



**Argentina  
programa  
4.0**



Ministerio de Economía  
Argentina

Secretaría de  
Economía del Conocimiento

***primero  
la gente***

# Clase 2: Clases JavaScript, Asincronismo y Promesas

# Presentación general del curso

Temas

# Agenda de hoy

- A. La evolución de las funciones constructoras
- B. Las clases JS
  - a. propiedades y métodos
  - b. getter y setter
  - c. propiedades privadas y estáticas
- C. Promesas JS
  - a. Estados: pending, fulfilled, rejected
  - b. Objetos: resolve, reject
  - c. Métodos: then(), catch(), finally()
- D. Funciones asincrónicas
  - a. Async - Await
  - b. Temporizadores en JS
    - i. SetTimeOut - SetInterval - ClearTimeOut - ClearInterval



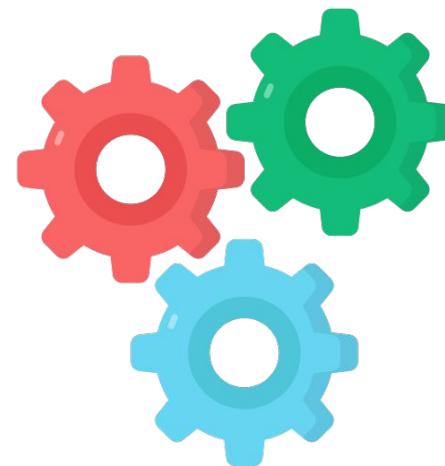
# **La evolución de las funciones constructoras**

# La evolución de las funciones constructoras

En el universo JavaScript, los objetos son algo cotidiano. Toda la estructura, o Core, del lenguaje está basado en objetos.

Aún así, JS es bastante permisible de poder definir toda una aplicación (frontend o backend), en una estructura de funciones, sin tener que caer en modelos o **paradigmas como MVC o MVVM**.

Veamos un poco cómo evolucionó JS incluyendo diferentes estructuras de objetos para que formen parte de su lenguaje, y qué se debe (y cómo) utilizar hoy de todo esto.



# La evolución de las funciones constructoras

Los objetos en JavaScript llegaron en diferentes etapas del lenguaje.

## OBJETO LITERAL

Los objetos literales JS llegaron a fines de la década del 90 a este lenguaje.

Pares clave-valor y funciones integradas, denominadas métodos.

## FUNCIONES CONSTRUCTORAS

Para ES5 (2009), se integraron las funciones constructoras.

Un paradigma más cercano al modelo de objetos que tienen otros lenguajes de programación, donde definimos una estructura padre, o template, y la instanciamos N cantidad de veces para utilizarla dentro de nuestra aplicación.

## CLASES JS

Finalmente, con ES6 (2015), llegaron las clases JS. Estas proponen un modelo de objetos basado en clases, simplificando la estructura confusa de las funciones constructoras, pero limitadas en general porque se engloban bajo el término Syntactic Sugar.



# La evolución de las funciones constructoras

**Las funciones constructoras** son una forma de crear objetos y definir propiedades y métodos para esos objetos.

Una función constructora se utiliza para crear objetos con las mismas propiedades y métodos.

Aquí vemos un ejemplo de cómo crear una función constructora en JavaScript para un objeto denominado **Producto**.

El mismo posee una serie de propiedades, y un método que nos devuelve el importe del Producto más el IVA del 21 % ya calculado.



Funciones constructoras

```
function Producto(nombre, importe) {  
    this.id = crearID()  
    this.nombre = nombre  
    this.importe = importe  
  
    this.importeConIVA = function() {  
        const IVA = 1.21  
        return this.importe * IVA  
    }  
}
```

# La evolución de las funciones constructoras

El uso de `this` en las funciones constructoras **se utiliza para referirse al objeto que se está creando o instanciando**, a través de dicha función constructora.

De esta forma, **definimos cada propiedad del objeto que se está creando, como una referencia del tipo sinónimo**.

Así, podremos instanciar (*clonar*) la función constructora N cantidad de veces, con diferentes nombres, y cada propiedad será correspondida con cada instancia y sus valores asociados.

```
● ● ● Funciones constructoras

function Producto(nombre, importe) {
    this.id = crearID()
    this.nombre = nombre
    this.importe = importe

    this.importeConIVA = function() {
        const IVA = 1.21
        return this.importe * IVA
    }
}
```



# La evolución de las funciones constructoras

Con la función constructora, podemos instanciar el objeto N cantidad de veces, de acuerdo a nuestra necesidad.

Cada instancia puede recibir un nombre y un importe diferente.

A través del método **importeConIVA()** obtendremos un retorno del cálculo del precio del producto, con el IVA del 21% adicionado.

```
● ● ● Funciones constructoras

// Creamos un objeto Producto usando la función constructora
const producto1 = new Producto("Teclado Bluetooth", 25900)
const producto2 = new Producto("Mouse inalámbrico", 4780)
const producto3 = new Producto("Placa gráfica GeForce", 159000)

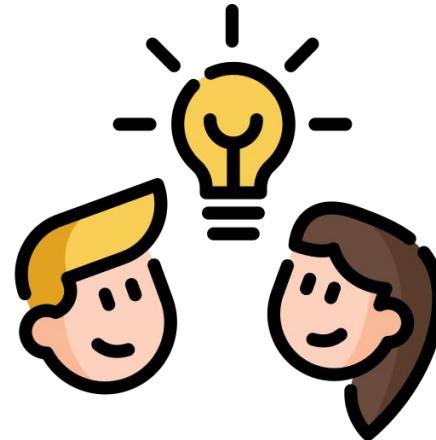
// Llamamos al método calcularIVA de cada objeto creado
console.log(producto1.calcularIVA()); // Salida: 3133.90
console.log(producto2.calcularIVA()); // Salida: 5783.80
console.log(producto3.calcularIVA()); // Salida: 192390
```



# La evolución de las funciones constructoras

Si bien, esto es una forma rápida y fácil de crear e instanciar objetos, a partir de una función constructora, el nombre elegido para éstas causaba “*mucho ruido*” para aquellos programadores que venían de lenguajes más estrictos a trabajar con JS.

**Así fue como nacieron las clases JS.**

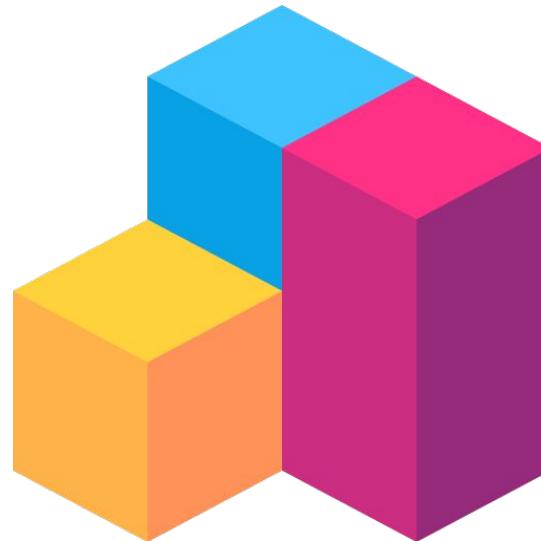


# Clases JS

# Clases JS

Las clases JS son básicamente una función constructora, pero evolucionada hacia una sintaxis más parecida a cómo se definen las clases en otros lenguajes de programación.

Esta evolución llegó a JS a partir de la versión EcmaScript 6. Si bien su estructura para definirla cambia respecto a las funciones constructoras, técnicamente son lo mismo.



# Clases JS

La estructura de una **Clase JS**, también posee un **constructor**, utiliza la palabra reservada **this** en cada una de sus **propiedades**, e integra **métodos** asociados.

Lo bueno de ella es que es mucho más limpia la forma de declararla.

Su estructura se asimila más a una clase Java o una clase C#.

```
● ● ● Clases JS

class Producto {
  constructor(nombre, importe) {
    this.id = crearID()
    this.nombre = nombre
    this.importe = importe
  }

  importeConIVA() {
    const IVA = 0.21
    return this.importe * IVA
  }
}
```



# Clases JS

Sé instancia de igual forma que una función constructora, y se utiliza también de igual forma.

**También, se recomienda crear cada clase JS en un archivo .JS dedicado o independiente.**



## Clases JS

```
const producto1 = new Producto("Macbook Air 13", 779500)
const producto2 = new Producto("Lenovo House 14", 219500)
const producto3 = new Producto("Notebook Gamer 17", 659500)
```

# Clases JS

En una clase JS podemos definir miembros estáticos.

Estos son métodos y/o propiedades que residen en el objeto constructor, y no en cada uno de los objetos creados a partir de una instancia de este.



# Clases JS

```
● ● ● Clases JS

class Producto {
    static acercaDe() {
        console.log("Copyright 2023 - jumpedu.org")
    }
    constructor(nombre, importe) {
        this.id = crearID()
        this.nombre = nombre
        this.importe = importe
    }

    importeConIVA() {
        const IVA = 0.21
        return this.importe * IVA
    }
}
```

Por ejemplo, **volviendo a la clase** **Producto**, si creamos esta para ser instanciada y luego distribuirla como una clase JS estándar que cualquier desarrollador pueda utilizarla, podremos agregarle un método estático con un Copyright, el cual será visible sobre la clase pura en sí, a través de la consola JS.



# Clases JS

```
const prod = new Producto("Macbook Air 13", 759900)
```

Una vez instanciada la clase, el objeto en cuestión, no hereda los miembros estáticos definidos en la clase.



The screenshot shows a browser's developer tools with the 'Console' tab selected. The console output is as follows:

```
Elements | Console | Sources >
prod.acercaDe()
✖ Uncaught TypeError: prod.acercaDe is not a function
at <anonymous>:1:6
```

A blue arrow points from the explanatory text above to this error message in the developer tools.

# Clases JS

Los **getters** y **setters** son funciones especiales que se utilizan para acceder y modificar propiedades de un objeto en JavaScript.

Los **getters** se utilizan para obtener el valor de una propiedad y los **setters** se utilizan para establecer el valor de una propiedad.

```
Clases JS

class Producto {
  constructor(nombre, importe) {
    this.id = crearID()
    this.nombre = nombre
    this.importe = importe
  }
  getNombre() {
    return this.nombre.toUpperCase() || ""
  }
  getImporte() {
    return this.importe || 0.00
  }
}
```



# Clases JS

La **ventaja de utilizar getters y setters** en una clase u objeto en JavaScript es que se puede controlar el acceso y la modificación de las propiedades de ese objeto.

En este ejemplo, vemos que el uso de los getters **getNombre()** y **getImporte()** retornan los valores de las propiedades homónimas, pero controlando que estas tengan datos.

Si no, retornan un fallback en su lugar.

```
Clases JS

class Producto {
    constructor(nombre, importe) {
        this.id = crearID()
        this.nombre = nombre
        this.importe = importe
    }
    getNombre() {
        return this.nombre.toUpperCase() || ""
    }
    getImporte() {
        return this.importe || 0.00
    }
}
```



# Clases JS

```
Clases JS

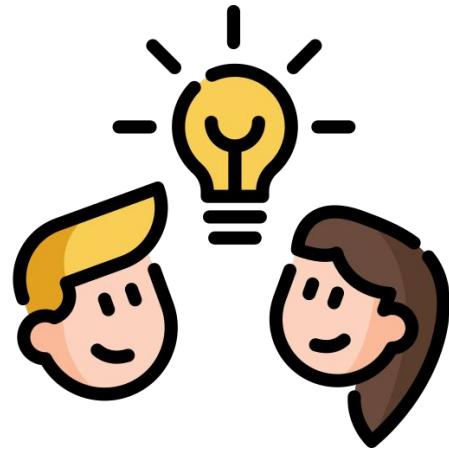
class Producto {
    constructor(nombre, importe) {
        this.id = crearID()
        this.nombre = nombre
        this.importe = importe
    }
    getNombre() {
        return this.nombre.toUpperCase() || ""
    }
    getImporte() {
        return this.importe || 0.00
    }
    setNombre(nombreProducto) {
        if (!nombreProducto) {
            console.error("Se esperaba un nombre válido")
        } else {
            this.nombre = nombreProducto.toUpperCase()
        }
    }
}
```

Los getters y setters **pueden proporcionar validación de datos, realizar cálculos o procesamiento adicionales antes de devolver o establecer el valor de una propiedad**, y en general, **aumentar el nivel de encapsulamiento y seguridad de los datos de un objeto.**

# Getters y Setters

Si bien JavaScript no es un lenguaje estricto en cuanto al uso de objetos y clases, es bueno que nos acostumbremos a trabajar de forma correcta, sobre todo si incluimos el uso de objetos dentro de nuestros desarrollos.

Por ello, implementar getters y setters en las clases y objetos en JavaScript para controlar el acceso y la modificación de las propiedades de un objeto, es un punto clave que nos hará más profesionales usarlo, además de prepararnos para en el futuro tener que interactuar con lenguajes de programación más estrictos.



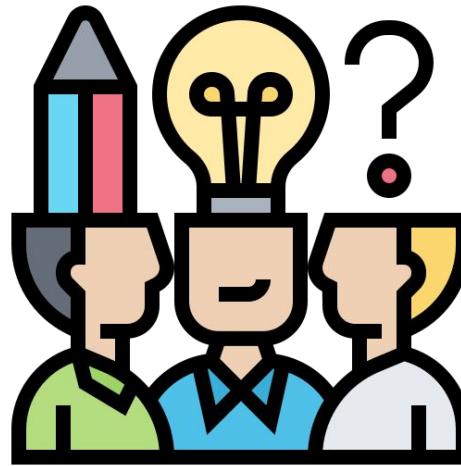
# Promesas JS

# Promesas JS

Seguimos avanzando con los puntos claves de JavaScript implementado en el Backend. En esta oportunidad, llega el turno de Promises.

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

Veamos entonces de qué tratan.



# Promesas JS

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

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

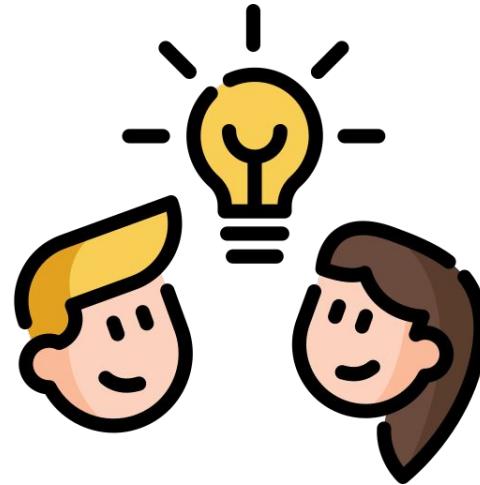
objeto.onComplete = ()=> {
    console.log("Tarea que se ejecuta al ocurrir el evento Complete");
}

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

# 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.**



# 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
}
```

# 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  
}
```

# 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 de una promesa JS**

# Estados de una promesa JS

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 JS

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 JS

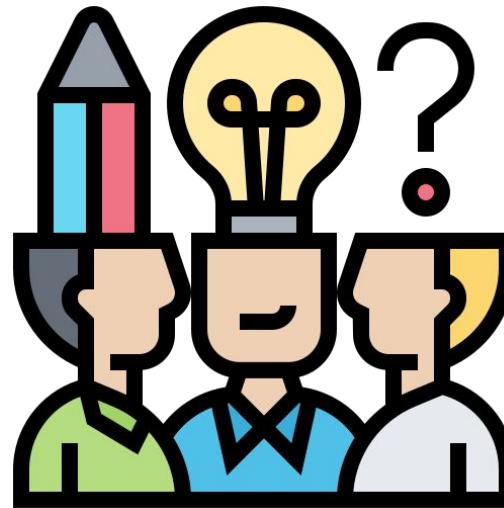
Una promesa en JavaScript tiene tres posibles estados:

| ESTADO           | DESCRIPCIÓN                                                                                                                                                                                                                                                           |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Pending</b>   | Este es el estado inicial de una promesa, cuando se crea pero aún no se ha resuelto ni rechazado.                                                                                                                                                                     |
| <b>Fulfilled</b> | 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. |
| <b>Rejected</b>  | 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.                               |

# Promesas JS

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 JS**

# Resultado de una promesa JS

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 JS

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 JS

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 JS

En cambio si la promesa es rechazada y se ejecuta el objeto **reject(...)**, el estado de la misma cambiará a **rejected**.

The diagram illustrates a code snippet for JavaScript Promises. The code defines a promise that generates a random number. If the number is less than 0.5, it is resolved successfully. Otherwise, it is rejected with an error message. A green box highlights the rejection logic. An arrow points from this highlighted area to a black box labeled "Promise {<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
  }
});
```

# Resultado de una promesa JS

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));
```

# Resultado de una promesa JS

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



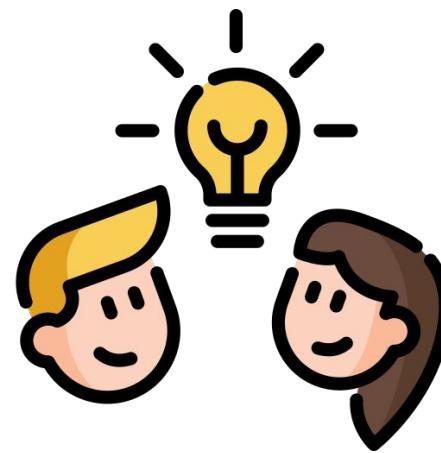
JS Promises

```
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 de una promesa JS

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.

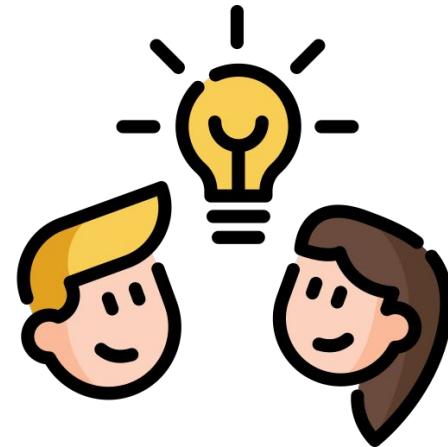
Podríamos interpretar esto como el fin del "*Ciclo de Vida*" de la promesa en cuestión.



# Promesas JS

Las promesas JS son clases completamente manejables por nosotros, para estructurar, por ejemplo, operaciones exhaustivas de nuestra aplicación, las cuales pueden demorar un tiempo indeterminado.

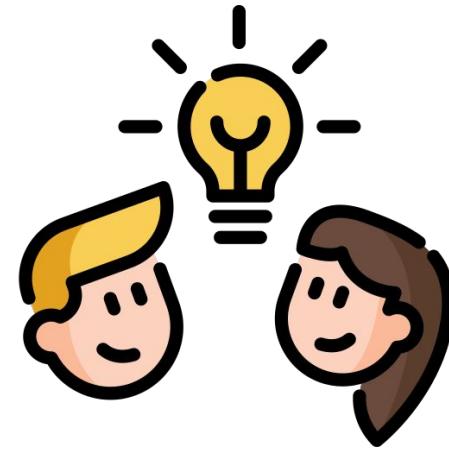
Allí, podremos recurrir a encapsular esta tarea dentro de una promesa JS, y así controlar su duración y que el código asociado se ejecute por etapas.



# Promesas JS

También, como las promesas llevan muchos años ya implementadas dentro del lenguaje JavaScript, tanto en backend como también en frontend (**nacieron en Node.js y se extendieron a JS frontend**), las mismas forman parte de otras tantas herramientas de este lenguaje.

Por ejemplo, en Frontend, la petición de datos a un servidor remoto se realiza con **fetch()** el cual puede utilizar Promises para controlar las etapas de ejecución asíncrona en el proceso de obtención de datos.



# Promesas JS

Ejemplo funcional de la función **fetch()** controlando la respuesta del servidor mediante JS Promises. Esta función es la utilizada actualmente en el Frontend para peticionar datos a una aplicación de backend.

```
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"))
```

# Promesas JS

Manejo controlado de la respuesta de una promesa.

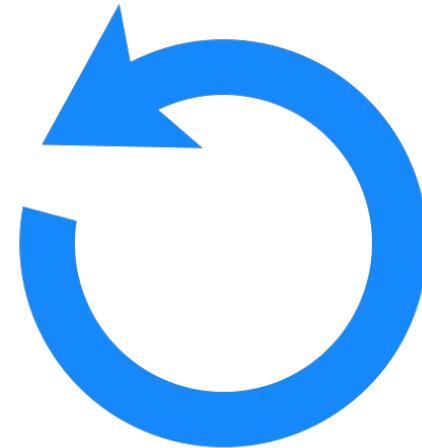
| ESTADO               | DESCRIPCIÓN                                                                                                                                                                                                                                                                        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

# 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.

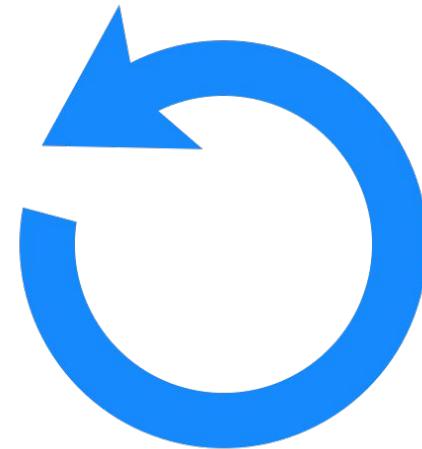


# Asincronismo

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

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

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

console.log("Inicio");

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

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



# 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");
```

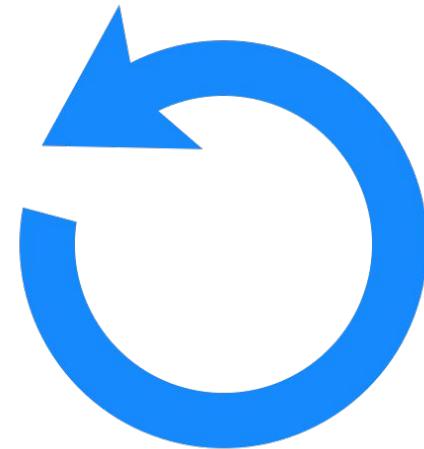
En este ejemplo elaborado, el **callback** simplemente imprime el resultado por consola.

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.

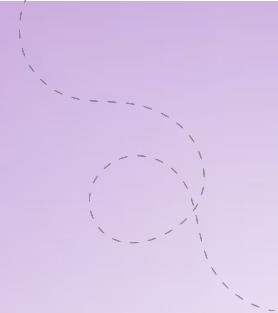
# Asincronismo

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 asíncrona, 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.

```
● ● ● Asincronismo  
  
async function tareaAsincrona() {  
    //función JS convertida en asincrónica  
    //Podemos esperar procesos que tienen un tiempo  
    //indefinido en terminar.  
}
```

# 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.

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));  
}
```

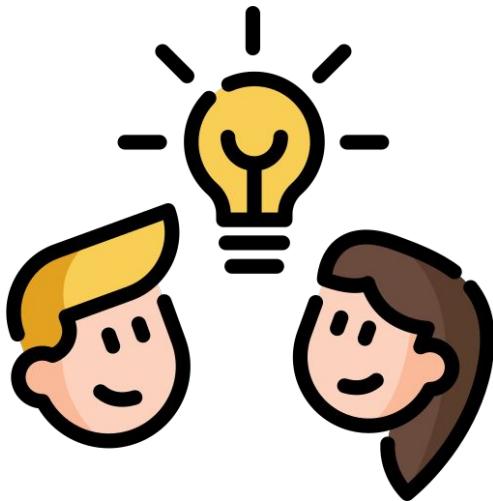
# Async - Await

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);
  }
}
```

# Desarrollo de aplicaciones para servidores



**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);
  }
}
```

# Temporizadores en JS

# Temporizadores en JS

Antes de existir Asincronismo y Promesas, para controlar los tiempos de ejecución de determinadas tareas, todo se hacía de la mano de los eventos y, eventualmente, del uso de las funciones temporizadoras de JS.

Estas son:

- **setInterval**
- **setTimeout**



# Temporizadores en JS

Las funciones `setTimeout()` y `setInterval()` son funciones integradas de JavaScript que permiten programar la ejecución de código en un momento específico o de forma repetida.

Veamos un poco en detalle, cada una de ellas.



# Temporizadores en JS

## **SetTimeout()**

La función setTimeout() se utiliza para programar una ejecución única de una función después de un cierto retraso en milisegundos.

La sintaxis es la siguiente:

```
setTimeOut(función, tiempo);
```

# Temporizadores en JS

**función:** la función contiene el código que se va a ejecutar después del retraso o tiempo preestablecido.

**tiempo:** el tiempo se especifica en milisegundos y refiere al espacio temporal que va a suceder desde que se ejecuta **setTimeOut**, hasta que se dispara la función asociada.

```
...  
  
setTimeout(function() {  
  console.log('Han pasado 2 segundos');  
}, 2000);
```

# Temporizadores en JS

Cuando definimos **setTimeOut**, podemos declarar esta función dentro de una constante. Esto nos permite contener a la función en la constante creada, y recurrir a un mecanismo de interrupción de este período de tiempo, si es que deseamos cancelar la ejecución del código asociado.

```
const timeout = setTimeout(()=> {
    console.log('Han pasado 2 segundos');
}, 2000);
```

# Temporizadores en JS

Si deseamos cancelar setTimeOut antes del tiempo estipulado de espera, podemos recurrir a la función **clearTimeout()**.

Entre paréntesis debemos definir el identificador único devuelto por **setTimeOut**.

```
const timeout = setTimeout(()=> {
    console.log('Han pasado 2 segundos');
}, 2000);

clearTimeout(timeout);
```

# Temporizadores en JS

La función **setInterval()** se utiliza para programar la ejecución repetida de una función con un cierto tiempo de espera, entre cada ejecución. Su sintaxis es la siguiente:

```
setInterval(function() {  
    console.log('Este mensaje se mostrará cada 2 segundos');  
, 2000);
```

# Temporizadores en JS

El tiempo de intervalo se especifica en milisegundos.

De igual forma que con **setTimeOut()**, podemos asociar a **setInterval** en una constante, para así poder cancelar su ejecución en el momento que deseemos.

```
setInterval(function() {  
    console.log('Este mensaje se mostrará cada 2 segundos');  
, 2000);
```

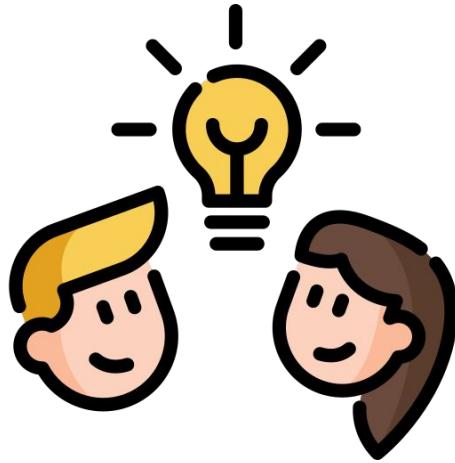
# Temporizadores en JS

Para lograr esto, invocamos a la función JS **clearInterval()**, indicándole entre los paréntesis el nombre de la constante que posee el identificador de **setInterval()**.

```
const cancelInterval = setInterval(function() {
    console.log('Este mensaje se mostrará cada 2 segundos');
}, 2000);

clearInterval(cancelInterval);
```

# Temporizadores en JS



**Es poco probable de que necesitemos utilizar estas funciones del lado del backend, pero debemos tenerlas presente por si alguna vez necesitamos generar un retraso forzado, antes de responder alguna petición cliente.**

# Muchas gracias.



Ministerio de Economía  
**Argentina**

Secretaría de  
Economía del Conocimiento

*primero  
la gente*