EXERCISES QUE TRABAJAREMOS EN EL CUE:

- EXERCISE 1: PROMESAS.
- EXERCISE 2: FUNCIONES ASYNC/AWAIT.

EXERCISE 1: PROMESAS

Otro componente nuevo que hace uso de callbacks, es el concepto de Promesas que se realizan con el objeto `Promise`. Éste representa la eventual finalización (o falla) de una operación asincrónica, y su valor resultante.

En sí, una **Promesa** es un `proxy` o *marcador de posición* para un valor, que no necesariamente se conoce cuando se crea. Le permite asociar controladores con el valor eventual del éxito o fracaso de una acción asincrónica.

Esto permite que los métodos asincrónicos devuelvan valores tal como los métodos síncronos, pero en vez de devolver sí o sí el valor final, ***promete*** proporcionar el valor en algún momento *en el futuro*.

La sintaxis de las Promesas consta de dos partes: primero, una configuración de la Promesa al momento de inicializar una nueva; y segundo, el llamado a la Promesa junto con los métodos que manejan un valor resuelto y un valor rechazado.

```
1  //Configuramos la promesa:
2  const miPromesa = new Promise((resolver, rechazar) => {
3          setTimeout(() => {
4                  resolver('foo');
5          }, 300);
6  });
7  //Llamamos a la promesa:
8  miPromesa
9    .then(manejadorResueltoA, manejadorRechazadoA)
10   .then(manejadorResueltoB, manejadorRechazadoB)
11   .then(manejadorResueltoC, manejadorRechazadoC);
```

Analizando primero la configuración de la Promesa, podemos ver que ésta recibe como parámetro un argumento de nombre **resolver,** y otro de nombre **rechazar**. Éstos corresponden a mensajes de éxito y de error al momento de ejecutar la acción asíncrona. Además, podremos notar el método `setTimeout()`, que realiza una acción asíncrona dentro de un tiempo delimitado en milisegundos (en este caso `300`).

Luego, en la llamada a la Promesa utilizamos el método **.then**, el cual recibe como parámetros dos métodos. El primer argumento es una función **callback** para el caso resuelto de la Promesa, y el segundo argumento es una función **callback** para el caso rechazado.

Para entender esto mejor, vamos a desarrollar el siguiente ejemplo:

```
<body>
        <div class="container">
                <div><b>Demostración de Promesa o Promise</b></div>
                <div>El objeto Promise representa la eventual finalización
o falla de una operación asincrónica y su valor resultante.</div>
                <br>
                <h2>Resultado:</h2>
                <h2 id="d1"></h2>
        </div>
</body>
```

```
let miPromesa = new Promise(function(miResolucion, miRechazo) {
        setTimeout(function() {
                if(true) { //En este caso la promesa se cumple
                        miResolucion("Funcionó :)");
                } else {
                        miRechazo("No funcionó :(")
                }
        }, 1000);
});
miPromesa.then(function(value) { //En caso de cumplirse
        document.getElementById("d1").innerHTML = value;
}, function(value) { //En caso de fallo
        alert(value)
});
```

En este caso, el éxito o el fracaso de la Promesa depende del valor del bucle **If**. Cuando es falso, la rechaza, y cuando es verdadero, la resuelve. Partiremos viendo el resultado cuando la función cumple la Promesa. En nuestro navegador, se verá así:

Demostración de Promesa o Promise

El objeto Promise representa la eventual finalización o falla de una operación asincrónica y su valor resultante.

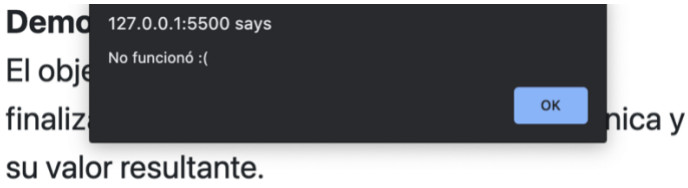
Resultado:

Funcionó :)

Ahora, vamos a ver el resultado cuando el bucle `If` es falso.

```
// CONFIGURACION:
let miPromesa = new Promise(function(miResolucion, miRechazo) {
        setTimeout(function() {
                if(false) { //En este caso la promesa NO se cumple
                        miResolucion("Funcionó :)");
                } else {
                        miRechazo("No funcionó :(")
                }
        }, 1000);
});
// LLAMADA A LA PROMESA:
miPromesa.then(function(value) { //En caso de cumplirse
        document.getElementById("d1").innerHTML = value;
}, function(value) { //En caso de fallo
        alert(value)
});
// This is just a sample script. Paste your real code (JavaScript, CSS or
HTML) here.
if('this_is' == /an_example/) {
        of_beautifer();
} else {
        var a = b ? (c % d) : e[f];
}
```

Si vamos a nuestro navegador, veremos el siguiente resultado indicando que la acción asíncrona devolvió un error, y fue manejado por la función de fallo del método `.then`.



Resultado:

El método `.then()` no es el único que podemos usar para manejar las respuestas de la Promesa. También existe el método `.catch()`:

```
miPromesa.catch(function(value) {
        console.log(value)
})
```

Un `.catch()` es en realidad solo un `.then()`, sin un espacio para una función `callback`, para el caso en que se resuelve la Promesa.

De esta manera hemos analizado toda la información necesaria para poder incorporar Promesas en nuestro desarrollo ES6, al igual que la incorporación de la nueva sintaxis del bucle `for/of`.

EXERCISE 2: FUNCIONES ASYNC/AWAIT

Las funciones async facilitan el uso de Promesas al reducir su sintaxis, solo requiriendo agregar la palabra clave `async` antes de cualquier función:

```
async function nombreFuncion(){

}
```

Una de sus principales características es que siempre devuelven Promesas. Podemos demostrarlo en nuestro navegador, usando el siguiente código:

```
// Funcion Normal:
function funcNormal() {
    return "Hola";
}

// Funcion Async:
async function funcAsync() {
```

```
    return "Hola";
}
```

Con solo mirar las 2 funciones, podemos ver fácilmente que tienen la misma funcionalidad de devolver la cadena `"Hola"`, con la única diferencia de que `funcAsync` es una función `async`. Nuestro primer instinto nos podría llevar a pensar que si aplicamos un `console.log()` a cada una de estas funciones, obtendríamos la cadena `"Hola"` de ambas partes, pero ese no es el caso. Dado que `funcAsync` es una función `Async`, tiene que devolver una Promesa, incluso si fuera una función completamente vacía. Si ejecutamos los siguientes comandos, y nos dirigimos a nuestro navegador, deberíamos ver el siguiente resultado en la consola:

```
// Muestra de resultados:
console.log(funcNormal())
console.log(funcAsync())
```

Como podemos ver, nuestra segunda función muestra efectivamente que devuelve una Promesa resuelta (`<fulfilled>`), con el valor retornado `'hola'`.

Una función asíncrona también se puede escribir usando la sintaxis de la función de flecha. Para hacerlo, basta con incorporar la palabra clave `async` antes del paréntesis. El siguiente código nos permite observar nuestra función `funcAsync`, escrita en el estilo de función de flecha:

```
// Funcion de Flecha con Async:
const funcAsync = async () => {
    return "Hola";
}
```

Ahora, continuaremos considerando el segundo elemento importante de cada función `async`: el operador `await`. "Await" en castellano significa *"esperar"* , o *"a la espera de"*; y este nombre es apropiado porque el

operador se utiliza para ***esperar*** a que se ejecute una Promesa, y consta de 2 partes: una "expresión", primordialmente Promesas; y un "valor retornado", que corresponde al valor de la Promesa resuelta o rechazada. A continuación, se muestra la sintaxis del operador `await`:

```
valorRetornado = await expresion
```

Dentro de esta sintaxis, la palabra clave `await` se coloca antes de una promesa (`"expresion"`), y hace que JavaScript espere hasta que dicha promesa devuelva un resultado (`"valorRetornado"`). Eso si, `await` solo hace *que el bloque de funciones* `async` espere, **no toda la ejecución** del programa.

Otro punto importante: solo se puede un usar el operador `await` dentro de una función `async`.

El bloque de código a continuación, muestra el uso de `async` y `await` juntos.

```
// Primera Promesa tarda 3 segundos.
function a() {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve('Esto tarda 3 segundos');
        }, 3000);
    });
}

// Segunda Promesa tarda 1 segundo.
function b() {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve('Esto tarda 1 segundo');
        }, 1000);
    });
}

// Definimos funcion async.
async function llamadaAsync() {
    console.log('INICIANDO...');

    // Primer await:
    const resultado1 = await a();
    console.log(resultado1);

    console.log('... pausa entre promesas ...')

    // Segundo await:
```

```
30      const resultado2 = await b();
31      console.log(resultado2);
32
33      console.log('...FIN');
34  }
35
36  llamadaAsync();
```

En este código definimos dos Promesas: `a()` y `b()`. La primera tarda 3 segundos en resolverse, mientras que la segunda tarda 1 segundo en hacerlo. Llamamos a ambas Promesas dentro de nuestra función Async por nombre `llamadaAsync()`, guardando los valores devueltos en `result1` y `result2` respectivamente. El resto del código dentro del alcance de la función `async`, es solo un par de llamadas `console.log()`, para tener algunos mensajes de referencia durante la ejecución. Finalmente, en la última línea de código llamamos a nuestra función async, obteniendo el resultado en nuestra consola:

```
INICIANDO...
Esto tarda 3 segundos
... pausa entre promesas ...
Esto tarda 1 segundo
...FIN
>
```

Como podemos ver en la ejecución de nuestro código, la función `async` se detuvo cuando llegó a nuestro primer `await`, y solo continuó una vez que obtuvo su valor de retorno. Aunque esto también sucedió para nuestra segunda expresión `await`, fue más notorio en la primera dada la demora de 3 segundos. Todo esto solo demuestra cómo el operador `await` puede detener completamente la ejecución de una función `async`, continuando solo después de obtener el resultado de una Promesa.

En este ejemplo, las funciones `async / await` nos dan la posibilidad de usar Promesas fácilmente en un estilo sintácticamente más limpio, permitiéndonos ejecutar múltiples mientras controlamos el flujo de ejecución. Si bien el ejemplo utilizado fue de naturaleza simple, podemos tener una base sólida de entendimiento de estas funciones, las cuales posteriormente podremos implementar a mayor escala.