Chapter 9 - Callbacks, promises & async/await

Asynchronous actions are the actions that we initiate now and they finish later. eg. setTimeout
Synchronous actions are the actions that initiate and finish one-by-one

Callback functions

A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete an action.

Here is an example of a callback:

```
function loadScript (src, callback) {
    let script = document.createElement ('script')
    script.src = src
    script.onload = () => callback (script)
    document.head.append (script)
}
```

Now we can do something like this:

```
loadScript ('https://cdn.harry.com', (script) => {
    alert ('script is loaded')
    alert (script.src)
});
```

This is called "callback-based" style of async programming. A function that does something asynchronously should provide a callback argument where we put the function to run after it's complete

Handling errors
We can handle callback errors by supplying error argument like this:

```
function loadScript (src, callback) {
  ...
  ...
  script.onload = () => callback (null, script);
  script.onerror = () => callback (new Error ('failed'));
  ...
}
```

Then inside of loadScript call:

```
loadScript ('cdn/harry', function (error, script) {
  ...
  if (error) {
    // handle error
    }
  else {
    // script loaded
    }
  });
```

Pyramid of Doom

When we have callback inside callbacks, the code gets difficult to manage

```
loadScript((...) {

    loadScript ..        ← Pyramid of Doom

        loadScript ...

            loadScript
                ...
```

As calls become more nested, the code becomes deeper and increasingly more difficult to manage, especially if we have real code instead of ...

This is sometimes called "callback hell" or "pyramid of doom"

The "pyramid" of these calls grows towards the right with every asynchronous action. Soon it spirals out of control. So this way of coding isn't very good!

Introduction to Promises

The solution to the callback hell is promises. A promise is a "promise of code execution". The code either executes or fails, in both the cases the subscriber will be notified.

The syntax of a Promise looks like this:

```
let promise = new Promise( function (resolve, reject) {
    // executor
});
```

(predefined in JS engine)

resolve and reject are two callbacks provided by javascript itself. They are called like this:

resolve (value) → If the job is finished successfully
reject (error) → If the job fails

The promise object returned by the new Promise constructor has these properties

1. State: Initially pending, then changes to either "fulfilled" when resolve is called or "rejected" when reject is called

2. result: Initially undefined, then changes to value if resolved (resolve(value)) or error when rejected (reject(error))

Consumers: then & catch
The consuming code can receive the final result of a promise through then & catch

The most fundamental one is then

```
promise.then( function (result) { /* handle */ },
              function (error) { /* handle error */ }
            );
```

If we are interested only in successful completions, we can provide only one function argument to .then():

```
let promise = new Promise (resolve => {
    setTimeout (() => resolve ("done"), 1000);
    });

promise.then (alert);
```

If we are interested only in errors, we can use null as the first argument: .then(null, f) or we can use .catch:

```
promise.catch (alert)
```

promise.finally (() => {}) is used to perform general cleanups

Quick Quiz: Rewrite the loadScript function we wrote in the beginning of this chapter using promises.

Promises Chaining

We can chain promises and make them pass the resolved values to one another like this

```
p.then (function (result) => {          // p is a promise
    alert (result); return 2;
    }).then ...
```

The idea is to pass the result through the chain of .then handlers.

Here is the flow of execution

1. The initial promise resolves in 1 seconds (Assumption)
2. The next .then() handler is then called, which returns a new promise (resolved with 2 value)
3. The next .then() gets the result of previous one and this keeps on going

Every call to .then() returns a new promise whose value is passed to the next one and so on. We can even create custom promises inside .then()

Attaching multiple handlers

We can attach multiple handlers to one promise. They dont pass the result to each other; instead they process it independently.

Let p is a promise

p.then(handler1) →

p.then(handler2) → Runs Independently

p.then(handler3) →

Promise API

There are 6 static methodes of Promise class:

1) Promise.all(promises) → Waits for all promises to resolve and returns the array of their results. If any one fails, it becomes the error & all other results are ignored

2) Promise.allSettled(promises) → Waits for all the promises to settle and returns their results as an array of objects with status and value.

3) Promise.race(promises) → Waits for the first promise to settle and its result/error becomes the outcome.

4) Promise.any(promises) → Waits for the first promise to fulfill (& not rejected), and its result becomes the outcome. Throws AggregateError if all the promises are rejected.

5) Promise.resolve(value) → Makes a resolved promise with the given value.

6) Promise.reject(error) → Makes a rejected promise with the given error.

Quick Quiz : Try all these promise APIs on your custom promises.

Async / Await

There is a special syntax to work with promises in javascript

A function can be made async by using async keyword like this :

```
async function harry(){
    return 7;
}
```

An async function always returns a promise. Other values are wrapped in a promise automatically

We can do something like this :

```
harry().then(alert)
```

So, async ensures that the function returns a promise and wraps non promises in it.

The await keyword

There is another keyword called await that works only inside async functions

```
let value = await promise;
```

The await keyword makes javascript wait until the promise settles and returns its value.

Its just a more elegant syntax of getting the promise result than promise.then + its easier to read & write

Error Handling

We all make mistakes. Also sometimes our script can have errors. Usually a program halts when an error occurs.

The try ... catch syntax allows us to catch errors so that the script instead of dying can do some thing more reasonable

The try ... catch Syntax

The try catch syntax has two main blocks: try and then catch

```
try {
  // try the code

} catch (err) {
  // error handling
  }
```

⇒ The err variable contains an error object

It works like this

1. first the code in try is executed
2. If there is no error, catch is ignored else catch is executed

try catch works synchronously
If an exception happens in scheduled code, like in setTimeout, then try... catch wont catch it :

```
try {
    setTimeout ( function () {
            // error code
        }
catch ...
```

→ Script dies and catch wont work

Thats because the function itself is executed later, when the engine has already left the try ... catch construct.

The error object
For all the built in errors, the error object has two main properties :

```
try {
  hey; // error variable not defined
  } catch (err) {
        alert (err.name)
        alert (err.message)
        alert (err.stack)
    }
```

Throwing Custom Error

We can throw our own error by using the throw syntax

```
if (age > 180) {
    throw new Error("Invalid Age")
    ...
```

We can also throw a particular error by using the built in constructor for standard errors:

```
let error = new SyntaxError(message)
            or
            new ReferenceError(message)
```

The finally clause

The try ... catch construct may have one more code clause: finally

If it exists it runs in all cases:

after try if there were no errors
after catch if there were errors

If there is a return in try, finally is executed just before the control returns to the outer code.