1. Introducción

Establecer un setTimeout para que el 300 milisegundos saque el mensaje TIMED OUT!

```
setTimeout(() => {
    console.log("TIMED OUT!")
}, 300);
```

Se define la función setTimeout, no tiene mucho más.

2. Cumplir una promesa

Es una variante del anterior en la que ya introducimos una promesa y tras cumplirse tarda 300 milisegundos en realizar la función de validación.

```
var promise = new Promise(function (fulfill, reject) {
 fulfill("FULFILLED!");
});

promise.then(setTimeout(() => {
    console.log("FULFILLED!")
}, 300));
```

Aquí ya se define una promesa. Se les pasan como argumento dos funciones, una de éxito y otra de error. En este caso, definimos la de éxito, que es la que nos interesa (sólo sacará una cadena de texto). Luego, con .then, indicamos como en el ejercicio anterior que tras 300 milisegundos pase el resultado de la función del parámetro fulfill.

3. Rechazar una promesa

Similar al anterior, pero en este caso con la función de error.

```
var promise = new Promise(function (fulfill, reject) {
    setTimeout(reject, 300, new Error("REJECTED!"));
});
```

```
function onReject (error) {
   console.log(error.message);
};

promise.then(0,onReject);
```

En este caso, el ejercicio pide operar con el .then, pero para mostrar el mensaje de error. Then puede manejar los dos parámetros de la promesa, pero para manejar el de error hay que pasar primero por encima del resolve (o fulfill). Esto lo hacemos pasando un valor como 0 o null, con lo que la función no tiene valor true, por tanto, no se completa y salta al error. En este caso, el temporizador ya está definido en la función reject.

4. Rechazar, o no rechazar

El objetivo del ejercicio es demostrar que la promesa solo se resuelve una vez, por lo que va a devolver el resultado de la validación o bien el de la función de error.

```
var promise = new Promise (function (fulfill, reject) {
   fulfill("I FIRED");
   var err = new Error ("I DID NOT FIRE");
   reject(err);
});

function onRejected(error) {
   console.log(error);
};
promise.then(onRejected,onRejected);
```

Aquí lo que pide el ejercicio es declarar la promesa con dos funciones. En caso de que la promesa pase a estado fulfilled devuelve el string “I FIRED”, mientras que de lo contrario genera un nuevo error que muestra el mensaje “I DID NOT FIRE”, que viene a través del parámetro reject. Como resolverá una o otra, pero no las dos a la vez, para probarlo se le pasa la función onRejected en los dos casos, tanto en caso de éxito como de fracaso, pero a la vez. Sin embargo, para que la promesa se resuelve no hace falta nada, porque va a devolvernos el mensaje “I FIRED”, que es lo que aparecerá por consola (lo cual le da un valor booleano de true), haciendo que el segundo parámetro no se llegue a ejecutar.

5. Siempre asíncrono

Las promesas siempre son asíncronas, porque de eso se trata. Para demostrarlo, el ejercicio pide mostrar un mensaje por consola que se define después, pero que llega antes que el que devuelve la promesa.

```
var promise = new Promise (function (fulfill, reject) {
    fulfill("PROMISE VALUE");
});

promise.then((fulfill) => console.log(fulfill));
console.log("MAIN PROGRAM");
```

El resultado de este ejercicio muestra primero el mensaje “MAIN PROGRAM” y posteriormente “PROMISE VALUE”, para demostrar que el segundo llega después que el primero. Para ello, construimos la promesa como de costumbre, le damos el valor adecuado para el caso de éxito, y de vuelta al código del programa principal, aunque definamos primero la acción que llevará a cabo then, actúa antes el console.log, dado que las promesas se resuelven cuando se pueda. En este caso, el programa principal tiene preferencia, y el console.log salta antes.

6. Atajos

Aquí no hay una solución válida. Esta tarea simplemente deja que probemos con las funciones resolve y reject, y el catch. Con que usemos todos, la da por buena. Por ejemplo, con este programa:

```
var promise = Promise.resolve('SECRET VALUE');
var promise = Promise.reject(new Error('SECRET VALUE'));

promise.catch(function (err) {
    console.error('QUIETOR!!');
    console.error(err.message);
});
```

Se pueden definir las funciones de resolve (o de fulfill, según lo ponga) y de reject por separado con las dos primeras líneas. El resto, sólo hace uso del catch para mostrar un primer aviso de error y a continuación el mensaje del objeto Error que se ha creado en la función (el ejercicio sí exige que se cree este objeto, qué le vamos a hacer...)

7. Promise After Promise

Como funciones que pueden devolver un valor, las promesas, o más concretamente, el método then, se puede encadenar para realizar varias operaciones seguidas. En este ejercicio, se suministran dos funciones (las pasa el propio programa al hacer el test) llamadas first() y second(). La idea es encadenarlas.

```
first().then(second).then(console.log);
```

Pues resulta que la solución más simple era la correcta. El resultado de first se lleva a second y luego se saca por consola, como pide el ejercicio.

8. Valores y promesas

En este ejercicio vemos que las promesas pueden devolver un valor que se puede seguir usando con otras funciones. El propósito aquí es hacer una concatenación con el título “DR.” y un nombre. En nuestro caso, vamos a lo grande y llamamos al Dr. Manhattan.

```
var promise = Promise.resolve("MANHATTAN");

function attachTitle(name) {
    return res=("DR. "+name);
};

promise.then(attachTitle).then(console.log);
```

La promesa nos devolverá el string “MANHATTAN”, y lo usaremos para concatenarlo a “DR.” con la función attachTitle. Para ello, esta función se le pasa a then, de manera que le pasa a la propia función attachTitle el resultado que devuelve la promesa cumplida (“MANHATTAN”) como parámetro. Asi, el segundo then recoge el res que devuelve attachTitle y saca por consola “DR. MANHATTAN”.