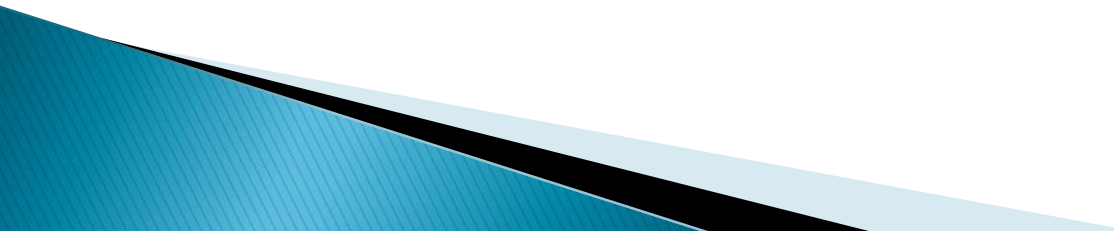


Promesas ES6

Programación Asíncrona

- ▶ Si hay algo que caracteriza a JavaScript, es la **asíncronía** que presentan algunas funciones.
 - ▶ Concretamente las que realizan **operaciones de entrada/salida como escritura o lectura del disco o una petición AJAX**
 - ▶ Ejemplos muy usados: consultas y manejo de datos en Firebase y/o servidores en la nube
- 

Formas de trabajar con asíncrono

1. Usando funciones “callback”
2. Usando promesas ES6
3. Usando observables Rxjs (programación reactiva)

Callbacks: ejemplo

```
function mostrar(error, resultado) {
  if (error)
    console.log("Ocurrió el error: " + error.message);
  else
    console.log(resultado);
}

console.log("1-comienzo programa");

//pido mis imágenes a mi servidor y las muestro en pantalla
getImagenesFromServer("http://server.com/imagenes/", mostrar);

console.log("2.-fin del programa");
//-----
function getImagenesFromServer(urlDatos, callback) {
  //defino variables para conectarme a urlDatos
  //conecto al servidor y lo consulto
  if (respuesta_servidor_ok) {
    callback(null, imagenes_recuperadas);
  } else {
    callback(new Error("error en la descarga"), null);
  }
}
```

Callbacks

- ▶ Las funciones conocidas como **callback**, son funciones que:
 - se pasan como **parámetros de otras funciones**,
 - que se **ejecutan o llaman dentro de éstas**,
 - se ejecutan a posteriori en **respuesta a cierto evento**.
- ▶ En el ejemplo anterior la función “mostrar” es un **callback**, ya que:
 - Se pasa como parámetro de la función “*getImagenesFromServer*”
 - Se ejecuta dentro de “*getImagenesFromServer*”:
callback (null,imagenes_recuperadas);
 - Se ejecuta cuando el servidor nos da una respuesta (correcta o errónea).

Callbacks

- ▶ El ejemplo anterior, muestra cómo hacer una petición a un servidor y recoger su respuesta:
 - sólo necesitaremos llamar a la función "getImagenesFromServer" pasándole laURL del servidor y la función (*callback*) que queramos ejecutar cuando obtengamos una respuesta válida.
 - Si queremos controlar la respuesta errónea, también podemos usar la misma función "callback".
- ▶ Este tipo de funciones se utilizan constantemente en JavaScript.
- ▶ Por ejemplo, cuando trabajamos con arrays y utilizamos métodos como **forEach** o **map** , estamos pasando una función como parámetro de otra, que se ejecuta por cada uno de los elementos del array.

Callbacks

- ▶ El ejemplo anterior, muestra cómo hacer una petición a un servidor y recoger su respuesta:
 - sólo necesitaremos llamar a la función "getImagenesFromServer" pasándole laURL del servidor y la función (*callback*) que queramos ejecutar cuando obtengamos una respuesta válida.
 - Si queremos controlar la respuesta errónea, también podemos usar la misma función "callback".
- ▶ Este tipo de funciones se utilizan constantemente en JavaScript.
- ▶ Por ejemplo, cuando trabajamos con arrays y utilizamos métodos como **forEach** o **map** , estamos pasando una función como parámetro de otra, que se ejecuta por cada uno de los elementos del array.

Callbacks

- ▶ En lugar de definir la función **mostrar** para luego pasarla como parámetro, podemos crear sobre la marcha la función en el momento de pasarla como parámetro:

```
console.log("1-comienzo programa");

//pido mis imágenes a mi servidor y las muestro en pantalla
getImagenesFromServer("http://server.com/imagenes/", function(error, resultado){
    if (error)
        console.log("Ocurrió el error: " + error.message);
    else
        console.log(resultado);
});

console.log("2.-fin del programa");

//-----
function getImagenesFromServer(urlDatos, callback) {
    //defino variables para conectarme a urlDatos
    //conecto al servidor
    if (respuesta_servidor_ok) {
        callback(null,imagenes_recuperadas);
    }else{
        callback(new Error("error en la descarga"),null);
    }
}
```


Características de las funciones Callbacks

- ▶ Pueden llevar parámetros o no.
 - El ejemplo anterior lleva 2 parámetros: error y resultado (es lo más habitual).

```
1 function ejemplo2(fn) {  
2     var nombre = "Pepe";  
3     fn(nombre);  
4 }  
5  
6 ejemplo2(function(nom) {  
7     console.log("hola " + nom);  
8 }); // "hola Pepe"
```

Ejemplos comunes de uso de Callbacks en JavaScript

▶ Añadir un gestor de eventos a un botón.

- La función `addEventListener()` , acepta otra función como parámetro que se ejecutará cuando se produzca el evento:

```
1 document.getElementById("btn1").addEventListener("click", function() {  
2   console.log("has pulsado el botón 1");  
3 });
```

▶ El método “map” de los arrays:

```
1 document.getElementById("btn1").addEventListener("click", function() {  
2   console.log("has pulsado el botón 1");  
3 });
```

Quiz sobre funciones JavaScript

- ▶ Realiza el siguiente Quiz para comprobar si lo has entendido:

[ENLACE AL QUIZ](#)

PROMESAS

- ▶ El ¿POR QUÉ? Necesitamos promesas: mira este [enlace](#)
- ▶ OTRO EJEMPLO:

Supongamos que vamos a comprar comida a un restaurante de comida rápida. Cuando terminamos de pagar por nuestra comida nos dan un ticket con un número, cuando llamen a ese número podemos entonces ir a buscar nuestra comida.

Ese *ticket* que nos dieron es nuestra **promesa**, ese ticket nos indica que eventualmente vamos a tener nuestra comida, pero que todavía no la tenemos.

Cuando llaman a ese número para que vayamos a buscar la comida entonces quiere decir que la promesa se completó.

Pero resulta que una promesa se puede completar correctamente o puede ocurrir un error, ¿Qué error puede ocurrir en nuestro caso? Por ejemplo puede pasar que el restaurante no tenga más comida, entonces cuando nos llamen con nuestro número pueden pasar dos cosas.

Nuestro pedido se resuelve y obtenemos la comida.

Nuestro pedido es rechazado y obtenemos una razón del por qué.

Pongamos esto en código:

PROMESAS

```
const ticket = getFood();  
  
ticket  
  .then(food => eatFood(food))  
  .catch(error => getRefund(error));
```

Promesas: ¿Qué son?

- ▶ Una promesa es un objeto que **representa un valor** que puede que esté **disponible ahora, o puede que esté disponible en un futuro, o incluso nunca lo esté.**
- ▶ Por tanto, como **no sabemos cuándo estará disponible**,
 - todas las **operaciones dependientes de haber conseguido ese valor**, hay que **postponerlas**.
 - Ej: una **petición GET para obtener un recurso**: imagen, archivo, un acceso a BD.
- ▶ La promesa **realiza esa operación costosa en tiempo, de forma asíncrona**
 - el get, el acceso a la BD, el cálculo complejo,...**y mientras, nuestro código JAVASCRIPT sigue ejecutándose de forma secuencial. (CUIDADO CON ESO.)**
- ▶ Pero **habremos dejado preparado en la promesa, ¿QUÉ CÓDIGO QUEREMOS QUE SE EJECUTE CUANDO EL DATO ESTÉ DISPONIBLE?.**
- ▶ De alguna manera **la promesa nos PROMETE, nos ASEGURA, que ese código que hemos dejado preparado, se va a ejecutar cuando el dato esté listo.**
- ▶ Así, **el resto del código puede seguir independiente de este dato** con su código en el hilo donde estaba (**es simple hilo, Javascript NO ES MULTIHILO**).

Promesas: ¿Cómo se crean?

- ▶ Las promesas se crean usando un constructor llamado **Promise** y pasándole como **parámetro una función** que recibe dos parámetros, **resolve y reject**, que nos permiten indicarle a esta que se resolvió o se rechazó.
 - **resolve** es, a su vez, una *función que llamaremos cuando tengamos el dato disponible*.
 - **reject**, es otra *función*, que *llamaremos cuando suceda un error*.

```
const promise = new Promise((resolve, reject) => {  
  const number = Math.floor(Math.random() * 10);  
  setTimeout(  
    () => number > 5  
      ? resolve(number)  
      : reject(new Error('Menor a 5')), 1000 ); });
```

- ▶ Para llamar esa promesa cuando la necesitemos:

```
promise  
  .then(number => console.log(number))  
  .catch(error => console.error(error));
```

Promesas: Estados

▶ Las promesas pueden estar en 3 estados posibles:

- **Pendiente:** estado inicial.

- **Resuelta:**

- ❑ Ocurre cuando llamamos a **resolve**.

- ❑ Cuando se resuelve, se ejecuta la función que pasamos al método **.then**

- **Rechazada:**

- ❑ Ocurre cuando llamamos a **reject**.

- ❑ Normalmente, cuando es rechazada *obtenemos un error que nos va a indicar la razón del rechazo.*

- ❑ Cuando se rechaza, se ejecuta la función que pasamos a **.catch**

También es posible pasar una segunda función a **.then** la cual se ejecutaría en caso de un error en vez de ejecutar el **.catch**

Promesas: ¿Cómo pasarles parámetros?

- ▶ El constructor de “**Promise()**” **no admite parámetros propios**.
- ▶ ¿**Cómo pasamos nuestros parámetros?**?. Por ejemplo:
 - ❑ En el ejemplo anterior, creamos una promesa que **se completa** luego de 1 segundo y **se resuelve** si el número generado es mayor a 5
 - ❑ ¿Y si *quiero que se resuelva cuando el número sea mayor que uno que yo le pase dinámicamente*, un “esperado” y el aleatorio entre 0 y máx? ¿dónde le paso esos parámetros?

▶ **SOLUCIÓN:**

- ❑ creamos una función que recibe los parámetros necesarios y

```
function aleatorioMayorQue (max = 10, expected = 5, delay = 1000) {  
  return new Promise((resolve, reject) => {  
    const number = Math.floor( Math.random() * max) );  
  
    setTimeout(  
      () => number > expected  
        ? resolve(number)  
        : reject(new Error('número menor al esperado')),  
      delay );  
  });  
}
```

Promesas: ¿Cómo pasarles parámetros?

- ▶ Para **llamar a esa función**:

```
randomDelayed(100, 75, 2500)  
  .then(number => console.log(number))  
  .catch(error => console.error(error));
```

- ▶ Cuando ejecutamos `randomDelayed(100, 75, 2500)`
 - creamos una promesa que luego de **2.5 segundos** se va a resolver siempre que el número generado (entre 0 y 100) sea mayor a 75.
 - Lo mismo que habíamos hecho antes, pero esta vez personalizable.

Promesas: Encadenadas

- ▶ Este patrón se llama **promise chaining** o encadenamiento de promesas.
- ▶ Básicamente **nos evita anidar código**, en vez de eso *una promesa puede devolver otra promesa y llamar al siguiente .then de la cadena*:

```
const array = [1, 2, 3]

addToArray(4, array)
  .then(()=>addToArray(5, array))
  .then(()=>addToArray(6, array))
  .then(()=>addToArray(7, array))
  .then(()=>console.log(array));

// (4 seg. de delay)-> [1,2,3,4,5,6,7]
```

- ▶ Donde, la función addToArray(...) es:

```
function addToArray (data, array) {
  return new Promise(function (resolve, reject) {
    setTimeout(function() {
      array.push(data)
      resolve(array)
    }, 1000);
    if (!array) {
      reject(new Error('No existe un array'))
    }
  })
}
```

Promesas: En paralelo

- ▶ Ver el siguiente enlace: [ENLACE](#)