SEP 08 2016

ES6 has introduced arrow functions which have three main benefits. First, they have a *concise* syntax. Secondly, they have *implicit returns*, which allows us to write these nifty one-liners.

Thirdly, *they don't rebind the value of* ***this*** when you use a arrow function inside of another function, which is really helpful for when you're doing things like click handlers and whatnot.

We're going to take a look at a whole bunch of examples as well as we're going to be using arrow functions all over the place in the ES6.io course.

I've got an array of names:

JavaScript

```
const names = ['Wes', 'Kait', 'Lux'];
```

I want to add `'Bos'` to the end of all three of these.

Normally, you'd do something like this:

JavaScript

```
const fullNames = names.map(function(name){
  return `${name} Bos`;
});

console.log(fullNames); // Wes Bos, Kait Bos, Lux Bos
```

We're going to use **backticks** here, which is our template strings. Don't worry exactly about what that is, if you're not sure just yet. We have a whole chapter on that coming up.

Anyway, It's going to give me Wes Bos, Kait Bos, Lux Bos in the entire array. It took this array, transformed it into whatever the item was, plus the name "Bos" on the end.

That makes sense to me, but this isn't an arrow function. Let's take a look at how we could rewrite that.

Turn it into an Arrow Function

The first thing you do with an arrow function is, you simply delete the keyword `function` and add in what's called a fat arrow. It looks like this: `=>`

JavaScript

```
const fullNames2 = names.map((name) => {
 return `${name} Bos`;
});

console.log(fullNames2); // Wes Bos, Kait Bos, Lux Bos
```

If you've come from other programming languages, you might have seen that before, but in JavaScript it's the first time we're seeing a fat arrow.

It'll do exactly the same thing as `function`. If you `console.log` it, there should be no surprises there. We get the exact same thing.

Removing Parens With Single Params

We can go even further with it where, if you only have *one parameter* you can take out the parentheses:

JavaScript

```
const fullNames3 = names.map(name => {
 return `${name} Bos`;
});

console.log(fullNames3); // Wes Bos, Kait Bos, Lux Bos
```

That's a bit of a stylistic choice. Some prefer the parenthesis regardless if you have one or more. In many callback fucntions (like our map function) it's nice to leave them out for a very clean syntax.

Arrow Function Implicit Return

What else could I do with this? I can use what's called an **implicit return**.

Hold on — what's a *explicit* `return`?

That's when you explicitly write `return` for what you want to `return`. But a lot of these callback functions that we write in JavaScript are just one-liners, where we *just return something immediately* in one line. We don't need a whole bunch of lines.

**So – if the only purpose of your arrow function is to return something, there is no need for the `return` keyword. **

Our three line function with an explicit return is now a single line function with an **implicit return**.

JavaScript

```
const fullNames4 = names.map(name => `${name} bos`);

console.log(fullNames4);  // Wes Bos, Kait Bos, Lux Bos
```

We did three things here:

1. delete the return
2. put it up on all of one line
3. delete the curly brackets

When you delete your curly brackets, it's going to be an implicit return, which means we do not need to specify that we are returning `${name} bos`.

It will just assume that we're doing so, and you can `console.log` it to see the same thing again.

No Arguments with Arrow Functions

Then finally, if you have no arguments at all — in our above examples obviously we need an argument — but if no arguments at all, you need to pass some empty parenthesis there.

Maybe we'll just return `Cool Bos`, and they'll all be Cool Bos at the end.

JavaScript

```
const fullNames5 = names.map(() => `Cool Bos`);

console.log(fullNames5); // Cool Bos, Cool Bos, Cool Bos
```

Another pattern you may see is developers using an underscore `_` in place of `()`:

JavaScript

```
names.map(_ => `Cool Bos`);
```

We call this a *throwaway variable* because we're actually creating a variable called `_` but not using it. It's important to note that the `_` **does not have any significance at all**. I could use any variable name here, we just throw it away.

JavaScript

```
names.map(x => `Cool Bos`);
names.map(WESBOS => `Cool Bos`);
names.map(_ya___Yayayayay => `Cool Bos`);
names.map(do_yaget_the_point => `Cool Bos`);
```

Personally I prefer to use `() =>` over `_ =>` when there are no params but I'll let you make that decision on your own.

Arrow Functions are Always Anonymous Functions

Another thing we need to know about arrow functions, at least right now, they may change this in future versions of JavaScript, is that arrow functions are always anonymous functions.

What is an anonymous function? Actually what's a named function?

A named function is something like this:

JavaScript

```
function sayMyName(name) {
  alert(`Hello ${name}`);
}
```

The benefit to using a named function is that if you have a stack trace, which means if you have an error and you want to figure out, where did this go wrong, sometimes a line number as to where it happened isn't very helpful, so you need to know actually the name of the function that it got called in.

If you use an arrow function, **you cannot name them**. None of our arrow functions have a name.

You can, however, put them in a variable. If I were to say something like this, and pass it a name, and create a function declaration that way.

JavaScript

```
const sayMyName = (name) => {alert(`Hello ${name}!`)}

sayMyName('Wes');
```

The thing we need to know about that is it is an anonymous function and it will not give us very good stack traces. However, if you're not too concerned with that, then you can absolutely go ahead.

This entry was posted in ES6, JavaScript. Bookmark the permalink.

12 Responses to *JavaScript Arrow Functions Introduction*

Jason Karns *says:*

September 8, 2016 at 5:16 pm

In regards to using `_` as a placeholder: although using `_` as a throwaway placeholder doesn't have explicit meaning in JavaScript, it *is* a convention across many other languages (haskell, ruby, etc). So it would be more immediately obvious to polyglots than using something other than `_`. Indeed, Ruby has special handling of `_` such that both the "Duplicate Variable" error and "Unused Variable" warning are suppressed. So the convention is hardcoded in at least one language.

And, of course, `_` also has special meaning in most REPLs (node and irb included) wherein `_` always contains the value of the previous expression.

Reply

Jason Karns *says:*

September 8, 2016 at 5:21 pm

I forgot to mention that lodash also has special handling for using `_` as a placeholder to its functions (specifically the partial application utilities). Of course, with lodash, the placeholder must be lodash itself, not the literal `_` variable name (though they are usually one and the same).

Reply

Jebin *says:*

September 8, 2016 at 9:53 pm

Adding to these – arrow functions cannot be generators.

Reply

Nyamka *says:*

September 9, 2016 at 11:03 pm

Thanks for the article. It was easy and concise.

Reply

Panayiotis Velisarakos *says:*

September 10, 2016 at 12:19 pm

This must be the best explanation of the arrow functions. Thanks for sharing!

Reply

Pingback: *JavaScript Arrow Functions Introduction — Front-End Front*

Pingback: *Дайджест свежих материалов из мира фронтенда за последнюю неделю №227 (4 — 11 сентября 2016) - itfm.pro*

Christopher Fujino *says:*

September 12, 2016 at 8:13 pm

It's important to note also that an arrow function uses `this` from the surrounding code, does not create its own. See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Reply

George Mauer *says:*

July 16, 2017 at 12:34 pm

I'm not sure what your point is about named functions. Browser stacktraces are pretty clever when it comes to printing anonymous function names from variables. In chrome latest:

```
const doThrow = () => { throw Error("ow") }
> undefined
doThrow()
> VM207:1 Uncaught Error: ow
> at doThrow (:1:31)
> at :1:1
function doThrow2() { throw Error("oh") }
> undefined
doThrow2()
> VM282:1 Uncaught Error: oh
> at doThrow2 (:1:29)
> at :1:1
```

Reply

Ashwin Dixit *says:*

August 10, 2017 at 4:21 pm

Thanks for this well-written and concise tutorial.

The code doesn't show properly in my browser, because of the heavy styling.
I have to select the code samples to make them visible.
It would be awesome if you could make it more readable.

Reply

Anh Tran *says:*

September 23, 2017 at 12:45 am

Just started to learn ES6 and love arrow functions a lot. It helps making the code cleaner and sometimes, easier to read it. Something like x => x + 2 is much more cleaner than function(x) { return x + 2; }.

Reply

Pingback: *JavaScript Arrow functions | The Agile Warrior*

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

Post Comment

Notify me of follow-up comments by email.

Notify me of new posts by email.