# JavaScript Avanzado

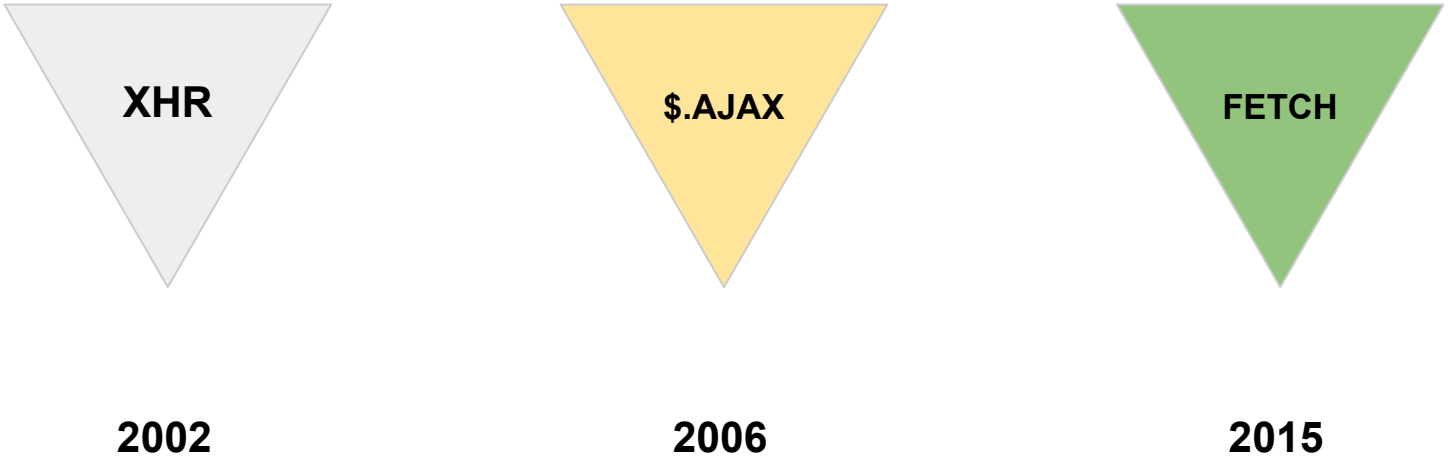
Introducción a AJAX - Fetch

# Agenda de hoy 📔

- Fundamentos de AJAX
  - XMLHttpRequest
  - $.ajax()
  - fetch()
    - status
    - response
  - Promises
    - .then()
    - .catch()
  - try - catch - finally
  - async await

# Evolución de peticiones a servidores en JS

# Evolución de peticiones a servidores con JS




Esta tecnología nace como una propuesta de Microsoft, inspirada en el mecanismo de funcionamiento de diálogo entre un navegador web y un servidor web.

# Evolución de peticiones a servidores con JS

```
const URL = 'https://api.servidorbackend.com/v1/catalogo';
const obtengoContenido = (URL) => {
      const xhr = new XMLHttpRequest()
            xhr.open("GET", URL, true)
            xhr.timeout = 5000
            xhr.send()
            xhr.addEventListener("loadend", ()=> {
               if (xhr.readyState == 4 && xhr.status == 200) {
                  contenidoJSON = JSON.parse(xhr.responseText)
                  contenidoJSON.forEach(cont => {
                     cardsAmostrar += retornoCardContenido(cont)
                  })
                  contenidoDOM.innerHTML = cardsAmostrar
               }
            })
            xhr.addEventListener("error", ()=> {
               contenidoDOM.innerHTML = retornoCardError()
               xhr.abort()
            })
            cargandoDOM.innerHTML = ""
}
```


El objeto **XMLHttpRequest** existe por retrocompatibilidad, pero lleva varios años ya en desuso.

La obtención de datos remotos se daba en un contexto basado en eventos, que controlaban el aspecto asincrónico del cual JS carecía.

# Evolución de peticiones a servidores con JS

$.ajax

```
$.ajax({
        url: URL,
        success: (data)=> {
           contenidoAPI = data.results
           contenidoAPI.forEach(usuario => {
                            HTMLTabla += retornoHTML(usuario)
                           })
           tablaResultados.innerHTML = HTMLTabla
        },
        error: ()=> {
           contenido.innerHTML = retornoError()
        }
     })
```


**JQuery Ajax** contiene internamente el manejo del objeto XHR, aunque estructura de forma ligeramente diferente el manejo de las peticiones.

Nos abstrae de los eventos y controles básicos en XHR para dedicarnos a mostrar el contenido obtenido, o un error ante cualquier falla o imprevisto.

# Evolución de peticiones a servidores con JS


Fetch

```
fetch('http://example.com/movies.json')
  .then(response => response.json());
  .then(data => console.table(data));
```

La API **Fetch** proporciona una interfaz JavaScript para acceder y manipular partes del canal HTTP, tales como peticiones y respuestas.

Es la evolución de XHR y se viene impulsando como estándar desde el año 2015 (ES6).

# Evolución de peticiones a servidores con JS


Fetch

```
fetch('http://example.com/movies.json')
  .then(response => response.json());
  .then(data => console.table(data));
```

Su funcionalidad está constituida a base de **Promesas JS**. Eventualmente también puede utilizarse mediante funciones asincrónicas JS.

# Promesas JS

# Promesas JS

Este concepto fue clave para comenzar a evolucionar los eventos en JS, aportando un valor agregado más importante dentro del mundo de las clases y objetos JS.

El manejo de eventos en JS era inicialmente la forma más efectiva de “*esperar a que algo suceda*”, y luego reaccionar.

# Promesas JS

Este concepto fue clave para comenzar a evolucionar los eventos en JS, aportando un valor agregado más importante dentro del mundo de las clases y objetos JS.

El manejo de eventos en JS era inicialmente la forma más efectiva de “*esperar a que algo suceda*”, y luego reaccionar.

# Promesas JS

Ese “*algo*” puede demorarse X cantidad de tiempo que no podemos calcular, por ello, debemos controlar la tarea siguiente mediante un evento que sí pueda esperar.

Eventos en JS

```
objeto.onComplete = ()=> {
  console.log("Tarea que se ejcuta al ocurrir el evento Complete");
}

objeto.onError = (err)=> {
  console.error("Si ocurre un error, se ejecuta este otro.", err);
}
```

# Fundamentos de Promesas JS

Como JS es un lenguaje de comportamiento sincrónico los eventos quedaban, en determinadas situaciones, muy limitados. Por ello, el equipo de Node JS, quienes trabajaban ya en un JS evolucionado del lado del servidor, desarrollaron la propuesta de **Promise**.

Promise contiene internamente, lo mejor de los eventos JS, y el condimento adicional necesario para abrazar más de cerca al modelo asincrónico que este lenguaje necesitaba.

# Fundamentos de Promesas JS

Ejemplo básico del uso de promesas a partir de la clase **Promise**.

JS Promises

```
return new Promise((resolve, reject) => {
  //controlar estados de la promesa y, resolverla o rechazarla
}
```

# Fundamentos de Promesas JS

Las promesas se crean instanciando la clase Promise, que acepta una **función callback** la cual se ejecutará asincrónicamente.

Esta función acepta a su vez dos parámetros: **resolve** y **reject**.

JS Promises

```
return new Promise((resolve, reject) => {
  //controlar estados de la promesa y, resolverla o rechazarla
}
```

# Fundamentos de Promesas JS

Con el primero de ellos manejaremos las tareas que van “*por el camino feliz*”, mientras que el segundo estado nos permitirá controlar qué hacer cuando ese camino “*no sea el esperado*”.

JS Promises

```
return new Promise((resolve, reject) => {
  //controlar estados de la promesa y, resolverla o rechazarla
}
```

# Estados

# Estados de una Promesa

Desde el momento en el cual **nace una promesa**, ésta **asume un estado predeterminado**. El mismo se denomina “*Pending*”.

JS Promises

```
const promesa =  new Promise((resolve, reject) => {
  const numeroAleatorio = Math.random();
})

//Si ejecutamos el objeto en la consola JS, nos devuelve su estado
promesa
Promise {<pending>}
```

# Estados de una Promesa

Una promesa en JavaScript tiene tres posibles estados:

| Estado | Descripción |
| --- | --- |
| **`pending`** | Este es el estado inicial de una promesa, cuando se crea pero aún no se ha resuelto ni rechazado. |
| **`fulfilled`** | Una promesa pasa a este estado cuando se ha resuelto correctamente, es decir, se ha ejecutado la operación asíncrona que representa sin errores. Cuando una promesa está en este estado, se puede acceder al valor de resultado proporcionado por la función **resolve**. |
| **`rejected`** | Una promesa pasa a este estado cuando se ha producido un error durante la ejecución de la operación asíncrona que representa. Con la promesa en este estado se puede acceder al motivo del rechazo proporcionado por la función **reject**. |

# Estados de una Promesa

Una vez que una Promesa JS se haya resuelto, o rechazado, la misma cambiará su estado inicial, **`pending`**, por el estado apropiado de acuerdo a su resolución o rechazo.

- **`fulfilled`**
- **`rejected`**

# Resultado de una Promesa

# Resultado de una Promesa

Código de una Promesa algo más funcional. En este, buscamos obtener un número aleatorio y **solo se resolverá la promesa si el número es mayor a 0.5**.

JS Promises

```
const promesa = new Promise((resolve, reject) => {
  const numeroAleatorio = Math.random();

  if (numeroAleatorio < 0.5) {
    resolve(numeroAleatorio); // Se resuelve con éxito
  } else {
    reject(new Error('El número random es mayor o igual a 0.5'));
    // la promesa se rechaza
  }
});
```

# Resultado de una Promesa

Tanto **resolve** como **reject** son objetos que integran a la promesa, y que utilizan un **retorno implícito** del resultado que cada uno de ellos maneja.

JS Promises

```
const promesa = new Promise((resolve, reject) => {
  const numeroAleatorio = Math.random();

  if (numeroAleatorio < 0.5) {
    resolve(numeroAleatorio); // Se resuelve con éxito
  } else {
    reject(new Error('El número random es mayor o igual a 0.5'));
    // la promesa se rechaza
  }
});
```

# Resultado de una Promesa

En el momento en el cual la promesa se resuelve y se ejecuta con ello el objeto **`resolve(...)`**, el estado de la promesa cambia automáticamente a **`fulfilled`**.

JS Promises

```
const promesa = new Promise((resolve, reject) => {
  const numeroAleatorio = Math.random();

  if (numeroAleatorio < 0.5) {
    resolve(numeroAleatorio); // Se resuelve con éxito
  } else {
    reject(new Error('El número random es mayor o igual a 0.5'));
    // la promesa se rechaza
  }
});
```

```
Promise {<fulfilled>}
```

# Resultado de una Promesa

En cambio si la promesa es rechazada y se ejecuta el objeto **`reject(...)`**,

el estado de la misma cambiará a **`rejected`**.

JS Promises

```
const promesa = new Promise((resolve, reject) => {
  const numeroAleatorio = Math.random();

  if (numeroAleatorio < 0.5) {
    resolve(numeroAleatorio); // Se resuelve con éxito
  } else {
    reject(new Error('El número random es mayor o igual a 0.5'));
    // la promesa se rechaza
  }
});
```

```
Promise {<rejected>}
```

# Resultado de una Promesa

A su vez, si la promesa devuelve **resolve**, entonces se invocará al método **.then()**.

Mientras que, si devuelve **reject**, el método invocado será **.catch()**.

JS Promises

```
promesa
  .then(resultado => console.log('La promesa se resolvió con éxito:', resultado))
  .catch(error => console.error('La promesa se rechazó debido a un error:', error));
```

# Fundamentos de Promesas JS

Manejo controlado de la respuesta de una promesa.

| Estado | Descripción |
| --- | --- |
| `.then(result)` | Mediante este método, podemos ir controlando cada tarea que se ejecuta, posterior a la respuesta de la promesa. Podemos encadenar tantos **.then()** como consideremos. El parámetro (`result`, en el ejemplo), corresponde al valor o resultado entregado por la promesa, mediante el objeto **resolve()**. |
| `.catch(err)` | Cualquier error que surja, o el mismo rechazo de la promesa, mediante el objeto **reject()**, será controlado por el método **.catch()**. El error resultante lo podemos ver mediante el parámetro **err**, o **error** y así tomar acciones apropiadas. |
| `.finally()` | Este método es opcional y se ejecutará siempre, si lo agregamos, independientemente de cuál haya sido el estado resultante de la promesa **resolve()** o **reject()**. |

# Resultado de una Promesa

Como alternativa, existe una tercera posible función, la cual se ejecuta siempre, indistintamente del resultado de la promesa: **`.finally()`**.

JS Promises

```
promesa
  .then(resultado => console.log('La promesa se resolvió con éxito:', resultado))
  .catch(err => console.error('La promesa se rechazó debido a un error:', err));
  .finally(() => console.warn('Mensaje alternativo que se muestra siempre'));
```

# Resultado y Estado de una Promesa

**Es importante destacar que, una vez que una promesa ha pasado a los estados de "resuelta" o "rechazada", permanecerá en ese estado y no se podrá cambiar a otro estado.**

**Podríamos interpretar esto como el fin del "*Ciclo de Vida*" de la promesa en cuestión.**

# Fundamentos de Promesas JS

Ejemplo funcional de **`fetch()`** controlando la respuesta del servidor mediante **JS Promises**.

JS Promises

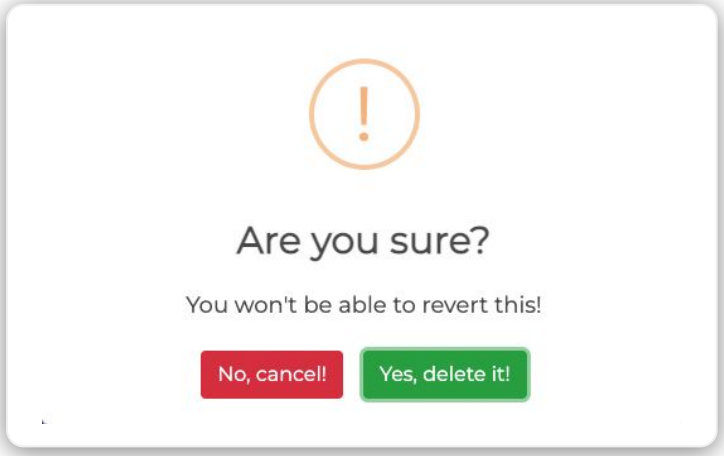
```
fetch('https://miservidorremoto.com/api/clientes')
      .then(response => response.json())
      .then(json => cargarTablaClientes(json))
      .catch(err => console.error("Se ha producido un error.", err)
      .finally(clientes => console.log("Finalizó la petición"))
```

# Fundamentos de Promesas JS

JS Promises

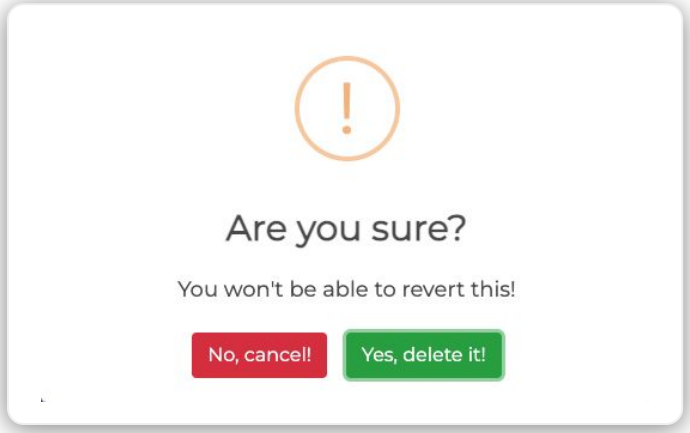
```
swalWithBootstrapButtons.fire({
  title: 'Are you sure?',
  text: "You won't be able to revert this!",
  icon: 'warning',
  showCancelButton: true,
  confirmButtonText: 'Yes, delete it!',
  cancelButtonText: 'No, cancel!',
  reverseButtons: true
}).then((result) => {
  if (result.isConfirmed) {
    swalWithBootstrapButtons.fire(
      'Deleted!',
      'Your file has been deleted.',
      'success'
    )
  } else if (
    /* Read more about handling dismissals below */
    result.dismiss === Swal.DismissReason.cancel
  ) {
    swalWithBootstrapButtons.fire(
      'Cancelled',
      'Your imaginary file is safe :)',
      'error'
    )
  }
})
```

**Sweet Alert 2** es una librería JS que permite solventar cuadros de diálogo interactivos que reemplazan a (**`alert`** - **`confirm`** - **`prompt`**) y más también.



# Fundamentos de Promesas JS

Cuando utilizamos esta librería para controlar un diálogo que depende de múltiples botones de acción, la funcionalidad de Sweet Alert se controla mediante una promesa, esperando que el usuario pulse un botón para recién allí disparar una acción asociada.



 # Espacio de prácticas 

# 👨‍💻 Espacio de prácticas 👨‍💻

Integra el uso de MOCKAPI.io, registrándote con tu cuenta de Gmail o Github:
Crea un endpoint utilizando como plantilla algún array de objetos anteriormente utilizado en los ejemplos del curso.

- Crea una función **obtenerDatos()** para leer el endpoint principal utilizando **fetch()**
- Crea una función **obtenerDato()** para obtener un producto por su **id**
    - utiliza el cuadro de diálogo prompt y **fetch** + **URL params**
- Crea una función **guardarDato()** para crear un nuevo recurso en el servidor
    - utiliza un objeto literal para armar los datos del nuevo recurso
    - grábalo utilizando **fetch** y el método **POST**

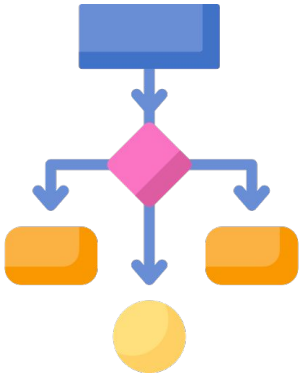
#  Espacio de prácticas 

Puesta en común del ejercicio

# try - catch - finally

# try - catch - finally

Una pieza clave que existe desde los orígenes de javascript, es el uso de la estructura **try - catch - finally**. Corresponde a un bloque de código que ejecuta una tarea específica en el código de JS y, ante cualquier error inesperado, atrapa el mismo y lo podemos controlar de forma efectiva.



# try - catch - finally


Su estructura es muy simple. Dentro del apartado **try {...}**, definimos una operación o tarea de JS. Si todo va bien, se ejecuta ese código y finaliza la operación de la instrucción en cuestión.

# try - catch - finally


En el apartado **catch{...}**, definimos una operación que sólo se ejecutará si por algún motivo ocurre un error inesperado dentro del bloque de ejecución contenido en **try{...}**. Catch nos permite capturar el error y/o manejarlo.

# try - catch - finally


Finalmente, y de manera opcional, podemos incluir la sentencia **finally {...}**.

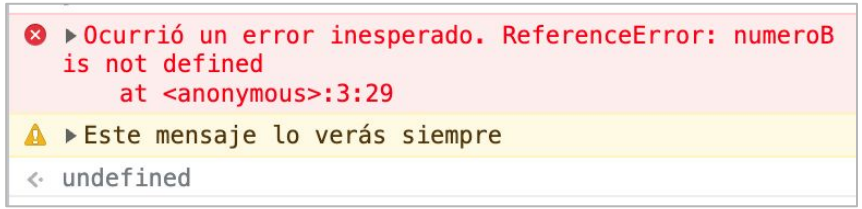
La misma ejecutará otro bloque de código, más allá de que el código controlado anteriormente, haya ido por buen camino, o se haya interceptado algún tipo de error.

# try - catch - finally

```
try catch finally

try {
  let numeroA = 21;
  let resultado = numeroA * numeroB;
      console.log("Resultado:", resultado);
} catch(error) {
  console.error("Ocurrió un error inesperado.", error);
} finally {
  console.warn("Este mensaje lo verás siempre");
}
```

En este bloque de código, intentamos multiplicar dos variables. La segunda de ellas no ha sido declarada, por lo cual, vamos camino hacia un error seguro.

Como resultado en la Consola JS, vemos que efectivamente se produjo un error. El mismo fue interceptado por la estructura **catch{}**, mientras que la estructura **finally{}** muestra su mensaje más allá del camino acontecido.



# try - catch - finally

try catch finally

```
try {
  let numeroA = 21;
  let numeroB = 75;
  let resultado = numeroA * numeroB;
      console.log("Resultado:", resultado);
} catch(error) {
  console.error("Ocurrió un error inesperado.", error);
} finally {
  console.warn("Este mensaje lo verás siempre");
}
```

```
Resultado: 1575
▶ Este mensaje lo verás siempre
undefined
```

Con el código bien definido, vemos que el camino de este algoritmo muestra la ejecución correcta del bloque **try {}** y debajo la ejecución del bloque **finally {}**.

Al ser imposible tener un control total sobre todos los posibles errores en los algoritmos, este tipo de herramientas son ideales para implementar. Aliviará mucho nuestra tarea cotidiana.

# try - catch - finally

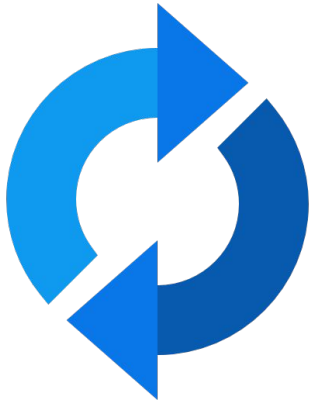
**Esta estructura de control de código, es común a casi todos los lenguajes de programación que se asimilan a JavaScript.**

**Es muy útil en la mayoría de los casos, y a su vez, es la esencia del funcionamiento que encontramos aplicado en el uso de Promesas JS, y sus métodos .then(), .catch() y .finally().**

# Asincronismo

# Asincronismo

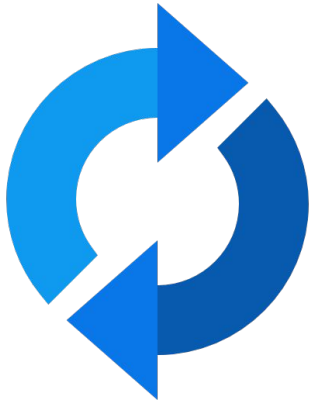
El asincronismo en JavaScript se refiere a la capacidad de realizar tareas de forma no bloqueante, es decir, que el programa no se detiene para esperar la respuesta de una tarea antes de continuar con las siguientes instrucciones.



Esto permite que el programa siga ejecutando otras tareas mientras espera una respuesta de otra tarea.

# Asincronismo

Para trabajar con asincronismo en JavaScript, se utilizan funciones asíncronas y callbacks. Las funciones asíncronas son aquellas que pueden ejecutar tareas de forma asíncrona, mientras que los callbacks son funciones que se ejecutan una vez que se haya completado alguna tarea definida como asíncrona.



Veamos a continuación, algunos ejemplos de código que ilustran cómo se aplica el asincronismo en JavaScript:

# Asincronismo

Asincronismo

```
function tareaAsincrona(callback) {
  setTimeout(function() {
    callback("Resultado de la tarea asíncrona");
  }, 1000);
}

console.log("Inicio");

tareaAsincrona(function(resultado) {
  console.log(resultado);
});

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

En este ejemplo, la función **tareaAsincrona** utiliza el método **`setTimeout`** para simular una tarea asíncrona que tarda un segundo en completarse.

La función recibe un argumento **callback**, que es una función que se ejecuta una vez que se ha completado la tarea asíncrona.

# Asincronismo

Asincronismo

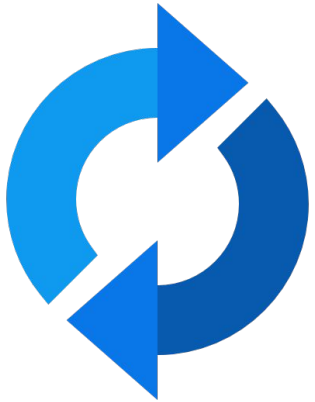
```
function tareaAsincrona(callback) {
  setTimeout(function() {
    callback("Resultado de la tarea asíncrona");
  }, 1000);
}

console.log("Inicio");

tareaAsincrona(function(resultado) {
  console.log(resultado);
});

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

En este ejemplo elaborado, el callback simplemente imprime el resultado por consola.

Al llamar a la función **tareaAsincrona**, se muestra primero el mensaje "*Inicio*", luego se inicia la tarea asíncrona y se muestra el mensaje "*Fin*", y finalmente se ejecuta el callback con el resultado de la tarea.

# Asincronismo

Y, para no caer en el uso de callback encadenando funciones dentro de otras funciones, JavaScript evolucionó, posterior al nacimiento de Promesas, integrando el uso de **funciones asincrónicas** en el lenguaje.



Esto es común en otros lenguajes de programación, pero en JS recién llegaron entre 2015 y 2016. Las promesas JS antecedieron al universo asincrónico en este lenguaje de programación.

# async - await

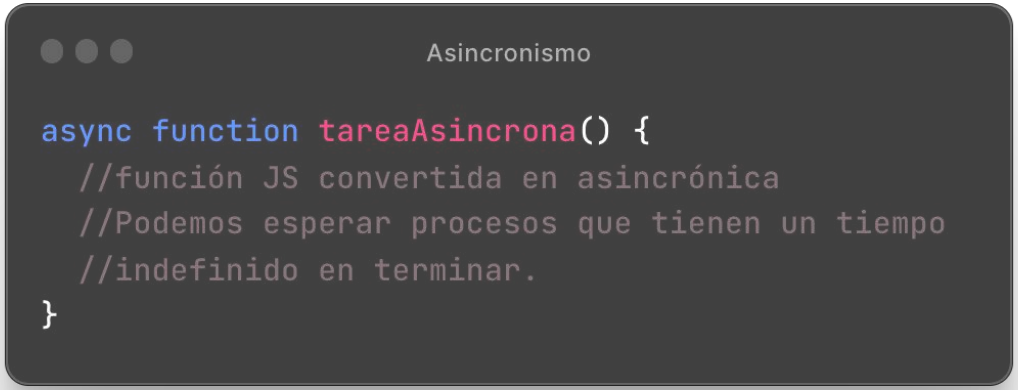
# async - await

En JavaScript, **`async`** y **`await`** son dos palabras clave que se utilizan para trabajar con funciones asíncronas de manera más sencilla.

Las mismas se combinan dentro de una función convencional, la cual se convierte en asincrónica, cuando anteponemos la palabra **async**.

# async - await

**async** se utiliza para declarar una función asíncrona. Una función asíncrona devuelve siempre una Promesa, aunque no se indique explícitamente.
Dentro de una función asíncrona, se pueden utilizar palabras clave como **await** para indicar que se debe esperar la respuesta de una tarea asíncrona antes de continuar con la ejecución del código.



# async - await

**`await`** se utiliza dentro de una función asíncrona para indicar que se debe esperar la respuesta de una tarea asíncrona antes de continuar con la ejecución del código.

Asincronismo

```
async function tareaAsincrona() {
  const resultado = await obteniendoDatos();
        console.table(JSON.parse(resutado));
}
```

# async - await

Cuando se utiliza **await**, la función se detiene en ese punto hasta que la tarea asíncrona se haya completado y se haya devuelto un resultado. El resultado de la tarea asíncrona se asigna a la variable/constante que se encuentra a la izquierda del operador await.

Asincronismo

```
async function tareaAsincrona() {
  const resultado = await obteniendoDatos();
        console.table(JSON.parse(resutado));
}
```

# async - await

Y, si deseamos tener un control total de la situación, podemos sumar al universo asincrónico, el uso de **try - catch**, para así tener todo el control ante algún posible error no previsto, sea nuestro o por parte de la petición de datos realizada.

Asincronismo

```
async function() {
  try {
    const resultado = await obteniendoDatos();
    console.table(JSON.parse(resutado));
  } catch (error) {
    console.error("Se ha producido un error", error);
  }
}
```

# async - await

**El uso de async y await hace que el código asíncrono sea más fácil de leer y entender, ya que se evita la anidación de callbacks y se utiliza una sintaxis más similar a la programación sincrónica.**

# async - await

Aquí tenemos un ejemplo de uso de **fetch()**, implementando asincronismo en lugar de promesas. El resultado es similar, aunque el código se estructura ligeramente distinto.

Asincronismo

```
async function() {
  try {
    const resultado = await fetch(URL);
    const data = await resultado.json();
    console.table(data);
  } catch (error) {
    console.error("Se ha producido un error", error);
  }
}
```

# ¡Gracias!