



Clase 3. Programación Backend

Programación sincrónica y asincrónica



OBJETIVOS DE LA CLASE

- Repasar las funciones en Javascript y conocer las nuevas declaraciones
- Comprender lo que es un callback y las promesas de JS
- Conocer el concepto y diferencias entre programación sincrónica y asincrónica en Javascript

CRONOGRAMA DEL CURSO

Clase 2



**Principios básicos de
Javascript**

Clase 3



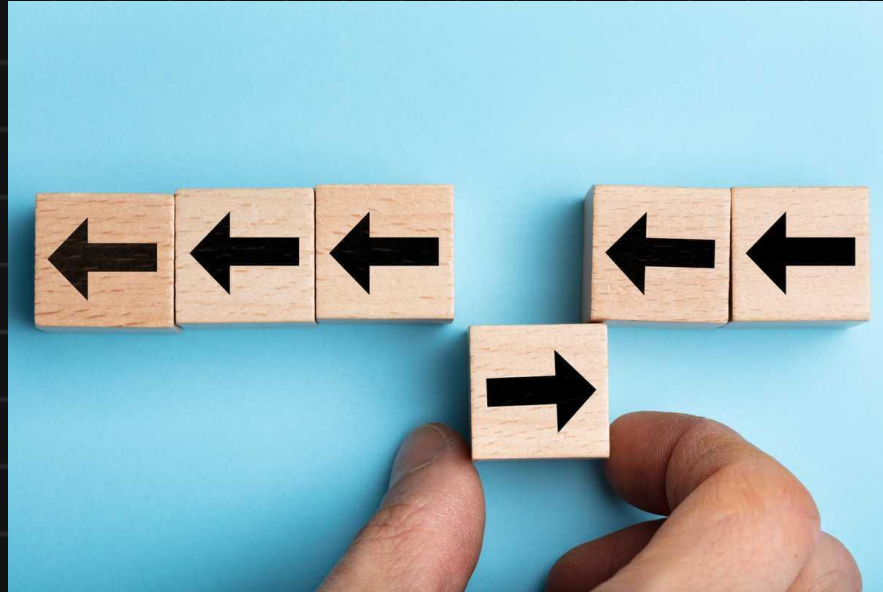
**Programación sincrónica y
asincrónica**

Clase 4



**Manejo de Archivos en
Javascript**

Funciones



Repasando...

- **Funciones:** Son un conjunto de instrucciones que realiza una tarea o calcula un valor.
- **Scope:** Se refiere al ámbito o alcance de una ejecución.
- **Closures:** Están conformados por una **función padre** que retorna una **función hija** donde esta última hace uso del scope de la primera.

Funciones, Scope y Closures

Veamos un ejemplo rápido... 



```
1 function saludar(nombre, pais) {  
2   const mensaje = `  
3   Hola ${nombre} 🖐️.  
4   ¿Cómo estás 😊?  
5   ¿Qué tal está ${pais} 🇨🇱?`  
6   console.log(mensaje)  
7 }  
8  
9 saludar('Cristian', 'Chile')
```

Template String

Veamos un ejemplo rápido... 

CODER HOUSE

- Nos permiten **definir** de una mejor manera, **la estructura** de los **objetos** desarrollados por el programador.
- Podemos definir **atributos** y **métodos** que serán parte del comportamiento de nuestro objeto.
- Estos últimos pueden ser de **instancia** o de **clase**.
- Los atributos de instancia se definen en el método **constructor**.

Clases

Veamos un ejemplo rápido... 

- Nos permiten **instanciar** un nuevo objeto a partir de una clase.
- Este procedimiento se realiza llamando al **constructor** de la clase y pasando los **argumentos**.

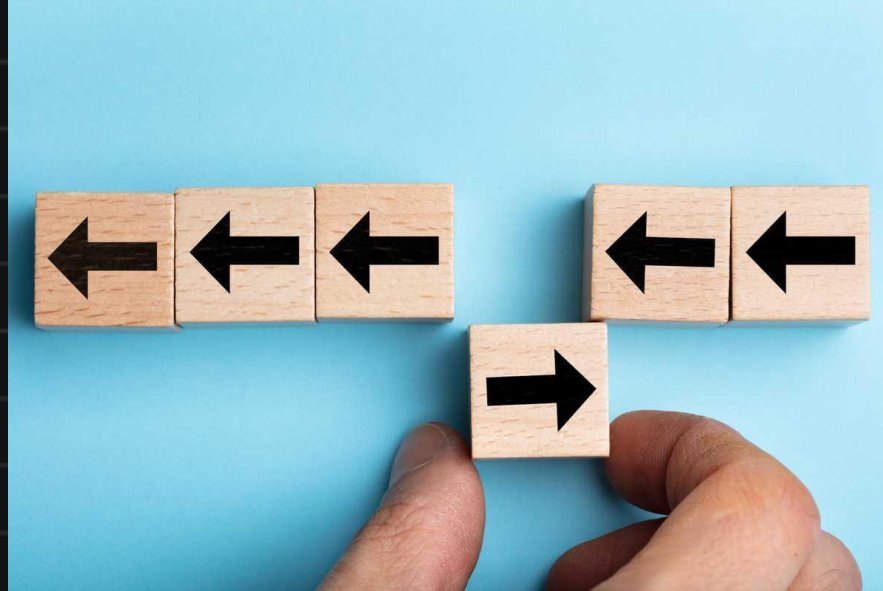
***Operador
new***

Veamos un ejemplo rápido... 



***¿Alguna pregunta hasta
ahora?***

Funciones 2.0



Nueva declaración de funciones



La **nueva sintaxis** consiste en **declarar únicamente los parámetros**, y luego **conectarlos** con el cuerpo de la función **mediante** el **operador =>** (flecha gorda, o 'fat arrow' en inglés). Veamos un ejemplo:

Nuevo estilo (simplificado):

```
const mostrar = (params) => {  
  console.log(params)  
}
```

*Llamada a la función: **mostrar(args)***

Funciones de un solo parámetro



En el caso de que la función reciba **un solo parámetro**, los **paréntesis** se vuelven **opcionales**, pudiendo escribir:

```
const mostrar = params => {  
  console.log(params)  
}
```

La función se podrá usar de la misma manera que las anteriores

Funciones de una sola instrucción



En el caso de que el cuerpo de la función conste de una **única instrucción**, las **llaves** se vuelven **opcionales**, el cuerpo se puede escribir en la misma línea de la declaración y el resultado de computar esa única línea se devuelve como resultado de la función, como si tuviera un “return” adelante. A esto se lo conoce como “return implícito”.

```
const mostrar = params => console.log(params)
```

En este caso la función devolvería “undefined” ya que console.log es de tipo void y por lo tanto no devuelve nada



Return implícito

Un ejemplo igualmente trivial pero más ilustrativo de return implícito sería el siguiente:

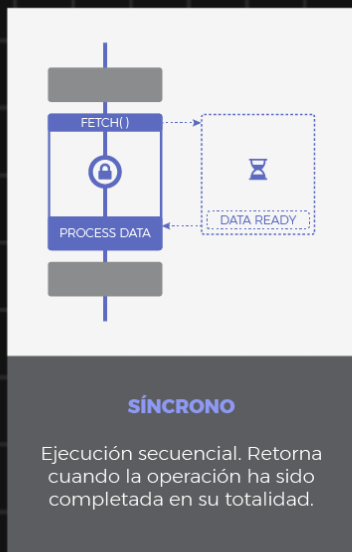
```
const promediar = (a, b) => (a + b) / 2
const p = promediar(4, 8)
console.log(p) // 6
```

Veamos un ejemplo rápido... 



***¿Alguna pregunta hasta
ahora?***

Sincronismo vs Asincronismo



Ejecución sincrónica vs. ejecución asincrónica

Ejecución Sincrónica

- Cuando escribimos **más de una instrucción** en un programa, **esperamos que** las instrucciones **se ejecuten** comenzando **desde la primera línea**, una por una, **de arriba hacia abajo** hasta llegar al final del bloque de código.
- Si una instrucción es una **llamada a otra función**, la **ejecución se pausa** y se procede a ejecutar esa función.
- Sólo **una vez ejecutadas** todas las instrucciones de esa función, el **programa retomará** con el flujo de instrucciones que venía ejecutando antes.

Comportamiento de una función: Bloqueante vs no-bloqueante

Cuando alguna de las instrucciones dentro de una función intente acceder a un recurso que se encuentre fuera del programa (por ejemplo, enviar un mensaje por la red, o leer un archivo del disco) nos encontraremos con dos maneras distintas de hacerlo: en forma bloqueante, o en forma no-bloqueante (blocking o non-blocking).

Operaciones bloqueantes



- En la mayoría de los casos, precisamos que el programa ejecute todas sus operaciones en forma secuencial, y sólo comenzar una instrucción luego de haber terminado la anterior.
- A las operaciones que obligan al programa a esperar a que se finalicen antes de pasar a ejecutar la siguiente instrucción se las conoce como **bloqueantes**.
- Este tipo de operaciones permiten que el programa se comporte de la manera más intuitiva.
- Permiten la ejecución de una sola operación en simultáneo.
- A este tipo de ejecución se la conoce como **sincrónica**.

Veamos un ejemplo rápido... 

Operaciones no-bloqueantes



- En algunos casos esperar a que una operación termine para iniciar la siguiente podría causar grandes demoras en la ejecución del programa.
- Por eso que Javascript ofrece una segunda opción: las operaciones **no bloqueantes**.
- Este tipo de operaciones permiten que, una vez iniciadas, el programa pueda continuar con la siguiente instrucción, sin esperar a que finalice la anterior.
- Permite la ejecución de varias operaciones en paralelo, sucediendo al mismo tiempo.
- A este tipo de ejecución se la conoce como **asincrónica**.

Concepto Ejecución Asíncronica

- Para poder usar funciones que realicen operaciones no bloqueantes debemos **aprender a usarlas adecuadamente**, sin generar efectos adversos en forma accidental.
- Cuando el código se ejecuta en forma sincrónica, establecer el orden de ejecución consiste en decidir qué instrucción escribir primero.
- Cuando se trata de **ejecución asíncronica**, sólo sabemos en qué orden comenzarán su ejecución las instrucciones, pero **no sabemos en qué momento ni en qué orden terminarán de ejecutarse**.

Veamos un ejemplo rápido... 



***¿Alguna pregunta hasta
ahora?***

Callbacks



Recordemos que... 

 ***FUNCIÓN === OBJETO*** 



```
1  const sumar = (a, b) => a + b;  
2  const resultado = { total: 0 }
```

 ***FUNCIÓN === PARÁMETRO*** 

FUNCIÓN === PARÁMETRO

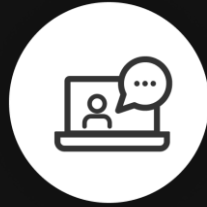


```
1 resultado.total = ejecutarOperacion(2, 3, sumar)
```

¡Vamos al código... !



- Definiremos una función llamada **operación** que reciba como parámetro dos valores y una función con la operación que va a realizar. Deberá retornar el resultado.
- Definiremos las siguientes funciones: suma, resta, multiplicación y división. Estas recibirán dos valores y devolverán el resultado. Serán pasadas como parámetro en la llamada a la función **operación**
- Todas las funciones tendrán que ser realizadas con sintaxis flecha.



***¿Alguna pregunta hasta
ahora?***

¡Hagamos un pequeño inciso...  !

Timers

setTimeout



setTimeout

- ❑ ***setTimeout(function, milliseconds, param1, param2, ...)***
 - Es una función nativa, no hace falta importarla.
 - La función ***setTimeout()*** recibe un callback, y lo ejecuta después de un número específico de milisegundos.
 - Trabaja sobre un modelo asincrónico no bloqueante.

Veamos un ejemplo rápido... 

setInterval

setInterval



- ❑ ***setInterval(cb, milliseconds, param1, param2, ...): Object***
 - Es una función nativa, no hace falta importarla.
 - La función ***setInterval()*** también recibe un callback, pero a diferencia de ***setTimeout()*** lo ejecuta una y otra vez cada vez que se cumple la cantidad de milisegundos indicada.
 - Trabaja sobre un modelo asincrónico no bloqueante.
 - El método ***setInterval()*** continuará llamando al callback hasta que se llame a ***clearInterval()*** o se cierre la ventana.
 - El objeto devuelto por ***setInterval()*** se usa como argumento para llamar a la función ***clearInterval()***.

Veamos un ejemplo rápido... 



***¿Alguna pregunta hasta
ahora?***

¡Ahora sí, volvamos a los Callbacks!

Callbacks: Algunas convenciones



- El **callback** siempre es el **último parámetro**.
- El **callback** suele ser una función que **recibe dos parámetros**.
- La **función llama** al callback **al terminar** de ejecutar todas sus operaciones.
- **Si la operación fue exitosa**, la función llamará al callback pasando null como primer parámetro y si generó algún resultado este se pasará como segundo parámetro.
- **Si la operación resultó en un error**, la función llamará al callback pasando el error obtenido como primer parámetro.

Ejemplo convenciones

Desde el lado del callback, estas funciones deberán saber cómo manejar los parámetros. Por este motivo, nos encontraremos muy a menudo con la siguiente estructura

```
const ejemploCallback = (error, resultado) => {  
  if (error) {  
    // hacer algo con el error!  
  } else {  
    // hacer algo con el resultado!  
  }  
};
```


¡Vamos al código... !



- Definiremos una función llamada **división** la cual recibirá por parámetro el **dividendo** y el **divisor**, seguido de un **callback** que es quien entregará el resultado.
- Debemos aplicar la convención explicada anteriormente, donde si la división nos da un **error**, este sea el **primer parámetro** del callback, de lo contrario el **resultado** se entregará en un **segundo parámetro** del callback.
- Cabe destacar que si no tenemos error, el primer parámetro será **null**.

Callbacks anidados

```
asyncFunctionA(data, array, function(err, result){
  asyncFuctionB(data, array, function(err,result){
    asyncFunctionC(data, array, function(err, result){
      asyncFunctionD(data, array, function(err, result){
        asyncFunctionE(data, array,function(err, result){
          asyncFunctionF(data, array, function(err,result){
            asyncFunctionH(data, array, function(err,result){
              //Do something
            })
          })
        })
      })
    })
  })
})
```

Concepto



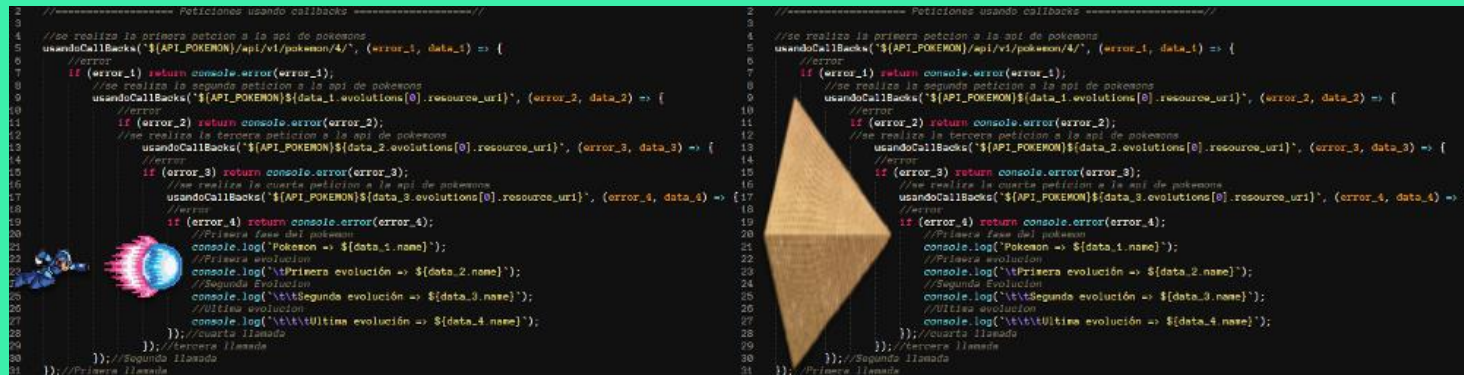
- Es un fragmento de código en el que **una función llama a un callback, y este a otro** callback, y este a otro, y así sucesivamente.
- Son **operaciones encadenadas**, en serie.
- Si el nivel de anidamiento es grande, se puede producir el llamado ***callback hell*** ó infierno de callbacks. También se conoce como ***pyramid of doom*** ó pirámide de la perdición.

Ejemplo Callback anidado

```
const copiarArchivo = (nombreArchivo, callback) => {  
  buscarArchivo(nombreArchivo, (error, archivo) => {  
    if (error) {  
      callback(error)  
    } else {  
      leerArchivo(nombreArchivo, 'utf-8', (error, texto) => {  
        if (error) {  
          callback(error)  
        } else {  
          const nombreCopia = nombreArchivo + '.copy'  
          escribirArchivo(nombreCopia, texto, (error) => {  
            if (error) {  
              callback(error)  
            } else {  
              callback(null)  
            }  
          })  
        }  
      })  
    }  
  })  
}
```

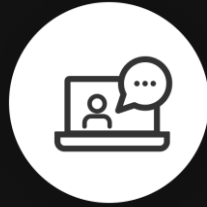
¡Atención!

Al tipo de estructura de código se le ha denominado **callbacks hell** o **pyramid of doom**, ya que las funciones se van encadenando de forma que la indentación del código se vuelve bastante prominente y dificulta la comprensión del mismo.



```
2 //===== Peticiones usando callbacks =====//
3
4 //se realiza la primera petición a la api de pokemons
5 usandoCallbacks(`${API_POKEEMON}/api/v1/pokemon/4/`, (error_1, data_1) => {
6   //error
7   if (error_1) return console.error(error_1);
8   //se realiza la segunda petición a la api de pokemons
9   usandoCallbacks(`${API_POKEEMON}${data_1.evolutions[0].resource_uri}`, (error_2, data_2) => {
10     //error
11     if (error_2) return console.error(error_2);
12     //se realiza la tercera petición a la api de pokemons
13     usandoCallbacks(`${API_POKEEMON}${data_2.evolutions[0].resource_uri}`, (error_3, data_3) => {
14       //error
15       if (error_3) return console.error(error_3);
16       //se realiza la cuarta petición a la api de pokemons
17       usandoCallbacks(`${API_POKEEMON}${data_3.evolutions[0].resource_uri}`, (error_4, data_4) => {
18         //error
19         if (error_4) return console.error(error_4);
20         //Primera fase del pokemon
21         console.log('Pokemon => ${data_1.name}');
22         //Primera evolución
23         console.log(`${data_1.name} => ${data_2.name}`);
24         //Segunda Evolución
25         console.log(`${data_2.name} => ${data_3.name}`);
26         //Última evolución
27         console.log(`${data_3.name} => ${data_4.name}`);
28       }); //cuarta llamada
29     }); //tercera llamada
30   }); //segunda llamada
31 }); //primera llamada
```

```
2 //===== Peticiones usando callbacks =====//
3
4 //se realiza la primera petición a la api de pokemons
5 usandoCallbacks(`${API_POKEEMON}/api/v1/pokemon/4/`, (error_1, data_1) => {
6   //error
7   if (error_1) return console.error(error_1);
8   //se realiza la segunda petición a la api de pokemons
9   usandoCallbacks(`${API_POKEEMON}${data_1.evolutions[0].resource_uri}`, (error_2, data_2) => {
10     //error
11     if (error_2) return console.error(error_2);
12     //se realiza la tercera petición a la api de pokemons
13     usandoCallbacks(`${API_POKEEMON}${data_2.evolutions[0].resource_uri}`, (error_3, data_3) => {
14       //error
15       if (error_3) return console.error(error_3);
16       //se realiza la cuarta petición a la api de pokemons
17       usandoCallbacks(`${API_POKEEMON}${data_3.evolutions[0].resource_uri}`, (error_4, data_4) => {
18         //error
19         if (error_4) return console.error(error_4);
20         //Primera fase del pokemon
21         console.log('Pokemon => ${data_1.name}');
22         //Primera evolución
23         console.log(`${data_1.name} => ${data_2.name}`);
24         //Segunda Evolución
25         console.log(`${data_2.name} => ${data_3.name}`);
26         //Última evolución
27         console.log(`${data_3.name} => ${data_4.name}`);
28       }); //cuarta llamada
29     }); //tercera llamada
30   }); //segunda llamada
31 }); //primera llamada
```

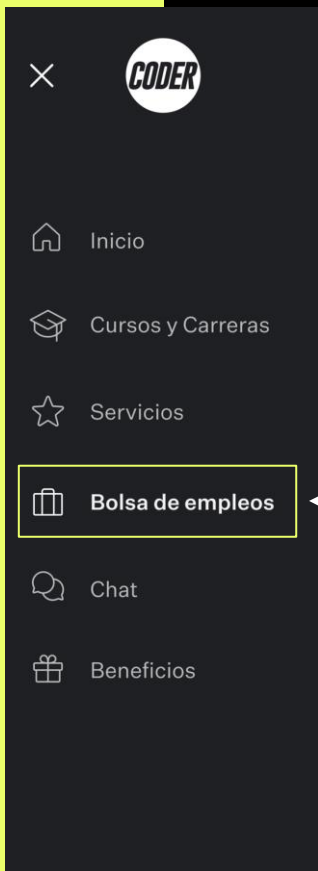


***¿Cómo vamos hasta
ahora?***



BREAK

¡5/10 MINUTOS Y VOLVEMOS!



Nuevo

¡Lanzamos la Bolsa de Empleos!

Un espacio para seguir **potenciando tu carrera** y que tengas más **oportunidades de inserción laboral**.

Podrás encontrar la **Bolsa de Empleos** en el menú izquierdo de la plataforma.

Te invitamos a conocerla y ¡postularte a tu futuro trabajo!

Conócela

Promesas



Promesas



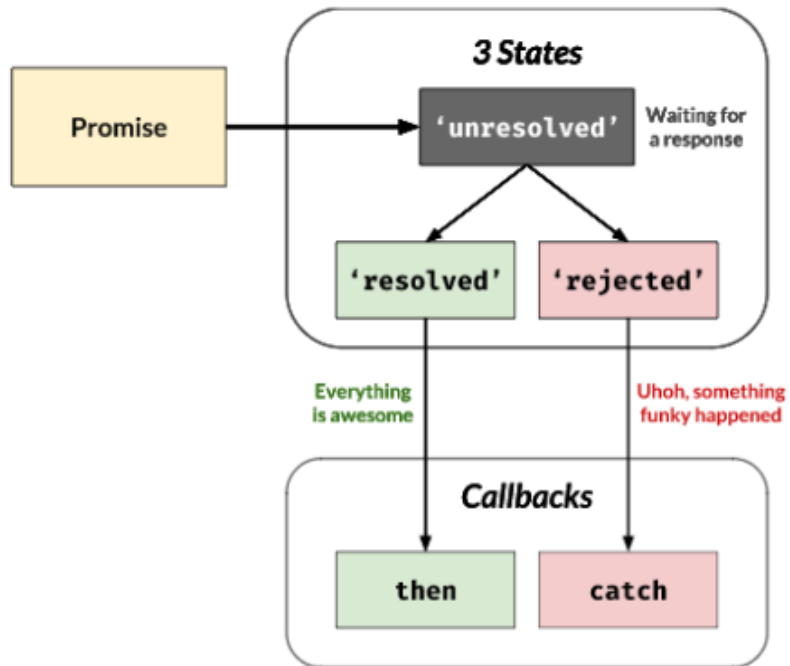
- Una Promesa es un objeto que encapsula una operación, y que permite definir acciones a tomar luego de finalizada dicha operación, según el resultado de la misma. Para ello, permite **asociar manejadores** que actuarán sobre un eventual valor (resultado) en caso de éxito, o la razón de falla (error) en caso de una falla.
- Al igual que con los callbacks, este mecanismo permite **definir desde afuera** de una función un bloque de código que **se ejecutará dentro** de esa función, dependiendo del resultado. A diferencia de los callbacks, en este caso se definirán dos manejadores en lugar de uno solo. Esto permite evitar *callback hells* como vimos previamente.

Estados de una promesa



El estado inicial de una promesa es:

- **Pendiente (pending):** Una vez que la operación contenida se resuelve, el estado de la promesa pasa a:
- **Cumplida (fulfilled):** la operación salió bien, y su resultado será manejado por el callback asignado mediante el método `.then()`.
- **Rechazada (rejected):** la operación falló, y su error será manejado por el callback asignado mediante el método `.catch()`.



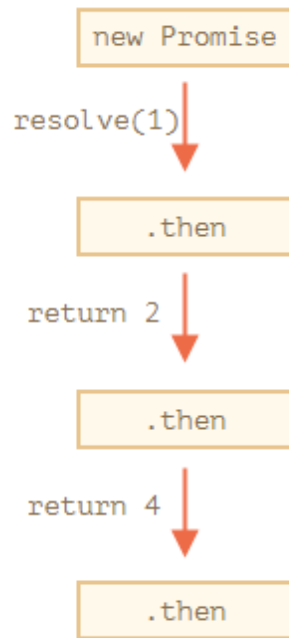
Veamos al IDE... 

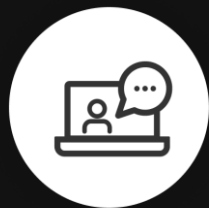


Encadenamiento de promesas

Una llamada a `promise.then()` devuelve otra promesa, para que podamos llamar al siguiente `.then()`.

```
new Promise(function (resolve, reject) {  
  setTimeout(() => resolve(1), 1000); // (*)  
})  
  .then(result => { // (**)  
    console.log(result); // 1  
    return result * 2;  
  })  
  .then(result => { // (***)  
    console.log(result); // 2  
    return result * 2;  
  })  
  .then(result => {  
    console.log(result); // 4  
    return result * 2;  
  });  
  
//1) La promesa inicial se resuelve en 1 segundo (*)  
//2) Entonces se llama el controlador .then (**).  
//3) El valor que devuelve se pasa al siguiente controlador .then (***)
```





¡Vamos al código!



- Desarrollaremos un **flujo de compra** (simulado usando la función *setTimeout*) de un ecommerce.
- Partiremos desde la **reserva de artículos** seguido de la **ejecución del pago** de los artículos y la posterior **notificación a proveedores** y el **usuario**.
- Recuerda que luego de una compra exitosa el **carrito de compra** debe ser vaciado.
- Todo esto usando encadenamiento de promesas.

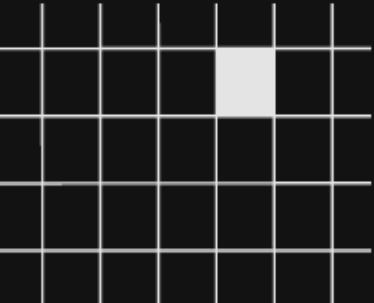
¿PREGUNTAS?





¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- Funciones
 - Callbacks
 - Promesas
 - Ejecución sincrónica/asincrónica
- 



OPINA Y VALORA ESTA CLASE

#DEMOCRATIZANDO LA EDUCACIÓN