5-6. Promises

We have up until now seen synchronous operations, like functions returning a value. 5-6-1

Something being synchronous means that it is blocking the code until it returns a final value.

Before we dig into promises, let's make sure we mean the same thing when we're talking about asynchronicity. 5-6-2

Asynchronicity 5-6-3

An asynchronous operation means that something operates independently. In short, instead of blocking (like a synchronous call), an asynchronous function would return immediately.

We could for example fire off a number of HTTP requests independently, and not wait for the previous one to finish. 5-6-4

Handling asynchronous code in JavaScript used to entail relying on callbacks, and would frequently result in callback hells. 5-6-5

```
callEndPoint("api/getidbyusername/hotcakes", function(result) {
    callEndPoint("api/getfollowersbyid" + result.userId, function(result) {
        callEndPoint("api/someothercall" + result.followers, function() {
            callEndPoint("api/someothercall" + result, function(result) {
                //...might go on, but we'll stop here
            });
        });
    });
});
```

Promises 5-6-6

The addition of Promises greatly simplified this by introducing a more organized alternative to callbacks.

Short description by MDN: 5-6-7

The Promise object is used for asynchronous computations.

A Promise represents a value which may be available now, or in the future, or never.

A Promise can be: 5-6-8

- fulfilled - when everything worked as intended
- rejected - when an error occurred during the task
- pending - when it's waiting to be fulfilled or rejected

Creating a Promise: 5-6-9

```
var promise = new Promise((resolve, reject) => {
    // Perform a task and then...

    if(/* things went well */) {
        resolve('Hooray!');
    }
    else {
        reject('Ouch');
        //Production code should return an appropriate error.
    }
});
```

Make use of a Promise: 5-6-10

```
promise.then(result => {
    console.log(result); // -> Hooray!
}).catch(err => {
    console.log(err); // -> Ouch!
});
```

`resolve` and `reject` can be used directly from the Promise object: 5-6-11

```
Promise.resolve(['JavaScript', 'Promise']).then(labels => {
    labels.forEach(label => console.log(label));
    // -> JavaScript
    // -> Promise
});
```

Promises are a core construct in **ES2015+**, and are used to represent ongoing asynchronous work: 5-6-12

```
let p = api.asyncOp()
    .then(() => anotherAsyncOp())
    .then(() => thirdAsyncOp());
```

Promises allow us to **break nesting** and express things in a slightly more readable manner.

Promises can be chained, using `.then` to take the information returned by the promise and handling the information whereafter a new `.then` can be chained. 5-6-13

5-6-14

```
new Promise(function(resolve, reject) {
        // A mock async action using setTimeout
        setTimeout(function() { resolve(10); }, 3000);
})
.then(function(num) { console.log('first then: ', num); return num * 2; })
.then(function(num) { console.log('second then: ', num); return num * 2; })
.then(function(num) { console.log('last then: ', num);});

// From the console:
// first then:  10
// second then:  20
// last then:  40
```

This would be a disaster using callbacks.

NOTE: All functions must return a Promise to make this work.

A Promise is a an object returned from an asynchronous operation, completed or failed. 5-6-15

The object will be available sometime in the future.

The Promise object takes an executor as a parameter which is a function that is passed with the arguments `resolve` and `reject`. 5-6-16

```
new Promise((resolve, reject) => {
    if(/*It went good*/)
        resolve(...);
    else
        reject(...);
    });
});
```

As said, instead of passing the function with the resolve and reject parameters we can also call resolve/reject directly on the Promise object. 5-6-17

Resolving a Promise 5-6-18

You can resolve a Promise using `Promise.resolve(value);` which will return a Promise object that is resolved with the given value.

```
Promise.resolve(value);
```

Rejecting a Promise 5-6-19

You can reject a Promise using `Promise.reject(reason);` which returns a Promise object that is rejected with the given reason.

```
Promise.reject('My reason to reject this Promise');
```

Thenables 5-6-20

Thenable is an object that returns a promise which a `then` function can be chained to.

```
getUserInformation(42)
    .then(info => {
        // Handle the returned information...
    });
```

The above example shows us that `then` can be fed with the returned value from the previous operation.

We can also chain `then` without any return values, like: 5-6-21

```
doSomething()
    .then(() => { ... });
```

Parallelise Promises 5-6-22

We can also parallelise promises using `Promise.all()` instead of doing them in sequences.

`Promise.all()` takes an array of promises, and will resolve when all of them are done.

```
Promise.all([promise1, promise2, promise3]).then(values => {
  console.log(values);
});
```

Error catching

5-6-23

If at any point a process fails though out the composed chain the `catch` will be ready to take care of the error.

```
getUserInformation(42)
    .then(info => {
        // Handle the returned information...
    })
    .catch(err => {
        // Handle the error case...
    });
```

This is a catch-all exception handler, shortcut for calling `.then(null, handler)` on a promise.

Async/await

5-6-24

The async/await keywords can be used to transform a regular function into a Promise, and pause the execution of an async function.

Using `async/await` we lose the `.then` chaining.

5-6-25

We get to write asynchronous code with a synchronous flow.

5-6-26

```
// Promise approach

function getJSON(){
    return new Promise(resolve => {
        service.get('https://example.com/misc/files/test.json')
            .then( function(json) {

                // The data from the request is available in a .then block
                // We return the result using resolve.
                resolve(json);
            });
    });
}

getJSON().then(json => {/* handle result */})
```

```
// Async/Await approach

// The async keyword will automatically create a new Promise and return it.
async function getJSONAsync(){

    // The await keyword saves us from having to write a .then() block.
    let json = await service.get('https://example.com/misc/files/test.json');

    // The result of the GET request is available in the json variable.
    // We return it just like in a regular synchronous function.
    return json;
}

getJSONAsync().then(json => {/* handle result */})
```