ANEXO II.

1. Los *callbacks*

Uno de los dos pilares en los que se sustenta la programación asíncrona en Javascript lo conforman los *callbacks*. **Un callback es una función A que se pasa como parámetro a otra B, y que será llamada en algún momento durante la ejecución de B (normalmente cuando B finaliza su tarea). Este concepto es fundamental para dotar a Node.js (y a Javascript en general) de un comportamiento asíncrono: se llama a una función, y se le deja indicado lo que tiene que hacer cuando termine, y mientras tanto el programa puede dedicarse a otras cosas.**

Un ejemplo lo tenemos con la función setTimeout de Javascript. A esta función le podemos indicar una función a la que llamar, y un tiempo (en milisegundos) que esperar antes de llamarla. Ejecutada la línea de la llamada a setTimeout, el programa sigue su curso y cuando el tiempo expira, se llama a la función callback indicada.

Probemos a escribir este ejemplo en un archivo llamado “callback.js” en nuestra subcarpeta "ProyectosNode/Pruebas/PruebasSimples":

```
setTimeout(function() {console.log("Finalizado callback");}, 2000);
console.log("Hola");
```

Si ejecutamos el ejemplo, veremos que el primer mensaje que aparece es el de “Hola”, y pasados dos segundos, aparece el mensaje de “Finalizado callback”. Es decir, hemos llamado a setTimeout y el programa ha seguido su curso después, ha escrito “Hola” por pantalla y, una vez ha pasado el tiempo estipulado, se ha llamado al *callback* para hacer su trabajo.

Utilizaremos *callbacks* ampliamente durante este curso. De forma especial para procesar el resultado de algunas promesas que emplearemos (ahora veremos qué son las promesas), o el tratamiento de algunas peticiones de servicios.

2. Las promesas

Las promesas son otro mecanismo importante para dotar de asincronía a Javascript. Se emplean para definir la finalización (exitosa o no) de una operación asíncrona. En nuestro código, podemos definir promesas para realizar operaciones asíncronas, o bien (más habitual) utilizar las promesas definidas por otros en el uso de sus librerías.

A lo largo de este curso utilizaremos promesas para, por ejemplo, enviar operaciones a una base de datos y recoger el resultado de las mismas cuando finalicen, sin bloquear el programa principal. Pero para entender mejor qué es lo que haremos, llegado el momento, conviene tener clara la estructura de una promesa y las posibles respuestas que ofrece.

Crear una promesa. Elementos a tener en cuenta

En el caso de que queramos o necesitemos crear una promesa, se creará un objeto de tipo *Promise*. A dicho objeto se le pasa como parámetro una función con dos parámetros:

- La función *callback* a la que llamar si todo ha ido correctamente
- La función *callback* a la que llamar si ha habido algún error

Estos dos parámetros se suelen llamar, respectivamente, *resolve* y *reject*. Por lo tanto, un esqueleto básico de promesa, empleando *arrow functions* para definir la función a ejecutar, sería así:

```
let nombreVariable = new Promise((resolve, reject) => {
    // Código a ejecutar
    // Si todo va bien, llamamos a "resolve"
    // Si algo falla, llamamos a "reject"
});
```

Internamente, la función hará su trabajo y llamará a sus dos parámetros en uno u otro caso. En el caso de *resolve*, se le suele pasar como parámetro el resultado de la operación, y en el caso de *reject* se le suele pasar el error producido.

Veámoslo con un ejemplo. La siguiente promesa busca los mayores de edad de la lista de personas vista en un ejemplo anterior. Si se encuentran resultados, se devuelven con la función *resolve*. De lo contrario, se genera un error que se envía con *reject*. Copia el ejemplo en un archivo llamado "prueba_promesa.js" en la carpeta "ProyectosNode/ Pruebas/PruebasSimples" de tu espacio de trabajo:

```
let datos = [
    {nombre: "Diego", telefono: "966112233", edad: 40},
    {nombre: "Carmen", telefono: "911223344", edad: 35},
    {nombre: "Victor", telefono: "611998877", edad: 15},
    {nombre: "Carolina", telefono: "633663366", edad: 17}
];

let promesaMayoresDeEdad = new Promise((resolve, reject) => {
    let resultado = datos.filter(persona => persona.edad >= 18);
    if (resultado.length > 0)
        resolve(resultado);
    else
        reject("No hay resultados");
});
```

La función que define la promesa también se podría definir de esta otra forma:

```
let promesaMayoresDeEdad = listado => {
    return new Promise((resolve, reject) => {
        let resultado = listado.filter(persona => persona.edad >= 18);
        if (resultado.length > 0)
            resolve(resultado);
        else
            reject("No hay resultados");
    });
};
```

Así no hacemos uso de variables globales, y el array queda pasado como parámetro a la propia función, que devuelve el objeto *Promise* una vez concluya. Deja definida la promesa de esta segunda forma en el archivo fuente de prueba.

Consumo de promesas

En el caso de querer utilizar una promesa previamente definida (o creada por otros en alguna librería), simplemente llamaremos a la función u objeto que desencadena la promesa, y recogemos el resultado. En este caso:

- Para recoger un resultado satisfactorio (*resolve*) empleamos la cláusula then.
- Para recoger un resultado erróneo (*reject*) empleamos la cláusula catch.

Así, la promesa anterior se puede emplear de esta forma (nuevamente, empleamos *arrow functions* para procesar la cláusula then con su resultado, o el catch con su error):

```
promesaMayoresDeEdad(datos).then(resultado => {
    // Si entramos aquí, la promesa se ha procesado bien
    // En "resultado" podemos acceder al resultado obtenido
    console.log("Coincidencias encontradas:");
    console.log(resultado);
}).catch(error => {
    // Si entramos aquí, ha habido un error al procesar la promesa
    // En "error" lo podemos consultar
    console.log("Error:", error);
});
```

Copia este código bajo el código anterior en el archivo "prueba_promesa.js" creado anteriormente, para comprobar el funcionamiento y lo que muestra la promesa.

Notar que, al definir la promesa, se define también la estructura que tendrá el resultado o el error. En este caso, el resultado es un vector de personas coincidentes con los criterios de búsqueda, y el error es una cadena de texto. Pero pueden ser el tipo de dato que queramos.

Para poder llamar a la promesa como la habíamos puesto inicialmente, debemos entender un poco más del método then().

El método then() retorna una Promesa. Recibe dos argumentos: funciones callback para los casos de éxito y fallo de Promise. Y seguirá esta estructura:

p.then(alCumplir[, enRechazo]);

```
p.then(function(value) {
   // cumplimiento
  }, function(reason) {
  // rechazo
});
```

Por lo tanto el código quedaría:

```
promesaMayoresDeEdad.then(function(value) {
      // cumplimiento
      console.log("Coincidencias encontradas:");
      console.log(value);
  }, function(reason) {
      // rechazo
      console.log("Error:", reason);
});
```

Usando funciones flecha:

```
promesaMayoresDeEdad.then(value => {
      // cumplimiento
      console.log("Coincidencias encontradas:");
      console.log(value);
  }, reason => {
      // rechazo
      console.log("Error:", reason);
});
```

Ejercicio

Crea una carpeta llamada "Ejercicio_Anexo_2" en tu espacio de trabajo, en la carpeta de "Ejercicios". Crea dentro un archivo fuente llamado "promesas.js", que sea una copia del archivo fuente "arrow_functions.js" del ejercicio anterior.

Lo que vas a hacer en este ejercicio es adaptar las dos funciones "nuevaPersona" y "borrarPersona" para que devuelvan una promesa.

- En el caso de "nuevaPersona", se devolverá con *resolve* el objeto persona insertado, si la inserción fue satisfactoria, o con *reject* el mensaje "Error: el teléfono ya existe" si no se pudo insertar la persona porque ya existía su teléfono en el vector
- En el caso de "borrarPersona", se devolverá con *resolve* el objeto persona eliminado, si el borrado fue satisfactorio, o con *reject* un mensaje "Error: no se encontraron coincidencias" si no existía ninguna persona con ese teléfono en el vector.

Modifica el código del programa principal para que intente añadir una persona correcta y otra equivocada (teléfono ya existente en el vector), y borrar una persona correcta y otra equivocada (teléfono no existente en el vector). Comprueba que el resultado al ejecutar es el que esperabas.