



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



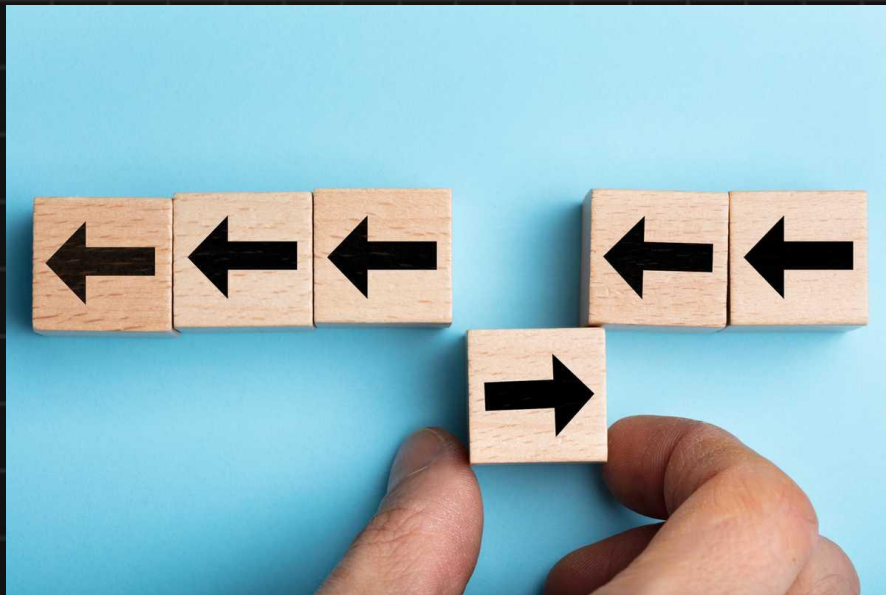
**Programación sincrónica y
asincrónica**

Clase 4



**Manejo de Archivos en
Javascript**

Funciones



Repasando...

Funciones en Javascript

A diagram illustrating the syntax of a JavaScript function. The code is: `function add (a, b) {
 return a + b;
}`. Handwritten labels in Spanish identify the parts: 'nombre' (name) points to 'add', 'parámetros' (parameters) points to '(a, b)', 'cuerpo/scope' (body/scope) points to the code inside the curly braces, and 'retorno' (return) points to the 'return' statement.

```
      nombre      parámetros  
      \          /  
function add ( a, b ) {  
    return a + b;    cuerpo/scope  
}  
      |  
    retorno
```

Declaración de funciones



Las funciones en Javascript tienen varias particularidades con respecto a otros lenguajes. Recordemos las formas para declarar una función:

Estilo clásico:

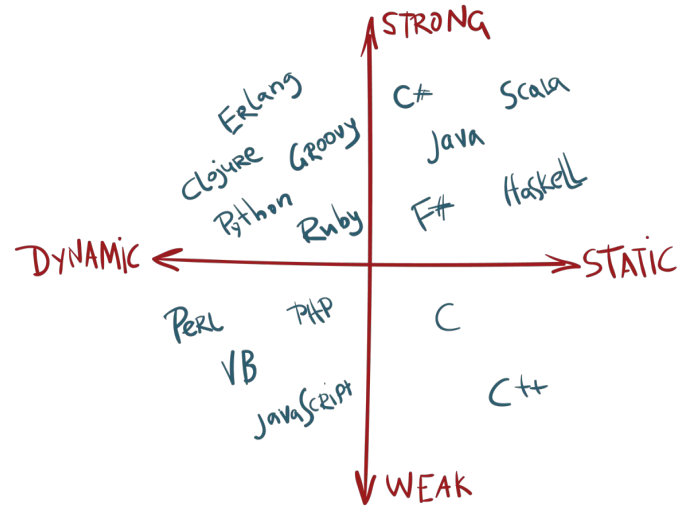
```
function mostrar(params) {  
    console.log(params)  
}
```

Llamada a la función: `mostrar(args)`

Declaración de funciones



Al ser **Javascript** un lenguaje que **no requiere especificar** el **tipo** de **dato** de sus **variables** (tipado dinámico), **tampoco** es necesario especificar el tipo de dato que devuelven las **funciones**, **ni** el tipo de dato de los **parámetros** que éstas reciben



Las funciones también son objetos

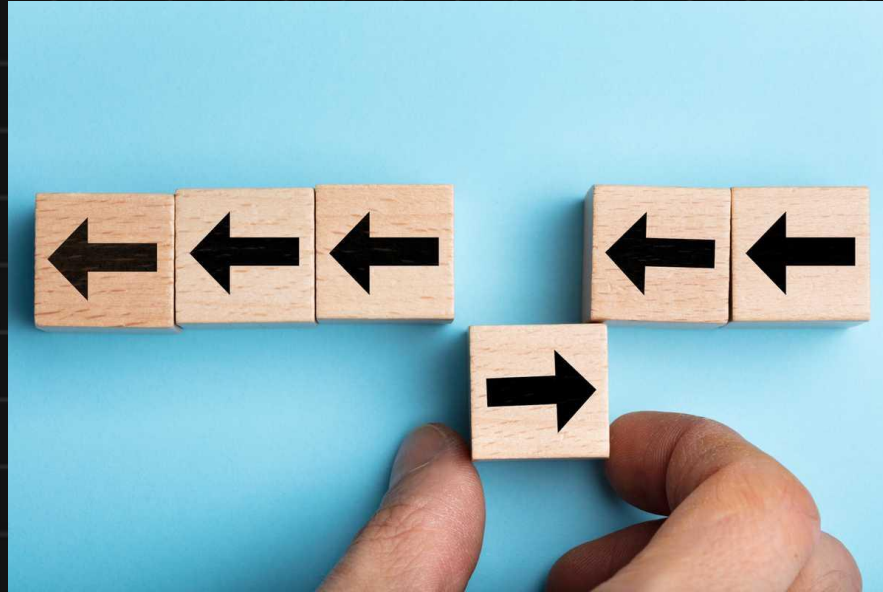


En JavaScript las **funciones se comportan como objetos**: es posible asignar una declaración de función a una variable.

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

La podemos ejecutar de la misma forma que una función clásica.

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
```

```
// -----
//     Ejemplos arrow function ó funciones flecha
// -----
const sumar = (a,b) => a + b    //sin llaves: retorno una sola línea

let op1 = 46, op2 = 57
let suma = sumar(op1, op2)
console.log(`La suma de ${op1} más ${op2} es igual a ${suma}`)
//-----
const sumar2 = (a,b) => {      //con llaves: múltiples instrucciones
    let s = a + b
    return s
}
console.log(`La suma2 de ${op1} más ${op2} es igual a ${sumar2(op1,op2)}`)
//-----
const dobleDe = a => a*2      //sin paréntesis: un solo argumento
console.log(`El doble de ${op1} es ${dobleDe(op1)}`)
//-----
const prtMensaje = () => {
    console.log('Hola')
}
prtMensaje()
//-----
// Cuando una arrow function retorna un objeto en una instrucción
// este debe ir entre paréntesis
const getPersona = () => ({nombre: 'Juan', edad: 34})
console.log(getPersona())
```

Callbacks



Callbacks...?

***Funciones como parámetros,
claro***

Funciones como parámetros..?!



Concepto



Como hemos visto, en Javascript es posible asignar una función a una variable. Esto es porque internamente, las funciones también son objetos (y las variables, referencias a esos objetos). Es por esto que **Javascript nos permite hacer que una función reciba como parámetro una referencia a otra función.**

```
const ejecutar = unaFuncion => unaFuncion()  
const saludar = () => console.log('saludos')  
ejecutar(saludar)
```

Ejemplos



Y como ya sabemos, donde puedo **usar una variable** puedo también **usar directamente el contenido** de esa variable. En el ejemplo, la función 'ejecutar' recibe una función anónima, y la ejecuta.

```
ejecutar(() => console.log('saludos'))
```

Esto también funciona con funciones anónimas con parámetros

```
const ejecutar = (unaFuncion, params) => unaFuncion(params)
const saludar = nombre => console.log(`saludos, ${nombre}`)
ejecutar(saludar, 'terricola')
```



Definiendo...



- Un callback es una **función que se envía como argumento a otra función.**
- La intención es que la función que hace de receptora ejecute la función que se le está pasando por parámetro.
- Podemos decir que la función “ejecutar” que usamos en el punto anterior “recibe un callback”.

¡Ejemplo!

Imaginemos que queremos que al finalizar una operación se ejecute un cierto código:

- Por ejemplo, queremos escribir un archivo y registrar en un log la hora en que se termine de escribir.
- Es probable que no se pueda saber con exactitud en qué momento va a finalizar.
- En algunos casos (ya veremos en cuáles) no podemos simplemente ejecutar la de escritura y luego, a continuación, guardar el log.
- En estos escenarios, las funciones deben recibir como último parámetro un callback, que (por convención) será ejecutado al finalizar la ejecución de la función.

Veamos una función inventada para entenderlo:

Ejemplo Callback

```
function escribirYLoguear(texto, callbackParaLoguear) {  
  // simulamos que escribimos en un archivo!  
  console.log(texto)  
  // al finalizar, ejecutamos el callback  
  callbackParaLoguear('archivo escrito con éxito')  
}  
  
escribirYLoguear('hola mundo de los callbacks!', (mensajeParaLoguear) => {  
  const fecha = new Date().toLocaleDateString()  
  console.log(`${fecha}: ${mensajeParaLoguear}`)  
})
```

En este ejemplo, “callbackParaLoguear” es una función anónima enviada como argumento a la función “escribirYLoguear” que obtiene la fecha de grabación y muestra un mensaje por pantalla



¡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, división y módulo. 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.

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!  
  }  
};
```

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



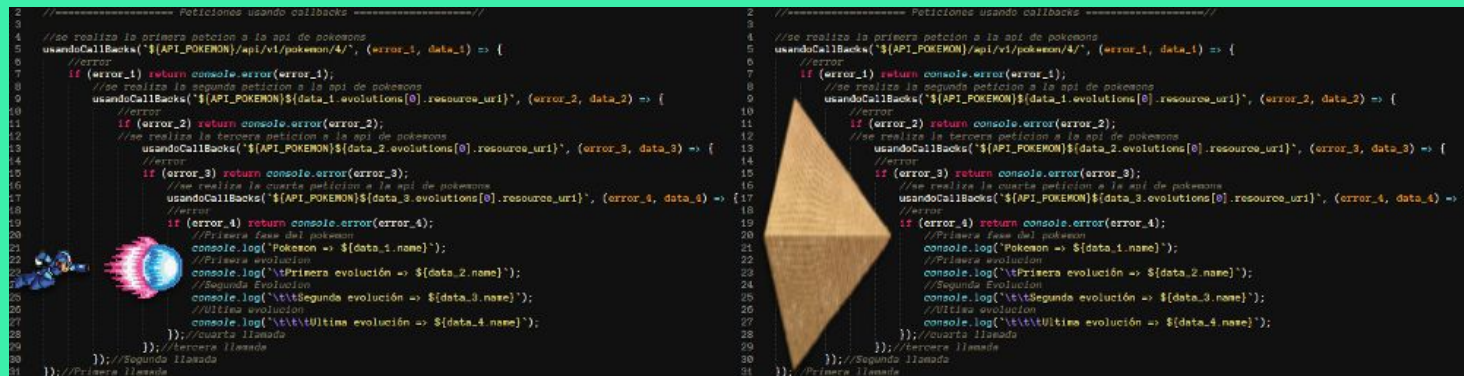
- Es un fragmento de código en el que **una función llama a un callback, y este a otro** callback, y este a otro, y así sucesivamente.
- Son **operaciones encadenadas**, en serie.
- Si el nivel de anidamiento es grande, se puede producir el llamado **callback hell** ó infierno de callbacks.
También se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

A 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.



The image displays two side-by-side code snippets, both titled "Peticiones usando callbacks", illustrating the concept of "callbacks hell" or "pyramid of doom". The left snippet shows a sequence of four nested callbacks for fetching Pokémon data and its evolutions, with each level of nesting increasing the indentation, creating a wide, shallow pyramid shape. The right snippet shows the same sequence of four nested callbacks, but with a different indentation style that creates a tall, narrow pyramid shape. A large, light-brown diamond is superimposed over the right snippet, highlighting the visual impact of the indentation. The code in both snippets is as follows:

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


Promesas



Promesas en

JS

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 veremos más adelante.

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()`.

PROMESAS



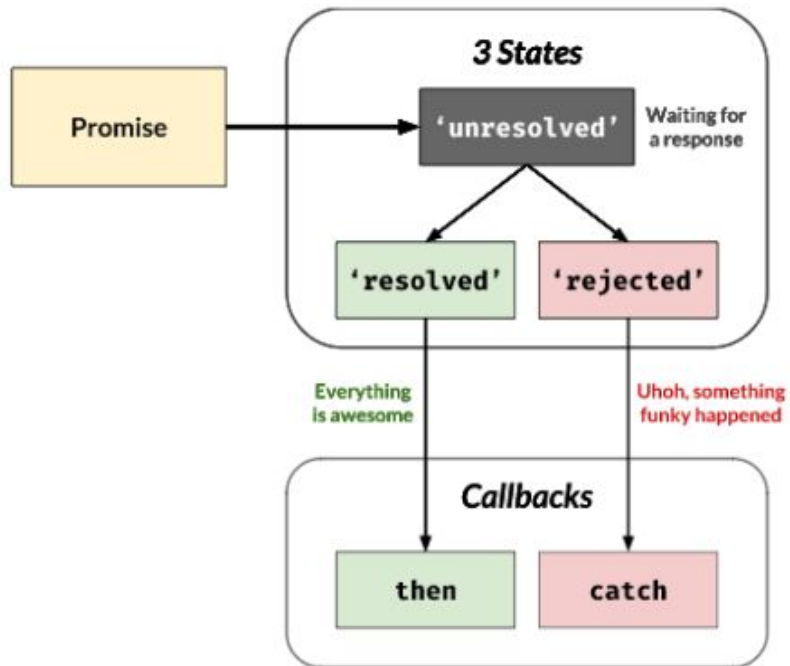
PROMESA PENDIENTE



PROMESA CUMPLIDA



PROMESA RECHAZADA



Promesas: creación



```
function dividir(dividendo, divisor) {  
  return new Promise((resolve, reject) => {  
    if (divisor == 0) {  
      reject('no se puede dividir por cero')  
    } else {  
      resolve(dividendo / divisor)  
    }  
  })  
}
```

Promesas: uso (sale bien)



```
dividir(10, 0)

  .then(resultado => {
    console.log(`resultado: ${resultado}`)
  })

  .catch(error => {
    console.log(`error: ${error}`)
  })

// muestra por pantalla:
// resultado: 5
```

Promesas: uso (sale mal)



```
dividir(10, 0)

.then(resultado => {
  console.log(`resultado: ${resultado}`)
})

.catch(error => {
  console.log(`error: ${error}`)
})

// muestra por pantalla:
// error: no se puede dividir por cero
```

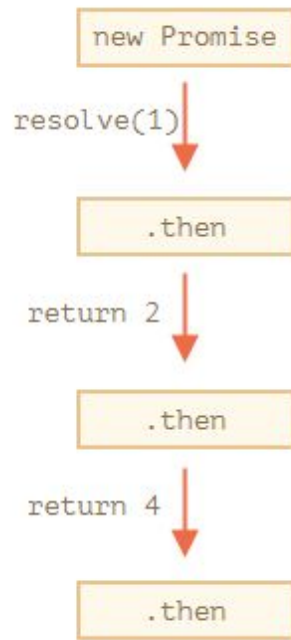



Encadenamiento de promesas

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!



Determinaremos en cada caso la salida que se registra en la consola

```
Promise.resolve(20)
  .then( x => x + 1 )
  .then( x => x * 2 )
  .then( x => {
    if(x==22) throw 'Error'
    else return 80
  })
  .then( x => 30 )
  .then( x => x / 2 )
  .then( console.log )
  .catch( console.log )
```

```
Promise.resolve(10)
  .then( x => x + 1 )
  .then( x => x * 2 )
  .then( x => {
    if(x==22) throw 'Error'
    else return 80
  })
  .then( x => 30 )
  .then( x => x / 2 )
  .then( console.log )
  .catch( console.log )
```

```
Promise.reject(30)
  .then( x => x + 1 )
  .then( x => x * 2 )
  .then( x => {
    if(x==22) throw 'Error'
    else return 80
  })
  .then( x => 30 )
  .then( x => x / 2 )
  .then( console.log )
  .catch( console.log )
```

Observación: `Promise.resolve(arg)` devuelve una promesa que siempre se resolverá en forma exitosa, y que devolverá como resultado el valor recibido como argumento.



BREAK

¡5/10 MINUTOS Y VOLVEMOS!

Sincronismo vs Asincronismo



Ejecución sincrónica vs. ejecución asincrónica

Ejecución Sincrónica: Repasemos

- 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.

Ejemplo Ejecución Sincrónica

```
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
```

- 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.**

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**.

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 que 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**.

Ejemplo Ejecución Asíncronica

```
const escribirArchivo = require('./escriArch.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
```

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.

Ejemplo Ejecución Asíncronica : Aclaración

Si queremos que el mensaje de 'finalizado' **salga después** de haber grabado el archivo, **anidaremos las instrucciones dentro del callback** de la siguiente manera:

```
escribirArchivo('hola mundo', () => {  
  console.log('terminé de escribir el archivo')  
  console.log('fin del programa')  
})
```

Esto funciona porque lo (único) que podemos controlar en este tipo de operaciones es que el callback siempre se ejecuta luego de finalizar todas las demás instrucciones involucradas en ese llamado.

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.

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 asíncronico 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()***.

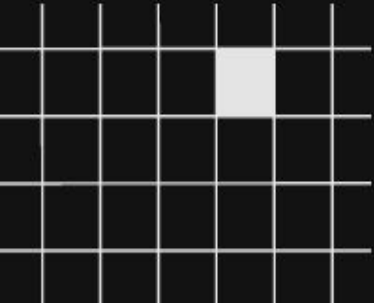
¿PREGUNTAS?





¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

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



OPINA Y VALORA ESTA CLASE

#DEMOCRATIZANDO LA EDUCACIÓN

CODER HOUSE