Async javascript

Writing this for MDN as a series of articles. Goal is to keep tone and audience throughout. That’s why I’m writing it as a single document.

Introduction: What and Why?

Synchronous code (baseline)

When we write Javascript we normally do it synchronously, every instruction is executed one at a time in the order they appear in the script. The script will finish once the last instruction is executed.

In the example below we log three items to the console.

```
console.log('1');
console.log('2');
console.log('3');
```

We will always get the same result: 1 2 3.

This works great for small pieces of code or scripts that don't produce output on the browser. But sooner rather than later you will want to work on larger scripts and you will find one of the first issues with Javascript: **It blocks rendering**.

Let's take the following HTML document.

```
<!DOCTYPE html>
<html lang='en'<html lang='en'>meta charset='UTF-8'>
    <meta name='viewport' content='width=device-wid<meta name='viewport' c
  </head>
  <body>

  </body>
</html>
</script>
```

And the following content in `myscript.js`:

```
// Capture the body tag as the third child of the document (zero based)
const content = document.documentElement.childNodes[2];
// Create an h2 element and assign it to the header constiable
const header = document.createElement('h2');
'h2'// Add content to the headerder.innerHTML = 'This was inserted';
// A'This was inserted'// Assign it as the first child of the body element
content.insertAdjacentElement('afterbegin', header);
```

The browser will load the page and when it finds the script it will load it and process it. Because the browser can not know ahead of time if the script will insert, remove or change content on the page it will pause rendering until the script is fully parsed and any changes to the page like inserting or removing nodes are done. The script can have any number of functions and interact with databases and other resources on your page and application.

This will happen for every script that you addd to the page; they will each load completely before moving onto the next. Working with smaller scripts may be fine but imagine if you're loading jQuery from a Content Delivery Network and then loading our script. Our HTML will look like this:

```
<!DOCTYPE html>
<html lang='en'<html lang='en'>meta charset='UTF-8'>
    <meta name='viewport' content='width=device-wid<meta name='viewport' c
    <</script>c='myscript.js'></script>
  </head>
  <body>
    <!-- cont</script><!-- content goes here -->
  </body>
</html>
```

Even with no content, this page will take longer to load as it now has to download jQuery before it can continue processing the page and its content, which may include other scripts.

Things will get slower when the code we write does heavy processing like database access or image manipulation.

The figure below gives you an example of what the execution order or stack

looks like for a synchronous blocking language like Javascript.

Before we jump in to asynchronous (async) code we'll look at two ways of writing synchronous code: Callbacks and try/catch blocks.

Callbacks

Callbacks are functions that are passed as parameters to other functions.

An example of a callback is the second parameter of an event listener. In the example below, clicking the button with id of `myButton` will trigger the function and produce the alert for the user.

```
myButton.addEventListener('click', function() {
  alert('You Clicked THE Button');
})
```

Another example is when we use `forEach` to loop through the items in an Array. In this case `forEach` will loop through the array

```
const gods = ['Apollo', 'Artemis', 'Ares', 'Zeus'];

gods.forEach(function (eachName, index){
  console.log(index + 1 + '. ' + eachName);
});
```

The two examples work because Javascript allows functions as parameters in other functions.

When we pass a callback function as an argument to another function, we are only passing the function definition. We are not executing the function in the parameter. The containing function can execute the callback anytime.

Note that the callback function is not executed immediately. It is "called back" (hence the name) somewhere inside the containing function's body synchronously.

Working with ‘this’

When the callback function uses the `this` object, we have to modify how we execute the callback function to preserve the this object context. Or else the this object will either point to the global window object (in the browser), if callback was passed to a global function. Or it will point to the object of the containing method.

```
const agentData = {
  id: 007,
  fullName: 'Not Set',
  // setUserName is a method on the agentData object
  setUserName: function (firstName, lastName)  {
    this.fullName = firstName + ' ' + lastName;
  }
}

function getUserInput(firstName, lastName, callback)  {
  if ((firstName.length > 3' '& (lastName.length > 3)) {
    callback (firstName, lastName);
  } else {
    console.log('data could not be saved.');
  }
}
```

When we create a new agent something unexpected happens. `agentData.fullName` returns the default value of Not Set but when we check for fullName in the window object (`window.fullName`) we get the name we expected.

```
getUserInput ('James', 'Bond', agentData.setUserName);
console.log (agentData.fullName);
console.log (window.fullName);
```

The problem has to do with `this`.

In most cases, the value of this is determined by how a function is called. It can’t be set by assignment during execution, and it may be different each time the function is called.

When we call a function directly, like we call getUserInput, the context is the root object (window in the case of the browser)

When a function is called as a method of an object, its this is set to the object the method is called on.

So how do we solve this problem?

Call and Apply

We can fix the problem with this using `call` or `apply`. These methods (available to all functions) are used to set the `this` object inside the function and to pass arguments to the functions.

Call takes the value to be used as the `this` object inside the function as the first parameter, and the remaining arguments to be passed to the function are passed as a comma separated list.

For `call` and `apply` to work we add an extra parameter that designates what objecgt we want to use to represent `this`, in this case we call it `callbackObj`.

The biggest difference is that, our new version of `getUserInput` uses the `call` method of the `callback` function with four parameters, including the new `callbackObj`.

```
const agentData = {
  id: 007,
  fullName: 'Not Set',
  // setUserName is a method on the agentData object
  setUserName: function (firstName, lastName)  {
    this.fullName = firstName + ' ' + lastName;
  }
}

function getUserInput(firstName, lastName, callback, callbackObj)  {
   if ((firstN' '.length > 3) && (lastName.length > 3)) {
    callback.call (callbackObj, firstName, lastName);
  } else {
    console.log('data could not be saved.');
  }
```

```
}
```

To verify that it works we add a new agent with our `getUserInput` function and, here is the key, we tell it that we want to use `agentData` as the object to represent this.

When we log the value of `agentData.fullName` we get the expected value. When we try to log the value of `window.fullName` we get undefined because `this` means `agentData` not the global window object.

```
getUserInput ('James', 'Bond', agentData.set'Bond'me, agentData);

console.log (agentData.fullName); // James Bond
console.log (window.fullName); // undefined
```

The Apply function's first parameter is also the value to be used as the `this` object inside the function, while the last parameter is an array of values to pass to the function.

The difference between `apply` and `call` is the signature of the method.

```
const agentData = {
  id: 007,
  fullName: 'Not Set',
  // setUserName is a method on the agentData object
  setUserName: function (firstName, lastName)  {
    this.fullName = firstName + ' ' + lastName;
  }
}

function getUserInput(firstName, lastName, callback, callbackObj)  {
  if ((firstNa' 'length > 3) && (lastName.length > 3)) {
  callback.apply (callbackObj, [firstName, lastName]);
  } else {
    console.log('data could not be saved.');
  }
}
```

The signature for `getUserInput` doesn't change when we use `apply` instead of `call`. The results are identical.

```
getUserInput ('James', 'Bond', agentData.set'Bond'me, agentData);

console.log (agentData.fullName); // James Bond
console.log (window.fullName); // Undefined
```

Try/Catch blocks

The second way to write synchronous code is to create `try/catch` blocks.

The idea behind `try/catch` blocks is that we need to run instructions in sequence but we must also do something if any of the commands fail.

We will use the following JSON for the example.

```
// data from the server
let json = '{"id":"007", "firstName":"James", "lastName": "Bond"}';
```

In the try/catch block below, we want to make sure that the data parses and do something if ther is an error.

In this example we just log a message and the error to console. In more complex examples the catch error may attempt connecting to the database again or ask the user to enter the data again if it wasn't complete or well formed.

```
try {
  // convert the text representation to JS object
  let user = JSON.parse(json);
  // Log the results to console
  console.log (user.id); // 007
  console.log (user.firstName);  // James
  console.log (user.lastName); // Bond
} catch(err) {
  console.log ('I\'m sorry Dave, I can\'t do that ');
  console.log (err.name);
```

```
  console.log (err.message);
}
'I\'m sorry Dave, I can\'t do that '
```

There is a third optional parameter, `finally`. It will happen regardless of whether the `try` block succeeds or the `catch` block is called. This gives the option of doing cleanup, closing database connections and doing other cleanup tasks your code needs to complete.

In the example below, the script will always log `All Done`, regardless if the JSON parsing was successful or not.

```
try {
  // convert the text representation to JS object
  let user = JSON.parse(json);
  // Log the results to console
  console.log (user.id); // 007
  console.log (user.firstName);  // James
  console.log (user.lastName); // Bond
} catch(err) {
  console.log ('I\'m sorry Dave, I can\'t do that ');
  console.log (err.name);
  console.log (err.message);
} finally {
  'I\'m sorry Dave, I can\'t do that '//This will always execute regardles
  console.log ('All Done');
}
```

Brief Introduction to Async Code

Async (short for asynchronous) code will execute without blocking rendering and returning a value when the code in the async block finishes.

Contrast the definition of async with the synchronous code we've been working so far where all statements are executed in the order they appear.

The example below uses both sync (synchronous) and async (asynchronous)

code to illustrate the difference. The console log statements outside of fetch will execute in the order they appear in the document.

```
console.log ('Starting');
fetch('https://s3-us-west-2.amazonaws/s3-us-west-2.amazonaws.com/spng') /
  .then((respons// 1// 1    if (response.ok) {
      console.log('It worked :)')
      return response.blob(); // 2
    } else {
      t'It worked :)''It worked :)'// 2('Network response was not ok.' );
    }
  })
  .then((myBlob) => {
    let objectURL = URL.createObjectURL(myBlob); // 3
    let myImage = document.cre'Network response was not ok.'// 3
    let myImage = document.createElement('img'); // 4
    myImage.src = objectURL; // 5r) => {
    console.log(
      'There has been a problem with your fetch operation: ',
      error.message
    );
  });
console.log ('All done!');
'There has been a problem with your fetch operation: '
```

The fetch function and it's associated chain uses the Promise API (discussed in more detail in a later document) to get a resource from the network and perform one or more tasks. It won't make the script wait until it's done but will continue working in the background until it completes.

The async nature of the Fetch API produces unexpected results. The sequence of messages logged to console:

1. Starting
2. All done!
3. It worked :)

The messages from both console.log calls outside fetch appear in their document order but the message signaling success doesn't appear until the chain started

with fetch completes.

Javascript is at its most basic a synchronous, blocking, single-threaded language; Only one operation can be in progress at a time.

Whether we want to run code synchronously or asynchronously will depend on what we're trying to do.

There are times when we want things to load and happen right away. The callback for a click event handler is a good example.

If we're running an expensive operation like querying a database and using the results to populate templates then we want to push this off the main thread and complete the task asynchronously.

Different ways of writing async javascript

Now that we've looked at synchronous and asynchronous code we'll look at different ways to write asynchronous code.

SetTimeout and SetInterval

SetTimeout and SetInterval provide ways to schedule tasks to run at a future point in time.

`setTimeout` allows you to schedule a task after a given interval.

`setInterval` lets you run a tasks periodically with a given interval between runs.

SetTimeout

setTimeout takes two parameters:

A string representing code to run and a number represting the time interval in miliseconds to wait before executing the code.

In the following example, the browser will wait two seconds before executing the anonymous function and presenting the alert message.

```
let myGreeting = setTimeout(function() {
  alert('Hello, Mr. Universe!');
}, 2000)
```

We're not required to write anonymous functions. The second version of the example uses `sayHi` as the name of the function. The rest of the code remains unchanged.

```
let myGreeting = setTimeout(function sayHi() {
  alert('Hello, Mr. Universe!');
}, 2000)
```

The code is rather messy. We can clean up the `setTimeout` call by taking the function outside the `setTimeout` call. The next iteration of our code defines `sayHi` first and then references the function by calling `sayHi` without parenthesis as the first parameter of `setTimeout`.

```
function sayHi() {
  alert('Hello Mr. Universe!');
}

let myGreeting = setTimeout(sayHi, 2000);
```

The last step in the demo is to pass parameters to the function we want to use in setTimeout.

This gets a little tricky.

First we configure the function to add the parameter and use it in the body of the function.

When we call `setTimeout` we pass the values for the function parameters as the third (and subsequent if there is more than one) parameters.

```
function sayHi(who) {
  alert('Hello ' + who + '!');
}

let myGreeting = setTimeout(sayHi, 2000, 'Mr. Universe');
```

All versions of the function will produce the same result... but they show different ways we can use setTimeout and the flexibility we have in writing the code.

setInterval

setTimeout works perfectly when we need to run the code once after a set period of time. But what happens when we need to run the code every x miliseconds?

That's where setInterval comes in. When we use this command, the code we attach to it will run every time the interval completes.

The example below creates a new date object and logs it to console. We then attach it to setInterval and execute it once every second. This will create the effect of a running clock that updates once per second.

```
function countTime() {
  let date = new Date();
  let time = date.toLocaleTimeString();
  document.getElementById('demo').innerHTML = time;
}

const createClock = setInterval(countTime, 1000);
```

ClearTimeout

This is less of an issue with `setTimeout` as it is with `setInterval` (discussed in later sections) but there may still be situations where you want to abort the execution of code inside a `setTimeout` call. For example, let's say that we set the timeout of a very expensive task.

```
function runExpensiveTask() {
  alert('Expensive Task has been completed!');
}

let myTaskRunner = setTimeout(runExpensiveTask, 30000);
```

And we want to stop it because we want to do something else in the page. To do it we call `clearTimeout` with the id of we assigned to `setTimeout` when we created it.

```
let forgetIt = clearTimeout(myTaskRunner);
```

clearInterval

With repetitive tasks like our clock example, we definitely want a way to stop the activity... otherwise we may end up getting errors when the browser can't complete any further versions of the task.

The stopTime() function uses `clearInterval` to clear the interval with the ID we created.

The example creates a button and attaches the stopTime function to the button's click event so when we click the button the interval will stop.

```
function stopTime() {
    clearInterval(createClock);
}

let myButton = document.getElementById('stopButton');
myButton.addEventListener('click', stopTime);
```

clearTimeout() and clearInterval() use the same list of entries to clear from. This means that you can use either method to remove a setTimeout or setInterval. For consistency, I use clearTimeout to clear setTimeout() entries and clearInterval to clear setInterval() entries.

Things to keep in mind

There are a few things to keep in mind when working with setTimeout and setInterval.

Recursive Timeout

There is another way we can use setTimeout: Use it recursively to call the same code repeatedly instead of using setInterval.

Compare the first example using a recursive setTimeout to run the code ever 1000 milliseonds.

```
let i = 1;

setTimeout(function run() {
  console.log(i);
  setTimeout(run, 100);
}, 100);
```

The second example uses setInterval to accomplish the same effect of running the code every 100 milliseconds.

```
let i = 1;

setInterval(function run() {
  console.log(i);
}, 100);
```

The difference between the two versions of the code is a subtle one.

Recursive setTimeout guarantees a delay between the executions; the code will run and then wait 100 milliseconds before it runs again. The 100 milliseconds will happen regardless of how long the code takes to run.

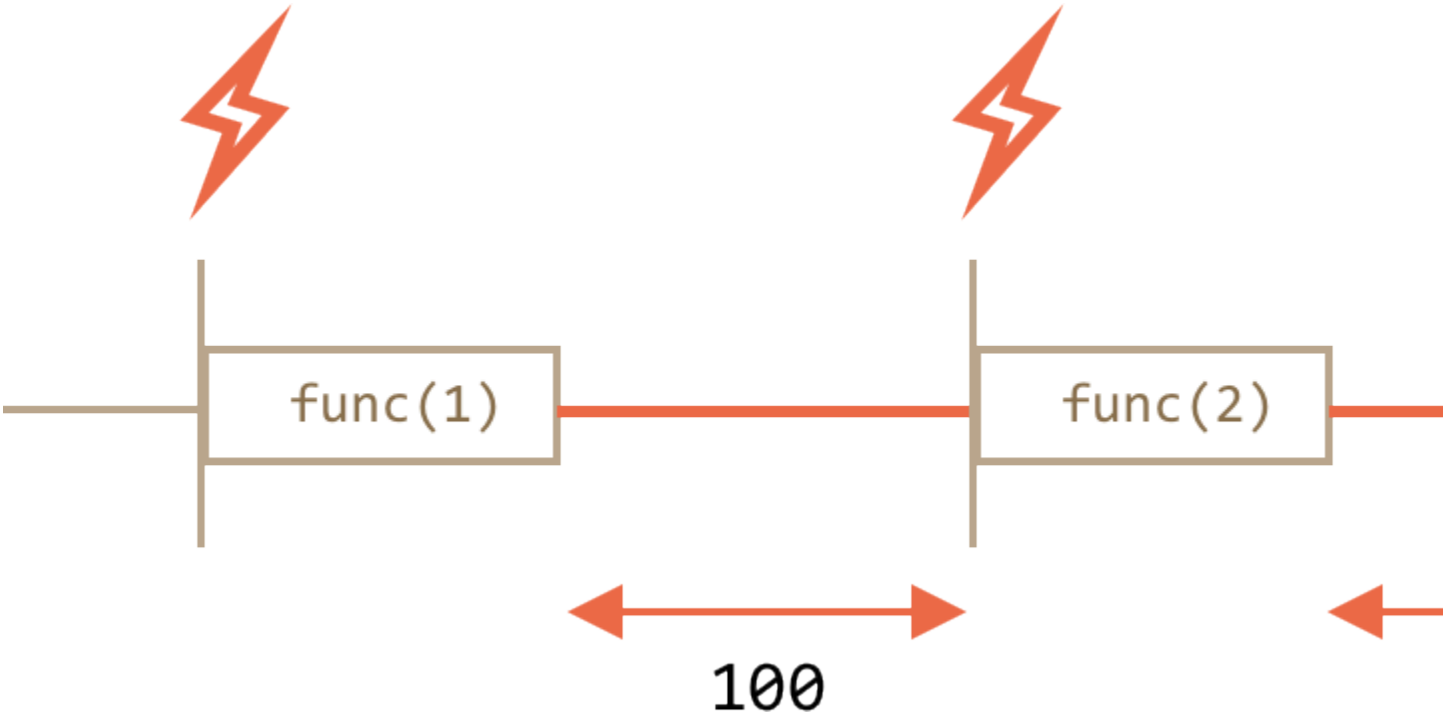

Figure 1: Using recursive setTimeout guarantees the interval will be the same between executions. Taken from Scheduling: setTimeout and setInterval

The example using setInterval does things differently. The interval we choose for setInterval includes the code we want to run in its execution. Let's say that the code takes 40 milliseconds to run, then the interval ends up being only 60 milliseconds.

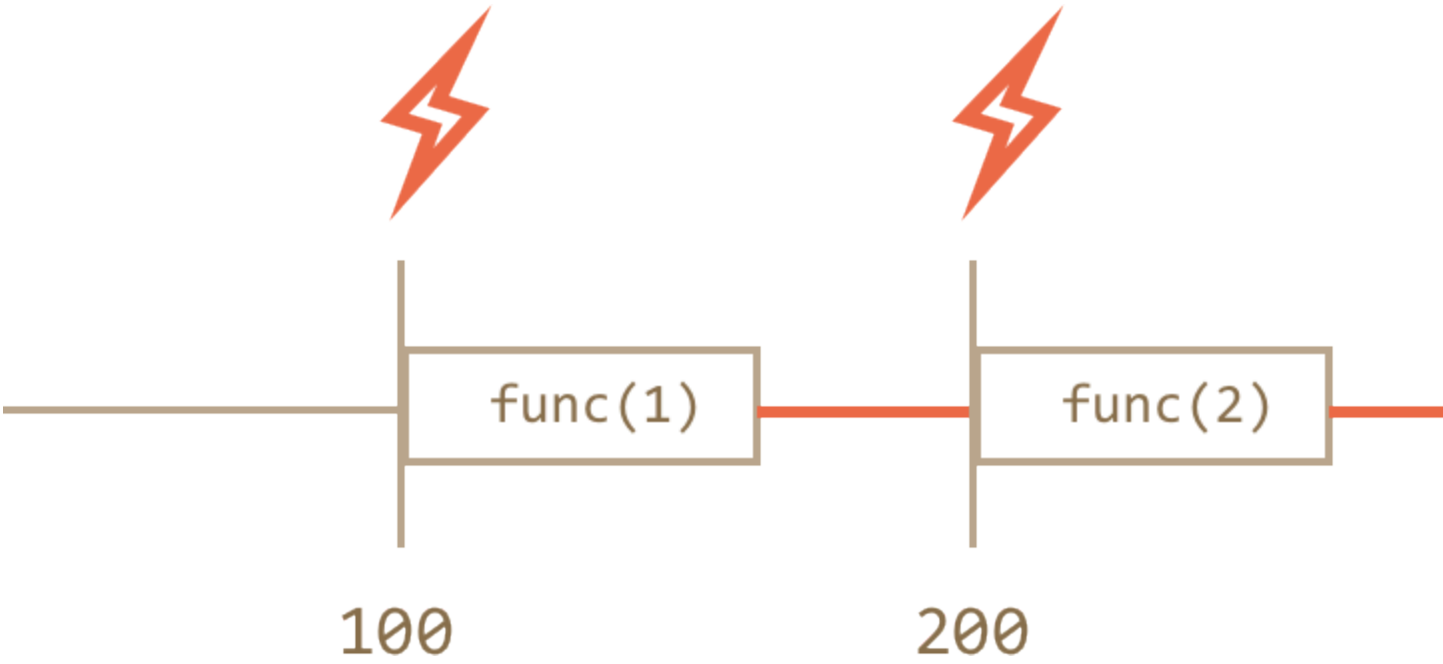

Figure 2: The interval we set for setInterval includes our own code execution. Taken from Scheduling: setTimeout and setInterval

Immediate Timeout

Using 0 as the value for setTimeout schedules the execution of func as soon as possible but only after the current code is complete.

For instance, the code below outputs “Hello”, then “World” as soon as the user clicks the OK button on the first alert dialogue:

```
setTimeout(function() {
  alert('Mr. Universe')
}, 0);

alert('hello');
```

When would you use them?

setTimeout and setInterval are useful when you need to schedule code execution.

Use setTimeout if you want to execute the code once after a given time elapses. Pay attention to the gotchas for using setTimeout and consider them additional alternatives

Use setInterval if you need the code to execute repeatedly at given intervals.

Request Animation Frame

Request Animation Frame is a specialized timing function that is primarily used with animation code of any kind (DOM elements, CSS, canvas, WebGL, or any other).The method tells the browser that you wish to perform an animation and requests that the browser call a function to update the animation before the next repaint. The method takes a callback as an argument to be invoked before the repaint

Before Request Animation Frame

When working directly with animations you've probably seen either of the two animation models show below

```
function draw() {
    // Drawing code goes here
}
setInterval(draw, 100);
```

The first example calls the draw function once every 100ms until the clearInterval function is called. An alternative to this code is to use a setTimeout function inside the draw function:

```
function draw() {
    setTimeout(draw, 100);
    // Drawing code goes here
}
draw();
```

This technique uses a recursive setTimeout call from inside the function to draw until clearTimeout is used to stop it.

You can also use the setInterval technique with a value to indicate how often the animation should run.

We arrived at our final value of 17 by running the following formula 1000 / 60Hz, rounded up.

```
// Lights, camera…function!
setInterval(function() {
    animateEverything();
}, 17);
```

Running RAF

The Request Animation Frame signature has only one parameter which is the function that you want to run.

Then we execute the function once to initiate the loop.

```
function draw() {
    // Drawing code goes here
    requestAnimationFrame(draw);
}
draw();
```

The smoothness of your animation is directly dependent on your animation's frame rate and it is measured in frames per second (fps). The higher this number is, the smoother your animation will look to a point.

Since most screens have a refresh rate of 60Hz, the fastest frame rate you can aim for is 60fps (frames per second). However, more frames, means more processing, which can often cause stuttering and skipping. This is what is meant by the term dropping frames or jank.

If we have a monitor that gives you 60Hz and you want to achieve 60 fps you have about 16.7 milliseconds to execute all your animation code. This is a reminder that we need to be mindful of the amount of code that we try to run for each animation loop.

Polyfill

Paul Irish whote a polyfill for older browsers that don't support Request Animation Frame natively.

When would you use it?

Request Animation Frame will work with all kinds of animations, CSS, WebGL, DOM manipulation.

Promises

```
var CACHE_NAME = 'my-site-cache-v1';
var urlsToCache = [
  '/',
  '/styles/main.css',
  '/script/main.js'
];

self.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open(CACHE_NAME)
    .then(function(cache) {
      console.log('Opened cache');
      return cache.addAll(urlsToCache);
    })
  );
});
```

```
fetch('https://s3-us-west-2.amazonaws.com/s.cdpn.io/32795/coffee2.png') //
  .then((response) => {
    if (response.ok) {
      console.log('it worked :)')
      return response.blob(); 'it worked :)'// 2      throw new Error('The
    }
  })
  .then((myBlob) => {
    let objectURL = URL.createObjectURL(myBlob); // 3
    let m'There was a problem with the network response.'// 3
    let myImage = document.createElement('img'); // 4
    myImage.src = objectURL; // 5rror) => {
    console.log(
      'There has been a problem with your fetch operation: ',
      error.message
    );
  });
'There has been a problem with your fetch operation: '
```

In this case it’ll perform the following tasks:

1. Retrieve the image from Codepen
2. Convert the response to an image blob
3. Create a URL object for the image blob
4. Create an image element
5. Make the `src` of the image the URL we created in the step 3
6. Attach the image to the document

Async/Await

New in ES2017 are `async` functions and the `await` keyword that will make writing async code easier to read, reason through and understand what caused any error that may get thrown. The hardest part, for me, of working with ES2016 and later is that I don't always see the reasoning behind the new code, the older version of the code still work just as fine.

Async / Await are different. They look a lot like the callback code that we used to work with in the ES5 days but they produce the same asynchronous result as if we were writing promises. It is very similar to how we'd write asynchronous code when using generators either natively or with a library like co

Async code running sequentially

Take the following code that represents sequential asynchronous calls

```
async function asyncFunc() {
    const result1 = await otherAsyncFunc1();
    console.log(result1);
    const result2 = await otherAsyncFunc2();
    console.log(result2);
}
```

And compare it with the code that produces the same result using promises:

```
function asyncFunc() {
    return otherAsyncFunc1()
    .then(result1 => {
        console.log(result1);
        return otherAsyncFunc2();
    })
    .then(result2 => {
        console.log(result2);
    });
}
```

As you can see the main difference is that `await` takes place of the then block. The code is cleaner and it makes more sense to me (not that the promise code is hard to read, just not as clean).

Async code running in parallel

The code works and it's cleaner but it's sequential. The `await` statements run sequentially and will wait for one promise to return before executing the next. There are times when we want to run all our promises in parallel either because we want the code to run fast or because we have enough promises that running them sequentially would slow the code execution too much.

To run promises in parallel we use `Promise.all`. Just like in promise based code we build an array of promises that will fulfill if all promises succeed or fail if anyone of those promises fail.

Here is the `async` / `await` code to log the result of two promises.

```
async function asyncFunc() {
    const [result1, result2] = await Promise.all([
        otherAsyncFunc1(),
        otherAsyncFunc2(),
    ]);
    console.log(result1, result2);
}
```

With the corresponding promise based code. See how similar the two are?

```
function asyncFunc() {
    return Promise.all([
        otherAsyncFunc1(),
        otherAsyncFunc2(),
    ])
    .then((result1, result2) => {
        console.log(result1, result2);
    });
}
```

Error handling

The final part of the equation is how to handle errors. To me this was the most surprising part of the exercise, going back to using `try` / `catch` blocks to handle errors just like the old synchronouse code we used to write, except that it's running the code sequentially and waits for each task to complete before performing the next.

```
async function fetchJson(url) {
    try {
        let request = await fetch(url);
        let text = await request.text();
        return JSON.parse(text);
    }
    catch (error) {
        console.log(`ERROR: ${error.stack}`);
    }
}
```

Recreating the font loader script with async and await

A few weeks ago I wrote a script to use Font Face Observer to make sure that readers got a consistent reading experience and that I could, as much as possible, control font behavior in my pages. The full script is shown below:

```
const mono = new FontFaceObserver('notomono-regular');
const sans = new FontFaceObserver('notosans-regular');
const italic = new FontFaceObserver('notosans-italics');
const bold = new FontFaceObserver('notosans-bold');
const bolditalic = new FontFaceObserver('notosans-bolditalic');

let html = document.documentElement;

html.classList.add('fonts-loading');
```

```
Promise.all([
  mono.load(),
  sans.load(),
  italic.load(),
  bolditalic.load()
]).then(() => {
  html.classList.remove('fonts-loading');
  html.classList.add('fonts-loaded');
  console.log('All fonts have loaded.');
}).catch(() =>{
  html.classList.remove('fonts-loading');
  html.classList.add('fonts-failed');
  console.log('One or more fonts failed to load')
});
```

A version of the script using `async` and `await` looks like this. Notice how we use try and catch blocks to control the process of our script.

```
const mono = new FontFaceObserver('notomono-regular');
const sans = new FontFaceObserver('notosans-regular');
const italic = new FontFaceObserver('notosans-italics');
const bold = new FontFaceObserver('notosans-bold');
const bolditalic = new FontFaceObserver('notosans-bolditalic');

let html = document.documentElement;

html.classList.add('fonts-loading');

async function loadFonts() {
  try {
    const results = await Promise.all([
      mono.load(),
      sans.load(),
      italic.load(),
      bold.load(),
      bolditalic.load()
```

```
    ]);
    html.classList.remove('fonts-loading');
    html.classList.add('fonts-loaded');
    console.log('All fonts have loaded.');
    return results;
  }
  catch (error) {
    html.classList.remove('fonts-loading');
    html.classList.add('fonts-failed');
    console.log('One or more fonts failed to load')
  }
}
```

Another example is how would our fetch call look in a async / await, try / catch blocks.

```
async function loadImage() {
  try {
    const response = await fetch('https://s3-us-west-2.amazonaws.com/s.cdp

    const myBlob = await response.blob();
    const objectURL = await URL.createObjectURL(myBlob);
    const myImage = await document.createElement('img');

    myImage.src = objectURL;

    document.body.appendChild(myImage);
  }
  catch(error) {
    console.log(
      'There has been a problem with your fetch operation: ', error.messag
    );
  }
}
```

Async functions and the await keyword are fully supported in modern browsers but not in older versions. How to handle the difference between supported and non supported browsers? We can use feature detection to work the promise code

and break early if the browser support promises.

Detecting promise support is a matter of testing if ‘Promise’ is available in the window object, otherwise, we can run an API that is supported in the older browsers you’re targetting. This is one way you can do it:

```
if (!'Promise' in window) {
  console.log('promises are not supported');
} else {
  console.log('promises are supported, yay!');
}
```

Detecting async functions is harder.