

Recuerden poner a  
grabar la clase



Clase 19

# Asincronismo

Diplomatura UNTREF



# Temario

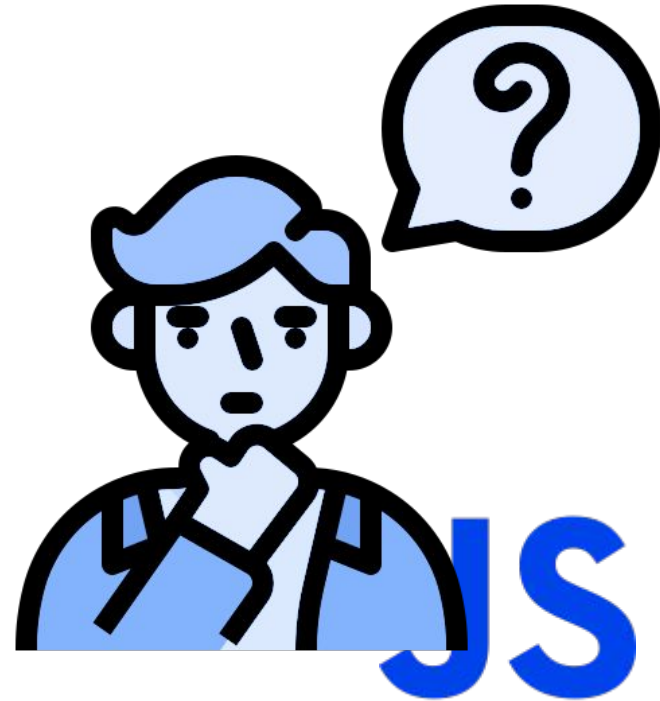
## Asincronismo:

- uso de setTimeout y setInterval
- clearTimeout y clearInterval
- comportamiento del motor JS funcionando de manera asincrónica
- Estructuras de control de ejecución:
  - Try
  - Catch
  - Finally

# Asincronismo

# Asincronismo

El asincronismo en JavaScript se refiere a la capacidad del lenguaje para ejecutar ciertas tareas de manera no bloqueante. En un programa JavaScript, las tareas pueden ser síncronas o asíncronas.



# sincronismo vs Asincronismo

## Síncrono:

En una operación síncrona, el programa se bloquea hasta que la tarea se completa.

Esto significa que si tienes una tarea que lleva tiempo, como cargar una imagen grande, el programa no continuará ejecutando otras instrucciones hasta que la imagen se haya cargado por completo.

Esto puede hacer que la interfaz de usuario se vuelva no receptiva.



## Asíncrono:

En una operación asíncrona, el programa continúa ejecutando otras instrucciones sin esperar a que la tarea se complete.

En lugar de bloquear el hilo de ejecución principal, se utiliza un mecanismo llamado "callbacks", "promesas" o "async/await" para manejar la finalización de la tarea en un momento posterior.

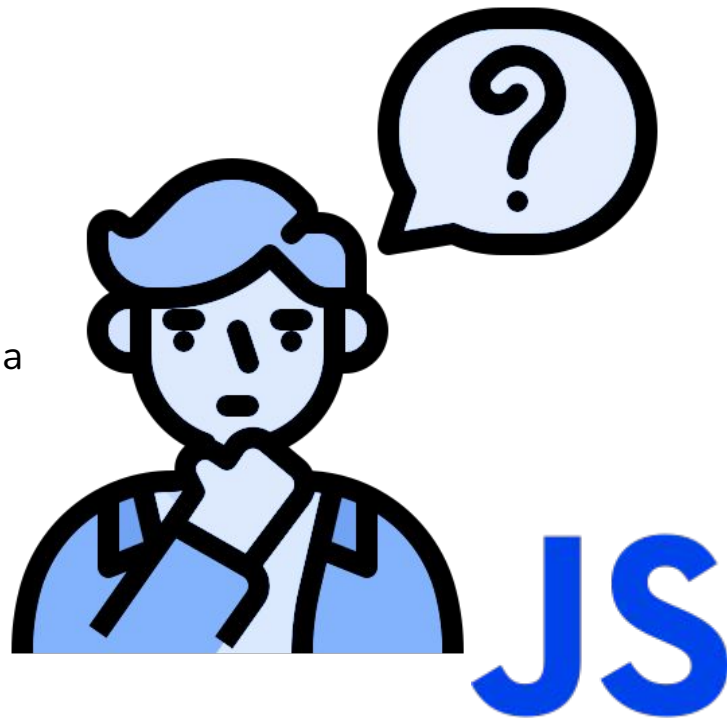
Esto permite que la interfaz de usuario siga siendo receptiva y que las operaciones de larga duración se ejecuten en segundo plano.

# Sincronismo vs. Asincronismo

El asincronismo es especialmente importante en JavaScript debido a su entorno de ejecución basado en un solo hilo (thread) en el navegador.

Si una tarea bloqueara el hilo principal, la experiencia del usuario se vería afectada negativamente.

Por lo tanto, JavaScript utiliza técnicas asíncronas para garantizar que las operaciones largas no bloqueen el hilo principal y que el programa siga siendo reactivo.



setTimeout y setInterval

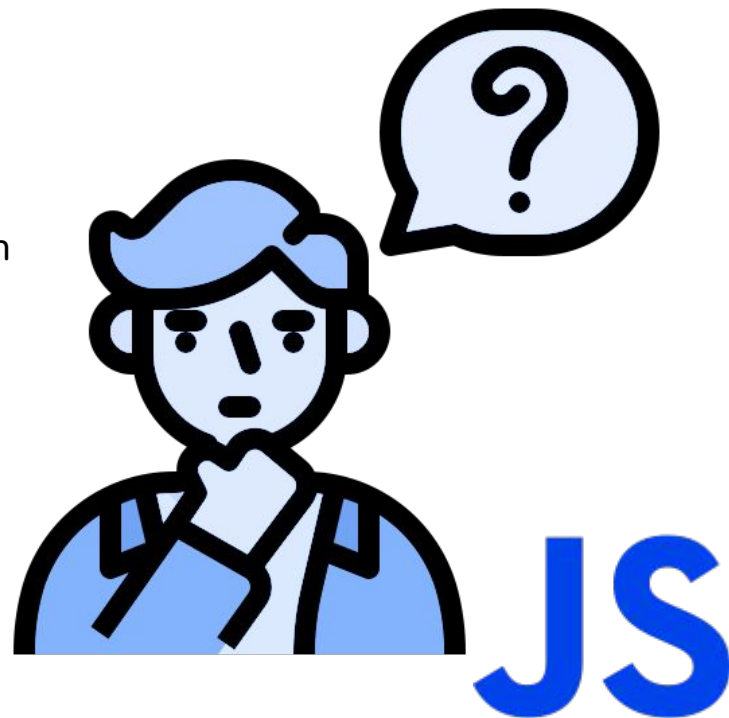


**setTimeout**

# setTimeout

**setTimeout** es una función en JavaScript que te permite programar la ejecución de una función (o la ejecución de un fragmento de código) después de un cierto período de tiempo especificado, medido en milisegundos.

En otras palabras, puedes decirle a JavaScript que ejecute una **función "X"** milisegundos después de que se haya llamado a **setTimeout**

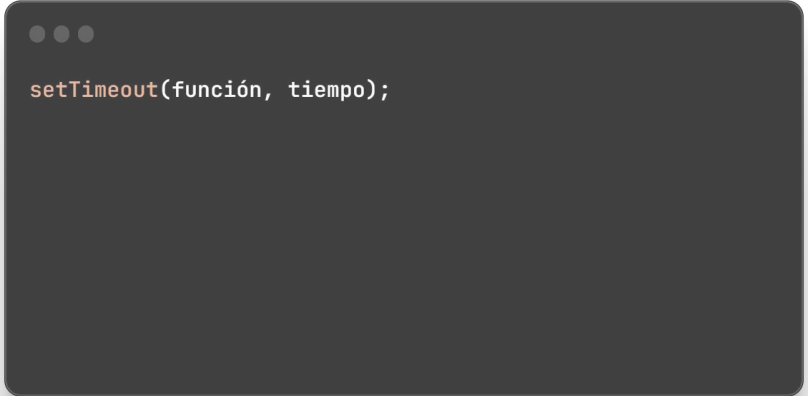


# setTimeout, sintaxis

La **sintaxis básica de setTimeout**:

**función:** La función que deseas ejecutar después de que haya transcurrido el tiempo especificado.

**tiempo:** El tiempo en milisegundos que debes esperar antes de que se ejecute la función.



```
setTimeout(función, tiempo);
```

# setTimeout en acción

## Ejemplo con setTimeout

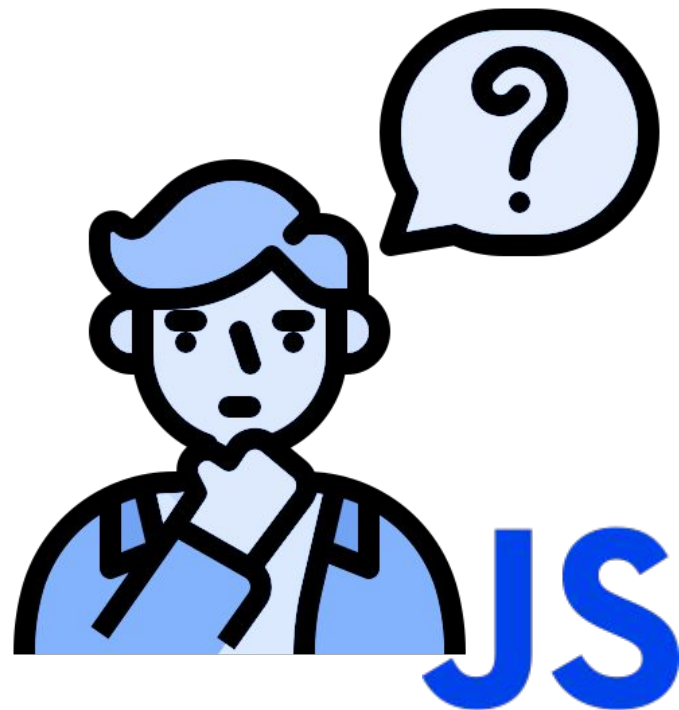
**mostrarMensaje** es una función que se ejecutará después de esperar 2000 milisegundos (2 segundos) debido a **setTimeout**.

```
// Ejemplo: Mostrar un mensaje después de 2 segundos
function mostrarMensaje() {
  console.log("Han pasado 2 segundos");
}

setTimeout(mostrarMensaje, 2000); // 2000 milisegundos (2 segundos)
```

## setTimeout consideraciones

**setTimeout** es útil cuando deseas retrasar la ejecución de una tarea, como mostrar un mensaje emergente después de que el usuario haya realizado una acción o implementar una lógica de temporización en una aplicación.

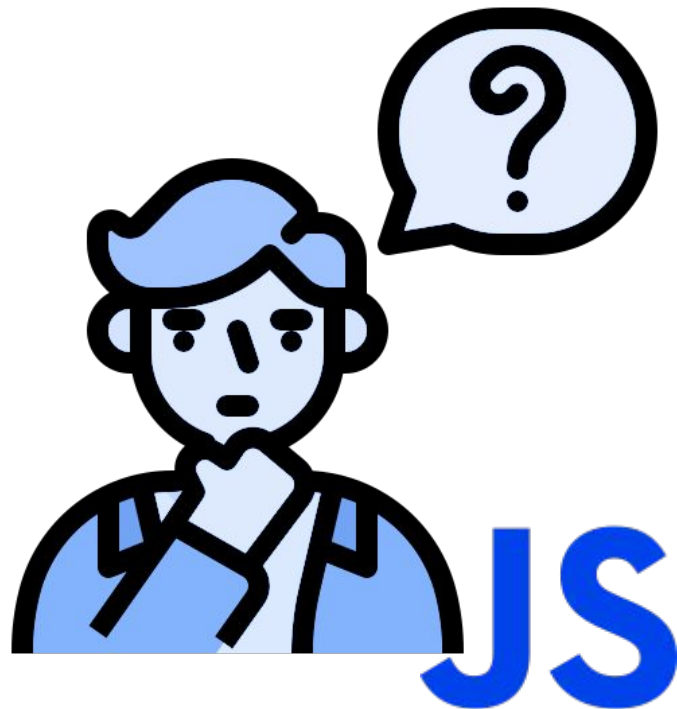


## setTimeout consideraciones

Es importante destacar que **setTimeout** no detiene la ejecución del resto del código;

Simplemente, programa la función especificada para ejecutarse después del tiempo indicado.

Esto significa que el código continuará ejecutándose después de la llamada a **setTimeout** sin esperar a que la función se ejecute.



# setTimeout consideraciones adicionales

## Valor de retorno:

La función `setTimeout` devuelve un identificador único que puede utilizarse más adelante para cancelar la ejecución planificada mediante `clearTimeout`.

Esto es útil si necesitas detener la ejecución programada antes de que ocurra.

```
const timeoutId = setTimeout(function() {  
  console.log("Esta función se ejecutará después de un  
  tiempo");  
}, 1000);  
  
// Para cancelar la ejecución programada  
clearTimeout(timeoutId);
```

# setTimeout consideraciones adicionales

## Funciones anónimas:

A menudo, se utiliza una función anónima como argumento para setTimeout.

Esto es útil cuando deseas ejecutar una tarea simple sin definir una función separada.

```
setTimeout(function() {  
    console.log("Ejecutando una función anónima después de  
un tiempo");  
}, 2000);
```



# setTimeout consideraciones adicionales

## Variables en el ámbito:

Cuando utilizas setTimeout, ten en cuenta el ámbito (scope) de las variables.

La función que se ejecutará posteriormente tendrá acceso a las variables dentro del ámbito donde se definió setTimeout.

Esto puede llevar a resultados inesperados si no se maneja adecuadamente.

```
let mensaje = "¡Hola!";

setTimeout(function() {
  console.log(mensaje); // Esto mostrará "¡Hola!" porque
  la función tiene acceso al ámbito externo
}, 1000);

mensaje = "¡Adiós!"; // Cambiamos el valor de la variable
después de programar setTimeout
```

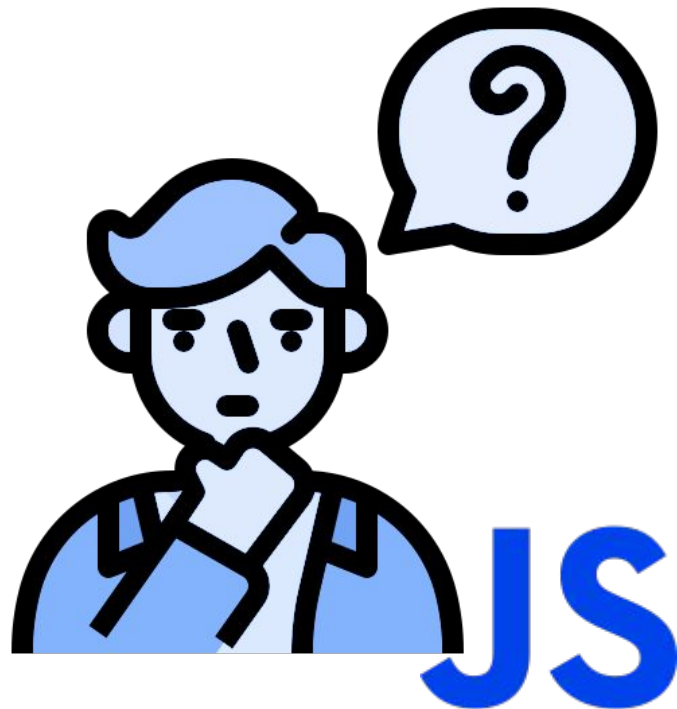
# setTimeout consideraciones adicionales

## Precisión del tiempo:

El tiempo especificado en `setTimeout` no es siempre exacto.

Puede haber una ligera variación debido a la capacidad de programación del sistema operativo y otros factores.

Por lo tanto, no debes depender de `setTimeout` para tareas críticas en tiempo real que requieran una precisión extrema.

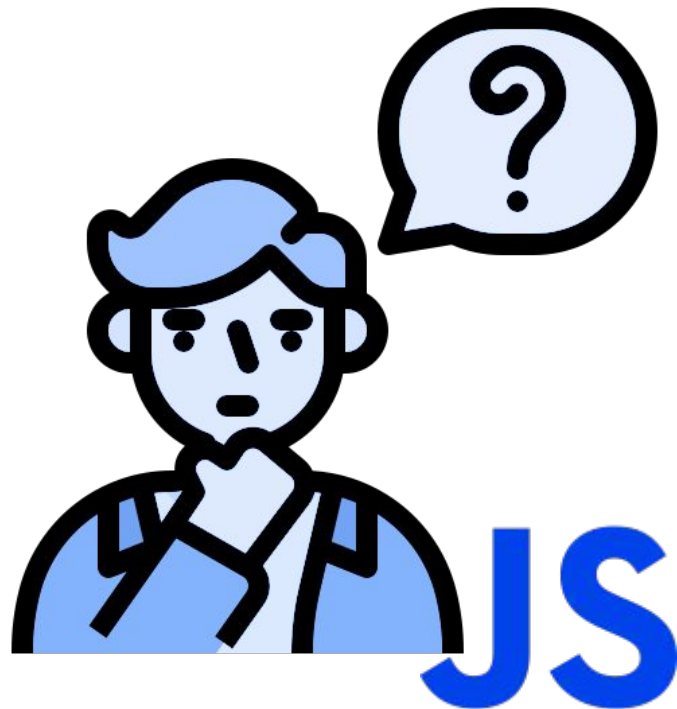


**setInterval**

# setInterval

**setInterval** es una función que se utiliza para ejecutar una función o un fragmento de código repetidamente a intervalos regulares de tiempo, que se especifican en milisegundos.

A diferencia de **setTimeout**, que ejecuta una función después de un retraso único, **setInterval** ejecuta la función a intervalos repetidos, una y otra vez, hasta que se detiene explícitamente.



# setInterval, sintaxis

La **sintaxis básica de setInterval**:

**función:** La función que deseas ejecutar a intervalos regulares.

**tiempo:** El tiempo en milisegundos entre cada ejecución de la función.

```
setInterval(función, tiempo);
```

# setInterval, en acción

En el siguiente ejemplo, **mostrarMensaje** es una función que se ejecutará cada segundo debido al uso de **setInterval**.

La variable **intervalo** contiene un identificador que se puede utilizar para detener la ejecución repetida de la función en cualquier momento mediante **clearInterval**.

```
// Ejemplo: Mostrar un mensaje cada segundo
function mostrarMensaje() {
  console.log("Este mensaje se mostrará cada segundo");
}

const intervalo = setInterval(mostrarMensaje, 1000); //
1000 milisegundos (1 segundo)
```

# setInterval, en acción

Es importante recordar que `setInterval` seguirá ejecutando la función indefinidamente a menos que se detenga utilizando `clearInterval`.

Por lo tanto, debes tener cuidado al usar `setInterval` para tareas que puedan ejecutarse de manera indefinida para evitar un uso excesivo de recursos del sistema.

```
// Detener la ejecución repetida después de 5 segundos
setTimeout(function() {
  clearInterval(intervalo);
  console.log("La ejecución se ha detenido después de 5 segundos");
}, 5000); // 5000 milisegundos (5 segundos)
```

**`setInterval` es útil para realizar tareas periódicas o para crear animaciones y actualizaciones en tiempo real en aplicaciones web.**

clearTimeout y clearInterval

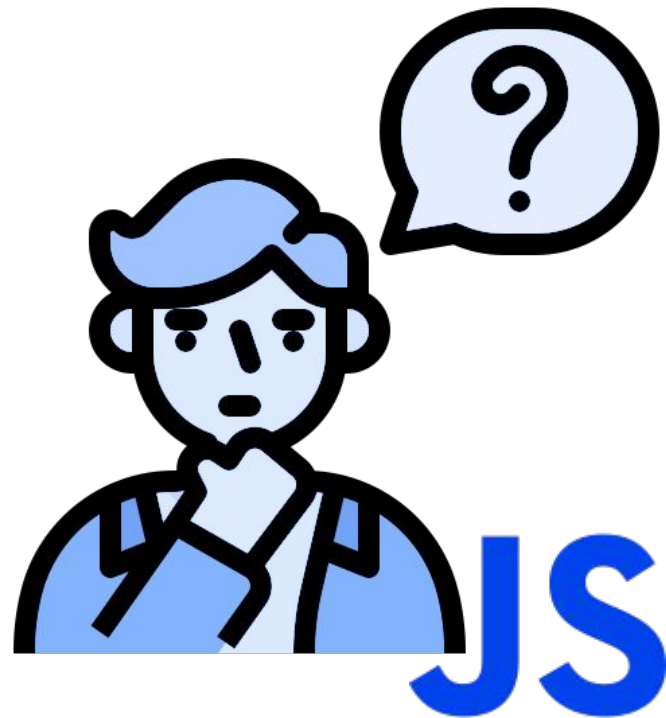


**clearTimeout**

# clearTimeout

**clearTimeout** es una función en JavaScript que se utiliza para cancelar la ejecución de una tarea programada que previamente hayas configurado con **setTimeout**.

En otras palabras, te permite detener una función o un fragmento de código que estaba esperando para ejecutarse en un tiempo futuro.



# clearTimeout, sintaxis

**identificador:** Es el identificador único que se devuelve cuando se llama a **setTimeout** para programar la ejecución de una función en el futuro.

Este identificador se utiliza para identificar y cancelar la tarea programada

```
clearTimeout(identificador);
```

## clearTimeout, en acción

En este ejemplo, **clearTimeout** se utiliza para cancelar la ejecución programada antes de que ocurra.

Esto evita que la función en setTimeout se ejecute después de 3 segundos.

```
// Programar una función para ejecutarse después de 3 segundos
const timeoutId = setTimeout(function() {
  console.log("Esta función se ejecutará después de 3 segundos");
}, 3000); // 3000 milisegundos (3 segundos)

// Cancelar la ejecución programada
clearTimeout(timeoutId);
```

# clearTimeout, utilidad

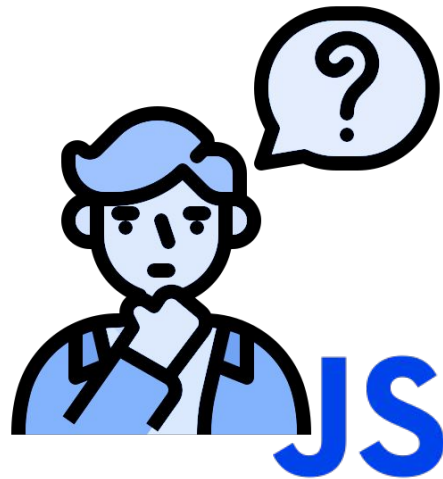
**clearTimeout** es útil cuando necesitas cancelar tareas programadas en situaciones como:

El usuario realiza una acción que requiere que una tarea programada se detenga.

Detectas ciertas condiciones en tu código que indican que la tarea programada ya no es necesaria.

Deseas evitar que una tarea programada se ejecute si ocurre un error o una excepción en tu código.

**clearTimeout** es una función importante para el manejo de tareas programadas en JavaScript, ya que te permite controlar y detener la ejecución de funciones en un tiempo futuro si es necesario.

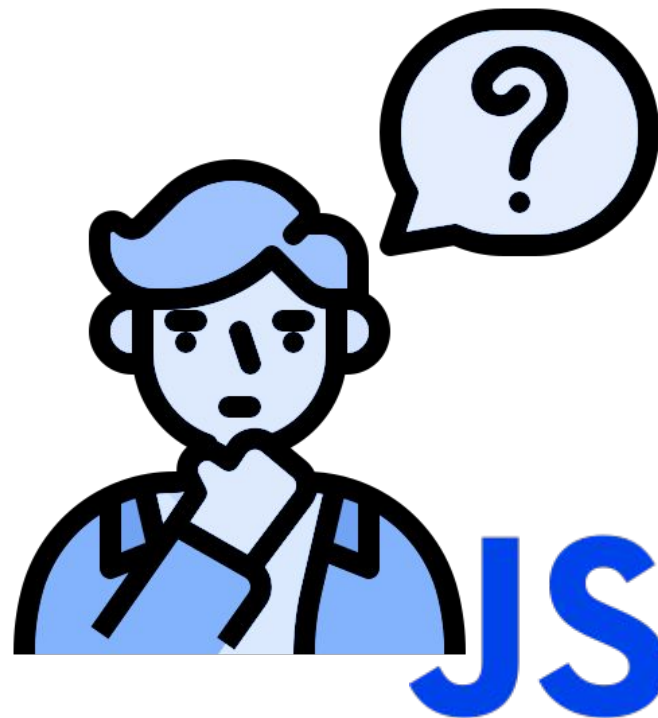


**clearInterval**

# clearInterval

es una función en JavaScript que se utiliza para detener la ejecución repetida de una función que previamente hayas configurado con setInterval.

Esta función te permite cancelar intervalos de tiempo regulares que se ejecutan periódicamente.



# clearInterval, sintaxis

La sintaxis básica de clearInterval es la siguiente:

**identificador:** Es el identificador único que se devuelve cuando se llama a **setInterval** para programar la ejecución periódica de una función.

Este identificador se utiliza para identificar y detener el intervalo de tiempo programado.

```
clearInterval(identificador);
```



# clearInterval, en acción

**clearInterval** se utiliza para detener la ejecución repetida del intervalo creado previamente con **setInterval**.

La función se detiene después de 5 segundos mediante **setTimeout**.

```
// Programar una función para ejecutarse cada segundo
const intervalo = setInterval(function() {
  console.log("Este mensaje se mostrará cada segundo");
}, 1000); // 1000 milisegundos (1 segundo)

// Detener la ejecución repetida después de 5 segundos
setTimeout(function() {
  clearInterval(intervalo);
  console.log("La ejecución se ha detenido después de 5 segundos");
}, 5000); // 5000 milisegundos (5 segundos)
```

# clearInterval, recomendaciones

**clearInterval** es útil cuando deseas detener la ejecución repetida de una función a intervalos regulares, por ejemplo:

Cuando una tarea periódica ya no es necesaria.

Cuando el usuario realiza una acción que requiere que se detenga un proceso repetido.

Para evitar que un proceso en segundo plano siga ejecutándose después de que el usuario abandone una página web.



# Comportamiento del motor JS funcionando de manera asincrona

# Comportamiento asincrono

El comportamiento asincrónico en JavaScript se refiere a la capacidad del lenguaje para realizar operaciones sin bloquear la ejecución de otras operaciones en curso.



# Comportamiento asincrono

Esto es fundamental para crear aplicaciones web responsivas y eficientes, ya que permite que ciertas tareas, como solicitudes de red, lectura/escritura de archivos o procesamiento costoso, se realicen en segundo plano sin interrumpir la ejecución principal del programa.



# Comportamiento asincrono

El mecanismo principal que permite esta asincronía es la callback, las promesas y, más recientemente, las funciones asíncronas (async/await).



# Comportamiento asincrono

## Callbacks:

Las callbacks son funciones que se pasan como argumentos a otras funciones.

Se utilizan para especificar qué debe hacerse cuando una operación asincrónica se complete.

```
function descargarDatos(url, callback) {  
  // Simular una solicitud de red asincrónica  
  setTimeout(function() {  
    // Datos descargados  
    const datos = 'Estos son los datos descargados';  
    callback(datos);  
  }, 1000);  
}  
  
descargarDatos('https://ejemplo.com/datos', function(datos) {  
  console.log(datos);  
});
```

# Comportamiento asincrono

## Promesas:

Las promesas son un patrón más estructurado y flexible para manejar operaciones asincrónicas.

Representan un valor que puede estar disponible ahora, en el futuro o nunca. Permiten encadenar operaciones asincrónicas de una manera más legible.

```
function descargarDatos(url, callback) {  
  // Simular una solicitud de red asincrónica  
  setTimeout(function() {  
    // Datos descargados  
    const datos = 'Estos son los datos descargados';  
    callback(datos);  
  }, 1000);  
}  
  
descargarDatos('https://ejemplo.com/datos', function(datos) {  
  console.log(datos);  
});
```



# Comportamiento asincrono

## Async/await:

Las funciones asíncronas (async) y la palabra clave await proporcionan una sintaxis más fácil de usar para trabajar con promesas.

await se utiliza dentro de una función async para esperar a que una promesa se resuelva antes de continuar con la ejecución del código.

```
async function descargarYMostrarDatos() {  
  try {  
    const datos = await descargarDatos('https://ejemplo.com/datos');  
    console.log(datos);  
  } catch (error) {  
    console.error(error);  
  }  
}  
  
descargarYMostrarDatos();
```

# Estructuras de control: try, catch y finally

# Estructuras de control

Las estructuras de control try, catch, y finally son utilizadas en muchos lenguajes de programación, incluido JavaScript, para manejar errores y excepciones de manera más controlada



# Estructuras de control

Estas estructuras permiten a los desarrolladores escribir código que puede manejar excepciones y realizar acciones específicas en caso de que ocurra un error.



# Estructuras de control

## try:

El bloque try se utiliza para envolver el código que se espera que pueda lanzar una excepción.

Si ocurre una excepción dentro del bloque try, el flujo de control se transfiere inmediatamente al bloque catch, saltándose cualquier código restante en el bloque try.

El bloque try se usa para "intentar" ejecutar una operación que podría causar un error.

```
try {  
    // Código que podría lanzar una excepción  
} catch (error) {  
    // Manejo de la excepción  
}
```

# Estructuras de control

## catch:

El bloque catch se utiliza para manejar excepciones que se generaron dentro del bloque try.

Dentro del bloque catch, puedes acceder al objeto de error (error) que contiene información sobre la excepción, como su mensaje y tipo. Puedes personalizar el manejo de errores en el bloque catch, como registrar información de error o tomar acciones específicas.

```
try {  
    // Código que podría lanzar una excepción  
} catch (error) {  
    // Manejo de la excepción  
}
```

# Estructuras de control

## finally:

El bloque finally es opcional y se coloca después del bloque try y, opcionalmente, después del bloque catch.

El código dentro del bloque finally se ejecutará siempre, ya sea que se haya lanzado una excepción o no.

Se utiliza para realizar tareas de limpieza o liberación de recursos, independientemente de si se produjo un error o no.

```
try {  
    // Código que podría lanzar una excepción  
} catch (error) {  
    // Manejo de la excepción  
} finally {  
    // Código que siempre se ejecutará  
}
```

# Estructuras de control en acción

La función `dividirNumeros` toma dos argumentos `a` y `b`,  
y trata de dividir `a` por `b`.

Dentro del bloque `try`, verificamos si `b` es igual a cero. Si  
es así, lanzamos una excepción personalizada  
utilizando `throw`. De lo contrario, realizamos la  
división y mostramos el resultado.

En el bloque `catch`, manejamos la excepción en caso de  
que se haya lanzado (cuando `b` es igual a cero) y  
mostramos un mensaje de error.

En el bloque `finally`, mostramos un mensaje para  
indicar que la operación de división ha finalizado, ya  
sea con éxito o con un error.



¿Dudas o consultas?





¡Muchas Gracias!