ES6 Arrow functions - The new fat and concise syntax in JavaScript

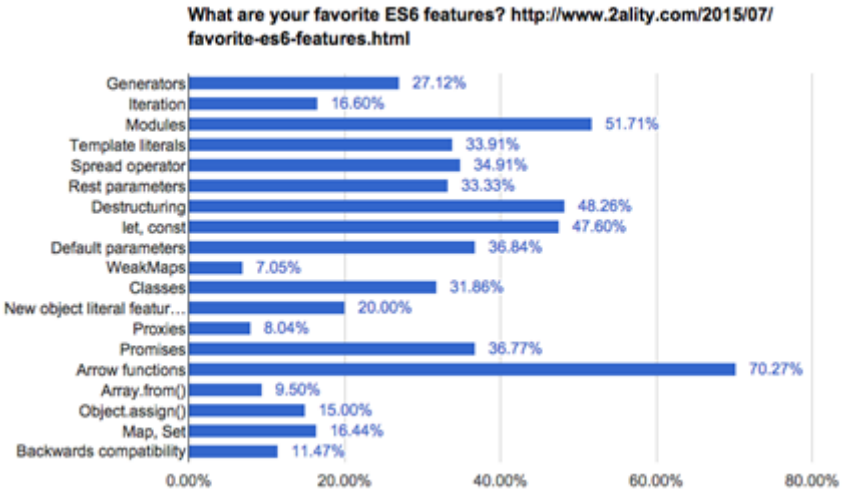
Kyle Pennell, 24 Jun 2016 CPOL

Arrow functions are a new ES6 syntax for writing JavaScript functions. They will save developers time and simplify function scope.

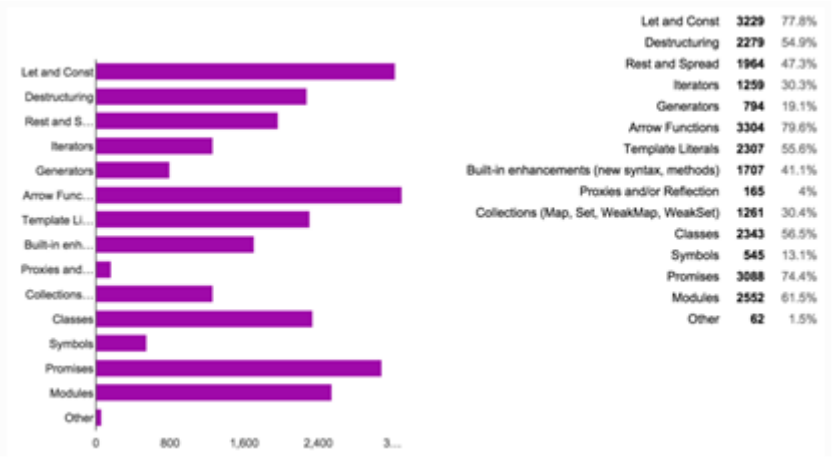
Editorial Note

This article is in the Product Showcase section for our sponsors at CodeProject. These articles are intended to provide you with information on products and services that we consider useful and of value to developers.

Arrow functions are a new ES6 syntax for writing JavaScript functions. They will save developers time and simplify function scope. Surveys show they are the most popular ES6 feature:



***Source:** Axel Rauschmayer survey on favorite ES6 features*



***Source:** Ponyfoo's survey on the most commonly used ES6 features*

The good news is that many major modern browsers support the use of arrow functions.

This post will cover the details of Arrow functions, specifically, how to use them, common syntaxes, common use cases, and gotchas/pitfalls.

What are Arrow functions?

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.

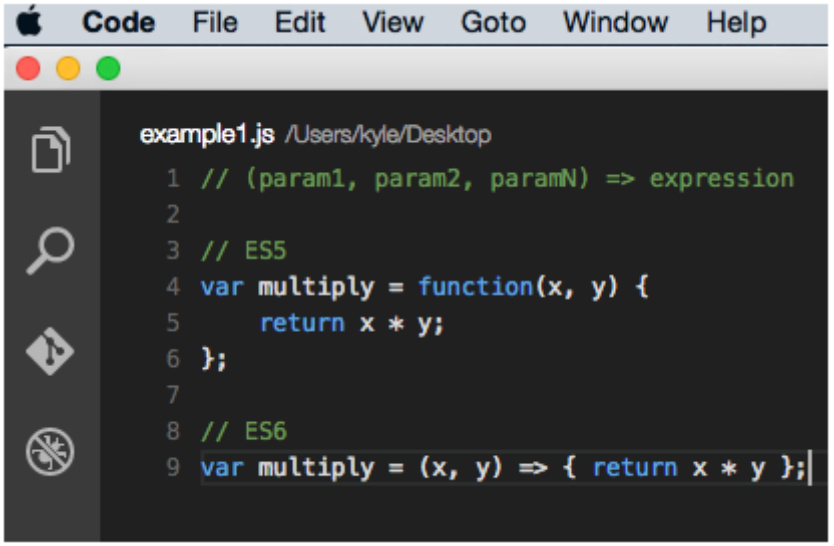
Arrow functions make our code more concise, and simplify function scoping and the this keyword. They are one-line mini functions which work much like Lambdas in other languages like C# or Python. (See also lambdas in JavaScript). By using arrow function we avoid having to type the `function` keyword, `return` keyword (it's implicit in arrow functions), and curly brackets.

Using Arrow Functions

There are a variety of syntaxes available in arrow functions. EcmaScript.org has a thorough list of the syntaxes and so does MDN. We'll cover the common ones here to get you started.

Let's compare how ES5 code with function expressions can now be written in ES6 using arrow functions.

Basic Syntax with Multiple Parameters (from MDN)



Code Example: http://codepen.io/DevelopIntelligenceBoulder/pen/wMdPoj?editors=101

The arrow function example above allows a developer to accomplish the same result with fewer lines of code and approximately half of the typing.

Curly brackets are not required if only one expression is present. The preceding example could also be written as:

Hide Copy Code

```
var multiply = (x, y) => x * y;
```

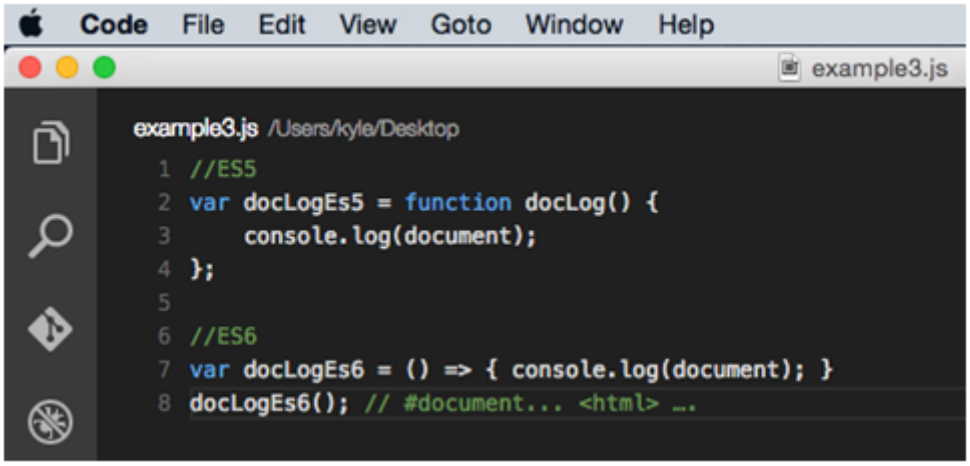
Basic Syntax with One Parameter

Parentheses are optional when only one parameter is present

http://codepen.io/DevelopIntelligenceBoulder/pen/PZmOWQ?editors=101

No Parameters

Parentheses are required when no parameters are present.



Code Example: http://codepen.io/DevelopIntelligenceBoulder/pen/GomOWO?editors=101

Object Literal Syntax

Arrow functions, like function expressions, can be used to return an object literal expression. The only caveat is that the body needs to be wrapped in parentheses, in order to distinguish between a block and an object (both of which use curly brackets).

Code Example: ***http://codepen.io/DevelopIntelligenceBoulder/pen/zrwPwx?editors=101***

Use Cases for Arrow Functions

Now that we've covered the basic syntaxes, let's get into how arrow functions are used.

One common use case for arrow functions is array manipulations and the like. It's common that you'll need to map or reduce an array. Take this simple array of objects:

Hide Copy Code

```
var smartPhones = [
    { name: 'iphone', price: 649 },
    { name: 'Galaxy S6', price: 576 },
    { name: 'Galaxy Note 5', price: 489 }
];
```

We could create an array of objects with just the names or prices by doing this in ES5:

Hide Copy Code

```
// ES5
console.log(smartPhones.map(function(smartPhone) {
  return smartPhone.price;
})); // [649, 576, 489]
```

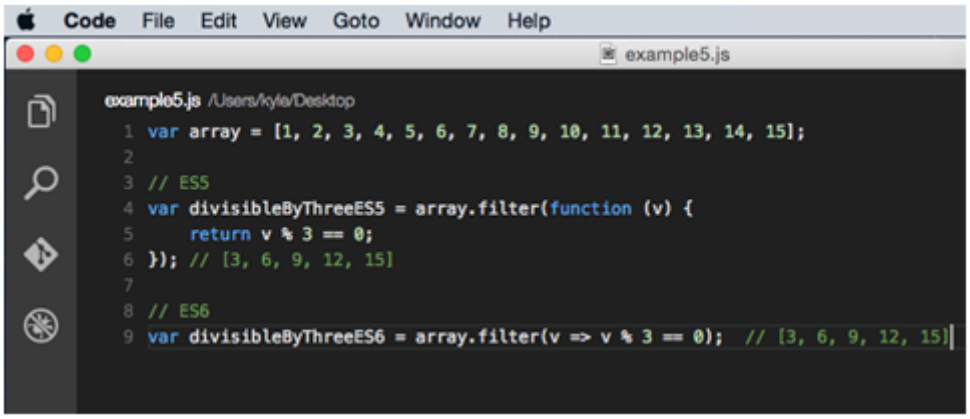
An arrow function is more concise and easier to read:

Hide Copy Code

```
// ES6
console.log(smartPhones.map(smartPhone => smartPhone.price)); // [649, 576, 489]
```

Code Example: ***http://codepen.io/DevelopIntelligenceBoulder/pen/jWmamX?editors=101***

Here's another example using the array filter method:



Code Example: ***http://codepen.io/DevelopIntelligenceBoulder/pen/RrVjgL?editors=101***

Promises and Callbacks

Code that makes use of asynchronous callbacks or promises often contains a great deal of `function` and `return` keywords. When using promises, these function expressions will be used for chaining. Here's a simple example of chaining promises from the MSDN docs:

Hide Copy Code

```
// ES5
aAsync().then(function() {
  return bAsync();
}).then(function() {
  return cAsync();
}).done(function() {
  finish();
});
```

This code is simplified, and arguably easier to read using arrow functions:

Hide Copy Code

```
// ES6
aAsync().then(() => bAsync()).then(() => cAsync()).done(() => finish);
```

Arrow functions should similarly simplify callback-laden NodeJS code.

What's the meaning of `this`?!

The other benefit of using arrow functions with promises/callbacks is that it reduces the confusion surrounding the `this` keyword. In code with multiple nested functions, it can be difficult to keep track of and remember to bind the correct `this` context. In ES5, you can use workarounds like the .bind method (which is slow) or creating a closure using `var self = this;`.

Because arrow functions allow you to retain the scope of the caller inside the function, you don't need to create `self=this` closures or use bind.

Developer Jack Franklin provides an excellent practical example of using the arrow function lexical this to simplify a promise:

Without Arrow functions, the promise code needs to be written something like this:

Hide Copy Code

```
// ES5
API.prototype.get = function(resource) {
    var self = this;
    return new Promise(function(resolve, reject) {
      http.get(self.uri + resource, function(data) {
        resolve(data);
      });
    });
```

Using an arrow function, the same result can be achieved more concisely and clearly:

Hide Copy Code

```
// ES6
 API.prototype.get = function(resource) {
   return new Promise((resolve, reject) => {
     http.get(this.uri + resource, function(data) {
       resolve(data);
     });
   });
 };
```

You can use function expressions if you need a dynamic `this` and arrow functions for a lexical `this`.

Gotchas and Pitfalls of Arrow Functions

The new arrow functions bring a helpful function syntax to ECMAScript, but as with any new feature, they come with their own pitfalls and gotchas.

Kyle Simpson, a JavaScript developer and writer, felt there were enough pitfalls with Arrow Functions to warrant this flow chart when deciding to use them. He argues there are too many confusing rules/syntaxes with arrow functions. Others have suggested that using arrow functions saves typing but ultimately makes code more difficult to read. All those `function` and `return` statements might make it easier to read multiple nested functions or just function expressions in general.

Developer opinions vary on just about everything, including arrow functions. For the sake of brevity, here are a couple things you need to watch out for when using arrow functions.

More about `this`

As was mentioned previously, the `this` keyword works differently in arrow functions. The methods call(), apply(), and bind() will not change the value of `this` in arrow functions. (In fact, the value of `this` inside of a function simply can't be changed–it will be the same value as when the function was called.) If you need to bind to a different value, you'll need to use a function expression.

Constructors

Arrow functions cannot be used as constructors as other functions can. Don't use them to create similar objects as you would with other functions. If you attempt to use `new` with an arrow function, it will throw an error. Arrow functions, like built-in functions (aka methods), don't have a prototype property or other internal methods. Because constructors are generally used to create classlike objects in JavaScript, you should use the new ES6 classes instead.

Generators

Arrow functions are designed to be lightweight and cannot be used as generators. Using the `yield` keyword in ES6 will throw an error. Use ES6 generators instead.

Arguments object

Arrow functions do not have the local variable arguments as do other functions. The `arguments` object is an array-like object that allows developers to dynamically discover and access a function's arguments. This is helpful because JavaScript functions can take an unlimited number of arguments. Arrow functions do not have this object.

How much use is there for Arrow Functions?

Arrow functions have been called one of the quickest wins with ES6. Developer Lars Schöning lays out how his team decided where to use arrow functions:

- Use `function` in the global scope and for `Object.prototype` properties.
- Use `class` for object constructors.
- Use `=>` everywhere else.

Arrow functions, like `let` and `const`, will likely become the default functions unless function expressions or declarations are necessary. To get a sense for how much arrow functions can be used, Kevin Smith, counted function expressions in various popular libraries/frameworks and found that roughly 55% of function expressions would be candidates for arrow functions.

Arrow functions are here (c'mon Safari!)–they are powerful, concise, and developers love them. Perhaps it's time for you to start using them!

This article is part of the web development series from Microsoft tech evangelists and engineers on practical JavaScript learning, open source projects, and interoperability best practices including Microsoft Edge browser.

We encourage you to test across browsers and devices including Microsoft Edge – the default browser for Windows 10 – with free tools on dev.microsoftedge.com, including virtual machines to test Microsoft Edge and versions of IE6 through IE11. Also, visit the Edge blog to stay updated and informed from Microsoft developers and experts.

License

Share

About the Author

Kyle Pennell

United States

Kyle is a Technical Instructor at DevelopIntelligence. He spends his time reading, coding, biking, and exploring live music in Denver. He enjoys trying to make technical concepts more approachable and likes tinkering with music and mapping APIs. You can follow his musings @kyleapennell.

Comments and Discussions