What Are Arrow Functions?

code (/?category=code) | 10/20/2015

One of my favorite parts of ES6 is arrow functions. In fact, at this point I encourage my team to always default to using them unless there is a clear reason not to. So, you you may be wondering what arrow functions are and why they are worth using. In this article, I will show some examples of arrow functions and how they can be used to make your JavaScript code cleaner and more readable.

Using Arrow Functions

Arrow functions are not in most browsers yet. They are part of the ES6 spec which began to get implemented this year in browsers, but there is still a long way to go for full browser-support. To get this to work in the browser, you'll need to install a transpiler. I highly recommend Babel (https://babeljs.io/). We are using it for some large production applications and it is a solid choice. If you're running Node, you'll either want to use 4.0 or up or run it with the `--harmony` flag on.

A Little Background

For those of you who are unfamiliar with JavaScript, a traditional function looks something like this:

```
function sayHi (name) {
    return `Hi ${name}`;
}
```

We can also declare a function as the value of a variable or as a value in an object:

```
let sayHi = function (name) {
    return `Hi ${name}`;
};

// Or
let person = {
    sayHi: function (name) {
        return `Hi ${name}`;
    }
}
```

This all works perfectly well, but the complexity comes when we use the `this` keyword in a function. It is really easy to lose the context of a function, which is why we used to create a magic variable to contain the context outside of the function:

```
function getPerson () {
    let self = this;

    return {
        setName: function (name) {
            self._name = name;
            return this;
        },
        sayHi: function () {
            return `Hi ${name}`;
        }
    };
}

getPerson().setName('Tyson').sayHi();
```

The reason we did that is that the `this` in a higher-order function would be a different context than a deeper `this`.

Another trick that can be used is to manually bind a function to a different context. This can be done on function execution via `call` and `apply`, or we can use `bind` to create a new copy of the function in the context of the object that we bind it to:

```
let tyson = {
    name: 'Tyson'
};

function sayHi () {
    return `Hi ${this.name}`;
}

let sayHiToTyson = sayHi.bind(tyson);
sayHiToTyson();
```

A Little Foreground

Arrow functions are a way to solve the problem of context and to write functions in a less tedious way.

That means that this:

```
let sayHi = function (name) {
    return `Hi ${name}`;
};
```

Becomes this:

```
let sayHi = (name) => {
    return `Hi ${name}`;
};
```

Okay, so we've saved a few characters. Big deal, right? Well, the real power of arrow functions is that we have to spend less time worrying about our context because of the simple rule that arrow functions inherit context from their parent.

That means that instead of this:

```
function getPerson () {
    this.name = 'Tyson';
    let self = this;

    return {
        sayHi: function () {
            return `Hi ${self.name}`;
        }
    };
}
```

We can do this:

```
let getPerson () => {
    this.name = 'Tyson';

    return {
        sayHi: () => {
            return `Hi ${this.name}`;
        }
    };
}
```

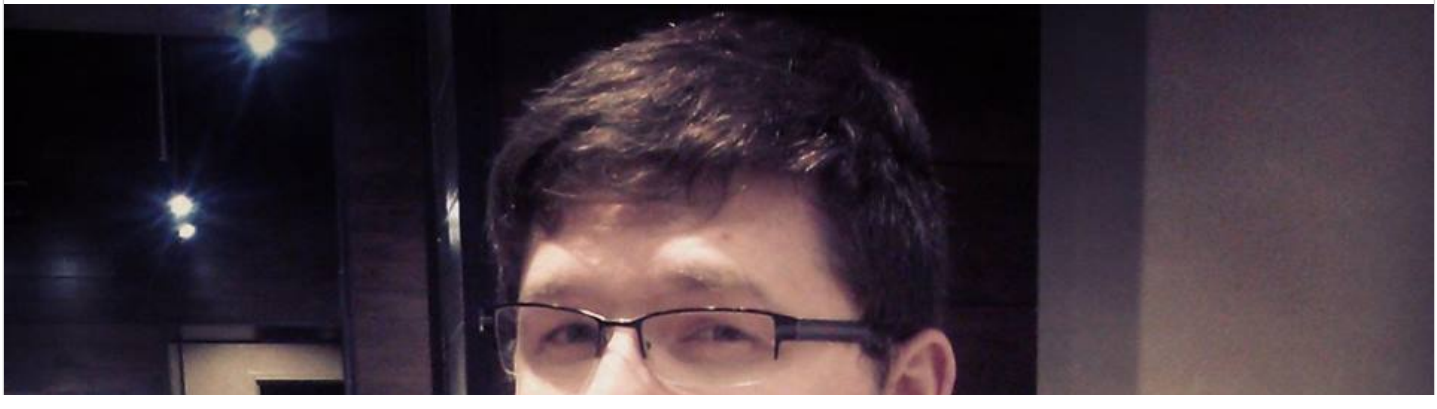
When you apply that kind of line trimming to a really big application, you will knock off a lot of busywork code that will no longer be needed.

If you feel like you're still writing too much code, you can use short-hand arrow functions like this for one-liners:

```
let person = {
    sayHi: (name => `Hi ${name}`; )
};
```

This will only work if you have a single argument, if you have more than one, you'll have to suck it up and wrap your arguments in parenthesis. There are probably worse fates.

So that is pretty much arrow functions in a nutshell. They've definitely been great for my code!



About the Author (/about)

Tyson Cadenhead is a Senior JavaScript Developer at Aloompa (http://aloompa.com). He lives in the greater Nashville area. His specialty is writing large, scalable JavaScript applications on the client and server side. His passions are for good design, usability, and clean, reusable code.

Tags: javascript (/?tag=javascript) es6 (/?tag=es6)

1

Related Posts:

(/blog/how-can-i-share-ejs-templates-between-the-client-side-and-the-server-side)

How Do I Share EJS Templates Between the Client-side and the Server-side? (/blog/how-can-i-share-ejs-templates-between-the-client-side-and-the-server-side)