UD6
Callbacks, Promises y Async/Await: en búsqueda del control de lo asíncrono.

1. Introducción

1.1. Hablemos de asincronía

Hasta ahora las acciones “asíncronas” más evidentes que hemos realizado en JavaScript han sido las llevadas a cabo por los métodos **“setTimeOut()”** y **“setInterval()”.** Ambas son capaces de romper el “ciclo lineal” de ejecución de instrucciones y poder ejecutar código en un orden/tiempos diferentes.

¿Cómo podemos definir una ejecución asíncrona? Podríamos decir que es una ejecución de un código que iniciamos o solicitamos en un momento dado, pero que por x razones se ejecutará más tarde, y una vez termine necesitaremos hacer algo en consecuencia.

Por ejemplo, veamos el código siguiente el cual es una función que carga un script pero mediante código javascript (sabemos que la carga de scripts puede acarrear un tiempo considerable):

```
 function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);  //Inyecta el código dinámicamente.
}
```

Puesto que la carga del script lleva su tiempo, y sabemos que si llamamos a esa función y acto seguido intentamos hacer uso de algún método que estuviera dentro de ese script, ¿Qué pasaría?

```
loadScript('/my/script.js');    // Imaginemos que el script contiene el método "nuevaFuncion(...)”. Justo llamar a loadScript, la ejecución continúa.

nuevaFunción();// Si llamamos al método "nuevaFuncion()"  ¿¿¿Qué pasaría??? Nos daría un error ya que quizás aún no haya sido cargado.
```

Probablemente el resultado fuera un “error” debido a que el navegador aún no le hubiera dado tiempo a cargar el script completamente, y por lo tanto sus funciones aún no son utilizables.

¿Cómo podríamos solucionar el problema? Pues existen principalmente 3 técnicas:

1. ***Callbacks***
2. ***Promises***
3. ***Async y Await.***

2. Callbacks

Los callbacks es una técnica que se utiliza en muchos lenguajes de programación. La definición más básica y simple de qué es una callback (función callback) es:

Son funciones que se pasan como parámetros de otras funciones y que se “llaman” dentro de estas cuando una determinada operación ha terminado,.

Normalmente las funciones callback se invocan cuando el código de la función que la recibe por parámetros ha terminado, y muchas veces se invoca con parámetros que indican o bien un resultado o bien un mensaje de error, etc.

Generalmente son usados en todas aquellas funciones que conlleven cierto tiempo de ejecución y no queremos bloquear el programa como pueden ser:

- Acceso a ficheros.
- Operaciones de red.
- Acceso a bases de datos.
- Carga de imágenes pesadas.
- Operaciones matemáticas complejas.

Veamos un ejemplo modificando el código anterior para asegurarnos que la función “nuevaFuncion()” está cargada y podemos hacer uso de ella:

```
function loadScript( src , callback ) { //2 argumentos: url del script y el callback
  let script = document.createElement('script');
  script.src = src;
  script.onload = () =>  callback(script) ; //Aquí es donde se llamará al callback
  document.head.append(script);
}

function callback(script) {
  //En este punto ya sabemos al 100% que la carga del script ha sido completada:
  nuevaFuncion( );
  console.log( "El script con la url " + script.src + "ha sido completamente
cargado. ");
}

loadScript  ("/script/script1.js" , callback );   //Hacemos la llamada.
```

2.1. Manejo de errores con callbacks (Handling errors).

Normalmente cuando hacemos uso de los callbacks lo haremos **usando 2 argumentos**, uno que representará un error, y si viene de vuelta es que se ha producido algún fallo o anomalía, y si viene de vuelta a “null” es que todo ha ido bien. Esto no es obligatorio hacerlo así, se podrían usar booleanos u otra estrategia, pero es bastante común verlo de la siguiente forma:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script); //Si ha ido bien, el error se manda
a null.
  script.onerror = () => callback(new Error(`Script load error for ${src}`), "");
```

```
  document.head.append(script);
}

//Llamamos a la función asíncrona y usamos un callback anónimo
loadScript('/my/script.js', function(error, script) {
  if (error) {
    // Manejar error
  } else {
    // Script cargado satisfactoriamente
  }
});
```

Importante: Daros cuenta en el último ejemplo que no hemos definido en ningún sitio la función callback de forma explícita, sino que hemos utilizado una función anónima. También podríamos haber codificado con una función no anónima:

```
//Llamamos a la función asíncrona
loadScript('/my/script.js', callback);

function callback(error, script) {
  if (error) {
    // Manejar error
  } else {
    // Script cargado satisfactoriamente
  }
}
```

2.2. Anidando callbacks: menudo follón!

Acabamos de aprender un “truco/técnica” de cómo podemos recuperar el hilo de ejecución de algo que se ejecuta asíncronamente. Los callbacks tienen un problema y es que el código se hace muy engorroso y complejo **cuando se “anidan” varias acciones asíncronas**. Por ejemplo, un caso muy común: Normalmente cuando el usuario hace login en un aplicación, tenemos que hacer una llamada al servidor para hacer el “log in” (envío de datos), si nos responde OK tenemos que volver a hacer una llamada para traernos sus datos/configuración, y así...varias acciones asíncronas. Para llevar un control de si todo ha ido bien o alguna de ellas ha fallado empezamos a anidar “callbacks” y nos volvemos locos:

```
loadScript('1.js', function(error, script) {
  if (error) {
      handleError(new Error(“Esta fallando en la carga del primer script”));
  } else {
      loadScript('2.js', function(error, script) {
      if (error) {
            handleError(“Esta fallando en la carga del segundo script”);
      }
      else {
```

```
      // ...
      loadScript('3.js', function(error, script) {
            if (error) {
                  handleError(error);
            } else {
              // ... Aquí es donde llegamos si la carga de todos los scripts ha
              // terminado bien y seguimos con la ejecución normal de código.
            }
      });
      }
      })
  }
});
```

Como se puede ver el código es un poco endemoniado: poco claro y además, imagina que por cada carga de script (donde están los "...") tuviéramos código específico después de cada carga, pues sería el "acabose".

Una opción intermedia para mejorar los problemas derivados de anidar "callbacks" es de la siguiente forma:

```
loadScript('1.js', step1);

function step1(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', step2);
    //loadScript('3.js', step3);
  }
}

function step2(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('3.js', step3);
  }
}

function step3(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...Aquí continuamos después de la carga de los 3 scripts...

  }
};
```

Curiosidad: Vamos a ver cómo se puede implementar la función “map” de javascript con callback:

```
function miMap(arr, callback) {
  let nuevoArray = [];
  for (let i = 0; i < arr.length; i++) {
      nuevoArray.push( callback(arr[i]) );
  }
  return nuevoArray;
}

//Vamos a hacer una prueba:
let array1 = miMap([1,2,3,4,5], function(elemento) {
  return elemento * 2;
}); // [2, 4, 6, 8, 10]
```

EJERCICIO 1 - Callback

A partir de una página html simple y de obtener mensajes a través de la consola, vamos a tratar de controlar mediante el uso de “callback” 3 llamadas “anidadas” a una función la cual simulará una llamada a un servidor. Esta simulación será simplemente un **“setTimeout()”** con X segundos de retraso. La primera llamada tardará 2 segundos en terminar de ejecutarse, la segunda 3 segundos y la tercera llamada 4 segundos.

NOTA: La segunda llamada a la función “pesada” debe esperar a que la primera llamada a dicha función termine. De eso, se trata el anidamiento. Suponemos que la segunda, depende del resultado de la primera (y así con el resto) (Suponemos). De eso se trata el anidamiento.

La función “**callback**” deberá NO ser anónima, y además deberá de tener 2 parámetros: uno para el objeto Error, y otro con el valor que trae desde su llamada (vamos a devolver el tiempo de espera, por ejemplo).

```
function llamadaServidor( numSegundos , callback)
{
  setTimeout( function() { console.log(`Llamada al servidor de ${numSegundos} sg.`);
}, numSegundos );
}
```

- E1.1. La salida del script debe de ir perfecta, sin fallos. Debe de obtener una salida similar a la siguiente:

```
Llamada al servidor de 2000 sg.
La primera llamada a tardado 2000
Llamada al servidor de 3000 sg.
La segunda llamada a tardado 3000
Llamada al servidor de 4000 sg.
La tercera llamada a tardado 4000
Hemos terminado todo perfecto
```

→ *Imprime la función asíncrona*
→ *Imprime su callback*
→ *Imprime la función asíncrona*
→ *Imprime su callback*
→ *Imprime la función asíncrona*

➜ *Imprime su callback*

- E1.2. Debe de forzar un error en la tercera llamada al servidor, y controlar que ha dado un error debido a que la llamada tarda más de 4 segundos (este es el motivo que debemos de programar, es decir, si una llamada al servidor tarda más de 4 segundos, que devuelva un error), además de emitir un mensaje de error por consola.

3. Promises

Las "promesas" son una nueva técnica que apareció para sustituir, en teoría, el uso de callbacks. Hace que el código sea menos engorroso, sobretodo a la hora de anidamiento de operaciones asíncronas.

Si estuviéramos en el contexto de la película "Regreso al futuro" las podríamos definir como:

<<Una promesa es un objeto que representa el valor de una operación que será resuelta en el futuro>>

Promises nos permiten esperar a que un determinado código termine su ejecución, antes de ejecutar la próxima sección de código. Su comportamiento es parecido al de los **"callbacks"**.

3.1. ¿Por qué es importante?

Porque nos ayudan a manejar operaciones asíncronas y esperar a que terminen y hacer algo (o no) cuando terminen. Un ejemplo muy claro de uso puede ser cuando nos traemos "datos" de un servidor. Normalmente tarda un poco en traer los datos, y ¿vamos a bloquear la web hasta que lleguen? ¿y si no llegan xq el servidor se ha caído?; estas situaciones "asíncronas" las manejaremos como tal, y debemos prever una respuesta tanto si todo funciona bien como si no.

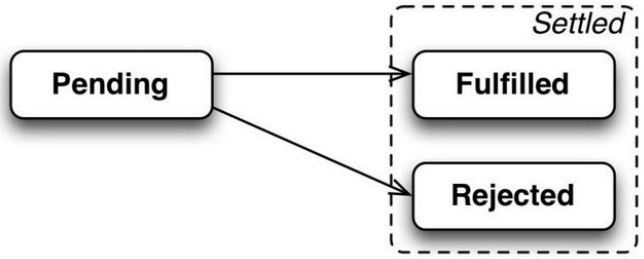
3.2. ¿Qué es una Promise?

En Javascript, Promise representa el resultado generado tras una operación asíncrona. Podemos hacer el símil de que Promise es como un "recipiente o envase" el cual es "adherido" (attached) a la respuesta o al resultado tras completarse una operación asíncrona.

Promise puede tener/estar en 1 de estos 3 estados:

- **Pending** ➜ La operación asíncrona no ha sido completada aún.
- **Fulfilled** **(Resolved)** ➜ La operación asíncrona se ha completado y Promise ya tiene un valor.
- **Rejected** ➜ La operación se ha completado con un error o fallo.

De estos 3 estados se dice que si la promesa aún no está resuelta está en estado "**Pending**" y si ha sido terminada pues estará o **"Fulfilled"** o **"Rejected"**. Una vez finaliza automáticamente se le llama "**Settled**".



3.3. Trabajando con Promises.

La sintaxis básica de una promesa es la siguiente:

```
//ES5
new Promise( /* Ejecutor */ function(resolver, rechazar) { ... } );
//ES6
new Promise( /* Ejecutor */ (resolver, rechazar) => { ... } );
```

- Resolver y rechazar (en inglés “resolve” y “reject”) son funciones “callbacks especiales”. Cuando invocamos la función **“resolve”** se dispara el consumidor **“.then”** y cuando invocamos la función **“reject”** se dispara la función consumidora **“.catch”**.

Se pueden crear Promesas de 2 formas:

1. La forma estándar de crear promesas es a través de su constructor y asignándola a una variable:

```
let miPrimeraPromise = new Promise( (resolve, reject) => {
  // Llamamos a resolve(...) cuando lo que estábamos haciendo finaliza con éxito, y
reject(...) cuando falla.
  // En este ejemplo, usamos setTimeout(...) para simular código asíncrono.
  setTimeout(function(){
      resolve("¡Éxito!"); // ¡Todo salió bien!
  }, 250);
});


// Y ahora consumimos la promesa:
miPrimeraPromise.then( (successMessage) => {
  // succesMessage es lo que sea que pasamos en la función resolve(...) de arriba.
  // No tiene por qué ser un string, pero si solo es un mensaje de éxito,
probablemente lo sea.
  console.log("¡Sí! " + successMessage);
});
```

2. La segunda forma de crearlas es haciendo uso de una función contenedora. La principal ventaja es que se nos facilita el poder usar parámetros/argumentos dentro de la promesa:

```
function delay(t)
{
  return new Promise( function(resolve, reject) {
      //Código que quieres que se ejecute asíncronamente, por ejemplo:
       setTimeout(() => resolve("Hemos terminado bien"), t );  // → Simulamos algo
q tarda "t" segundos y se llamará al callback "resolve" al terminar.
  });
}

// Y ahora consumimos la promesa:
// delay(4000).then( texto => console.log(texto));
```

Primero se invoca a una función, y esta devuelve una Promise, la cual a su vez tiene 2 callbacks como argumentos: **"resolved" y "reject"**: resolved será llamado cuando la promesa termine bien (y se pasará a su ".then"), y el reject si algo ha ido mal (y se pasará al ".catch").

¿Y ahora, qué hacemos con la promesa que acabamos de crear (delay)? Pues vamos a ver cómo utilizarla:

```
// Lo que se ejecutará cuando la promesa termine y lo haga bien.
function logExito() {  console.log('Promise ejecutada con éxito');   }
// Lo que se ejecutará cuando la promesa termine y lo haga mal.
function logFallo() {  console.log("Promise ha fallado");  }
// Aquí llamamos a la promesa y fíjate en los métodos especiales then y catch:
delay(2000).then(logExito).catch(logFallo);
```

3.3.1. Consumidores de promesas:

Los consumidores dentro del contexto de promesas son los métodos "**then**", "**catch**" y "**finally**". Una cosa importante a resaltar es que estos métodos, a la vez que consumen promesas, **también retornan promesas**, dando la posibilidad de **encadenar promesas**.

3.3.1.1. then:

Sin duda este es el más importante y será ejecutado cuando. Tiene 2 argumentos:

```
promise.then(
  function(result) { /* Maneja un resultado satisfactorio */ } ,
  function(error)  { /* Maneja un error */ }
);
```

➔ **Primer argumento:** El primer argumento se invoca cuando la promesa es "fullfilled" positivamente (resolved), y recibe el resultado.

➙ **Segundo argumento:** El segundo argumento se invoca cuando la promesa es “fullfilled” incorrectamente (rejected).

Ejemplo: partiendo de la siguiente promesa (creada por su constructor) vamos a ver los dos casos: resolved y rejected.

```
let promise = new
Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"),
1000);
});
//Ejecutamos el .then como “resolve”
promise.then(
  result => alert(result) , //Muestra
"done!" después de 1 segundo.
  error => alert(error) // No se ejecuta
);
```

```
let promise = new
Promise(function(resolve, reject) {
  setTimeout(() => reject(new
Error("Whoops!")), 1000);
});

//Ejectuamos .then como “reject”:
promise.then(
  result => alert(result) , //No se
ejecuta
  error => alert(error) //Muestra "Error:
Whoops!" después de 1 segundo.
);
```

3.3.1.2. catch:

Catch es uno de los consumidores de promesas que serán llamados cuando un error ha ocurrido. Si todo va bien, nunca será ejecutado. Veamos un ejemplo en el que se encadenan varias promesas:

```
const a = () => new Promise((resolve) => setTimeout(() => { console.log('a'),
resolve() }, 1e3));
const b = () => new Promise((resolve) => setTimeout(() => { console.log('b'),
resolve() }, 1e3));
const c = () => new Promise((resolve, reject) => setTimeout(() => {
console.log('c'); reject('Oops!') }, 1e3));
const d = () => new Promise((resolve) => setTimeout(() => { console.log('d'),
resolve() }, 1e3));

//Ahora vamos a consumirlas:
a().then(b).then(c).then(d).catch( console.error );
```

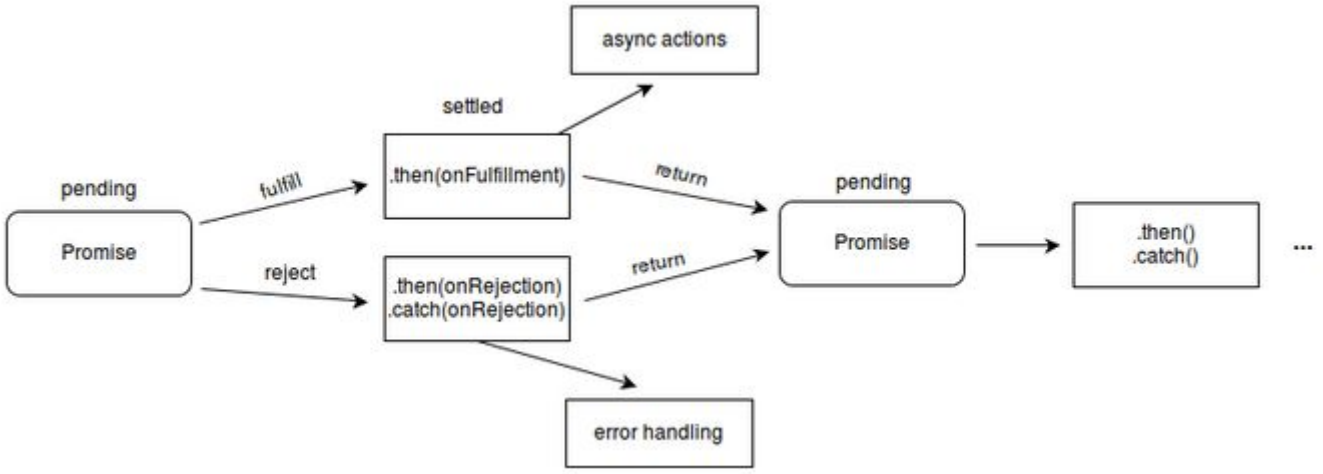
3.3.1.3. finally:

Finally será invocado siempre que la promesa llegue a estado “settled”, es decir, terminada.

```
new Promise((resolve, reject) => {
  /* do something that takes time, and then call resolve/reject */
})
  // runs when the promise is settled, doesn't matter successfully or not
  .finally(() => stop loading indicator)
  .then(result => show result, err => show error)
```

3.3.2. Encadenando promesas

En el siguiente gráfico se ilustra el proceso de encadenamiento de promesas:

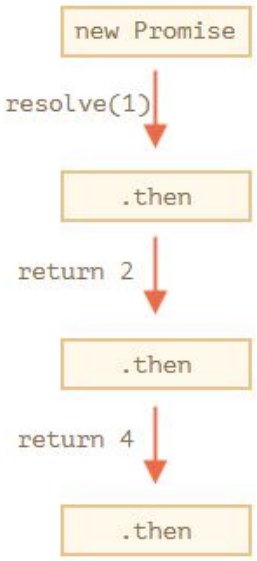


Uno de los puntos fuertes de las promesas frente a los callbacks es la capacidad de simplificar la encadenación de acciones asíncronas sin tener que anidar. Veamos el siguiente ejemplo:

```
new Promise(function(resolve, reject) {
      setTimeout(() => resolve(1), 1000); // (*)
}).then( function(result) { // (**)
      alert(result); // 1
      return result * 2;
})).then(function(result) { // (***)
      alert(result); // 2
      return result * 2;
}).then(function(result) {
      alert(result); // 4
      return result * 2;
});
```

Importante: La cadena de promesas funciona correctamente porque **cada llamada a "promise.then" devuelve una promesa**: Cada promesa devuelta por cada **".then"** retroalimenta a la entrada del siguiente ".then".

Podemos ver un gráfico de cómo sería la ejecución encadenada de promesas:



2.4. Métodos de las clase Promise

Son 4 métodos “estáticos”:

1. Promise.reject
2. Promise.resolve
3. Promise.all
4. Promise.race

2.4.1. Promise.resolve y Promise.reject

Estos 2 métodos son utilizados para resolver o rechazar una promesa de forma inmediata; además, se puede pasar un argumento que será pasado al siguiente **“.then”**. Ejemplos:

```
Promise.resolve('Yay!!!').then(console.log).catch(console.error);
```

```
Promise.reject('Oops 🔥').then(console.log).catch(console.error);
```

Importante: ¿Te has fijado en el código del “.then” ? Vaya chulada, directamente, sin indicar nada está pasando el parámetro de entrada al “console.log”. Y con el “.catch” está pasando lo mismo. Una forma más rudimentaria sería:

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

Vamos a ver un ejemplo muy parecido a uno anterior en el que encadenamos la ejecución de varias promesas:

```
const a = () => new Promise((resolve) => setTimeout(() => { console.log('a'),
resolve() }, 1e3));
const b = () => new Promise((resolve) => setTimeout(() => { console.log('b'),
resolve() }, 1e3));
const c = () => new Promise((resolve, reject) => setTimeout(() => {
console.log('c'); reject('Oops!') }, 1e3));
const d = () => new Promise((resolve) => setTimeout(() => { console.log('d'),
resolve() }, 1e3));

//Ahora vamos a consumirlas:
Promise.resolve()
.then(a)
.then(b)
.then(c)
.then(d)
.catch(console.error)
```

3.4.2. Promise.all()

Un método muy útil es **“all()”**. Con este podemos ejecutar una serie de promesas en orden y cuando hayan terminado todas **“satisfactoriamente”** obtendremos un array de resultados de todas ellas. La sintaxis para usar este método es:

```
Promise.all( [Promise1, Promise2, ...] ).then( onFulfilled , onRejected );
```

Veamos un ejemplo:

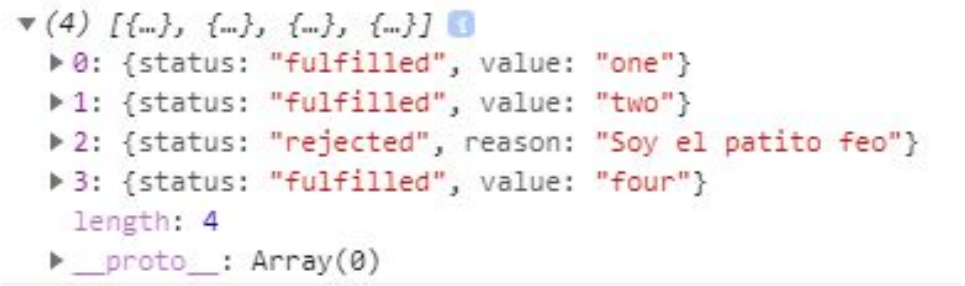
```
function promiseCall(waitSecond, returnData)
{
    return new Promise( function (resolve, reject) {
        //Tarea requiere asincronía:
        setTimeout( function() { resolve(returnData); }, waitSecond );
    });
};

// Consumiendo Promise 1 - 4 in Promise.all()
Promise.all([p1, p2, p3, p4]).then(function (value) {
        console.log(value);  //value es un array con todos los resultados.
    }
).catch ( function (reason) {
    // Not Called
    console.log(reason);
});
```

- En el caso de que una (o todas) las promesas sean “rejected” se disparará el “.catch”. Incluso si fuera la última promesa en terminar su ejecución, se descarta el resultado de las promesas bien resueltas, y se muestra únicamente el mensaje de error de la promesa “rejected”.

3.4.3 Promise.allSettled()

Este método estático para ejecutar una serie de promesas al mismo tiempo es muy parecido al anterior pero se diferencia en que le da igual que las promesas lleguen al estado resolved o rejected: simplemente espera a que todas terminen, y cuando esto ocurre, devuelve un array de resultados similar al de la siguiente foto:

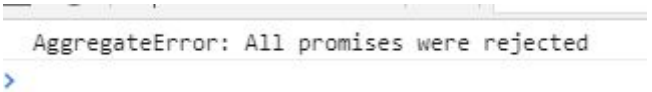


Nos devuelve un array de objetos por cada promesa, y cada objeto nos informa si el estado de la promesa ha sido **“fullfilled”** o **“rejected”**, junto a su correspondiente resultado. Ojo!!! Si es **resolved**, nos devuelve el resultado con el parámetro **“value”**, pero si es **rejected** usa el parámetro **“reason”**.

3.4.4 Promise.any()

Se utiliza para ejecutar en paralelo una serie de promesas pero devuelve el resultado de la primera que llegue al estado “**resolved**”. Si esto ocurre, descarta la ejecución de las otras promesas que no hayan terminado aún.

¿Y qué ocurre si todas las promesas terminan “rejected”? Cuando esto ocurre todas se devuelve el siguiente mensaje de error:



3.4.5 Promise.race()

Promise.race() es un método estático que nos ayuda en aquellas situaciones en las que queremos ejecutar varias promesas pero solo debemos mostrar o manipular **el resultado de la primera que haya terminado**. Fíjese que a la hora de usarlo **debemos de añadir un array con todas las promesas de la carrera**. Veámoslo con un ejemplo:

```
var promise1 = new Promise((resolve, reject) => {
      setTimeout(() => resolve('Promise-one'), 500);
```

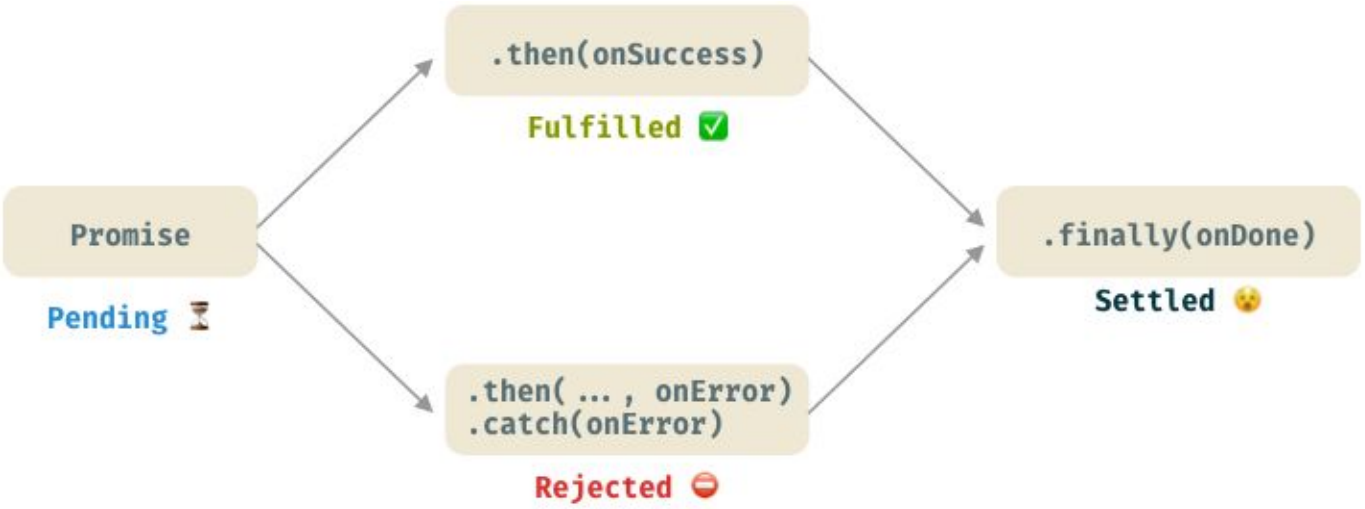
```
});

var promise2 = new Promise((resolve, reject) => {
      setTimeout(() => resolve('Promise-two'), 100);
});

Promise.race([promise1, promise2]).then((value) => {
      console.log(value);
// Ambas han terminado con "resolve", pero solo la promise2 es resuelta ya que
// ha terminado antes que promise1.
});
```

- La diferencia con el operador “any” es que al comando “race” le da igual que la primera promesa que termina haya sido “resolved” o “rejected”. Sea como sea, la primera que termina es la que se devuelve.

3.5. Promesas: resumiendo conceptos



EJERCICIO 2: Promises

En el comentario “Tu código” debes de escribir una promesa que simule un delay haciendo uso de la función setTimeout:

```
function delay(ms) {
      // Tu código
}
delay(3000).then(() => alert('runs after 3 seconds'));
```

EJERCICIO 3: Promises

El siguiente código está escrito con “callbacks”. Trata de sustituir los callbacks por promesas y escribe cómo harías uso de las promesas (es decir, primero tienes que hacer las modificaciones para intercambiar callbacks por promesas, y una vez lo tengas, llama a la función y captura la salida de la promesa). Para llamar a la función loadScript vas a necesitar una url. Puedes usar la siguiente url:

https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Script load error for ${src}`));

  document.head.append(script);
}
```

EJERCICIO 4: Promises

Con los apuntes, y haciendo búsquedas por internet realiza dos ejercicios que hagan uso de los métodos **Promise.all** y **Promise.race**. Comenta el código explicando perfectamente su funcionamiento.

4. Async / Await

4.1 Funciones Async (asíncronas)

Este tipo de funciones siempre tiene que ir precedido de la palabra “**async**” he indicará que la función a la que acompaña, internamente, devolverá una promesa y devolverá un valor cuando sea “resolved”. Se utilizan para ejecutar código que requiera ejecutarse en segundo plano. Veamos un ejemplo:

```
async function funcionAsincrona() {
  return 1;       //Equivale a:   return Promise.resolve(1);
}

funcionAsincrona().then(alert); // 1
```

Si te fijas en el ejemplo, al ejecutar la función podemos tratarla como una promesa y utilizar los métodos consumidores “.then” , “.catch” y “.finally”.

4.2 Await

Await es una palabra reservada, la cual solo va a funcionar dentro de funciones asíncronas. Esta esperará dentro de la funciones asíncronas a que la promesa a la que se la relaciona esté en estado “settles” y devuelva un valor. Por lo tanto 2 cosas importantes:

1. Solo funciona dentro de funciones async.
2. Solo se utiliza delante de una promesa para esperar que esta termine.

Veamos un ejemplo:

```
async function probando() {

            let promise = new Promise((resolve, reject) => {
                setTimeout(() => reject("BAD DONE!"), 1000);
            });

            try{
                let result = await promise; // espera hasta que la promesa se resuelva
            }
            catch(error)
            {
                return error;
            }
            return result; // "done!"
        }

        probando().then(console.log).catch(console.error);
```

- La idea principal de las funciones **async** es englobar código que se debe ejecutar en segundo plano, es decir, código que requiere tiempo en su ejecución, y **await** se utiliza internamente para “esperar” por cada de una de las acciones. Esas acciones (pesadas) deben de estar a su vez englobadas o ejecutadas como una promesa para que “await” pueda esperar y consumirla.

4.3. Manejar errores

¿No te has preguntado qué pasa si la promesa a la que “await” está esperando a ejecutar lanza un reject? Pues tan simple como emplear un try/catch. Ejemplo:

```
async function llamadaServidor() {

  try {
      let response = await fetch('/no-user-here');
      let user = await response.json();
      alert(“hola”+ user.name);
  } catch(err) {
      // catches errors both in fetch and response.json
      alert(err);
```

```
  }
}
llamadaServidor();
```

EJERCICIO 5: async/await

Convierta las siguientes líneas de código haciendo uso de **async/await** en vez de **.then/catch**:

- Para resolver este ejercicio hay que tener en cuenta que el método “fetch” el cual se utiliza para hacer llamadas HTTP a un servidor, e internamente, cuando se finaliza, **devuelve una promesa**.
- Para probarlo bien, podéis fabricar un archivo nombre.json con contenido Json y cargarlo desde local. O también podéis llamar a un servidor gratis que sirva información JSON como puede ser “https://httpbin.org/json”.
- Si hubiera algún error en el código, arreglarlo antes de hacer la modificación.

```
function loadJson(url) {
  return fetch(url)
      .then(response => {
      if (response.status == 200) {
         return response.json();
      } else {
         throw new Error(response.status);
      }
      })
}

loadJson('no-such-user.json') // (3)
  .catch(alert); // Error: 404
```

EJERCICIO 6 : Revoltillo

Partiendo del siguiente código:

```
const posts = [
    {title: "Post1", body: "This is post one"},
    {title: "Post2", body: "This is post two"}
];

function getPosts(){
    setTimeout(()=>
    {
        let output =``;
        posts.forEach( (post, index) => {
            output += `<li>${post.title}</li>`;
        })
        document.body.innerHTML = output;
```

```
    }
    , 1000);
}

function createPost (newPost) {
    setTimeout( () => { posts.push(newPost) } , 2000);
}

getPosts();
createPost({title: "Post3", body: "This is post three"});
```

Vamos a resolver el problema que podrá usted observar; el problemilla es que el método **"createPost"** tarda 2 segundos en ejecutarse, y el **"getPosts"** tarda un segundo, y da igual el orden en el que llames a los dos métodos que el tercer post que debe mostrarse nunca se muestra. El resultado final debe de ser así:

```
Post1
Post2
Post3
```

- **Ejercicio 6.1:** Busque una solución al problema haciendo uso de **callbacks**.
- **Ejercicio 6.2:** Busque una solución al problema haciendo uso de **promises**.
- **Ejercicio 6.3:** Busque una solución al problema haciendo uso de **async/await**.

EJERCICIO 7 : Uno sencillo de callbacks (es tan fácil que Laureano muchas vueltas)

Complementa el siguiente código de forma que se imprima por consola la salida: "1 2 3".

```
function useCallback(callback)
{
      callback(1);
      callback(2);
      callback(3);
}

function callback(sentence) {
}
```

Salida:

```
1
2
3
```

5. Enlaces

Buen tutorial sobre promesas:

https://adrianmejia.com/promises-tutorial-concurrency-in-javascript-node/

https://nodejs.dev/understanding-javascript-promises

Buen video tutorial sobre async/await (20 min.):

https://youtu.be/USuhP9F56UE

https://www.oscarblancarteblog.com/2019/03/15/javascript-async-await/

Buen tutorial sobre async/await:

https://itnext.io/a-beginners-guide-to-async-await-in-javascript-97750bd09ffa

Explicación de la diferencia entre el segundo argumento de .then y .catch:

https://stackoverflow.com/questions/24662289/when-is-thensuccess-fail-considered-an-antipattern-for-promises