# Async functions: better async?

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 promise to an array that will fulfill if all promises succeed or fail if anyone of t hose 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` may look 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')
  }
}
```

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.

Or we can choose not to care about older browsers and only support current browsers that will work with the features we want.

Which one you use is up to you.

# Links and References

- https://developers.google.com/web/fundamentals/getting-started/primers/async-functions
- https://jakearchibald.com/2014/es7-async-functions/
- http://exploringjs.com/es2016-es2017/ch_async-functions.html