# ECMAScript 6: arrow functions and method definitions

[2012-04-12] esnext, dev, javascript



**Follow-up blog post from 2013-08-11:** "Callable entities in ECMAScript 6".

In JavaScript, one aspect of creating a function inside a method is difficult to get right: handling the special variable `this`. ECMAScript.next will make things easy by introducing two constructs: arrow functions and method definitions. This blog posts explains what they are and how they help.

## 1. Terminology

For this blog post, we distinguish two kinds of callable entities:

- A *subroutine* exists on its own and is called directly. In general, "function" would be a better word, but that has a broader meaning in JavaScript. Hence, JavaScripters normally say "non-method function" to a subroutine. I'm only using the term "subroutine", because I couldn't find a better one (other possibilities are: callback, procedure).
- A *method* is part of an object `o` and called via an object (which isn't necessarily the same object as `o`).

In JavaScript, both subroutines and methods are implemented by functions. For example:

```
var obj = {
    myMethod: function () {
        setTimeout(function () { ... }, 0);
    }
}
```

`myMethod` is a method, the first argument of `setTimeout()` is a subroutine. Both are implemented by functions. Before we can explain the implications of that, we need to make a few more definitions. Whenever a function is called, it is in two kinds of *scopes* (or contexts): Its *lexical scopes* are the syntactic constructs that surround it (a trait of the source code or the *lexicon*). Its *dynamic scopes* are the function that called it, the function that called that function, etc. Note the nesting that occurs in both cases. *Free variables* are variables that aren't declared within a function. If such variables are read or written, JavaScript looks for them in the surrounding lexical scopes.

Functions work well as method implementations: They have a special variable called `this` that refers to the object via which the method has been invoked. In contrast to other free variables, `this` isn't looked up in the surrounding lexical scopes, it is handed to the function via the invocation. As the function receives `this` dynamically, it is called *dynamic `this`*.

Functions don't work well as implementations of subroutines, because `this` is still dynamic. The subroutine call sets it to `undefined` in strict mode [1] and to the global object, otherwise. That is unfortunate, because the subroutine has no use for its own `this`, but it shadows the `this` of the surrounding method, making it inaccessible. For example:

```
var jane = {
    name: "Jane",

    logHello: function (friends) {
        var that = this;  // (*)
        friends.forEach(function (friend) {
            console.log(that.name + " says hello to " + friend)
        });
    }
}
```

The argument of `forEach` is a subroutine. You need the assignment at (*) so that it can access `logHello`'s `this`. Clearly, subroutines should have *lexical `this`*, meaning that `this` should be treated the same as other free variables and looked up in the enclosing lexical scopes. `that = this` is a good work-around. It simulates lexical `this`, if you will. Another work-around is to use `bind`:

```
var jane = {
    name: "Jane",

    logHello: function (friends) {
        friends.forEach(function (friend) {
            console.log(this.name + " says hello to " + friend)
        }.bind(this));
    }
}
```

Now the argument of `forEach` has a fixed value for `this`. I can't be changed, not even via `call` or `apply`. There are three problems with any simulation of lexical `this`:

- You have to know how JavaScript's quirky `this` works (which you should neither want to nor need to).
- You have to constantly be alert as to when to simulate lexical `this`. That choice should be automatic and not require extra thought. A simulation incurs a performance and memory cost, so you'll want to avoid it if you don't need it.
- There is more to type and more visual clutter.

Note that even CoffeeScript's fairly elegant arrow functions have problems #1 and #2.

## 2. Arrow functions

ECMAScript.next's *arrow functions* are better suited for defining subroutines than normal functions, because they have lexical `this`. Using one for `forEach` looks as follows.

```
let jane = {
    name: "Jane",

    logHello: function (friends) {
        friends.forEach(friend => {
            console.log(this.name + " says hello to " + friend)
        });
    }
}
```

The "fat" arrow `=>` (as opposed to the thin arrow `->`) was chosen to be compatible with CoffeeScript, whose fat arrow functions are very similar.

Specifying arguments:

```
    () => { ... } // no argument
     x => { ... } // one argument
(x, y) => { ... } // several arguments
```

Specifying a body:

```
x => { return x * x }  // block
x => x * x  // expression, equivalent to previous line
```

The statement block behaves like a normal function body. For example, you need `return` to give back a value. With an expression body, the expression is always implicitly returned. Having a block body in addition to an expression body means that if you want the expression to be an object literal, you have to put it in parentheses.

Note how much an arrow function with an expression body can reduce verbosity. Compare:

```
let squares = [ 1, 2, 3 ].map(function (x) { return x * x });
let squares = [ 1, 2, 3 ].map(x => x * x);
```

### 2.1. Implementing lexical `this`

For arrow functions, lexical `this` is implemented as follows. The arrow function

```
x => x + this.y
```

is mostly syntactic sugar for

```
function (x) { return x + this.y }.bind(this)
```

That expression creates two functions: First, the original anonymous function with the parameter `x` and dynamic `this`. Second, the *bound function* that is the result of `bind`. While an arrow function behaves as if it had been created via `bind`, it consumes less memory: Only a single entity is created, a specialized function where `this` is directly bound to the `this` of the surrounding function.

### 2.2. Arrow functions versus normal functions

An arrow function is different from a normal function in only three ways: First, it always has a bound `this`. Second, it can't be used as a constructor: There is no internal method [[Construct]] (that allows a normal function to be invoked via `new`) and no property `prototype`. Therefore, `new (() => {})` throws an error. Third, as arrow functions are an ECMAScript.next-only construct, they can rely on new-style argument handling (parameter default values, rest parameters, etc.) and don't support the special variable `arguments`. Nor do they have to, because the new mechanisms can do everything that `arguments` can.

Apart from these simplifications, there is no observable difference between an arrow function and a normal function. For example, `typeof` and `instanceof` can be used as before:

```
> typeof () => {}
'function'

> () => {} instanceof Function
true
```

## 3. Syntactic variants under discussion

The following syntactic variants are still being discussed and might not be added to ECMAScript.next.

- Omitting the parameters:

```
=> { ... }
```

With JavaScript's automatic semicolon insertion [2], there is a risk of such an expression being wrongly considered as continuing a previous line. Take, for example, the following code.

```
var x = 3 + a
=> 5
```

These two lines are interpreted as

```
var x = 3 + (a => 5);
```

However, arrow functions will usually appear in expression context, nested inside a statement. Hence, I wouldn't expect semicolon insertion to be much of a problem. If JavaScript had significant newlines [3] (like CoffeeScript) then the problem would go away completely.

- **Omitting the body:**

```
x =>
```

That's a function with a single parameter that always returns `undefined`. It is a synonym for the `void` operator [4]. I'm not sure how useful that is.

- **Named arrow functions:** JavaScript already has *named function expressions*, where you give a function a name so that it can invoke itself. That name is local to that function, it doesn't leak into any surrounding scopes. Named arrow functions would work the same. For example:

```
let fac = me(n) => {
    if (n <= 0) {
        return 1;
    } else {
        return n * me(n-1);
    }
}
console.log(me); // ReferenceError: me is not defined
```

## 3.1. Parsing arrow functions

Most JavaScript parsers have a two-token look-ahead. How then is such a parser supposed to distinguish between the following two expressions?

```
(x, y, z)
(x, y, z) => {}
```

The first expression is the comma operator applied to three variables, in parentheses. The second expression is an arrow function. If you want to distinguish them at the beginning (at the

opening parenthesis), you have to look head many tokens, until you either encounter the arrow or not.

To parse both of the above with a limited look-ahead, one uses a trick called *cover grammar*: One creates a grammar rule that covers both use cases, parses and then performs post-processing. If an arrow follows the closing parenthesis, some previously parsed things will raise an error and the parsed construct is used as the formal parameter list of an arrow function. If no arrow follows, other previously parsed things will raise an error and the parsed construct is an expression in parentheses. Some things can only be done in a parenthesized expression:

```
(foo(123))
```

Other things can only be done in a parameter list. For example, declaring a rest parameter:

```
(a, b, ...rest)
```

Several things can be done in both contexts:

```
(title = "no title")
```

The above is an assignment in expression context and a declaration of a default parameter value in arrow function context.

## 4. Possible arrow function feature: optional dynamic this

One arrow function feature, that has been deferred and might still be added, is the ability to switch to dynamic `this`. The use case for that feature is as follows. In jQuery, some arguments are subroutines that have `this` as an implicit parameter:

```
$(".someCssClass").each(function (i) { console.log(this) });
```

You currently cannot write `each`'s argument as an arrow function, because `call` and `apply` cannot override the arrow function's bound value for `this`. You would need to switch to dynamic `this` to do so:

```
$(".someCssClass").each((this, i) => { console.log(this) });
```

Normally, no parameter can have the name `this`, so the above is a good marker for an arrow function with dynamic `this`.

Shouldn't there be a simpler solution for optional dynamic `this`? Alas, two seemingly simpler approaches won't work.

### 4.1. Non-solution: switching between dynamic and lexical `this`

One can conceivably change between dynamic and lexical `this`, depending on how a function is invoked. If it is invoked as a method, use dynamic `this`. If it is invoked as a subroutine, use lexical `this`. The problem is that that can lead to the function being invoked the wrong way: If the function is intended to be a subroutine, but invoked as a method, it will lose the link to the `this` of its lexical context. If the function is intended to be a method, but is invoked as a function, it assumes that it accesses its instance, but will instead access the `this` of its surroundings. ECMAScript 5 strict mode solves this issue by failing quickly: If a method is invoked as a subroutine, its `this` is undefined, leading to an error the first time it accesses a property. And subroutines can't even do anything with their `this`, unless it is bound, in which case it will never change. ECMAScript.next will work the same, but make it harder to use a function the wrong way: Arrow functions will be the default subroutine and have fixed lexical `this`, method definitions (see below) will be the default method and can only be invoked via an object (with the problematic exception of methods that are extracted from their object).

Another problem with switching between the two kinds of `this` is security-related: You can't control how a function you write will be used by clients, opening the door to inadvertently exposed secrets. Example: Let's pretend there are "thin arrow functions" (defined via `->`) that switch between dynamic and lexical `this`, on the fly.

```
let objectCreator = {
    create: function (secret) {
        return {
            secret: secret,
            getSecret: () -> {
                return this.secret;
            }
        };
    },
    secret: "abc"
}
```

This is how one would normally use `obj`:

```
let obj = objectCreator.create("xyz");
// dynamic this:
console.log(obj.getSecret()); // xyz
```

This is how an attacker could get access to `objectCreator.secret`:

```
let obj = objectCreator.create("xyz");
let func = obj.getSecret;
// lexical this:
console.log(func()); // abc
```

## 4.2. Non-solution: Let `call` or `apply` override the bound value of `this`.

That is problematic, because `call` or `apply` can accidentally break an arrow function that relies on `this` being lexical. Hence, this is too brittle a solution.

### 4.3. Arguing in favor of simplicity

I don't think that we need optional dynamic `this` for arrow functions, it partially destroys their simplicity. Furthermore, having dynamic `this` available in something that is not a method goes against the grain of object-orientation. Every time an API hands an argument to a subroutine via `this`, it should instead introduce a real parameter. jQuery already allows you to do that for the `each` method:

```
$(".someCssClass").each(function (i, ele) { console.log(ele) });
```

Another option is to use a normal function. Or you can wrap a normal function around an arrow function and pass `this` from the former to the latter, as an additional (prefixed) parameter. In the following code, method `curryThis` performs such wrapping.

```
$(".someCssClass").each(
    (that, ele) { console.log(that) }.curryThis()
);
```

`curryThis` can be implemented as follows [5]:

```
Function.prototype.curryThis = function () {
    var f = this;
    return function () {
        var a = Array.prototype.slice.call(arguments);
        a.unshift(this);
        return f.apply(null, a);
    };
};
```

## 5. Method definitions

With arrow functions, the choice between dynamic `this` and lexical `this` has become automatic:

1. Need to define a subroutine? Use an arrow function and automatically have lexical `this`.
2. Need to define a method? Use a normal function and automatically have dynamic `this`.

#2 still isn't optimal: Seeing the keyword `function` when defining a method is misleading. And you still have to think about two kinds of functions – people might accidentally use arrow functions to define methods (which will lead to errors as soon as `this` is accessed). To address this issue, ECMAScript.next gives methods their own syntactic construct: the *method definition*. It is simply a more compact way of defining a method. Method definitions also enable another ECMAScript.next feature: `super` references [6]. For those, a function needs to know which object it is stored in. A method definition automatically adds that information to a function. Method

definitions can appear in two contexts: in class declarations and in object literals.

### 5.1. Method definitions in class declarations

One proposal that is currently being discussed for ECMAScript.next is called "maximally minimal classes" – minimal syntactic sugar for current practices (background on the class discussion: [7]). An example:

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    // Method definition:
    dist() {
        return Math.sqrt((this.x*this.x)+(this.y*this.y));
    }
}
```

If accepted, class declarations would be great for newcomers and for tool support. It would also hopefully make it easier to share code between JavaScript frameworks (which currently tend to have different inheritance APIs). The foundations don't change, a class declaration will internally be translated to a constructor function. `dist` is a method definition for `Point.prototype.dist`. As you can see, there is less to type and you won't be tempted to incorrectly use an arrow function.

### 5.2. Method definitions in object literals

The first step towards having dedicated syntax for methods are class declarations, as explained above. The next step is to also provide method syntax for object literals. For example:

```
let jane = {
    name: "Jane",

    // Method definition:
    logHello(friends) {
        friends.forEach(friend => {
            console.log(this.name + " says hello to " + friend)
        });
    }
}
```

Again, you don't see that `logHello` is a function, underneath. It looks like a method and will have dynamic `this`. That leaves us with one more use case: What if you want to add methods to an existing object, say:

```
obj.method1 = function (...) { ... };
obj.method2 = function (...) { ... };
```

Here, you still see old-school functions. However, ECMAScript.next will allow you to use an object literal, instead:

```
Object.assign(obj, {
    method1(...) { ... },
    method2(...) { ... }
});
```

## 6. What do we really need?

In order to find out what is needed by actual JavaScript code, Kevin Smith surveyed a wide variety of code bases (jQuery, Facebook, Node.js, etc.). They contained a total of 20667 function expressions. Among them, he found the following distribution:

| | |
|---|---|
| **Arrow function candidates (AFCs)** | 55.7% |
| Expression body (`return` *expr*) | 8.9% |
| Object literal body (`return {...}`) | 0.14% |
| Block with single statement | 20% |
| Block with multiple statements | 26.7% |
| **Methods** (in object literal, with `this`) | 35.77% |
| **AFCs or methods** | 91.46% |
| **Functions with `this`** (outside object literal) | 8.54% |

Terminology:

- Arrow function candidates (AFCs) are function expressions without dynamic `this`; they don't refer to `this` in (the immediate scope of) the body.
- "Expression body" is a superset of "object literal body".
- Function expressions in an object literal that don't refer to `this` are considered AFCs. In that case, the surrounding object is usually a namespace.

Many functions with (dynamic) `this` are probably about adding methods to an existing object. They could then be handled by ECMAScript.next's `Object.assign`. Furthermore, as the survey was only to provide a rough estimate and the additional parsing effort was not worth it, the category "functions with `this`" in the table includes functions with a bound `this`, such as

```
function (...) { ... }.bind(this)
```

Obviously, those are actually AFCs.

These findings mean that JavaScript most urgently needs easy ways to define subroutines (with lexical `this`) and methods. ECMAScript.next therefore makes the right choices. Another

implication is that arrow functions with dynamic `this` are not that important, even for current JavaScript code. Lastly, having to parenthesize an object literal in an expression body is rarely an issue: only 0.14% of the function expressions return one.

## 7. Conclusion

You will hardly ever see old-school functions in ECMAScript.next:

- Constructor functions will (hopefully) be replaced by class declarations.
- Functions as subroutines will be replaced by arrow functions.
- Functions as methods will be replaced by method definitions.

I suspect that this clear separation of concerns is a bit difficult to get used to for experienced JavaScript programmers. But, in my opinion, it will make JavaScript a simpler language in the long run. Current functions playing triple duty is difficult to understand. However, they won't go away, they are still available whenever you need the additional power.

## 8. More material on arrow functions

- "What is the meaning of this?" by Douglas Crockford
- "JavaScript Fat City" by Angus Croll

## 9. References


1. JavaScript's strict mode: a summary
2. Automatic semicolon insertion in JavaScript
3. What JavaScript would be like with significant newlines
4. The void operator in JavaScript
5. Uncurrying `this` in JavaScript
6. A closer look at super-references in JavaScript and ECMAScript.next
7. Myth: JavaScript needs classes

Name

**Angus Croll** • 5 years ago

Nice article. I like your approach

>>Apart from these limitations, there is no observable difference between an arrow function and a normal function

Also arguments object is not referenceable and function cannot be named. (http://javascriptweblog.wor...

2 ^ v • Reply • Share ›

**Axel Rauschmayer** Mod ➚ Angus Croll • 5 years ago

Thanks! I've edited to mention the difference in parameter handling. I've also added a link to your blog post.

2 ^ v • Reply • Share ›

**Angus Croll** ➚ Axel Rauschmayer • 5 years ago

thanks Axel!

^ v • Reply • Share ›

**a1exlism** • 8 months ago

Err.... Find probs above those examples, forEach() does not work as before.

^ v • Reply • Share ›

**Ivan Fraixedes** • 2 years ago

Hi Axel,
Thanks for this post and many others.

I wanted to repot you 2 typos in code snippets

One in section 4.1
getSecret: () -> {

The arrow symbol is wrong

And in section 4.3
(that, ele) { console.log(that) }.curryThis()

The arrow is missed in the arrow function declaration

^ v • Reply • Share ›

**Andrew Hewitt** • 2 years ago

Wow, great writeup, thanks! I've been diving into ES6 lately and loving arrow functions to my surprise. Thanks for teaching me that I could define an object { methodName(){ ... } } without using "function".

I've always found the use of "function" to be odd between handing around as callbacks (subroutines) and then as methods. Now I can get rid of the word from my code almost entirely using arrow functions and that bit about defining methods in object literals while at the same time improving readability. Thanks again!

Reply • Share ›

**Henrique Silvério** • 2 years ago

Thanks for explanations, looking forward your "Exploring ES6" book. =]

Curious about the `arguments`, I run a simple test in Firefox, and it works.

```
var sum = (a, b) => {
    console.log(arguments);
    return a + b;
};
// undefined

sum(2, 2);
// 4
// Arguments { 0: 2, 1: 2, 2 more }
```

Is it a change in the arrow functions behaviors or I confusing me something?

Reply • Share ›

**PaulGrenier** • 3 years ago

You don't need to .bind(this) in your second example, Array#forEach includes a 'thisArg' parameter in the signature:
var jane = { name: "Jane",

logHello: function (friends) {
friends.forEach(function (friend) {
console.log(this.name + " says hello to " + friend)
}, this);
}
}

Reply • Share ›

**Florent** • 4 years ago

Thanks for the article.

Just note that the [].forEach() function has a second optional parameter to specify the value of `this` within the callback:
https://developer.mozilla.o...

Florent

Reply • Share ›

**Paul** • 4 years ago

Nice to see that more patterns from functional language are adopted by JavaScript.

Reply • Share ›

**Xose Lluis** • 4 years ago

Many thanks for another really excellent article (as usual).

A minor point I've realized while reading here (another excellent read):
http://dmitrysoshnikov.com/...
on how Function.bind works.

You mention that arrow functions are almost syntactic sugar for .bind, which really makes the whole thing quite more clear, but one important difference (along with the 2 functions vs 1 function thing that you mention) is that bound functions have a [[Construct]] property (at least in ES5), so (contrary to arrow functions) they can be used as constructors (in which case will use the "freshly provided this" instead of the bound one)

^ | v • Reply • Share ›

**aaronfrost** • 5 years ago

Thanks Axel. Great explanation of these two new features.

^ | v • Reply • Share ›

**Michael Smith** • 5 years ago

Nice article. I'm looking forward to being able to use arrow functions, though most of my JavaScript is client-side, so I may be waiting a while.

JavaScript is a curious language: Arrow functions aside, if you create a function in the global scope, it's guaranteed to be a method. The only way to create one of these non-method functions is to create it within another function. I realize this is trivia, but it's interesting to me because in many other languages, functions created in the global scope cannot be methods.

^ | v • Reply • Share ›

**Axel Rauschmayer** Mod ➚ Michael Smith • 5 years ago

Agree with everything except for “The only way to create one of these non-method functions is to create it within another function.” Even inside a method, a current function has all the facilities of a method. The best current implementation of a non-method function is either via that = this or via bind(this). Both of which only pick up the `this` of the surrounding scope and thus will also work in global scope (where `this` is defined and points to the global object).

^ | v • Reply • Share ›

Dr. Axel Rauschmayer
Twitter, Mastodon

## Most popular (last 30 days)

ECMAScript 6 modules: the final syntax

Classes in ECMAScript 6 (final semantics)

ECMAScript 2017 (ES8): the final feature set

Converting a value to string in JavaScript

Iterating over arrays and objects in JavaScript

ECMAScript 6's new array methods

The final feature set of ECMAScript 2016 (ES7)

ECMAScript 6: merging objects via Object.assign()

Destructuring and parameter handling in ECMAScript 6

SimpleHTTPServer: a quick

way to serve a directory