Promises

WARNING! This is an advanced topic which is aimed at those of you finding the unit, and programming in general, relatively straightforward. You should only attempt this if you have completed **all** previous exercises and are confident with your answers. If you are struggling with the unit it is recommended that you catch up with previous topics first. You can then return to this topic later, when you are confident with all previous topics.

Using Promises

Introduction

A promise is an object which "promises" to do a particular asynchronous, background task which will complete at some point in the future. An AJAX request is a typical example; promises are used extensively in AJAX development). One of two functions are called depending on whether it fulfilled that "promise":

- a "resolve" function if it did fulfil the promise;
- and a "reject" function if it didn't.

Promises allow you to write code with intuitive, clean syntax in this form:

```
promise.then(resolveFunction).catch(rejectFunction);
```

where "promise" is the promise object. What this code means is:

```
Perform the task associated with a promise, then run a
and catch any  error conditions with an error handling
(rejectFunction)".
```

Hopefully you can see that this code is cleaner than the typical callback-based AJAX code. In general, **Promises allow us to write code for asynchronous processes (i.e. callback based processes, where a function runs at some future point) in a cleaner and more sequential way.** This is because the "resolve" (fulfilment) function ***will only run once the asynchronous task (e.g. AJAX call) has completed***.

Promises produce cleaner code; however, learning how to write a promise in the first place is conceptually harder than the "classic" approach to asynchronous tasks such as AJAX. However, **using** pre-written Promises is easy, so once you've written your promise, coding does become easier!

How to implement a promise

A promise has an associated ***task***, which is a background task which may either succeed (fulfilling the promise) or fail (rejecting the promise). This task is implemented as a ***function*** which ***must be passed into the Promise constructor as a parameter***. This task function ***must*** take two parameters, each of which is themselves a callback function:

- the ***resolve*** function, which is the function which runs if the promise is fulfilled;
- the ***reject*** function, which is the function which runs if the promise is rejected.

Each of these is specified by the code calling the promise; the parameter to ***then()*** ends up as the ***resolve*** function and the parameter to ***catch()*** ends up as the ***reject*** function.

The task function must then call these two functions as appropriate, i.e. if the task succeeds, it must call the ***resolve*** function and if it fails, it must call the ***reject*** function.

For example, if we are developing a promise-based AJAX background task, the "resolve" function would be the AJAX callback to parse the JSON (or whatever) and the "reject" function the AJAX error handler. We might call the "reject" function if the service returns an HTTP error code (in the 400-599 range)

Here is an example of such a Promise associated with an AJAX request.

```
var p = new Promise(
            (resolve,reject)=> // this is the task func
            {
                var xhr2 = new XMLHttpRequest();
                xhr2.open('GET', 'webservice.php');
                xhr2.addEventListener("load", (e)=>
                {
                    if(e.target.status>=400 && e.target
                    {
                        // pass the HTTP code to the re
                        reject(e.target.status);
                    }
                    else
                    {
                        // pass the XMLHttpRequest to r
                        resolve(e.target);
                    }
                } );
                xhr2.send();
```

```
            }
        );
```

So how can we we use the promise object in a friendly syntax such as:

```
promise.then(resolveFunction).catch(rejectFunction);
```

What we need is to ***write a function which creates and returns the Promise***. Here is an example:

```
function makeAjaxPromise(url)
{
    var p = new Promise(
            (resolve,reject)=> // this is the task func
            {
                var xhr2 = new XMLHttpRequest();
                xhr2.open('GET', url);
                xhr2.addEventListener("load", (e)=>
                {
                    if(e.target.status>=400 && e.target
                    {
                        reject(e.target.status);
                    }
                    else
                    {
                        resolve(e.target);
                    }
                } );
                xhr2.send();
            }
        );

    return p;
}
```

The ***makeAjaxPromise()*** method creates the Promise object (containing the task function) and returns it. This allows us to write this code:

```
makeAjaxPromise('webservice.php').then(ajaxSuccess).cat
```

Because ***makeAjaxPromise()*** returns a Promise object, it can be placed in the chain as above. The ajaxSuccess() and handleError() codes would just be standard AJAX callback functions:

```
function ajaxSuccess(xmlHTTP)
{
    var data = JSON.parse(xmlHTTP.responseText);
    // do something with the data..
}

function handleError(error)
{
    alert('Error: ' + error);
}
```

In real code, we wouldn't give the promise-returning function a name like ***makeAjaxPromise()***. We would give it an intuitive name, like, say, ***httpGet()***. Here's the above example rewritten with this function name, and also directly returning the promise object, rather than storing it in a variable "p":

```
function httpGet(url)
{
    return new Promise(
            (resolve,reject)=> // this is the task func
            {
                var xhr2 = new XMLHttpRequest();
                xhr2.open('GET', url);
                xhr2.addEventListener("load", (e)=>
                {
                    if(e.target.status>=400 && e.target
                    {
                        reject(e.target.status);
                    }
                    else
                    {
                        resolve(e.target);
                    }
                } );
                xhr2.send();
            }
        );
}
```

We can now write intuitive code such as:

```
httpGet('webservice.php').then(ajaxSuccess).catch(hand
```

Chaining "then" calls

We can chain several "then" calls. The idea is that each function in a "then" call is called sequentially. We could have a promise chain looking something like this:

```
httpGet('webservice.php').then(parseJson).then(showJson
```

How might this work? The idea is that ***parseJson()*** itself returns a further Promise. Here is our ***parseJson()*** function:

```
function parseJson(xmlHTTP)
{
    return new Promise ( (resolve,reject) =>
        {
            var parsedData = JSON.parse(xmlHTTP.respons
            if(parsedData.length==0)
            {
                reject("No matching results!");
            }
            else
            {
                resolve(parsedData);
            }
        }
    );
}
```

Note how we pass the parsed JSON data into the ***resolve()*** function if the parsing succeeded (at least 1 result was present in the JSON). Thus ***parseJson()*** itself returns a Promise, and we can call a **further *then()*** on the result of ***parseJson()*** i.e.

```
httpGet(url).then(parseJson).then(showJsonResults);
```

Because ***showJsonResults()*** will be the resolve function of the Promise returned by ***parseJson()***, and we pass the parsed JSON to this resolve function, it follows that ***showJsonResults()*** will received parsed JSON as (typically) a JavaScript array of objects, and we can extract the data in the usual way:

```
function showJsonResults(data)
{
    var html = "";
```

```
    for(var i=0; i<data.length; i++)
    {
        html = html + `Depart ${data[i].depart} Arrive
    }
    document.getElementById("results").innerHTML = htm
}
```

Exercise 1

Make a copy of your AJAX search script from Topic 7 (the version with just a plain search, no DOM) and change it so that the AJAX is performed using promises.

New style AJAX requests: the fetch API

Above we saw how to do AJAX requests using promises. However, standards-compliant browsers (such as Firefox and Chrome) now have a ***built in*** API - the fetch API- which uses promises. You can use the fetch API without having to write an AJAX promise yourself. Here is an example, in which we are searching for flights to a particular destination on a particular date;

```
function ajaxrequest() {
    var a = document.getElementById("destination").valu
    var b = document.getElementById("date").value;

    fetch('/flights.php?destination='+a+'&date='+b).the
        then(showJSONResults).catch(ajaxerror);
}


function ajaxresponse(response) {
    return response.json();
}

function ajaxerror(code) {
    alert('error; ' + code);
}

function showJSONResults(jsonData) {
    var html = "";
    for(var i=0; i<jsonData.length; i++) {
        html = html + `Depart ${jsonData[i].depart} Arr
    }
    document.getElementById("response").innerHTML = htr
}
```

```
}
```

Note how we use a promise chain with:

```
fetch('/flights.php?destination='+a+'&date='+b).then(a
        then(parseJSON).catch(ajaxerror);
```

fetch() contacts the given URL. We ***then*** call our ***ajaxresponse()*** function to handle the response. This works in a different way to the standard AJAX callback from "classic" AJAX. Rather than being passed an XMLHTTP object, it takes a ***response object*** as a parameter. This has a couple of interesting methods:

- ***json()***, which returns a Promise which passes parsed JSON to its resolve function.
- ***text()***, which returns a Promise which passes text to its resolve function.

Because each of these return Promises, we can chain our AJAX callback function to further functions which interpret the response, for example a function to display our JSON results (***showJSONResults()*** here).

Exercise 2

Make a further copy of your AJAX exercise (Topic 7) and change it to use the fetch API.

References

A number of articles were used to draw up these notes, which are also useful further reading. These include:

- Mozilla article on promises
- Google article on promises
- David Walsh's article
- David Walsh's article on fetch