

LIND-2C Programación Web

Asincronismo

Mg. Pablo Ezequiel Inchausti
Ing. David Túa

UNIVERSIDAD DEL CEMA
UCEMA

Jueves 14 de septiembre, 09:00 hs

No queremos perder el tiempo...

Cuando estamos trabajando en el navegador, como lo hacen las aplicaciones web, será muy común intentar realizar operaciones que demoran un tiempo. Ejemplos de esto pueden ser accesos a servicios en la nube, bases de datos, etc. Es en este escenario donde nos encontramos con una disyuntiva: ¿esperamos que dicha operación termine para continuar o no? Para responder esta pregunta, con las ventajas y desventajas de elegir cada opción, nos introduciremos en el terreno del asincronismo.



Orígenes del asincronismo

Vamos a comenzar el análisis de este tema analizando un ejemplo. Supongamos que estamos construyendo una pantalla web como la siguiente:

Cliente: LOPEZ, RAMON ANGEL

Movimientos de la cuenta

Saldo al 31/07/2020: \$10.000

02/08/2020	Extracción ATM	(\$1.000)
22/08/2020	Depósito por caja	\$2.000
02/09/2020	Plazo Fijo	(\$5.000)
02/10/2020	Acreditación Plazo Fijo	\$5.500
24/12/2020	Deposito por caja	\$500

Saldo al 30/12/2020: \$12.000

Cotización del dólar de día: \$200 / \$220

Orígenes del asincronismo

En esta pantalla, podemos asumir que debemos obtener la siguiente información:

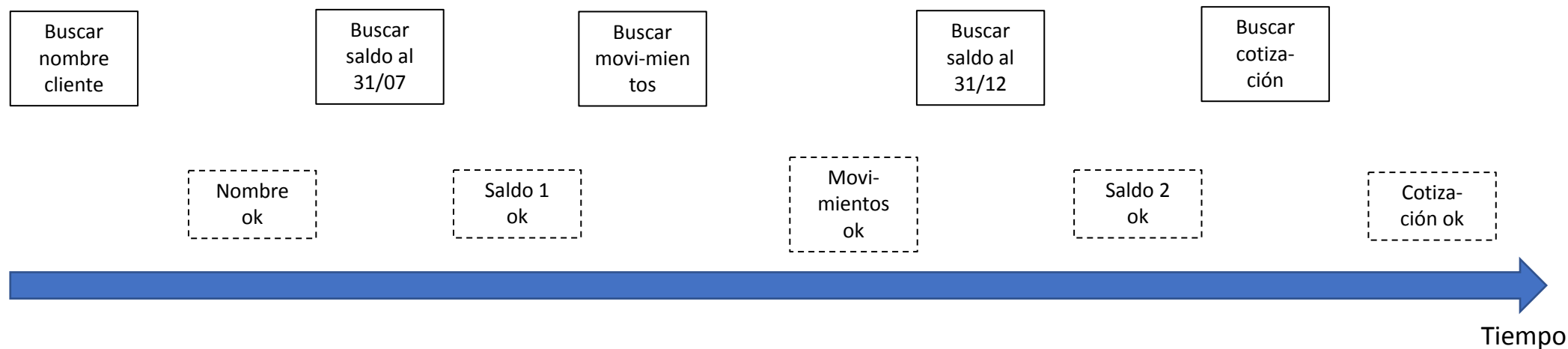
1. Nombre del cliente
2. Saldo de la cuenta al 31/07/2020
3. Movimientos de la cuenta desde el 31/07/2020 al 31/12/2020
4. Saldo de la cuenta al 31/12/2020
5. Cotización del dólar al día de la fecha

Luego, podemos asumir también que dicha información se recupera desde 5 “lugares” distintos. Y también sería lógico pensar que la recuperación de cada uno de los 5 puntos no será instantánea sino que demorará cierto tiempo, que podrá ser desde pocos milisegundos hasta varios segundos. Pensemos que si tenemos que recuperar la información desde “internet”, ir a buscar los datos y traerlos seguro demandará un tiempo considerable.

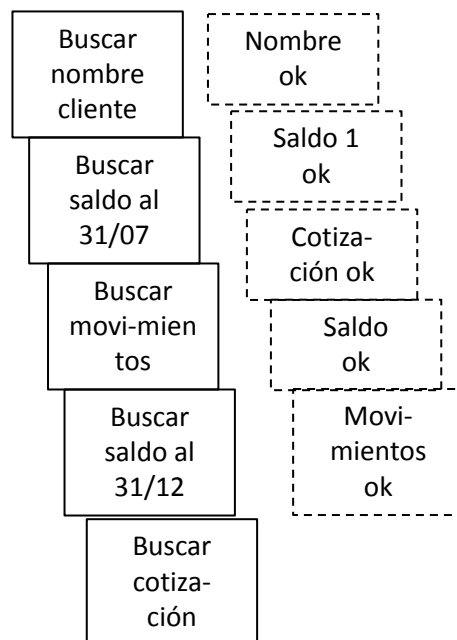
Entonces, teniendo en cuenta lo antedicho, tenemos dos alternativas para resolver esta problemática: hacerlo de forma sincrónica o asincrónica. Veamos de qué se trata cada una de ellas mediante el siguiente diagrama.

Sincronismo vs. Asincronismo. Diagrama de Tiempo

Operaciones
sincrónicas



Operaciones
asincrónicas



Sincronismo vs. Asincronismo. Ventajas y desventajas

Analicemos entonces las características que aparecen al utilizar cada una de estas técnicas.

Sincronismo

- ✓ Existirán muchos tiempos “muertos”, por tanto, el tiempo punta a punta será mucho mayor
- ✓ A consecuencia de lo anterior, existirán momentos donde la pantalla quedará freezada a la espera de la recepción de los datos
- ✓ La lógica de programación resultará sencilla pues es la que venimos utilizando hasta ahora
- ✓ Conocemos perfectamente el orden en que volverán nuestras datos

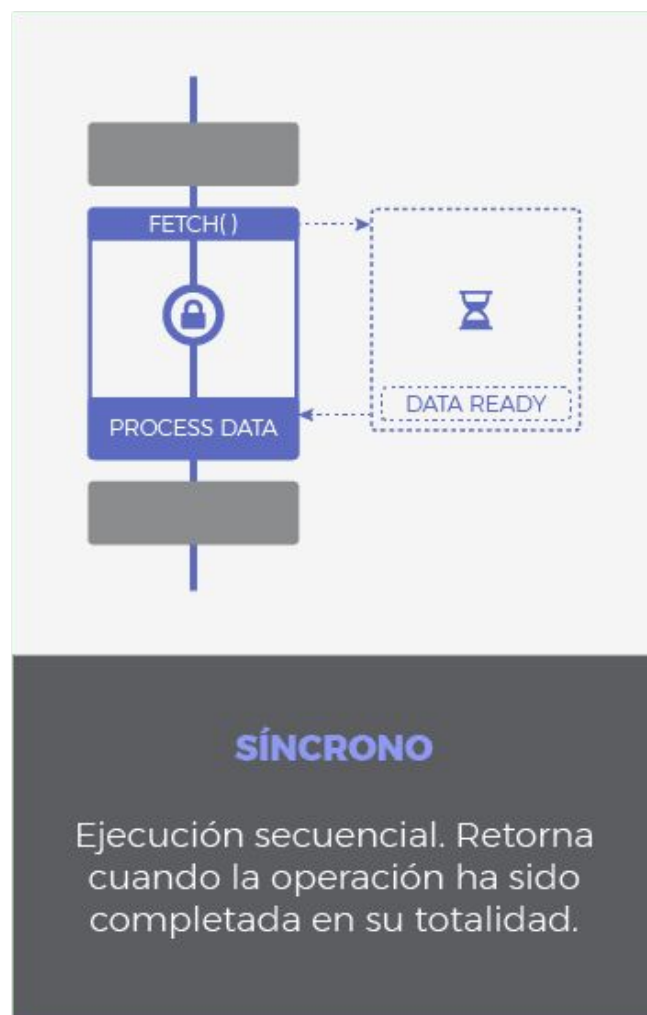
Asincronismo

- ✓ No existirán tiempos “muertos”, por tanto, el tiempo punta a punta será óptimo
- ✓ A consecuencia de lo anterior, no existirá el efecto freezado en la pantalla
- ✓ La lógica de programación será más compleja que la que venimos utilizando hasta ahora, pues existen procesos que se lanzarán recién cuando vuelva el resultado de cada consulta
- ✓ No conocemos con antelación el orden en que volverán los datos, podrán volver en forma desordenada respecto a las consultas. Y esto podrá condicionar nuestra lógica.

Conclusión: en cada caso deberemos analizar qué opción nos conviene. Pero, en la programación web y sobre todo en el frontend, resultarán mucho más importantes las ventajas del asincronismo pues existirán constantemente “tiempos muertos” que provocarán una muy mala experiencia si utilizáramos funciones síncronas.

Sincronismo vs. Asincronismo. Recapitulación

Podemos entonces conceptualizar una operación sincrónica y una asíncrona mediante los siguientes diagramas:



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.

```
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.

Operaciones bloqueantes

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, acceder a una base de datos fuera del ámbito del navegador, etc), nos encontraremos con dos maneras distintas de hacerlo: en forma bloqueante, o en forma no-bloqueante (blocking o non-blocking). Veamos entonces cómo funciona una operación bloqueante:

```
1 console.log('iniciando');
2 let dato = obtenerDatoSincronico();
3 console.log('Dato: ', dato)
4 console.log('finalizando');
5
6 function obtenerDatoSincronico(){
7     //demora 1000 ms
8     return 'Jorge';
9 }
```

```
iniciando
Dato: Jorge
finalizando
```

- Este tipo de operaciones permiten que el programa se comporte de la manera más intuitiva.
- Sigue las reglas de la ejecución **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. Es por ello 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. Permitirá, por tanto, la ejecución de varias operaciones en paralelo, sucediendo al mismo tiempo. A este tipo de ejecución se la conoce como **asíncrona**.

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 asíncrona, sólo sabemos en qué orden comenzarán su ejecución las instrucciones, pero no sabemos en qué momento ni en qué orden terminarán de ejecutarse.

Para analizar una operación no-bloqueante será de utilidad conocer primero la instrucción `setTimeout`.

Operaciones no-bloqueantes

La instrucción **setTimeout** invoca a una función después de un número específico de milisegundos. Trabaja sobre un modelo **asincrónico no-bloqueante** y su sintaxis es la siguiente:

```
setTimeout(funcion, milisegundos, param1, param2, etc)
```

Donde param1, param2, etc son los parámetros de la función invocada. Veamos entonces un ejemplo:

```
1 console.log('iniciando');
2 obtenerDatoAsincronico();
3 console.log('finalizando');
4
5 function obtenerDatoAsincronico(){
6     setTimeout(()=>console.log('Dato: Jorge'), 1000);
7 }
```

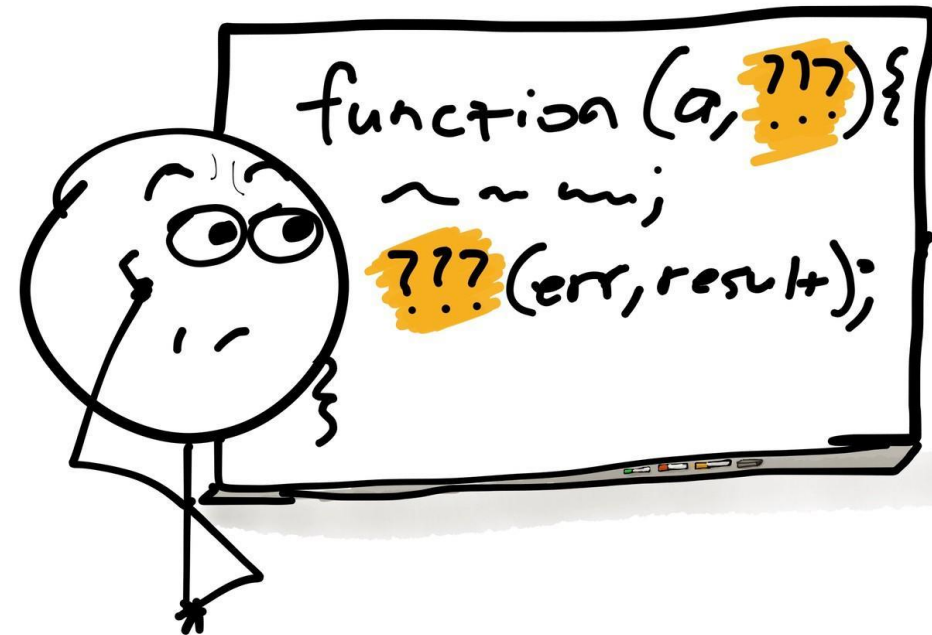
```
iniciando
finalizando
Hint: hit control+c
Dato: Jorge
```

Analicemos paso a paso este programa:

- Se imprime la palabra “iniciando”
- Se invoca a la función obtenerDatoAsincronico. Esta función, llama a la instrucción setTimeout que es no-bloqueante. Por tanto, no bloquea sino que sigue ejecutando el programa.
- Vuelve de la rutina, y se imprime la palabra “finalizando”
- Un segundo después, se ejecuta la función que se pasó por parámetro al setTimeout, y por tanto se imprime el dato

Callbacks

Hasta aquí hemos aprendido los conceptos de la programación sincrónica y la asincrónica, así como también qué son las operaciones bloqueantes y las no-bloqueantes. Ahora bien, ¿cómo programamos de una forma u otra? La respuesta a esta pregunta nos la darán los callbacks. Un correcto manejo de los mismos es vital en el mundo de la programación web.



Funciones como objetos

En JavaScript las funciones se comportan como objetos, por tanto, es posible asignar una declaración de función a una variable.

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

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

Recordemos que, al utilizar la función flecha, podemos omitir las llaves cuando está compuesta de una única línea. Además, en este caso, existe un return implícito.

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

Todas estas instrucciones efectúan lo mismo: permiten almacenar una función en una variable para luego ejecutarla de la siguiente manera:

```
mostrar('Hola a tod@s!');
```

```
Hola a tod@s!
```

Funciones como parámetro de otra función

Si podemos almacenar una función en una variable, entonces podemos enviar dicha función como parámetro a otra función. Veamos el siguiente ejemplo.

```
1  const finalizar = () => console.log('Fin de la función');
2
3  saludar(finalizar);
4
5  function saludar(f){
6      console.log('Hola a tod@s!');
7      f();
8  }
```

```
Hola a tod@s!
Fin de la función
```

Analicemos el programa y su salida:

- En la primer línea, definimos una función que imprime el mensaje “Fin de la función” y la asigna a la variable finalizar
- En la segunda línea, se invoca a la función “saludar”, enviando como parámetro la variable “finalizar” que recordemos contiene la función
- Al invocar a la función “saludar”, se imprime el mensaje “Hola a tod@s!” y luego se ejecuta la función enviada como parámetro. Es por ello que se imprime el mensaje “Fin de la función”.

Callback

Un callback es una función que se envía como parámetro 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 “saludar” que utilizamos en el punto anterior “recibe un callback”. El callback recibido es la función contenida en la variable “finalizar”.

Algo importante a destacar, es que la función que se envía como parámetro (el callback) no necesariamente debe ser almacenada en una variable.

```
1  saludar(() => console.log('Fin de la función'));  
2  
3  function saludar(f){  
4      console.log('Hola a tod@s!');  
5      f();  
6  }
```

```
Hola a tod@s!  
Fin de la función
```

Vemos en este caso que enviamos la función como parámetro sin asignarla a ninguna variable.

Callback. Un ejemplo más completo

Analicemos un caso un poco más completo de un callback.

```
1  saludar((nombre) => console.log(`Hola ${nombre}!`));  
2  
3  function saludar(f){  
4      console.log('Comienzo de la función...');  
5      let nombre = "Pepe";  
6      f(nombre);  
7  }
```

```
Comienzo de la función...  
Hola Pepe!
```

Al igual que en el ejemplo anterior, la función que se envía como parámetro se ejecuta al final de la función saludar. La novedad en este caso es que este callback utiliza un dato que no era conocido al momento de enviar la función como parámetro. Esta es la esencia del asincronismo, ya que podremos ejecutar una función cuando termine otro evento (en este caso, cuando recibimos el nombre a saludar).

Callback. Algunas convenciones

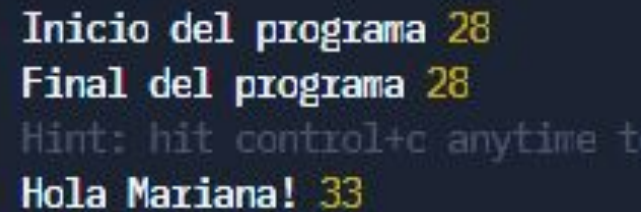
- ✓ El callback siempre es el último parámetro.
- ✓ El callback suele ser una función que recibe dos parámetros.
- ✓ La función llama al callback al terminar de ejecutar todas sus operaciones.
- ✓ Si la operación fue exitosa, la función llamará al callback pasando null como primer parámetro y si generó 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.
- ✓ 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!  
  }  
};
```

Callback. Entendiendo el asincronismo

Con todos los conceptos ya aprendidos, veamos cómo podemos utilizar un callback para disponer de una función no-bloqueante que opere de forma asincrónica.

```
1 console.log('Inicio del programa',
2           new Date().getSeconds());
3
4 buscarNombre((err,nombre)=>{
5   if (err){
6     console.log('Error!', err,
7               new Date().getSeconds())
8   } else {
9     console.log(`Hola ${nombre}!`,
10              new Date().getSeconds())
11   }
12 });
13
14 console.log('Final del programa',
15           new Date().getSeconds());
16
17 function buscarNombre(f){
18   setTimeout( ()=> {
19     let nombre = 'Mariana';
20     f(null, nombre);
21   }, 5000);
22 }
```



Inicio del programa 28
Final del programa 28
Hint: hit control+c anytime t
Hola Mariana! 33

La función `buscarNombre` está simulando un acceso a la base de datos que demora 5 segundos. Vemos que estos 5 segundos ocurren de forma no-bloqueante, ya que continúa la ejecución del programa sin esperar a obtener el nombre. Recién cuando el nombre fue recuperado (5 segundos después), se ejecuta la función esperada, el callback que enviamos por parámetro y que saluda al nombre recuperado de la base de datos.

setInterval

Para finalizar el capítulo de callbacks, conozcamos la instrucción **setInterval**, que es muy parecida a **setTimeout** e incluso tiene la misma sintaxis. También invoca a una función después de un número específico de milisegundos trabajando sobre un modelo asíncronico no-bloqueante. A diferencia de **setTimeout**, se seguirá ejecutando luego del tiempo especificado tantas veces hasta tanto se llame a **clearInterval** o se cierre la ventana (el valor devuelto por **setInterval** se usará como parámetro para el método **clearInterval**).

```
1 console.log('Inicio del programa');
2
3 let id = setInterval(()=>console.log('Hola!!!!'),1000);
4
5 setTimeout(()=>clearInterval(id),5100);
6
7 console.log('Fin del programa');
```

```
Inicio del programa
Fin del programa
Hint: hit control+c any
Hola!!!!
Hola!!!!
Hola!!!!
Hola!!!!
Hola!!!!
```

Algo importante a destacar, es que tanto para **setTimeout** como para **setInterval**, el primer argumento es un callback, ya que es una función que enviamos como parámetro y se ejecuta al finalizar el tiempo especificado.

Promesas

Los callbacks fueron el primer mecanismo con el que contamos los programadores para implementar el asincronismo en un programa Javascript. Con el paso del tiempo, y algunas dificultades que surgieron con su uso, tomó cada vez más fuerza un nuevo mecanismo que implicó el uso de promesas. Repasaremos en este capítulo qué son, para qué las podemos utilizar y qué problemas vienen a resolver.



Callbacks anidados

```
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)  
            }  
          })  
        }  
      })  
    }  
  })  
}
```

```
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  
            })  
          })  
        })  
      })  
    })  
  })  
})
```

- 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.

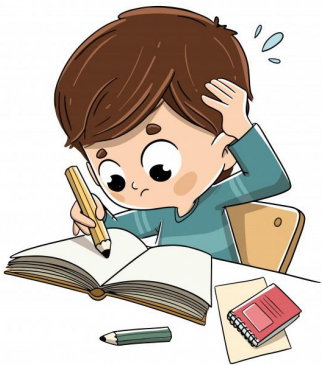
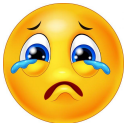
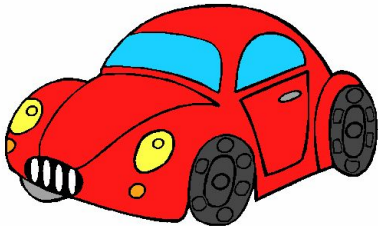
Callbacks hell

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



En estos casos se verá muy comprometida la legibilidad del código y caeremos fácilmente en errores de funcionamiento. Este es uno de los principales problemas que tuvieron los callback y por lo que empezó a ser mucho más común el uso de promesas. La mayoría de las librerías asíncronas que se utilizan hoy en día permiten el uso de promesas.

Promesas



Promesas. Visión técnica

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

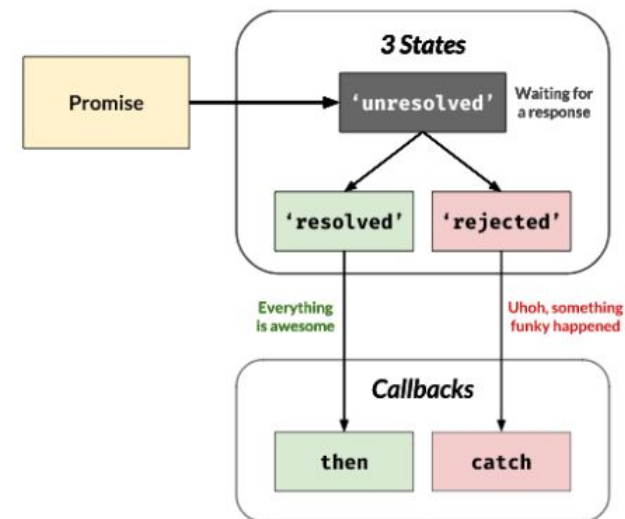
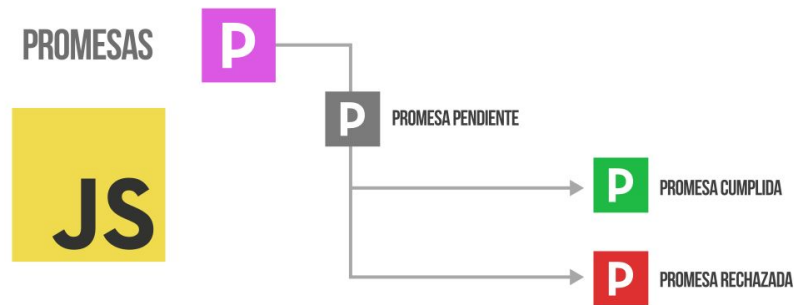
El estado inicial de una promesa es:

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()`.



Creación y uso de promesas



demo
u04.01

UNIVERSIDAD DEL CEMA
UCEMA

A continuación vamos a utilizarla. Veamos un **caso** donde resultará una promesa **exitosa**:

2

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

Veamos entonces un ejemplo del uso de promesas. Primero deberemos crear la promesa:

1

Finalmente, veamos un **caso** donde resultará una promesa **fallida**:

3

```
dividir(10, 2)  
  .then(resultado => {  
    console.log(`resultado ${resultado}! `)  
  })  
  .catch(error => {  
    console.log(`error ${error}! `)  
  })  
  
// muestra por pantalla  
//      resultado: 5
```

```
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

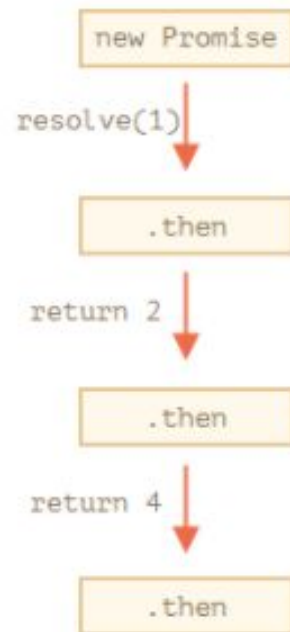


demo
u04.02

UNIVERSIDAD DEL CEMA
UCEMA

Veamos ahora de qué forma evitamos el callback hell utilizando promesas.

Lo más importante es saber que cuando `promise.then()` retorna algo, devuelve otra promesa. Y dicha promesa será tomada por el siguiente `.then()`.



```
1  new Promise(  
2    function (resolve, reject) {  
3      setTimeout(() => resolve(1), 1000);  
4    }  
5    .then(result => {  
6      console.log(result);  
7      return result * 2;  
8    })  
9    .then(result => {  
10     console.log(result);  
11     return result * 2;  
12   })  
13   .then(result => {  
14     console.log(result);  
15     return result * 2;  
16   })  
17   .then(result => {  
18     console.log(result);  
19     return result * 2;  
20   })  
21  
22   // 1) La promesa inicial se resuelve en 1 segundo(*)  
23   // 2) Entonces se llama al controlador .then en (**)  
24   // 3) El valor que devuelve  
25   //      se pasa al siguiente controlador .then (***)  
26  
27
```

Un caso real



demo
u04.03

UNIVERSIDAD DEL CEMA
UCEMA

Para finalizar el capítulo de promesas, veamos un caso real y extremadamente útil en el desarrollo web: Axios, una librería que nos permitirá invocar a un servicio de backend mediante el uso de promesas.

```
1  <!DOCTYPE html>
2  <html lang="es">
3  <head>
4      <meta charset="utf-8">
5      <title>Utilizando promesas</title>
6      <script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
7  </head>
8  <body>
9      <span><span>
10     <script>
11         axios.get('https://jsonplaceholder.typicode.com/users')
12             .then(resp=>console.log(resp))
13             .catch(e => console.log('Error!!!!', e));
14     </script>
15 </body>
16 </html>
```

Muchas Gracias!!

UNIVERSIDAD DEL CEMA
UCEMA

ucema.edu.ar