



An Introduction to JavaScript ES6 Arrow Functions

September 21, 2015 / in ES2015/ES6, JavaScript Language / by Alex Gorbachev

Like 42

Tweet

Share

Share

<https://www.addtoany.com/share?url=https%3A%2F%2Fstrongloop.com>

introduction-to-javascript-es6-arrow-functions%2F&title=An%20Introduction%20to%20JavaScript%20ES6%20
Fat arrow functions, also known as just arrow functions, are a brand new feature in ECMAScript 2015 (formerly ES6).

Rumor has it that the `=>` syntax was adopted over `->` due to CoffeeScript [<http://coffeescript.org/>] influence and the similarity of sharing `this` context.

Arrow functions serve two main purposes: more concise syntax and sharing lexical `this` with the parent scope. Let's examine each in detail.

New Function Syntax

Classical JavaScript function syntax doesn't provide for any flexibility, be that a 1 statement function or an unfortunate multi-page function. Every time you need a function you have to type out the dreaded `function () {}`. More concise function syntax was one of the many reasons why CoffeeScript [<http://coffeescript.org/>] gained so much

Questions?

Contact us

for training and professional services to see how Node.js can accelerate your mobile, IoT, and next-generation web API.

Recent Posts

Introducing Raymon
Cat Person who Occ
Evangelizes
January 31, 2017 - 7:01

The Future of Node.js
"Community Focus
and Inclusive Node
January 24, 2017 - 3:21

Introducing Joe Sep
Lead Developer Evar
Friends Call Me Joe
Can Too"
January 24, 2017 - 2:41

Categories

API Connect (15)

API Tip (11)

Arc (24)

BACN (3)

Case Studies (3)

Cloud (11)

Community (251)

momentum back in the day. This is especially pronounced in the case of tiny callback functions. Lets look at a Promise chain:

```
function getVerifiedToken(selector) {
  return getUsers(selector)
    .then(function (users) { return users[0]; })
    .then(verifyUser)
    .then(function (user, verifiedToken) { return verifiedToken; })
    .catch(function (err) { log(err.stack); });
}
```

Above is more or less plausible piece of code written using classical JavaScript `function` syntax. Here is what the same code could look like rewritten using the arrow syntax:

```
function getVerifiedToken(selector) {
  return getUsers(selector)
    .then(users => users[0])
    .then(verifyUser)
    .then((user, verifiedToken) => verifiedToken)
    .catch(err => log(err.stack));
}
```

A few important things to notice here:

1. We've lost `function` and `{ }` because all of our callback functions are one liners.
2. We've lost `()` around the argument list when there's just one argument (rest arguments are an exception, eg `(...args) => ...`)
3. We've lost the `return` keyword because when omitting `{ }`, single line arrow functions perform an implicit return (these functions are often referred to as **lambda functions** in other languages).

It's important to reinforce the last point. **Implicit return only happens for single statement arrow functions.** When arrow function is declared with `{ }`, even if it's a single statement, **implicit return does not happen**:

[Espanol](#) (7)
[Events](#) (30)
[Express](#) (21)
[How-To](#) (162)
[JavaScript Language](#) (20)
[ES2015/ES6](#) (6)
[LoopBack](#) (127)
[Mobile](#) (32)
[News](#) (125)
[Node core](#) (6)
[Node DevOps](#) (22)
[Node.js v0.12](#) (17)
[Performance Tip](#) (11)
[Portuguese](#) (31)
[Press](#) (14)
[Product](#) (30)
[API Gateway](#) (2)
[Resources](#) (8)
[StrongNode](#) (7)
[Uncategorized](#) (2)

Archives

[February 2017](#)
[January 2017](#)
[December 2016](#)
[November 2016](#)
[October 2016](#)
[September 2016](#)
[August 2016](#)
[July 2016](#)
[June 2016](#)
[May 2016](#)
[April 2016](#)
[March 2016](#)
[February 2016](#)
[January 2016](#)
[December 2015](#)
[November 2015](#)
[October 2015](#)
[September 2015](#)
[August 2015](#)
[July 2015](#)
[June 2015](#)
[May 2015](#)
[April 2015](#)

```
const getVerifiedToken = selector => {
  return getUsers()
    .then(users => users[0])
    .then(verifyUser)
    .then((user, verifiedToken) => verifiedToken)
    .catch(err => log(err.stack));
}
```

Here's the really fun bit. Because our function has only one statement, we can still get rid of the `{ }` and it will look almost exactly like CoffeeScript [<http://coffeescript.org/>] syntax:

```
const getVerifiedToken = selector =>
  getUsers()
    .then(users => users[0])
    .then(verifyUser)
    .then((user, verifiedToken) => verifiedToken)
    .catch(err => log(err.stack));
```

Yep, the example above is completely valid ES2015 syntax (I was also surprised that it compiles fine [<http://babeljs.io/repl/#?>]). When we talk about single statement arrow functions, it doesn't mean the statement can't be spread out to multiple lines for better comprehension.

There's one caveat, however, with omitting `{ }` from arrow functions – how do you return an empty object, eg `{ }`?

```
const emptyObject = () => {};  
emptyObject(); // ?
```

Unfortunately there's no way to distinguish between empty block `{ }` and an object `{ }`. Because of that `emptyObject()` evaluates to `undefined` and `{ }` interpreted as empty block. To return an empty object from fat arrow functions you have to surround it with brackets like so `({ })`:

```
const emptyObject = () => ( {} );  
emptyObject(); // {}
```

[March 2015](#)

[February 2015](#)

[January 2015](#)

[December 2014](#)

[November 2014](#)

[October 2014](#)

[September 2014](#)

[August 2014](#)

[July 2014](#)

[June 2014](#)

[May 2014](#)

[April 2014](#)

[March 2014](#)

[February 2014](#)

[January 2014](#)

[December 2013](#)

[November 2013](#)

[October 2013](#)

[September 2013](#)

[August 2013](#)

[July 2013](#)

[June 2013](#)

[May 2013](#)

[April 2013](#)

[March 2013](#)

Here's all of the above together:

```
function () { return 1; }  
() => { return 1; }  
() => 1  
  
function (a) { return a * 2; }  
(a) => { return a * 2; }  
(a) => a * 2  
a => a * 2  
  
function (a, b) { return a * b; }  
(a, b) => { return a * b; }  
(a, b) => a * b  
  
function () { return arguments[0]; }  
(...args) => args[0]  
  
() => {} // undefined  
() => ({} ) // {}
```

Lexical this

The story of clobbering `this` in JavaScript is a really old one. Each `function` in JavaScript defines its own `this` context, which is as easy to get around as it is annoying. The example below tries to display a clock that updates every second using jQuery:

```
$('.current-time').each(function () {  
  setInterval(function () {  
    $(this).text(Date.now());  
  }, 1000);  
});
```

When attempting to reference the DOM element `this` set by `each` in the `setInterval` callback, we unfortunately get a brand new `this` that belongs to the callback itself. A common way around this is to declare `that` or `self` variable:

```
$('.current-time').each(function () {  
  var self = this;  
  
  setInterval(function () {  
    $(self).text(Date.now());  
  }, 1000);  
});
```

```
    }, 1000);  
});
```

The fat arrow functions allow you to solve this problem because they don't introduce their own `this`:

```
$('.current-time').each(function () {  
  setInterval(() => $(this).text(Date.now()), 1000);  
});
```

What about arguments?

One of the caveats with arrow functions is that they also don't have their own `arguments` variable like regular functions:

```
function log(msg) {  
  const print = () => console.log(arguments[0]);  
  print(`LOG: ${msg}`);  
}  
  
log('hello'); // hello
```

To reiterate, fat arrow functions don't have their own `this` and `arguments`. Having said that, you can still get all arguments passed into the arrow functions using rest parameters (also known as spread operator):

```
function log(msg) {  
  const print = (...args) => console.log(args[0]);  
  print(`LOG: ${msg}`);  
}  
  
log('hello'); // LOG: hello
```

What about yield?

Fat arrow functions can't be used as generators. That's it – no exceptions, no caveats and no workarounds.

Bottom Line

Fat arrow functions are one of my favorite additions to JavaScript. It might be very tempting to just start using `=>` instead of `function` everywhere. I've seen whole libraries written just using `=>` and I don't think it's the right thing to do because of the special features that `=>` introduces. I recommend using arrow functions only in places where you explicitly want to use the new features:

- Single statement functions that immediately return (lambdas)
- Functions that need to work with parent scope `this`

ES6 Today

How can you take advantage of ES6 features today? Using transpilers in the last couple of years has become the norm. People and large companies no longer shy away. Babel [\[https://babeljs.io/\]](https://babeljs.io/) is an ES6 to ES5 transpiler that supports all of the ES6 features.

If you are using something like Browserify [\[http://browserify.org/\]](http://browserify.org/) in your JavaScript build pipeline, adding Babel [\[https://babeljs.io/\]](https://babeljs.io/) transpilation takes only a couple of minutes [\[https://babeljs.io/docs/using-babel/#browserify\]](https://babeljs.io/docs/using-babel/#browserify). There is, of course, support for pretty much every common Node.js build system like Gulp, Grunt and many others.

What About The Browsers?

The majority of browsers are catching up [\[https://kangax.github.io/compat-table/es6/\]](https://kangax.github.io/compat-table/es6/) on implementing new features but not one has full support. Does that mean

you have to wait? It depends. It's a good idea to begin using the language features that will be universally available in 1-2 years so that you are comfortable with them when the time comes. On the other hand, if you feel the need for 100% control over the source code, you should stick with ES5 for now.

You may also be interested in...

- [Write Your Own Co Using ES2015 Generators](https://strongloop.com/strongblog/write-your-own-co-using-es2015-generators/)
[\[https://strongloop.com/strongblog/write-your-own-co-using-es2015-generators/\]](https://strongloop.com/strongblog/write-your-own-co-using-es2015-generators/)
- [An Introduction To JavaScript ES6 Modules](https://strongloop.com/strongblog/an-introduction-to-javascript-es6-modules/)
[\[https://strongloop.com/strongblog/an-introduction-to-javascript-es6-modules/\]](https://strongloop.com/strongblog/an-introduction-to-javascript-es6-modules/)
- [Introducing IBM Node.js Tools for Eclipse Beta](https://strongloop.com/strongblog/introducing-ibm-node-js-tools-for-eclipse-beta/)
[\[https://strongloop.com/strongblog/introducing-ibm-node-js-tools-for-eclipse-beta/\]](https://strongloop.com/strongblog/introducing-ibm-node-js-tools-for-eclipse-beta/)

Like 42  Tweet   Share 

Share

[\https://www.addtoany.com/share?url=https%3A%2F%2Fstrongloop.com/introduction-to-javascript-es6-arrow-functions%2F&title=An%20Introduction%20to%20JavaScript%20ES6%2

[< Previous Post](#)

[Next Post >](#)

Compose APIs, Build, Deploy and Monitor Node

Install API Connect

```
$ npm install -g apiconnect
```

Create your API Connect project

```
$ apic loopback
```

Launch API Designer

```
$ cd your-project
```

```
$ apic edit
```

API Designer will run on your local browser

An IBM Bluemix account is required to use the Designer. [Register for IBM Bluemix](#)

[Click To Get Started](#)



It's FREE!