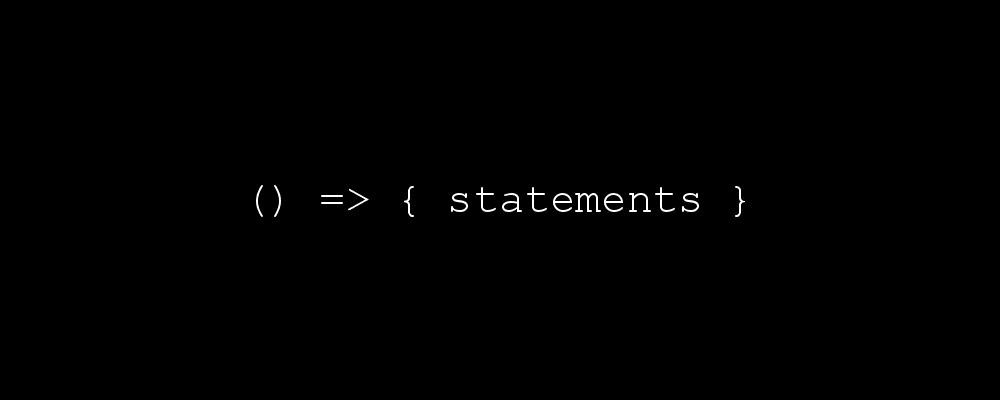
**Arrow functions** – also called “fat **arrow**” **functions**, from CoffeeScript (a transcompiled language) are a more concise syntax for writing **function** expressions. They utilize a new token, => , that looks like a fat **arrow**. **Arrow functions** are anonymous and change the way this binds in **functions**.

**JavaScript: Arrow Functions for Beginners**



Last week I published [this post](https://codeburst.io/javascript-the-keyword-this-for-beginners-fb5238d99f85) on the keyword this for beginners. One of the topics that wasn’t covered in that article was arrow functions. The topic was simply too large to cover in that post, so this article is here to make up for it. Read on to learn about the basics of arrow functions!

**Preface**

Hey! I’m Brandon. I created [codeburst.io](https://codeburst.io) and I write JavaScript tutorials and articles to help beginners better understand the inner workings of Web Development. If you have any questions about the article, leave a comment and I’ll get back to you, or find me on twitter [@brandonmorelli](http://twitter.com/brandonmorelli). Lastly, when you’re ready to really dive into Web Development, Check out the [Best Courses for Learning Full Stack Web Development](https://codeburst.io/best-udemy-courses-for-learning-full-stack-web-development-45e2bd3ec28b)

**Benefit #1: Shorter Syntax**

Lets take a look at a regular function:

function funcName(params) {  
 return params + 2;  
 }

funcName(2);  
// 4

This above code indicates one of the two reasons for creating arrow functions: **shorter syntax.** The exact same functions can be expressed as an arrow function with only one line of code:

var funcName = (params) => params + 2

funcName(2);  
// 4

Pretty cool. This example is obviously an extreme simplification, but hopefully illustrates my point. Lets take a look at the syntax of arrow functions a little more in-depth:

**(**parameters**) => {** statements **}**

If we have no parameters, we express an arrow function like this:

**() => {** statements **}**

When you only have one parameter, the opening parenthesis are optional:

parameters **=> {** statements **}**

Finally, if you are returning an expression, you remove the brackets:

parameters **=>** expression

// is equivalent to:

function (parameters){  
 return expression;  
}

Alright, you know the syntax, how about an example? Open up your Chrome Developer Console (Windows: Ctrl + Shift + J)(Mac: Cmd + Option + J) and type the following:

var double = num => num \* 2

As you can see we’re assigning the variable double to an arrow function. The arrow function has one parameter, num . As there is only one parameter, we’ve omitted parenthesis around the parameter. Since we want to return the value of num \* 2 we’ve also omitted the brackets around the expression to return. Let’s invoke the function and see it in action:

double(2);  
// 4

double(3);  
// 6

**Benefit #2: No binding of this**

*Before we move on, you should have a good understanding of the keyword this and how it works. If you want to learn, or need a refresher, read* [*my post*](https://codeburst.io/javascript-the-keyword-this-for-beginners-fb5238d99f85) *on the subject before continuing.*

Unlike a regular function, an arrow function does not bind this. Instead, this is bound lexically (i.e. this keeps its meaning from its original context).

An example should make this clearer. In your console lets create a constructor function then create an instance of it:

function Counter() {  
 this.num = 0;  
}

var a = new Counter();

As you should know from my last article, with a constructor function the value of this is bound to the new object being created, in this case, the a object. That is why we can console.log a.num and get 0

console.log(a.num);  
// 0

What if we want to increase the value of a.num every second? We can use the setInterval() function. setInterval() is a function that calls another function after a set number of milliseconds. Let’s add it to our Counter function:

function Counter() {  
 this.num = 0;

this.timer = setInterval(function add() {  
 this.num++;  
 console.log(this.num);  
 }, 1000);  
}

Code looks the same as before, except we’ve added a variable, this.timer and set it equal to our setInterval function. Every 1000 milliseconds (one second) the code will run. this.num will increment by one, then it will be logged to the console. Let’s try it out. In the console create an instance of Counter again:

var b = new Counter();  
// NaN  
// NaN  
// NaN  
// ...

As you’ll see, the function will log to the screen every second. But the result isn’t what we expect. NaN (Not a Number) is being logged. So, what went wrong? First thing is first, stop they annoying interval by running:

clearInterval(b.timer);

Let’s back up. Our setInterval function isn’t being called on a declared object. It also isn’t being called with the new keyword (only the Counter() function is). And lastly, we’re not using call, bind, or apply. setInterval is just a normal function. In fact, the value of this in setIntervalis being bound to the global object! Lets test this theory by logging the value of this:

function Counter() {  
 this.num = 0;

this.timer = setInterval(function add() {  
 console.log(this);  
 }, 1000);  
}

var b = new Counter();

As you’ll see, the window object is logged out every second. Clear the interval by running:

clearInterval(b.timer);

Back to our original function. It was logging NaN because this.num was referring to the num property on the window object ( window.num which doesn’t exist), and not the b object ( b.num ) we had just created.

So how do we fix this? With an arrow function! We need a function that doesn’t bind this. With an arrow function, the this binding keeps its original binding from the context. Lets take our original Counter function and replace our setInterval with an arrow function.

function Counter() {  
 this.num = 0;

this.timer = setInterval(() => {  
 this.num++;  
 console.log(this.num);  
 }, 1000);  
}

var b = new Counter();  
// 1  
// 2  
// 3  
// ...

As you’ll see, the console begins logging increasing numbers — it works! The original this binding created by the Counter constructor function is preserved. Inside the setInterval function, this is still bound to our newly created b object!

You can clear the interval with:

clearInterval(b.timer);

For proof of concept we can again try logging this from within our arrow function. We’ll create a variable called that in our Counter function. We’ll then log out true if the value of this in our setInterval function is equal to the value of this (via that ) in the parent Counter function:

function Counter() {  
 var that = this;

this.timer = setInterval(() => {  
 console.log(this === that);  
 }, 1000);  
}

var b = new Counter();  
// true  
// true  
// ...

As expected, we log true each time! Again, clear the interval with:

clearInterval(b.timer);

**Conclusion**

Hopefully this article helped you see the two main benefits of arrow functions:

1. Shorter Syntax
2. No binding of this

As a disclaimer, there is more to arrow functions than what was explained in this article. But this should give you a great base of knowledge for further learning! As always, leave a comment if you have any great resources on the subject for others to explore.

Generator function