JavaScript 1 - Module 5

In this module we will be looking at the following topics:

- Asynchronous JavaScript
- Communicating with APIs
- Postman
- CORS
- JWT

Asynchronous Code

In this lesson we will cover the basics of working with asynchronous code in JavaScript. We will explore the `Promise` object and the `async` and `await` keywords.

JavaScript Timing Events

JavaScript can be executed in time-intervals.

This is called **timing events**.

Timing Events

The `window` object allows execution of code at specified time intervals. The two key methods to use for timing events with JavaScript are:

- `setTimeout(function, milliseconds)`
  Executes a function, after waiting a specified number of milliseconds.
- `setInterval(function, milliseconds)`
  Same as setTimeout(), but repeats the execution of the function continuously.

> The `setTimeout()` and `setInterval()` are both methods of the HTML DOM Window object.

The `setTimeout()` Method

```
window.setTimeout(function, milliseconds);
```

The `window.setTimeout()` method can be written without the window prefix.

- The first parameter is a **function** to be executed.
- The second parameter indicates the number of milliseconds before execution.

> There are 1000 milliseconds in one second.

`setTimeout()` Example

Click a button. Wait 3 seconds, and the page will alert "Hello":

```
<button>Try it</button>
```

```
function myAnnoyingFunction() {
  alert('Hello');
}

const btn = document.querySelector ("button");
btn.addEventListener("click", function() {
  setTimeout(myAnnoyingFunction, 3000);
});
```

Codepen example

How to Stop the Execution?

The `clearTimeout()` method stops the execution of the function specified in setTimeout().

```
window.clearTimeout(timeoutVariable)
```

The `window.clearTimeout()` method can be written without the window prefix.

The `clearTimeout()` method uses the variable returned from `setTimeout()`:

```
myVar = setTimeout(function, milliseconds);
clearTimeout(myVar);
```

If the function has not already been executed, you can stop the execution by calling the clearTimeout() method:

```
function myAnnoyingFunction() {
  alert('Hello');
}

let timer; // Variable to handle the timing event

const start = document.querySelector ("button#try");
start.addEventListener("click", function() {
  timer = setTimeout(myAnnoyingFunction, 3000);
});

const stop = document.querySelector ("button#stop");
stop.addEventListener("click", function() {
    clearTimeout(timer);
    console.log("Thank you");
});
```

Codepen example

The setInterval() Method

The `setInterval()` method repeats a given function at every given time-interval.

```
window.setInterval(function, milliseconds);
```

The `window.setInterval()` method can be written without the window prefix.

- The first parameter is the function to be executed.
- The second parameter indicates the length of the time-interval between each execution.

setIntervall() Example

This example executes a function called "myTimer" once every second (like a digital watch).

```
<div id="demo"> </div>
```

```
let timer = setInterval(myTimer, 1000);
const out = document.getElementById("demo");

function myTimer() {
    let d = new Date();
    out.innerHTML = d.toLocaleTimeString();
}
```

Codepen example

How to Stop the Execution?

The `clearInterval()` method stops the executions of the function specified in the `setInterval()` method.

```
window.clearInterval(timerVariable)
```

The `window.clearInterval()` method can be written without the window prefix.

The `clearInterval()` method uses the variable returned from `setInterval()`:

```
myVar = setInterval(function, milliseconds);
clearInterval(myVar);
```

Same example as above, but we have added a "Stop"-button:

```
<div id="demo"> </div>
<button>Stop it</button>
```

```
let timer = setInterval(myTimer, 1000);
const out = document.querySelector("div#demo");

function myTimer() {
    let d = new Date();
    out.innerHTML = d.toLocaleTimeString();
}

const btn = document.querySelector("button");
btn.addEventListener("click", () => clearInterval(timer));
```

Codepen example

Synchronous and Asynchronous Code

Consider the following code:

```
console.log("Start!");
const name = "Lasse";
const greeting = `Hello, my name is ${name}!`;
console.log(greeting);
console.log("End!");
// "Start"
// "Hello, my name is Lasse!"
// "End"
```

This code:

- Declares a string called name.
- Declares another string called greeting, which uses name.
- Outputs the greeting to the JavaScript console.

We should note here that the browser effectively steps through the program one line at a time, in the order we wrote it. At each point, the browser waits for the line to finish its work before going on to the next line. It has to do this because each line depends on the work done in the preceding lines.

That makes this **a synchronous program**.

It would still be synchronous even if we called a separate function, like this:

```
function makeGreeting(name) {
  return `Hello, my name is ${name}!`;
}

console.log("Start!");
const name = "Lasse";
const greeting = makeGreeting(name);
console.log(greeting);
console.log("End!");
// "Start"
// "Hello, my name is Lasse!"
// "End"
```

Here, `makeGreeting()` is a **synchronous function** because the caller has to wait for the function to finish its work and return a value before the caller can continue.

```
console.log('Start');

setTimeout(() => {
    const name = "Lasse";
    const greeting = `Hello, my name is ${name}!`;
    console.log(greeting);
}, 2000);

console.log('End');
// "Start"
// "End"
// "Hello, my name is Lasse!"
```

Here we use `setTimeout()` to delay the execution of the greeting, making this **an asynchronous program**.

Callbacks

Callbacks

An event handler is a particular type of callback. **A callback is just a function that's passed into another function**, with the expectation that the callback will be called at the appropriate time. As we just saw, callbacks used to be the main way asynchronous functions were implemented in JavaScript.

However, callback-based code can get hard to understand when the callback itself has to call functions that accept a callback. This is a common situation if you need to perform some operation that breaks down into a series of asynchronous functions. For example, consider the following:

```
function doStep1(init) {
  return init + 1;
}

function doStep2(init) {
  return init + 2;
}

function doStep3(init) {
  return init + 3;
}

function doOperation() {
  let result = 0;
  result = doStep1(result);
  result = doStep2(result);
  result = doStep3(result);
  console.log(`result: ${result}`);
}

doOperation();
```

Here we have a single operation that's split into three steps, where each step depends on the last step.

In our example, the first step adds 1 to the input, the second adds 2, and the third adds 3.

Starting with an input of 0, the end result is 6 (0 + 1 + 2 + 3).

As a synchronous program, this is very straightforward. But what if we implemented the steps using callbacks?

```
function doStep1(init, callback) {
  const result = init + 1;
  callback(result);
}

function doStep2(init, callback) {
  const result = init + 2;
  callback(result);
}

function doStep3(init, callback) {
  const result = init + 3;
  callback(result);
}

function doOperation() {
  doStep1(0, (result1) => {
    doStep2(result1, (result2) => {
      doStep3(result2, (result3) => {
        console.log(`result: ${result3}`);
      });
    });
  });
}

doOperation(); // result: 6
```

Because we have to call callbacks inside callbacks, we get a deeply nested `doOperation()` function, which is much harder to read and debug. This is sometimes called "**callback hell**" or the "**pyramid of doom**" (because the indentation looks like a pyramid on its side).

When we nest callbacks like this, it can also get very hard to handle errors: often you have to handle errors at each level of the "pyramid", instead of having error handling only once at the top level.

For these reasons, most modern asynchronous APIs don't use callbacks. Instead, the foundation of asynchronous programming in JavaScript is the **Promise**.

Promises

The `Promise` object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

Using Promises

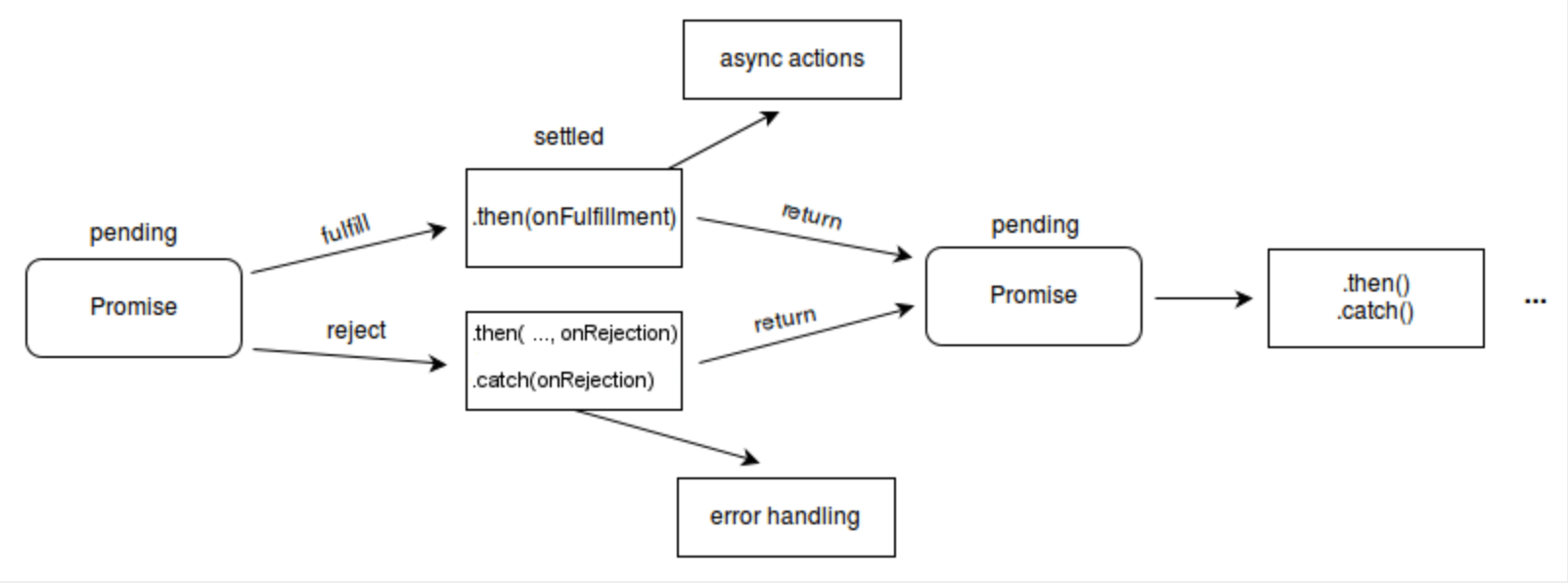
A `Promise` is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.

A `Promise` is in one of these states:

- *pending*: initial state, neither fulfilled nor rejected.
- *fulfilled*: meaning that the operation was completed successfully.
- *rejected*: meaning that the operation failed.

The eventual state of a pending promise can either be **fulfilled with a value** or **rejected with a reason** (error). When either of these options occur, the associated handlers queued up by a promise's then method are called. If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached.

A promise is said to be settled if it is either fulfilled or rejected, but not pending.



Chained Promises

The methods `Promise.prototype.then()`, `Promise.prototype.catch()`, and `Promise.prototype.finally()` are used to associate further action with a promise that becomes settled.

As the `Promise.prototype.then()` and `Promise.prototype.catch()` methods return promises, they can be chained.

The `.then()` method takes up to two arguments:

- the first argument is a **callback function** for the fulfilled case of the promise, and
- the second argument is a **callback function** for the rejected case.

Each `.then()` returns a newly generated promise object, which can optionally be used for chaining; for example:

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("foo");
  }, 300);
});

myPromise
  .then(handleFulfilledA, handleRejectedA)
  .then(handleFulfilledB, handleRejectedB)
  .then(handleFulfilledC, handleRejectedC);
```

Processing continues to the next link of the chain even when a `.then()` lacks a callback function that returns a Promise object. Therefore, a chain can safely omit every rejection callback function until the final `.catch()`.

Handling a rejected promise in each `.then()` has consequences further down the promise chain. Sometimes there is no choice, because an error must be handled immediately. In such cases we must throw an error of some type to maintain error state down the chain. On the other hand, in the absence of an immediate need, it is simpler to leave out error handling until a final `.catch()` statement. A `.catch()` is really just a `.then()` without a slot for a callback function for the case when the promise is fulfilled.

```
myPromise
  .then(handleFulfilledA)
  .then(handleFulfilledB)
  .then(handleFulfilledC)
  .catch(handleRejectedAny);
```

Using Arrow Function Expressions for the callback functions, implementation of the promise chain might look something like this:

```
myPromise
  .then((value) => `${value} and bar`)
  .then((value) => `${value} and bar again`)
  .then((value) => `${value} and again`)
  .then((value) => `${value} and again`)
  .then((value) => {
    console.log(value);
  })
  .catch((err) => {
    console.error(err);
  });
  .finally(/* Maintenance, eg. Stop loading spinner */)
```

Note: For faster execution, all synchronous actions should preferably be done within one handler, otherwise it would take several ticks to execute all handlers in sequence.

Basic Example

```
const myFirstPromise = new Promise((resolve, reject) => {
  // We call resolve(...) when what we were doing asynchronously was successful,
  // and reject(...) when it failed (not applicable here)
  // In this example, we use setTimeout(...) to simulate async code.
  // In reality, you will probably be using something like XHR or an HTML API.
  setTimeout(() => {
    resolve("Success!"); // Yay! Everything went well!
  }, 250);
});

myFirstPromise.then((successMessage) => {
  // successMessage is whatever we passed in the resolve(...) function above.
  // It doesn't have to be a string, but if it is only a succeed message,
  // it probably will be.
  console.log(`Yay! ${successMessage}`);
});
```

Codepen: myFirstPromise

Codepen: myFirstPromise with chaining

Promise.all()

The `Promise.all()` method takes an iterable of promises as an input, and returns a single `Promise` that resolves to an array of the results of the input promises.

This returned promise will fulfill when all of the input's promises have fulfilled, or if the input iterable contains no promises.

It rejects immediately upon any of the input promises rejecting or non-promises throwing an error, and will reject with this first rejection message / error.

Promise.all()

Example: Using Promise.all()

`Promise.all()` waits for all fulfillments (or the first rejection).

```
const p1 = Promise.resolve(3);
const p2 = 1337;
const p3 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("foo");
  }, 100);
});

Promise.all([p1, p2, p3]).then((values) => {
  console.log(values); // [3, 1337, "foo"]
});
```

Codepen

> Tip: Increase the timeout to see that it waits.

If the iterable contains non-promise values, they will be ignored, but still counted in the returned promise array value (if the promise is fulfilled):

```
// this will be counted as if the iterable passed is empty,
// so it gets fulfilled
const p = Promise.all([1, 2, 3]);
// this will be counted as if the iterable passed contains only
// the resolved promise with value "444", so it gets fulfilled
const p2 = Promise.all([1, 2, 3, Promise.resolve(444)]);
// this will be counted as if the iterable passed contains only
// the rejected promise with value "555", so it gets rejected
const p3 = Promise.all([1, 2, 3, Promise.reject(555)]);

// using setTimeout we can execute code after the queue is empty
setTimeout(() => {
  console.log(p);  // Promise { <state>: "fulfilled", <value>: Array[3] }
  console.log(p2); // Promise { <state>: "fulfilled", <value>: Array[4] }
  console.log(p3); // Promise { <state>: "rejected", <reason>: 555 }
});
```

Async Await

An **async function** is a function declared with the `async` keyword, and the `await` keyword is permitted within it.

The `async` and `await` keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to explicitly configure promise chains.

async function

How to use promises > async and await

```
function resolveAfter2Seconds() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('resolved');
    }, 2000);
  });
}

async function asyncCall() {
  console.log('calling');
  const result = await resolveAfter2Seconds();
  console.log(result); // expected output: "resolved"
}

asyncCall();
```

Codepen

try...catch

The `try...catch` statement is comprised of a `try` block and either a `catch` block, a `finally` block, or both. The code in the `try` block is executed first, and if it throws an exception, the code in the `catch` block will be executed. The code in the `finally` block will always be executed before control flow exits the entire construct.

```
try {
  nonExistentFunction();
} catch (error) {
  console.error({name: error.name, message: error.message});
} finally {
  console.log("Do some more stuff either way")
}
/* Expected output: [object Object]
{ "name": "ReferenceError", "message": "nonExistentFunction is not defined"}
"Do some more stuff either way"*/
```

Codepen

Let's take a look at a practical example of `async` `await` . In this example we will get data from an API, parse it as JSON and then return the data.

```
async function fetchProducts() {
  try {
    const url = 'https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json';
    // after this line, our function will wait for the `fetch()` call to be settled
    // the `fetch()` call will either return a Response or throw an error
    const response = await fetch(url);
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    // after this line, our function will wait for the `response.json()` call to be settled
    // the `response.json()` call will either return the parsed JSON object or throw an error
    const data = await response.json();

    // Console log out the name of the first product received
    console.log(data[0].name);
  }
  catch (error) {
    console.error(`Could not get products: ${error}`);
  }
}

fetchProducts(); // "baked beans"
```

Codepen

Demo: The Promise of Amiibos

Todos

Mollify

Read Synchronous and Asynchronous Code

Read Asynchronous Callbacks

Read Promises

Read Async Await

Read Race Conditions, and do the Lesson Task

Read setTimeout and setInterval