# ¡Les damos la bienvenida!

¿Comenzamos?

**Unidad 02.** PROGRAMACIÓN BACKEND

# Programación backend Avanzada

CODERHOUSE

# Objetivos de la clase

- Conocer las características de ECMAScript
- Aplicar los conceptos incorporados en el desarrollo backend

# Objetivos de la clase

- Repasar las funciones en Javascript
- Comprender un callback y cómo se relacionan las funciones con éstos.
- Usar promesas en Javascript
- Concretar la diferencia entre programación sincrónica y asincrónica.

# Javascript y ES7

# Ejemplo en vivo

- ✓ Utilización del operador exponencial y manejo de array con includes.

# Ejemplo de uso de ** y Array.includes

exponential_includes.js

```
//Exponential  ** permite hacer el equivalente de la operación Math.pow(base,exp), para elevar un valor base a un exponente dado.
let valoresBase = [1,2,3,4,5,6] //Tenemos un conjunto de valores base.
let nuevosValores = valoresBase.map((numero,indice)=>numero**indice);
console.log(nuevosValores); // resultado: [ 1, 2, 9, 64, 625, 7776 ]
/**
 * El código mostrado arriba toma un arreglo de valores base y, con ayuda del operador map, utiliza el operador exponencial para elevar el
 * valor base, por su índice: (1**0,2**1,3**2,4**3,5**4,6**5)
 */

//Includes: Corrobora si algún elemento existe dentro del arreglo.
let nombres = ['Juan','Camilo','María','Ana','Humberto'];
if(nombres.includes('Camilo')){//includes devolverá sólo true o false según sea el caso, por lo cual podemos usarlo dentro del if
    console.log("Camilo sí aparece dentro del arreglo")
}else{
    console.log("Nombre no encontrado en la base de datos")
}
```

CODERHOUSE

# Javascript y ES8

# Ejemplo en vivo

- ✓ Utilización de operadores nullish
- ✓ Utilización de variables privadas

# Ejemplo de uso de Object.entries, Object.keys, Object.values

entries_values_keys.js

```
}
let parLlaveValor = Object.entries(impuestos)
console.log(parLlaveValor) //resultado: [ [impuesto1,2341], [impuesto2,341], [impuesto3,4611], [impuesto4,111] ]
/**
 * Notamos cómo Object.keys obtiene en arreglos individuales la propiedad con su valor, en caso de que necesitemos utilizarlas por separado.
 */
let soloPropiedades = Object.keys(impuestos)
console.log(soloPropiedades)//resultado: [impuesto1,impuesto2,impuesto3,impuesto4]
/**
 * Ahora podemos obtener sólo las propiedades del objeto, sin necesidad de su valor, este método es MUY ÚTIL en códigos profesionales,
 * sin embargo, por cuestiones de complejidad se abordará en elementos prácticos más adelante.
 */
let soloValores = Object.values(impuestos)
console.log(soloValores)//resultado : [2341,341,4611,111]
/**
 * Teniendo sólo los valores del objeto, podemos utilizarlos para hacer un total (En este ejemplo nos apoyamos de un método ya existente
 * conocido como .reduce)
 */
let impuestosTotales = soloValores.reduce((valorInicial,valorAcumulado)=>valorAcumulado+valorInicial);
console.log(impuestosTotales);// 7404, total de todos los impuestos.
```

Nota: reduce es un método complejo al momento de utilizarlo, puede aprovecharse este ejemplo para ahondar en él.

# Javascript y ES9

# Ejemplo en vivo

- ✓ Utilización básica de operador rest y operador spread en los objetos.

# Ejemplo de uso de spread operator y rest operator

spread_rest.js

```js
//Dados los siguientes objetos
let objeto1 = {
    propiedad1:2,
    propiedad2:"b",
    propiedad3:true
}
let objeto2 = {
    propiedad1:"c",
    propiedad2:[2,3,5,6,7]
}
//SPREAD OPERATOR Nos sirve para hacer una destructuración del objeto, para poder utilizar sólo las propiedades que queremos.
let  {propiedad1,propiedad2} = objeto1; //Tomamos el objeto1 y lo "rompemos" para obtener sólo las primeras dos propiedades.
let objeto3 = {...objeto1,...objeto2} //Spread operator también se puede utilizar para vaciar propiedades de un objeto en otro objeto nuevo.

console.log(objeto3); // resultado :{ propiedad1: 'c', propiedad2: [ 2, 3, 5, 6, 7 ], propiedad3: true }
//Notamos cómo, si dos elementos comparten el mismo nombre de propiedad, se superponen, por lo que propiedad1 y propiedad2 del objeto uno ya
//no existen dentro del objeto 3, sino que fueron "superpuestos" por propiedad1 y propiedad2 del objeto 2.

//REST OPERATOR nos servirá para obtener un objeto SÓLO con las propiedades RESTANTES del objeto que hayamos destructurado, por ejemplo:
let objeto4 = {
    a : 1,
    b : 2,
    c : 3
}
let  {a,...rest} = objeto4; //Indicamos que queremos trabajar con la propiedad a, y guardar en un objeto el resto de las propiedades de ese
//objeto, en caso de que los necesitemos más adelante.
console.log(rest); //resultado: { b: 2, c: 3 }
```

# Utilización ES6-ES9

Pongamos en práctica algunos de los módulos vistos en los ejemplos

Duración: **15-20 min**

CODERHOUSE

ACTIVIDAD EN CLASE

# Utilización ES6-ES9

**Descripción de la actividad.**

Dados los objetos indicados en la siguiente diapositiva:

- ✓ Realizar una lista nueva (array) que contenga todos los tipos de productos (no cantidades), consejo: utilizar **Object.keys** y **Array.includes.** Mostrar el array por consola.
- ✓ Posteriormente, obtener el total de productos vendidos por todos los objetos (utilizar **Object.values)**

```
const objetos =  [
    {
        manzanas:3,
        peras:2,
        carne:1,
        jugos:5,
        dulces:2
    },
    {
        manzanas:1,
        sandias:1,
        huevos:6,
        jugos:1,
        panes:4
    }
]
```

# Javascript y ES10

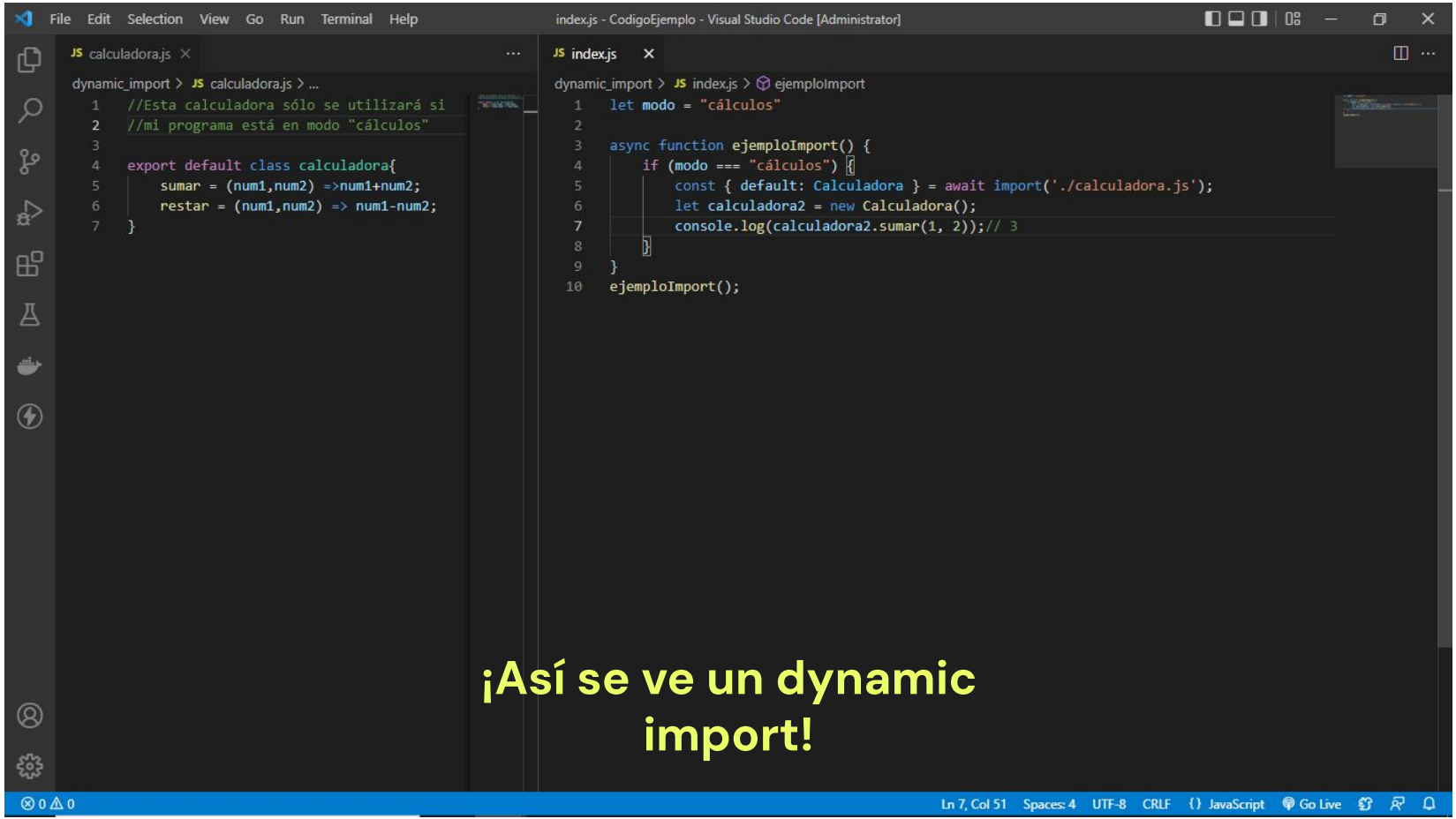
# Dynamic import

Uno de los problemas principales de los imports tradicionales, es que terminamos importando TODOS LOS MÓDULOS, aun cuando no estamos utilizando todos al mismo tiempo.

Con dynamic import, esto cambia.

Dynamic import permite importar **sólo los módulos** que necesito según una situación particular, lo cual permite optimizar la utilización de recursos, al pedir a la computadora sólo lo que estaré utilizando.

Es utilizado principalmente en códigos que utilizan el patrón de diseño Factory (se abordará más adelante).


File Edit Selection View Go Run Terminal Help
index.js - CodigoEjemplo - Visual Studio Code [Administrator]
calculadora.js
dynamic_import > calculadora.js > ...
1 //Esta calculadora sólo se utilizará si
2 //mi programa está en modo "cálculos"
3
4 export default class calculadora{
5 sumar = (num1,num2) =>num1+num2;
6 restar = (num1,num2) => num1-num2;
7 }
index.js
dynamic_import > index.js > ejemploImport
1 let modo = "cálculos"
2
3 async function ejemploImport() {
4 if (modo === "cálculos") {
5 const { default: Calculadora } = await import('./calculadora.js');
6 let calculadora2 = new Calculadora();
7 console.log(calculadora2.sumar(1, 2));// 3
8 }
9 }
10 ejemploImport();
¡Así se ve un dynamic import!
Ln 7, Col 51 Spaces: 4 UTF-8 CRLF {} JavaScript Go Live

CODERHOUSE

# Ejemplo en vivo

- ✓ Validación de cadena con trim
- ✓ Aplanado de Array con múltiple anidación

# Ejemplo de uso de los métodos trim y flat

trim_flat.js

```
let cadena1 = `              hola`//El mensaje no debería enviarse de esa manera, ya que consume espacio innecesario a almacenar.
console.log(cadena1.trim());//resultado: "hola"
//Podemos validar también si me enviaron un mensaje vacío, para saber si almacenarlo o no enviar el mensaje (simulando un chat)
let mensajes=[];
let intentoDeMensaje=`                          `
if(intentoDeMensaje.trim().length>0){//Por lo menos hay un caracter (no espacio) para enviar al usuario, entonces es un mensaje válido.
    mensajes.push(intentoDeMensaje.trim());
}else{//Si la condición entra a else, es porque el mensaje venía vacío y por lo tanto no deberíamos guardarlo ni enviarlo al otro usuario
    console.log("Mensaje vacío, para poder enviar un mensaje, favor de escribir algo")
}
/**
 * Sin el método trim, permitimos muchas brechas de seguridad al momento de querer procesar cadenas de texto, de modo que es bueno limitarlas
 * a un formato que dominemos (sin espacios extras ni cadenas aparentemente vacías)
 */

//Uso de flat
let arrayAnidado = [1,32,4,5,6,[1,4,5,1],[3411,3,4]]//Anteriormente, para poder obtener los valores internos de cada arreglo, usábamos
// una técnica llamada "Recursividad", esta vez es más sencillo.
console.log(arrayAnidado.flat());//resultado: [1,32,4,5,6,1,4,5,1,3411,3,4], notamos que ya no hay más anidación y podemos manejarlos mejor.
```

CODERHOUSE

# Javascript y ES11

# Ejemplo en vivo

- ✓ Explicación de asignación de variable a partir de un nullish, para entender su diferencia con el operador OR ||
- ✓ Explicación de una variable privada en una clase.

# Ejemplo de operador nullish y variables privadas

nullish_privateVariables.js

```
/**
 * El operador nullish difiere del operador ||, ya que || también ignora valores falseys.
 */
let variableDePrueba = 0 //Reemplazar esta variable de prueba por diferentes valores null, undefined, y falseys

let variableAsignable = variableDePrueba || "Sin valor";
console.log(variableAsignable); // Tal vez yo necesitaba el valor 0, pero al utilizar ||, se toma el valor por default
let variableConNullish = variableDePrueba??"Sin valor";
console.log(variableConNullish);//Notamos que a este punto sí podemos tomar el valor 0 que necesitamos.
/**
 * Notamos que con Nullish realmente sólo nos interesa que el valor NO SEA undefined o null. Todo lo demás podemos asignarlo,
 */

// Sobre variable privada
class Persona {
    #fullname;//Primero declaramos la variable antes del constructor para poder construirla con los valores del constructor
    constructor(nombre,apellido){
        this.nombre=nombre;
        this.apellido=apellido;
        this.#fullname=`${this.nombre} ${this.apellido}` //Asignamos el valor de la variable privada.
    }
    //Esta variable la podemos utilizar de manera interna. No se puede acceder a ella por fuera.
    getFullName = () =>{
        return this.#fullname; //La única forma en la cual podemos obtener el valor de esa variable privada es pidiéndola desde un método
        //Permite utilizar variables, pero evitando que se puedan modificar desde fuera (por seguridad).
    }
    #metodoPrivado = () =>{//También podemos declarar métodos privados, para que sólo sean usados dentro de la clase, y no llamados por fuera
        //Aquí realizamos tareas que queremos que se ejecuten sólo de manera interna en la clase, no podemos mandar a llamar este método
    }
}
let instancia1 = new Persona('Mauricio','Espinosa')

console.log(instancia1.getFullName())// Mauricio Espinosa
```

CODERHOUSE

# Hands on lab

En esta instancia de la clase **repasaremos** algunos de los conceptos vistos en clase con una aplicación

**¿De qué manera?**
El profesor, demostrará cómo hacerlo y tú lo puedes ir replicando en tu computadora. Si surgen dudas las puedes compartir para resolverlas en conjunto de la mano de los tutores.

Tiempo estimado: **20 minutos**

CODERHOUSE

HANDS ON LAB

# Registrador de tickets de eventos

¿Cómo lo hacemos? **Se creará una clase que permitirá llevar una gestión completa de usuarios que deseen acceder a dichos eventos.**

- ✓ Definir clase TicketManager, el cual tendrá un arreglo de eventos que iniciará vacío
- ✓ La clase debe contar con una variable privada “precioBaseDeGanancia”, la cual añadirá un costo adicional al precio de cada evento.
- ✓ Debe contar con el método “getEventos” El cual mostrará los eventos guardados.
- ✓ Debe contar con el método “agregarEvento” El cual recibirá los siguientes parámetros:
  - nombre
  - lugar
  - precio (deberá agregarse un 0.15 del valor original)
  - capacidad (50 por defecto)
  - fecha (hoy por defecto)

  El método deberá crear además el campo id autoincrementable y el campo “participantes” que siempre iniciará con un arreglo vacío.

HANDS ON LAB

# Registrador de tickets de eventos

- ✓ Debe contar con un método “agregarUsuario” El cual recibirá:
  - id del evento (debe existir, agregar validaciones)
  - id del usuario

  El método debe evaluar que el evento exista y que el usuario no haya estado registrado previamente (validación de fecha y capacidad se evitará para no alargar el reto)
  Si todo está en orden, debe agregar el id del usuario en el arreglo “participantes” de ese evento.
- ✓ Debe contar con un método “ponerEventoEnGira” El cual recibirá:
  - id del evento
  - nueva localidad
  - nueva fecha

  El método debe copiar el evento existente, con una nueva localidad, nueva fecha, nuevo id y sus participantes vacíos (Usar spread operator para el resto de las propiedades)

# Break

¡10 minutos y volvemos!

# Ejemplo map con callback

- ✓ Utilizaremos la función map, haciendo énfasis en el callback, para entenderlo de manera convencional
- ✓ Se explicará el funcionamiento interno de la función map para analizar el momento de utilización del callback.

# Ejemplo de uso de un callback en la función map

callbacks1.js > ...

```js
//Utilizaremos este arreglo de prueba
let valoresOriginales = [1,2,3,4,5];
//Estamos acostumbrados a leer una función map de la siguiente forma:
let nuevosValores = valoresOriginales.map(x=>x+1); //nuevosValores tendrá: [2,3,4,5,6]
/**
 * Sin embargo, lo que metemos dentro de la función map es una función (flecha, más específicamente), que indica que se sume en 1 el valor
 * del número que esté en el arreglo.
 * ¿Siempre tenemos que sumar 1? ¡No! Nosotros podemos meter la operación que queramos, ¡y map la ejecutará de manera interna!
 */
let otrosValores = valoresOriginales.map(x=>x*2); //otrosValores tendrá: [2,4,6,8,10]
let masValores = valoresOriginales.map(x=>"a"); //masValores tendrá: ["a","a","a","a","a"]
/**
 * Notamos que, no importa cuánto cambie la función que esté metiendo dentro de map, map está hecho para RECIBIR UNA FUNCIÓN COMO PARÁMETRO
 * y poder ejecutarla cuando lo considere pertinente. Ahora. Si estructuramos el callback por fuera.
 */
const funcionCallback = (valor) => { //Función que evalúa si el valor del arreglo es un número par
    if(valor%2===0){
        return valor
    }
    else{
        return "no es par"
    }
}
const evaluacionDePares = valoresOriginales.map(funcionCallback); //Estoy pasando la función completa como argumento de la función map
console.log(evaluacionDePares) // el resultado será: ["no es par",2,"no es par",4,"no es par"];
```

CODERHOUSE

# Ejemplo descomposición de función map

- ✓ Se hará descomposición de la función map para poder analizarla por dentro. El objetivo es localizar en qué punto la función “map” llamaría de manera interna el callback

# Ejemplo de recreación interna de función map para localizar su callback

callbacks2.js > ...

```
1  //Usaremos el arreglo de prueba
2  let arregloDePrueba = [1,2,3,4,5];
3  const miFuncionMap = (arreglo,callback) =>{
4      let nuevoArreglo = [];
5      for(let i=0;i<arreglo.length;i++){
6          let nuevoValor = callback(arreglo[i]);//Nota como, el callback que recibí aquí arriba lo estoy ejecutando en este punto
7          nuevoArreglo.push(nuevoValor);
8      }
9      return nuevoArreglo;
10 }
11 //pongamos en comparación nuestra nueva función CON UN CALLBACK y la función map
12 let nuevoArregloPropio = miFuncionMap(arregloDePrueba,x=>x*2); //El nuevo arreglo será: [2,4,6,8,10]
13 let nuevoArregloConMap = arregloDePrueba.map(x=>x*2); //El arreglo será: [2,4,6,8,10]
14 /**
15  * Nota que no hay diferencia. Acabamos de recrear la función map para entender su funcionamiento interno y viendo DÓNDE está utilizando el
16  * callback que enviamos como parámetro
17  */
18
19
20 /**
21  * EXTRA: Si queremos que la función se ejecute sobre el mismo arreglo y no tener que pasarlo como parámetro, debemos agregar nuestra nueva
22  * función en el prototipo del objeto Array
23  */
24 Array.prototype.miPropiaFuncionMap = function(callback){
25     let nuevoArreglo = [];
26     for(let i=0;i<this.length;i++){
27         let nuevoValor = callback(this[i]);
28         nuevoArreglo.push(nuevoValor);
29     }
30     return nuevoArreglo;
31 }
32 let arregloPrueba = [1,2,3,4,5];
33 let nuevosValores = arregloPrueba.miPropiaFuncionMap(x=>x+1);
```

CODERHOUSE

# Ejemplo callback con operaciones

- ✓ Se crearán cuatro funciones: sumar, restar, multiplicar y dividir.
- ✓ Además, se proporcionará otra función operación, que recibirá como callback cualquiera de estas tres funciones para ejecutarla.

# Ejemplo de callbacks utilizando diferentes operaciones

```
const sumar = (numero1,numero2) =>numero1+numero2;
const restar = (numero1,numero2) => numero1-numero2;
const multiplicar = (numero1,numero2) =>numero1*numero2;
const dividir = (numero1,numero2) => numero1/numero2;

const realizarOperacion = (numero1,numero2,callback) =>{
    console.log("¡Voy a realizar una operación, no sé cuál, pero lo haré!");
    let resultado = callback(numero1,numero2);
    /**
     * No sabemos cuál de las 4 funciones será, pero eso a nosotros no nos importa, nosotros sólo la ejecutamos y devolvemos el resultado.
     */
    console.log(`El resultado de la operación que no supe cuál fue, es: ${resultado}`)
}

realizarOperacion(2,5,sumar);// El resultado de la operación que no supe cuál fue, es: 7
realizarOperacion(2,5,restar);// El resultado de la operación que no supe cuál fue, es: -3
realizarOperacion(2,5,multiplicar);// El resultado de la operación que no supe cuál fue, es: 10
realizarOperacion(2,5,dividir) //El resultado de la operación que no supe cuál fue es: 0.4
/**
 * Analicemos, realizarOperacion recibe una función de callback y la ejecuta por dentro, peroooooo... ¡No tiene idea de qué hace la función, sólo
 * la ejecuta! Así que hay que tener siempre mucho cuidado de lo que pasamos por callback, ya que en caso de pasar una función que no sea
 * compatible con los valores que esté trabajando la función, podríamos romper el código al cual le pasamos el callback.
 */
```

# 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 de convenciones

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

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

# Callbacks anidados

# Callbacks anidados

En algún momento el mundo laboral te exige hacer más que sólo una suma o una resta. Nos encontraremos con procesos que requieren operaciones de más pasos. Si nosotros trabajamos con callbacks, podemos encadenar un conjunto de operaciones secuenciales.

Así, **un callback puede llamar a otro callback, y este puede llamar a otro callback, y este a otro...**

```
// Callback Hell


a(function (resultsFromA) {
    b(resultsFromA, function (resultsFromB) {
        c(resultsFromB, function (resultsFromC) {
            d(resultsFromC, function (resultsFromD) {
                e(resultsFromD, function (resultsFromE) {
                    f(resultsFromE, function (resultsFromF) {
                        console.log(resultsFromF);
                    })
                })
            })
        })
    })
});

```

CODERHOUSE

# Ejemplo de 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)
                        }
                    })
                }
            })
        }
    })
}
```

# Creando nuestra primera promesa

# Ejemplo

- ✓ Se creará una promesa, haciendo énfasis en los casos de resolución (resolve) y en los casos de rechazo (reject).

# Ejemplo de creación de una promesa

promises1.js

```
promises1.js > dividir
const dividir = (dividendo,divisor) =>{
    return new Promise((resolve,reject)=>{//Nota que al crear una promise, estamos pasando un callback con dos parámetros: resolve y reject
        if(divisor===0){
            reject('No se pueden hacer divisiones entre cero')
            /**
             * Rechazamos la operación porque no es posible trabajar con una división entre cero, no puedo cumplirle al usuario la promesa que
             * le hice sobre dividir sus números.
             */
        }else{
            resolve(dividendo/divisor);
            /**
             * Si los valores son válidos, entonces sí puedo cumplir la promesa que hice al usuario de dividir sus números, por lo tanto,
             * utilizaremos el valor
             */
        }
    })
}
```

# Ejemplo

- ✓ Se utilizará la función “dividir” que se creó en el ejemplo anterior, para profundizar sobre los operadores .then y .catch

# Ejemplo de uso de una promesa

```
/**
 * Una vez creada nuestra promesa, es hora de comenzar a utilizarla.
 */
dividir(6,3) //Llamamos la función como cualquier caso.
.then(resultado=>{
    console.log(resultado) //en este caso, al no ser división entre 0, la promesa se cumplirá y el resultado será 2
    /**
     * Programamos el then para recibir cualquier "RESOLVE" por parte de la promesa (es decir, utilizamos then para recibir los casos en los
     * cuales sabemos que la función va a salir bien). el parámetro "resultado" será el valor que devuelva el resolve de la promesa.
     */
})
.catch(error=>{
    console.log(error)
    /**
     * Programamos también un catch para recibir cualquier "REJECT" por parte de la promesa (es decir, utilizamos catch para ATRAPAR los
     * errores que nos arroje la promesa, para poder entender la razón por la cual nuestra promesa no pudo cumplirse correctamente), el
     * parámetro "error" será el valor que devuelva el reject dentro de la promesa.
     */
})

//Otro ejemplo
dividir(5,0)
.then(resultado=>{
    console.log(resultado)
})
.catch(error=>{
    console.log(error)
    /**
     * En este caso, como el divisor sí es cero, la promesa no podrá resolverse y entrará en este bloque catch, indicando la razón por la cual
     * no pudo resolverse dicha promesa. De esta manera, tenemos control sobre los casos donde todo salga bien, pero TAMBIÉN controlamos los
     * casos donde algo salga mal.
     */
})
```

CODERHOUSE

# Sincronismo vs Asincronismo

# Ejemplo de ejecución sincrónica

- ✓ En todo momento, sólo se están ejecutando las instrucciones de una sola de las funciones a la vez. O sea, debe finalizar una función para poder continuar con la otra.
- ✓ El fin de una función marca el inicio de la siguiente, y el fin de ésta, el inicio de la que le sigue, y así sucesivamente, describiendo una secuencia que ocurre en una única línea de tiempo.

```
function funA() {
    console.log(1)
    funB()
    console.log(2)
}
function funB() {
    console.log(3)
    funC()
    console.log(4)
}
function funC() {
    console.log(5)
}

funA()

//Al ejecutar la función funA()
//se muestra lo siguiente por pantalla:
1
3
5
4
2
```

CODERHOUSE

# Ejemplo de ejecución asincrónica

- ✓ En el ejemplo no se bloquea la ejecución normal del programa y se permite que este se siga ejecutando.
- ✓ La ejecución de la operación de escritura “comienza” e inmediatamente cede el control a la siguiente instrucción, que escribe por pantalla el mensaje de finalización.
- ✓ Cuando la operación de escritura termina, ejecuta el callback que informará por pantalla que la escritura se realizó con éxito.

```
const escribirArchivo = require('./escrArch.js')

console.log('inicio del programa')

// el creador de esta funcion la definió
// como no bloqueante. recibe un callback que
// se ejecutará al finalizar la escritura.
escribirArchivo('hola mundo', () => {
  console.log('terminé de escribir el archivo')
})

console.log('fin del programa')

// se mostrará por pantalla:
// > inicio del programa
// > fin del programa
// > terminé de escribir el archivo
```

CODERHOUSE

# Ejemplo en vivo

- ✓ Explicación sobre el uso de una función asíncrona aplicando async/await.
- ✓ Usaremos la misma promesa de dividir con la que hemos estado trabajando a lo largo de la clase.

# Ejemplo del entorno async/await

```
const dividir = (dividendo,divisor) =>{
    return new Promise((resolve,reject)=>{//Nota que al crear una promise, estamos pasando un callback con dos parámetros: resolve y reject
        if(divisor===0){
            reject('No se pueden hacer divisiones entre cero')
            /**
             * Rechazamos la operación porque no es posible trabajar con una división entre cero, no puedo cumplirle al usuario la promesa que
             * le hice sobre dividir sus números.
             */
        }else{
            resolve(dividendo/divisor);
            /**
             * Si los valores son válidos, entonces sí puedo cumplir la promesa que hice al usuario de dividir sus números, por lo tanto,
             * utilizaremos el valor
             */
        }
    })
}

const funcionAsincrona = async() =>{
    /**
     * Estamos inicializando un entorno completo asíncrono, todo lo que esté dentro de las llaves de la función se comportará no-bloqueante con
     * el exterior.
     */
    try{
        //Encerramos la operación a realizar en un bloque try, porque al ser una promesa, PODRÍA NO CUMPLIRSE, así que hay que estar prevenidos
        let resultado = await dividir(10,5) //Ya no hay .then, ahora sólo ESPERAMOS por el resultado de la promesa.
        console.log(resultado);
    }
    catch(error){
        //El bloque catch es obligatorio después de un try{} y sirve igual que el .catch, para poder ATRAPAR errores.
        console.log(error);
    }
}
funcionAsincrona(); //Ya que el entorno de ejecución asíncrono vive en una función, hay que ejecutarla al final.
```

CODERHOUSE

# Hands on lab

En esta instancia de la clase **ahondaremos sobre creación de promesas y el uso de async await** con un ejercicio práctico

**¿De qué manera?**
El profesor demostrará cómo hacerlo y tú lo puedes ir replicando en tu computadora. Si surgen dudas las puedes compartir para resolverlas en conjunto de la mano de los tutores.

Tiempo estimado: **15 minutos**

CODERHOUSE

HANDS ON LAB

# Calculadora positiva con promesas

¿Cómo lo hacemos? **Se crearán un conjunto de funciones gestionadas por promesas y un entorno ASÍNCRONO donde podremos ponerlas a prueba**

- ✓ Definir función suma:
  - Debe devolver una promesa que se resuelva siempre que ninguno de los dos sumandos sea 0
  - En caso de que algún sumando sea 0, rechazar la promesa indicando “Operación innecesaria”.
  - En caso de que la suma sea negativa, rechazar la promesa indicando “La calculadora sólo debe devolver valores positivos
- ✓ Definir función resta:
  - Debe devolver una promesa que se resuelva siempre que ninguno de los dos valores sea 0
  - En caso de que el minuendo o sustraendo sea 0, rechazar la promesa indicando “Operación inválida
  - En caso de que el valor de la resta sea menor que 0, rechazar la promesa indicando “La calculadora sólo puede devolver valores positivos”

HANDS ON LAB

# Calculadora positiva con promesas

- ✓ Definir una función multiplicación:
  - Debe devolver una promesa que se resuelva siempre que ninguno de los dos factores sea negativo
  - Si el producto es negativo, rechazar la oferta indicando “La calculadora sólo puede devolver valores positivos
- ✓ Definir la misma función división utilizada en esta clase.
- ✓ Definir una función asíncrona “cálculos”, y realizar pruebas utilizando async/await y try/catch

# Muchas gracias.

#DemocratizandoLaEducación