Module 3 – Vanilla JavaScript: Promises, Async and Callbacks

# Lab Objectives

To utilise promises within JavaScript

See Asynchronous actions in practice

Provide real world examples of callback use

Promises

Promises are a value which represent that something will happen. They allow you to execute one action only when another has finished, you can also use a value output by the previous action, and you can chain these together.

Promises can seem confusing at first, but put simply, if there is an action you may need to wait for such as a connection to a server, or a download that may need to complete, the best way to execute code immediately after this action has finished is to wrap the action in a promise. The initial action goes in the initialisation of the promise, featuring a resole and reject statement, and the code to be executed after this action has been complete is placed in a .then function on the variable the promise is assigned to.

A commented example of this is shown below:

// Initialisation of a new promise

let newPromise = new Promise((resolve, reject) => {

// Action to be performed goes in here

if (1+1 === 2) {

resolve('If statement is true');

} else {

reject('If statement is false');

}

})

// using the promise

// code to be executed if the promise resolves goes in the then function

// Code to be executed if the promise is rejected goes in the catch function

newPromise.then((message) => {

console.log(message);

}).catch((message) => {

console.log(message);

});

Task: write down what will happen when the promise resolves successfully, and if it gets rejected

Task: think about and write a much more complex promise, which returns an object used in the then statement

Promises are an incredibly easy way to make code asynchronous, they ensure that code is executed exactly when it needs to be, and that all other code can continue executing without being held up by things such as http requests or libraries that take a large amount of time to load.

1. **Async**

Async functions are an extension of promises, they allow a function to be executed alongside the main body of code, rather than within it, which is incredibly important when there may be a delay in the actions it is performing. A further introduction is the await keyword, which will pause execution of the async code until a promise has resolved.

An example is shown below with a call to an example API. The code prints out

to the console and in the background the network request is made without

pausing the console printouts:

async function googleCall() {

console.log('request started');

let googleResponse = await fetch('https://jsonplaceholder.typicode.com/todos/1');

return googleResponse

}

console.log(1);

console.log(2);

googleCall().then((response) => response.json())

.then((json) => console.log(json));

console.log(3);

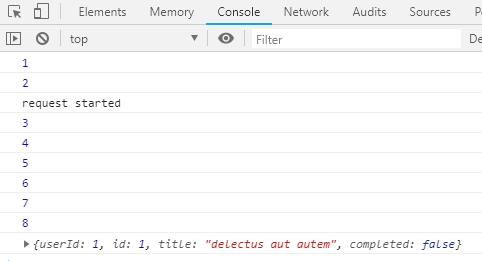
console.log(4);

console.log(5);

console.log(6);

console.log(7);

console.log(8);



You can see above that the console printed ‘request started’ at the correct point in time, as this is when the function was called, however the resulting JSON data from the request did not print out until all other code had finished running. This is because of the amount of time it took for the HTTP request to complete, and for the request to then be transformed into JSON by the json function.

**Task:** Write your own asynchronous function similar to the one above making the request to a different test API and use the json returned, passing it into other functions or printing individual items to the console

1. **Callbacks**

Callbacks are a way to make sure functions are executed in the correct order. They are an older method primarily used before the introduction of promises, however it is still useful to understand how they work in case you work with any external JavaScript libraries that still use this method.

The idea behind a callback is that instead of calling two functions directly after each other, meaning that if one function has a delay the other will be executed first, we pass a the second function as an argument to the first, and only begin executing this after the first function has completed.

**Task:** Explain the difference between the following two pieces of code

function functionOne() {

setTimeout(() => {

console.log('Function one output');

}), 500

}

function functionTwo() {

console.log('Function two output');

}

functionOne();

functionTwo();

//------------------------------------------------------

function functionOne(callback) {

setTimeout(() => {

console.log('Function one complete output');

callback();

}, 500);

}

function functionTwo() {

console.log('Function two output');

}

functionOne(functionTwo);

**Task:** now you’ve seen an example try creating a callback yourself, including a delayed function such as a http request and a separate function that does something with the data returned from this