JavaScript async and await

Asynchronous JavaScript has never been easy. For a while, we used callbacks. Then, we used promises. And now, we have asynchronous functions.

Asynchronous functions make it easier to write asynchronous JavaScript, but it comes with its own set of gotchas that makes life hard for beginners.

In this 2-part series, I want to share everything you need to know about asynchronous functions.

Asynchronous functions

Asynchronous functions contains the `async` keyword. You can use it in a normal function declaration:

```
async function functionName (arguments) {
  // Do something asynchronous
}
```

You can also use it in an arrow-function.

```
const functionName = async (arguments) => {
  // Do something asynchronous
}
```

Asynchronous functions always return promises

It doesn't matter what you `return`. The returned value will always be a promise.

```
const getOne = async _ => {
  return 1
}

const promise = getOne()
console.log(promise) // Promise
```

Note: You should know what are JavaScript Promises and how to use them before you move on. Otherwise, it'll start to get confusing. Use this article to help you get familiar with JavaScript promises.

The await keyword

When you call a promise, you handle the next step in a `then` call, like this:

```
const getOne = async _ => {
  return 1
}

getOne()
  .then(value => {
    console.log(value) // 1
  })
```

The `await` keyword lets you wait for the promise to resolve. Once the promise is resolved, it returns the parameter passed into the `then` call.

```
const test = async _ => {
  const one = await getOne()
  console.log(one) // 1
}

test()
```

Return await

There's no need to `await` before returning a promise. You can return the promise directly.

(If you `return await` something, you resolve the original promise first. Then, you create a new promise from the resolved value. `return await` effectively does nothing. No need for the extra step).

```
// Don't need to do this
const test = async _ => {
  return await getOne()
}

test()
  .then(value => {
    console.log(value) // 1
  })
// Do this instead
const test = async _ => {
  return getOne()
}
```

```
test()
  .then(value => {
    console.log(value) // 1
  })
```

Note: If you don't need `await`, you don't need to use an async function. The example above can be rewritten as follows:

```
// Do this instead
const test = _ => {
  return getOne()
}

test()
  .then(value => {
    console.log(value) // 1
  })
```

Handling errors

If a promise results in an error, you handle it with a `catch` call, like this:

```
const getOne = async (success = true) => {
  if (success) return 1
  throw new Error('Failure!')
}

getOne(false)
  .catch(error => console.log(error)) // Failure!
```

If you want to handle an error in an asynchronous function, you need to use a `try/catch` call.

```
const test = async _ => {
  try {
    const one = await getOne(false)
  } catch (error) {
    console.log(error) // Failure!
  }
}

test()
```

If you have multiple `await` keywords, error handling can become ugly…

```
const test = async _ => {
  try {
    const one = await getOne(false)
  } catch (error) {
    console.log(error) // Failure!
  }

  try {
    const two = await getTwo(false)
  } catch (error) {
    console.log(error) // Failure!
  }

  try {
    const three = await getThree(false)
  } catch (error) {
    console.log(error) // Failure!
  }
}

test()
```

There’s a better way.

We know that asynchronous functions always return a promise. When we call a promise, we can handle errors in a `catch` call. This means we can handle any errors from our asynchronous function by adding `.catch`.

```
const test = async _ => {
  const one = await getOne(false)
  const two = await getTwo(false)
  const three = await getThree(false)
}

test()
  .catch(error => console.log(error)))
```

Note: The Promise `catch` method lets you catch one error only.

Multiple awaits

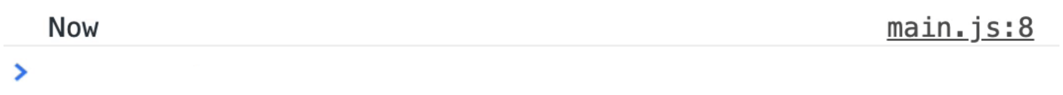
`await` blocks JavaScript from executing the next line of code until a promise resolves. This may have the unintended consequence of slowing down code execution.

To show this in action, we need to create a delay before resolving the promise. We can create a delay with a `sleep` function.

```
const sleep = ms => {
  return new Promise(resolve => setTimeout(resolve, ms))
}
```

`ms` is the number of milliseconds to wait before resolving. If you pass in `1000` into `sleep`, JavaScript will wait for one second before resolving the promise.

```
// Using Sleep
console.log('Now')
sleep(1000)
  .then(v => {
    console.log('After one second')
  })
```



```
Now                                                  main.js:8
```

Let's say `getOne` takes one second to resolve. To create this delay, we pass `1000` (one second) into `sleep`. After one second has passed and the `sleep` promise resolves, we return the value 1.

```
const getOne = _ => {
  return sleep(1000).then(v => 1)
}
```

If you `await getOne()`, you'll see that it takes one second before `getOne` resolves.

```
const test = async _ => {
  console.log('Now')

  const one = await getOne()
  console.log(one)
}

test()
```

Now let’s say you need to wait for three promises. Each promise has a one-second delay.

```
const getOne = _ => {
  return sleep(1000).then(v => 1)
}

const getTwo = _ => {
  return sleep(1000).then(v => 2)
}

const getThree = _ => {
  return sleep(1000).then(v => 3)
}
```

If you `await` these three promises in a row, you’ll have to wait for three seconds before all three promises get resolved. This is not good because we forced JavaScript to wait two extra seconds before doing what we need to do.

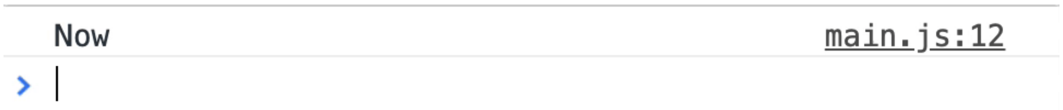
```
const test = async _ => {
  const one = await getOne()
  console.log(one)

  const two = await getTwo()
  console.log(two)

  const three = await getThree()
  console.log(three)

  console.log('Done')
}

test()
```



```
Now                                   main.js:12
>
```

If `getOne`, `getTwo` and `getThree` can be fetched simultaneously, you’ll save two seconds. You can fetch these three promises at the same time with `Promise.all`.

There are three steps:

1. Create the three promises
2. Add all three promises into an array
3. `await` the array of promises with `Promise.all`

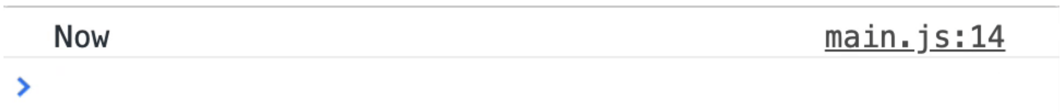
Here's what it looks like:

```
const test = async _ => {
  const promises = [getOne(), getTwo(), getThree()]
  console.log('Now')

  const [one, two, three] = await Promise.all(promises)
  console.log(one)
  console.log(two)
  console.log(three)

  console.log('Done')
}

test()
```



```
Now                                                    main.js:14
>
```

That's all you need to know about basic asynchronous functions! I hope this article clears things up for you.

Note: This article is a modified excerpt from Learn JavaScript. If you find this article useful, you might want to check it out.

Next up, we're going to look at asynchronous functions and its behavior in loops.

If you enjoyed this article, please tell a friend about it! Share it on Twitter. If you spot a typo, I'd appreciate if you can correct it on GitHub. Thank you!