function(event) {

console.log('Mensaje del hilo principal:', event.data);

**// Realizar alguna tarea en segundo plano**

const result = event.data \* 2;

**// Enviar el resultado de vuelta al hilo principal**

self.postMessage(result);

};

**Operaciones de entrada y operaciones de salida**

**¿Qué son las operaciones de entrada y salida en JavaScript?**

Las operaciones de entrada y salida en JavaScript incluyen acciones como leer o escribir datos desde/hacia archivos, enviar o recibir datos a través de una red, o interactuar con el usuario a través de la entrada estándar (teclado) y la salida estándar (pantalla).

**¿Para qué sirven las operaciones de entrada y salida en JavaScript?**

Las operaciones de entrada y salida son fundamentales para interactuar con el entorno externo en el que se ejecuta el código JavaScript. Por ejemplo:

* Leer y escribir archivos: Permite a una aplicación leer datos de archivos de almacenamiento o escribir datos en ellos, lo que es esencial para manejar datos persistentes.
* Realizar solicitudes HTTP: Permite a una aplicación web comunicarse con servidores remotos, lo que es necesario para cargar recursos, enviar datos a servicios web, etc.
* Interactuar con el usuario: Permite a una aplicación recibir entrada del usuario (por ejemplo, a través de formularios en una página web) y mostrar información al usuario (por ejemplo, mensajes de error o resultados de operaciones).

**Ejemplo**

**// Ejemplo de entrada (lectura de datos del usuario)**

let nombre = prompt("Por favor, ingresa tu nombre:");

console.log("¡Hola, " + nombre + "! Bienvenido a nuestra página.");

**// Ejemplo de salida (mostrar información al usuario)**

console.log("Este es un mensaje de prueba.");

**// Ejemplo de operación asíncrona (simulando una solicitud de red)**

console.log("Realizando solicitud de red...");

setTimeout(function() {

console.log("¡Solicitud completada!");

}, 2000); **// Simula un retraso de 2 segundos**

**¿Qué son las operaciones bloqueantes?**

Las operaciones bloqueantes son aquellas que detienen la ejecución del código hasta que la operación se completa. Durante este tiempo, el hilo de ejecución se queda inactivo, lo que significa que no puede continuar con otras tareas. En JavaScript, las operaciones de entrada y salida (I/O) como la lectura de archivos o las solicitudes de red síncronas suelen ser operaciones bloqueantes.

**¿Para qué sirven las operaciones bloqueantes en JavaScript?**

Las operaciones bloqueantes son útiles cuando se necesita asegurar que una operación se complete antes de continuar con el siguiente paso del proceso. Sin embargo, pueden tener un impacto negativo en la capacidad de respuesta y el rendimiento del sistema, especialmente en aplicaciones de servidor donde se deben manejar múltiples solicitudes simultáneas.

**¿Qué son las operaciones no bloqueantes?**

Las operaciones no bloqueantes son aquellas que permiten que el código continúe ejecutándose mientras la operación se está realizando. Esto significa que el hilo de ejecución no se detiene y puede seguir realizando otras tareas mientras espera la finalización de la operación. En JavaScript, las operaciones asíncronas, como las devoluciones de llamada (callbacks), las promesas y async/await, se utilizan para realizar operaciones no bloqueantes.

**¿Para qué sirven las operaciones no bloqueantes en JavaScript?**

Las operaciones no bloqueantes son útiles cuando se desea optimizar la eficiencia y la capacidad de respuesta de una aplicación, especialmente en aplicaciones de servidor donde se deben manejar múltiples solicitudes simultáneas. Permiten realizar operaciones de manera eficiente sin detener la ejecución del código, lo que puede mejorar significativamente el rendimiento del sistema.

**Que es lifo**

En JavaScript, el término LIFO se aplica comúnmente en el contexto de estructuras de datos, específicamente en la implementación de pilas.

Una pila en JavaScript es una estructura de datos que sigue el principio LIFO: el último elemento que se añade a la pila es el primero en ser retirado. Esto significa que los elementos se agregan y eliminan de la pila en el mismo extremo, conocido como "top" o "cima" de la pila.

Las pilas en JavaScript se pueden implementar fácilmente utilizando un array. Algunas operaciones comunes en una pila son:

* **push:** Añade un elemento a la cima de la pila.
* **pop:** Elimina y devuelve el elemento en la cima de la pila.
* **peek:** Devuelve el elemento en la cima de la pila sin eliminarlo.
* **isEmpty:** Verifica si la pila está vacía.

Las pilas son útiles en muchas situaciones de programación, como la evaluación de expresiones aritméticas, la gestión de llamadas a funciones (stack frames), el procesamiento de historiales de navegación en aplicaciones web, y en algoritmos de búsqueda y recorrido de árboles, entre otros casos.

class Pila {

constructor() {

this.items = [];

}

push(elemento) {

this.items.push(elemento);

}

pop() {

if (this.isEmpty()) {

return "La pila está vacía";

}

return this.items.pop();

}

peek() {

return this.items[this.items.length - 1];

}

isEmpty() {

return this.items.length === 0;

}

print() {

console.log(this.items.toString());

}

}

// Ejemplo de uso de la pila

let pila = new Pila();

pila.push(1);

pila.push(2);

pila.push(3);

pila.print(); // Output: 1,2,3

console.log(pila.pop()); // Output: 3

console.log(pila.peek()); // Output: 2

pila.print(); // Output: 1,2