JavaScript Promises

Learning Objectives

- Understand what promises are
- Understand the advantages of promises over callbacks
- Understand how to consume promises with `async`/`await`
- Understand how to consume promises with `then`

Material

Async the Easier Way

Have a look at the code below:

```
console.log("Down")
    setTimeout(() => {
        console.log("the")
        setTimeout(() => {
            console.log("Rabbit")
            setTimeout(() => {
                console.log("Hole")
            }, 3000);
        }, 2000);
    }, 1000);
```

What do you think of it - is it easy to read?

What about this:

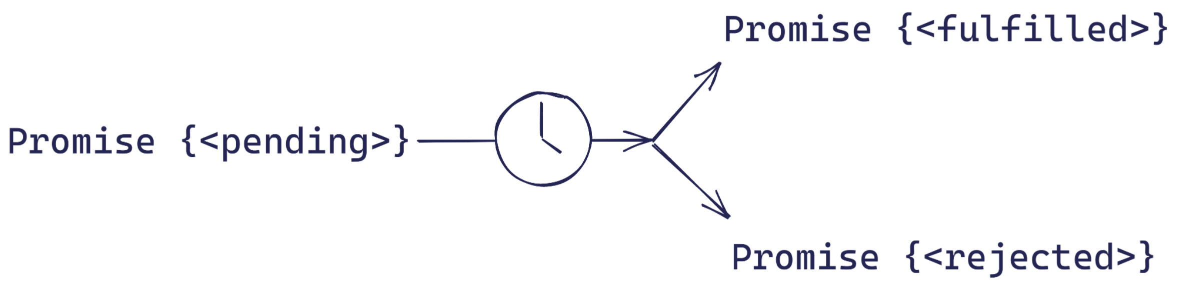
```
    console.log("Down");
    await pause(1000);
    console.log("the");
    await pause(2000);
    console.log("Rabbit");
    await pause(3000);
    console.log("Hole");
```

You might not yet know what the `await` keyword does but I think you'll agree that this second version is easier to follow. Unlike the previous example, the code reads as a top-to-bottom list of instructions, rather than a tangled mess of nested callbacks.

The 2nd example has been simplified by using promises.

Introducing Promises

A promise is an object which represents an asynchronous task. A promise begins in the pending state and, once the asynchronous task is finished, transitions to either fulfilled or rejected. In the example above, the pause function (definition not shown) is returning a promise.



A place you'll likely use promises is in the fetch Web API. A call to `fetch(url)` will make an HTTP GET request to the given url (the type of request your browser makes under the hood when you click a link). Network requests take a long time, however, so the fetch function returns a promise which represents the request.

```
const fetchRequest = fetch('https://www.multiverse.io/en-GB');

console.log(fetchRequest);
```

If we ran this, the logs would show us that fetchRequest is a pending promise:

```
>> Promise {<pending>}
```

Async + Await

If we want to use the data returned in the network response, we first need to wait for the promise to resolve. To do this, we can use the `await` keyword, however this can **only be used in functions labelled `async`**.

```
async function getMultiversePage() {
    const data = await fetch('https://www.multiverse.io/en-GB');
    // do stuff with the data
}
```

Here, `data` is the actual information we asked for in the network request. The `await` keyword does 2 things:

- it **pauses the code inside the async function** until the promise has finished pending
- it gives us the unwrapped value of the promise

Async functions always return a promise. Take the code below:

```
async function double(x) {
    return 2 * x
}

console.log(double(10))
```

Even though there is no asynchronous logic, so we aren't having to wait for anything, the output is still a promise:

```
>> Promise { 20 }
```

Consuming with `then`

An alternative to async/await is to use a promise's `then` method:

```
fetch(`https://www.multiverse.io/en-GB`)
    .then(data => {
        // do stuff with the data
    })
```

Like await, `then` waits for the promise to finish. However, when using `then`, we have to supply it with a callback which it calls once the promise is finished pending, passing in the unwrapped value (called `data` here).

The advantage of `then` is that it can be used outside of async functions; however, pay attention to the order of execution:

```
const myPromise = pause(1000); // resolves after 1s
console.log(1)
myPromise.then(() => {
    console.log(2)
});
console.log(3)
>> 1
>> 3
>> 2
```

Note that the "3" was printed before the "2". Javascript will finish executing the code after the `then` and come back to it once the promise is resolved and the call stack is clear.

Although `then` uses a callback style, we can chain `then`s together to avoid the messy nesting we saw earlier:

```
fetch('http://example.com/multiverse.json')
  .then(response => response.json())
  .then(data => console.log(data));
```

Error Handling

Promises are often used when handling operations such as reading from a computer's file system or making network requests. These types of operations are prone to fail so we should always include error handling. Async/await errors can be handled with a normal try/catch block:

```
async function getMultiversePage() {
    try {
        const data = await fetch("https://www.multiverse.io/en-GB");
        // do stuff with the data
    } catch (err) {
        // handle error
    }
}
```

Errors produced when using then can be caught using a promise's catch method:

```
fetch("https://www.multiverse.io/en-GB")
  .then(response => response.json())
  .catch(err => /* handle error*/)
```

Core Assignments

Promise Quiz

Clone this repository.

The repository consists of a series of JavaScript files to test your understanding of promises. The quiz uses the in-built Node file-system module. A call to:

```
readFile("./animal.txt", { encoding: "utf-8" })
```

tells node to load `animal.txt` into memory as a string. The process of reading and writing to a file on disk (storage) is very slow compared to dealing with data in RAM (memory) - such as the variables you create in your code - so Node's `readFile` function returns a **promise** so as not to block execution whilst the file loads.

For each of the questions, predict what the output to the console will be. Once you've made your prediction, run the file with node and see what the answer is. If you predicted incorrectly, try and work out where the gap in your understanding is.

Extension Assignments

Print With Delay

Create a function `printWithDelay(arr, delay)` which prints each element of an array, pausing for the specified delay after each console.log. You will find the `pause` function below useful, which returns a promise resolving after the given amount of time.

```
function pause(time) {
    return new Promise(resolve => {
        setTimeout(resolve, time);
    })
}
```

Additional Resources

Promises - the Eggs of JavaScript