Lesson Tuesday

Intermediate JavaScript (/intermediate-javascript) / Test-Driven Development and Environments with JavaScript (/intermediate-javascript/test-driven-development-and-environments-with-javascript) / ES6 Arrow Notation

Text

Arrow functions are one of the most popular and useful new features in ES6. In fact, we've already been using arrow functions with Jest. Now it's time to delve more deeply into why they are useful and how we can use them in our code. There are a few reasons why arrow functions are so popular. We'll be focusing primarily on one of these reasons: arrow functions change the way `this` is bound inside of a nested function.

Arrow Functions and `this`

The value of `this` changes based on where we are calling it in our code, but it always represents the object inside of which our code is executing. `this` can be a very confusing topic in JavaScript, especially for beginners, and it's not always clear what `this` is bound to — meaning, what object `this` actually represents. You

may have dealt with this issue in Introduction to Programming; it's very common (and frustrating) for new developers to try to use `this`, only to find that it's `undefined`.

The best way to simplify `this` is to think of it within the context of object-oriented programming. Specifically, we should probably only use `this` if we're calling a function on a specific object.

In fact, this is what we commonly do inside of classes. (If classes still feel like a weird new concept, you can also think of this as being similar to what we've done with prototype methods in the past.)

Let's look at some code that isn't going to work as expected. The reason, as you might guess, is related to `this`. Go ahead and put the following code in the DevTools console:

```
class Box {

  constructor() {
    this.stuff = [];
  }

  addJunk(array) {
    console.log(this);
    array.forEach(function(thing) {
      this.stuff.push(thing);
    });
  }
}
```

In this example, we create a `Box` class. It has one property: `this.stuff`, an empty array.

Next, we have a prototype method `Box.prototype.addJunk(array)`, but written using `class` syntax. It seems like a straightforward method — we pass in an array and then for each element in the

array, we push that element into `this.stuff`, the property of our `Box` object.

Well, if we try it in the console, we'll see that `Box.prototype.addJunk(array)` doesn't work correctly. Add this code in the console:

```
> let box = new Box();
> box.addJunk(["broken pencils", "busted rubber bands", "ch
eckers pieces"]);
```

We'll get the following `console.log()` message:

```
Box {stuff: Array(0)}
```

This `console.log()` message confirms that `this` is set to our `Box` object, which before the loop to add the new Junk has nothing in the `stuff` property. This is what we expect to see.

However, we'll also get the following error:

```
Uncaught TypeError: Cannot read property 'stuff' of undefin
ed
```

What just happened? When our code tried to read `this.stuff` inside the loop, it threw an error because `this` is `undefined`.

And why would that happen? We just used a `console.log(this);` to verify that `this` is exactly what we thought it was, the `Box` object.

However, we used `console.log(this)` *outside* of the loop. This is a weird thing about JavaScript. We entered a loop and `this` lost its scope. What gives? Well, in JavaScript, every time a function is

created, so too is a new scope. So, it's not that we've entered a loop that's causing `this` to be undefined, it's because we're calling `this` within the callback function (with the new scope!) that we pass into `Array.prototype.forEach()`. The problem with the scope of the callback function is that it's not tied to any object, so `this` ends up being `undefined`.

To reiterate this and see the value of `this` in the context of our `Box` object, we've added commentary and a few additional `console.log()` statements that describe what `this` represents at every level of our script. You can read through the following commentary, or copy and paste the following code into your DevTools console to try it out. The last message you'll see is the same as before: a TypeError about `this` being `undefined`.

```
// 'this' is set to JavaScript global object, which
// is the window object when run in the browser
console.log("this at the top level of our script", this);

class Box {

  constructor() {
    // 'this' is the Box object
    console.log("this in the constructor", this);
    this.stuff = [];
  }

  addJunk(array) {
    // 'this' is the Box object
    console.log("this in addJunk method", this);

    array.forEach(function(thing) {
      // 'this' is undefined, because
      // the callback function does not belong to
      // any object type
      console.log("this in addJunk's forEach loop callbac
k", this);
      this.stuff.push(thing);
    });
  }
}
const newBox = new Box();
newBox.addJunk(["broken pencils", "busted rubber bands", "c
heckers pieces"]);
```

Well, this is not ideal at all. We want `this` to represent our `Box` object within the `Array.prototype.forEach()` callback function!

Traditionally (before arrow notation came along, that is), JavaScript developers dealt with the issue by manually binding `this` to the object they wanted it to represent. There are several ways to bind `this`, but here's the easiest approach:

```
addJunk(array) {
  let that = this;
  array.forEach(function(thing) {
    that.stuff.push(thing);
  });
}
```

We declare a variable named `that` inside our function and set it to the value of `this`. The inner function has access to `that`, which is just a reference to `this`, and so we can push stuff into the box. This works because the variable `that` won't change until we tell it to, whereas the value of `this` always depends on where it is scoped (where in our code it's being called from).

This approach is a hack, though, and JavaScript's default behavior really isn't great. Do we really want to use the above hack any time there's a callback function inside of an object's method?

Fortunately, we can fix this problem by using arrow notation. With arrow notation we can create an **arrow function expression**, which is also commonly called an **arrow function**. Arrow functions do not perform any binding of `this`, which means that `this` is set to the scope that the arrow function was defined in. For the `Box.prototype.addJunk()` method, this means that inside of an arrow function, `this` will remain unchanged and still set the `Box` object.

Let's update `Box.prototype.addJunk()` to use an arrow function.

```
addJunk(array) {
  // The arrow function is defined in the scope of
  // the Box.prototype.addJunk() method, which
  // belongs to the Box object.
  array.forEach((thing) => {
    // The arrow function's own scope is anywhere
    // inside of the arrow function.
    // 'this' is not bound to the arrow function's own scop
e, but
    // instead to the scope in which the arrow function was
defined,
    // the Box object.
    this.stuff.push(thing);
  });
}
```

And this is the arrow function, separated out:

```
(thing) => {
  this.stuff.push(thing);
}
```

What we've done here is taken away the `function()` and replaced it with `() =>`. Parameters still go inside the parentheses as needed.

When to Use Arrow Functions

So when should we use arrow notation instead of the notation we've used in the past? Well, it's becoming increasingly common to use arrow notation regardless of whether you're concerned about binding `this` or not. That's because arrow syntax is more concise and because its behavior is predictable and helps ensure that our code works how we'd expect it to.

However, **you should never use arrow notation to define an object's method or an object's constructor function.** Using arrow functions in any other application, or inside of an object's method is acceptable. To review a list of all differences between other functions and arrow functions, visit the MDN documentation (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions).

More Arrow Function Examples

So how exactly does this syntax look in situations other than with `Array.prototype.forEach()`?

Here are some examples:

Unnamed Functions

Here's an unnamed function without arrow notation:

```
function (name) {
  return "hi " + name;
}
```

Here's the same function with arrow notation:

```
(name) => {
  return "hi " + name;
}
```

Named Functions

How about for a named function? This is how we've done it so far:

```
function greeting(name) {
  return "hi " + name;
}
```

To do this with arrow notation, we need to save the function in a variable like this:

```
const greeting = (name) => {
  return "hi " + name;
};
```

This may look pretty strange at first, but it will become more familiar over time, and we will revisit arrow functions in React.

You can continue to use function declarations or function expressions instead of arrow function expressions. However, the one case where you must use an arrow function is when you need the value of `this` to not change when a new scope is created by a function inside of an object's method.

And again, you should not use arrow functions to create constructors or object methods.

Syntactic Sugar for Arrow Notation

It's also possible to use arrow notation to make our code even more concise, though it will look even more abstract as a result. Specifically, if the body of the function is a single line, we can omit both the brackets *and* the return keyword. Let's take a look:

```
const greeting = name => "hi " + name;
```

We can even omit the parentheses around the example above as long as there's just one parameter.

While this is very concise, it can be harder to read and understand, especially for new developers, so don't use this syntax unless you feel very comfortable with it. In fact, there are a few gotchas with this syntax.

If the function has two arguments, you can't omit the parentheses:

```
const greeting = (greeting, name) => greeting + " " + name;
```

And if the code is multi-line, you can't omit the brackets *or* the `return` keyword:

```
const greeting = (greeting, name) => {
  const uppercasedGreeting = greeting.toUpperCase();
  const uppercasedName = name.toUpperCase();
  return uppercasedGreeting + " " + uppercasedName;
};
```

This is a contrived example because we could easily reduce this function to one line. The point is that as soon as we have multiple lines in the body of a function using arrow notation, we need to use brackets and the `return` keyword.

For more information on arrow notation, see Arrow function expressions in the Mozilla documentation (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions).

You will see arrow notation a lot in documentation and in JavaScript libraries, like Jest — and the longer you code in JavaScript, the more likely you are to use it regularly. We recommend getting familiar

with arrow notation and using it regularly once you are comfortable with it.

Previous (/intermediate-javascript/test-driven-development-and-environments-with-javascript/es6-classes)
Next (/intermediate-javascript/test-driven-development-and-environments-with-javascript/es6-template-literals)

Lesson 37 of 49
Last updated February 11, 2023

disable dark mode

(http://www.epicodus.com)