ES6 Arrow Functions in Depth

The daily saga of es6-in-depth articles continues. Today we'll be discussing Arrow Functions. In previous articles we've covered destructuring and template literals. I strive to cover all the things when it comes to the ES6 feature-set – and eventually we'll move onto ES7. I find that writing about these features makes it way easier for them to become ***engraved in my skull*** *as well.*

Nicolás Bevacqua

Published a year ago | 6 minute read | 7

Since you're reading these articles, I suggest you set up Babel and `babel-node`, and follow along by copying the self-contained examples into a file. You can then run them using `babel-node yourfile` in the terminal. Running these examples on your own and maybe tweaking them a little bit **will help you better internalize these new features** – even if you're just adding `console.log` statements to figure out what's going on.

Now onto the topic of the day.

We've already kind of went over *arrow functions* in previous articles, using them in passing without a lot of explaining going on. This article will focus mainly in arrow functions and keep the rest of ES6 in the back burner. I think that's the best way to write about ES6 – making a single feature "stand out" in each article, and gradually adding the others and interconnecting the different concepts so that we can understand *how they interact together*. I've observed **a lot of synergy** in

concepts so that we can understand *how they interact together*. I've observed **a lot of synergy** in ES6 features, which is *awesome*. It's still important to make a gradual dive into ES6 syntax and features and not jump into the water as it's warming up, because otherwise you'll have a bad time adjusting to the new temperature – that was probably a bad analogy, moving on.

Using Arrow Functions in JavaScript

Arrow functions are available to many other modern languages and was one of the features I sorely missed a few years ago when I moved from C# to JavaScript. Fortunately, they're now part of ES6 and thus available to us in JavaScript. The syntax is quite expressive. We already had anonymous functions, but sometimes it's nice to have a terse alternative.

Here's how the syntax looks like if we have a single argument and just want to return the results for an expression.

```
[1, 2, 3].map(num => num * 2)
// <- [2, 4, 6]
```

The ES5 equivalent would be as below.

```
[1, 2, 3].map(function (num) { return num * 2 })
// <- [2, 4, 6]
```

If we need to declare more arguments *(or no arguments)*, we'll have to use parenthesis.

```
[1, 2, 3, 4].map((num, index) => num * 2 + index)
// <- [2, 5, 8, 11]
```

You might want to have some other statements and not just an expression to return. In this case you'll have to use bracket notation.

```
[1, 2, 3, 4].map(num => {
  var multiplier = 2 + num
  return num * multiplier
})
// <- [3, 8, 15, 24]
```

You could also add more arguments with the parenthesis syntax here.

```
[1, 2, 3, 4].map((num, index) => {
  var multiplier = 2 + index
  return num * multiplier
})
// <- [2, 6, 12, 20]
```

At that point, however, chances are you'd be better off using a named function declaration for a number of reasons.

- `(num, index) =>` is only marginally shorter than `function (num, index)`
- The `function` form allows you to name the method, improving code quality
- When a function has multiple arguments and multiple statements, I'd say it's improbable that six extra characters will make a difference
- However, naming the method might add just enough context into the equation that those six extra characters (plus method name) become really worthwhile

extra characters (plus method name) become really worthwhile

Moving on, if we need to return an object literal, we'll have to wrap the expression in parenthesis. That way the object literal won't be interpreted as a statement block (which would result in a silent error or worse, a **syntax error** because `number: n` isn't a valid expression in the example below. The first example interprets `number` as a label and then figures out we have an `n` expression. Since we're in a block and not returning anything, the mapped values will be `undefined`. In the second case, after the label and the `n` expression, `, something: 'else'` makes no sense to the compiler, and a `SyntaxError` is thrown.

```
[1, 2, 3].map(n => { number: n })
// [undefined, undefined, undefined]
[1, 2, 3].map(n => { number: n, something: 'else' })
// <- SyntaxError

[1, 2, 3].map(n => ({ number: n }))
// <- [{ number: 1 }, { number: 2 }, { number: 3 }]
[1, 2, 3].map(n => ({ number: n, something: 'else' }))
/* <- [
  { number: 1, something: 'else' },
  { number: 2, something: 'else' },
  { number: 3, something: 'else' }]
*/
```

A cool aspect of arrow functions in ES6 is that they're bound to their lexical scope. That means that you can say goodbye to `var self = this` and similar hacks – *such as using* `.bind(this)` – to preserve the context from within deeply nested methods.

```
function Timer () {
  this.seconds = 0
  setInterval(() => this.seconds++, 1000)
}
```

```
var timer = new Timer()
setTimeout(() => console.log(timer.seconds), 3100)
// <- 3
```

Keep in mind that the lexical `this` binding in ES6 arrow functions means that `.call` and `.apply` won't be able to change the context. Usually however, that's more of a feature than a bug.

Conclusions

Arrow functions are neat when it comes to defining anonymous functions that should probably be *lexically bound anyways*, and they can definitely make your code more terse in some situations.

There's no reason why you should be turning all of your function declarations into arrow functions unless their arguments and expression body are descriptive enough. I'm a big proponent of named function declarations, because they improve readability of the codebase without the need for comments – which means I'll have *"a hard time"* adopting arrow functions in most situations.

That being said, I think arrow functions are particularly useful in most functional programming situations such as when using `.map`, `.filter`, or `.reduce` on collections. Similarly, arrow functions will be really useful in asynchronous flows since those typically have a bunch of callbacks that just do argument balancing, a situation where arrow functions really shine.