Most popular (last 30 days)

ECMAScript 2017: the final feature set

ECMAScript 6 modules: the final syntax

ES proposal: import() – dynamically importing ES modules

Making transpiled ES modules more spec-compliant

ES proposal: Shared memory and atomics

Classes in ECMAScript 6 (final semantics)

Communicating between Web Workers via MessageChannel

Most popular (all time)

ECMAScript 6 modules: the final syntax

Classes in ECMAScript 6 (final semantics)

Iterating over arrays and objects in JavaScript

ECMAScript 6's new array methods

The final feature set of ECMAScript 2016 (ES7)

WebAssembly: a binary format for the web

Basic JavaScript for the impatient programmer

Google Dart to "ultimately ... replace JavaScript"

Six nifty ES6 tricks

Google's Polymer and the future of web UI frameworks

Blog archive

► 2017 (4)

► 2016 (38)

Free email newsletter: "ES.next News"

2013-08-11

Callable entities in ECMAScript 6

Labels: dev, esnext, javascript

Update 2013-11-28: There won't be generator arrow functions in ECMAScript 6 (details).

In ECMAScript 5, one creates all callable entities via functions. ECMAScript 6 has more constructs for doing so. This blog post describes them.

1. The status quo: ECMAScript 5

In ECMAScript 5, functions do triple duty:

- As normal functions: you can directly call functions.
- As methods: you can assign a function to the property of an object and call it as a method, via that object.
- As constructors: you can invoke functions as constructors, via the `new` operator.

The three main problems with this approach are:

1. It confuses people.
2. You can use a function the wrong way, e.g. call a constructor or a method as a normal function.
3. Functions used as normal functions shadow the `this` of surrounding constructors or methods [1]. That's because `this` is always *dynamic* (provided by each function call), but should be *lexical* in this case, like normal variables that are resolved via surrounding scopes if they are not declared within a function.

Let's first look at ECMAScript 6's callable entities and then at how they help with these problems.

2. ECMAScript 6's callable entities

The following subsections explain ECMAScript 6's callable entities and what ECMAScript 5 constructs and patterns they correspond to. If you are confused about the difference between a function expression and a function declaration, consult [3].

2.1. Function expression → arrow function

In ECMAScript 5, a function used as a normal function inside a constructor or a method shadows `this`:

```
function GuiComponent() {  // constructor
    var that = this;
    var domNode = ...;
    domNode.addEventListener('click', function () {
        console.log('CLICK');
        that.handleClick();  // `this` is shadowed
    });
}
```

ECMAScript 6 has *arrow functions* [2] that have a more compact syntax and don't have their own `this`, their `this` is lexical:

(Ad, please don't block.)

90% Unlimited Downloads Choose from Over 300,000 Vectors, Graphics & Photos.

ads via Carbon

Dr. Axel Rauschmayer

Free online books by Axel

Speaking JavaScript [up to ES5]

Exploring ES6

JavaScript training:
Ecmanauten

- ► 2015 (65)
- ► 2014 (55)
- ▼ 2013 (98)
  - ► December (6)
  - ► November (3)
  - ► October (4)
  - ► September (6)
  - ▼ August (6)
    - We need intelligence, not intellect
    - Protecting objects in JavaScript
    - Why all objects are truthy in JavaScript
    - Callable entities in ECMAScript 6
    - The flag /g of JavaScript's regular expressions
    - Directories for JavaScript resources
  - ► July (8)
  - ► June (11)
  - ► May (11)
  - ► April (12)
  - ► March (8)
  - ► February (10)
  - ► January (13)
- ► 2012 (177)
- ► 2011 (380)
- ► 2010 (174)
- ► 2009 (68)
- ► 2008 (46)
- ► 2007 (12)
- ► 2006 (1)
- ► 2005 (2)

Labels

dev (592)
javascript (400)
computers (316)
life (194)
jslang (179)
esnext (156)
apple (107)
webdev (95)
mobile (83)
scitech (50)
hack (49)
mac (47)
google (39)


```
function GuiComponent() {  // constructor
    var domNode = ...;
    domNode.addEventListener('click', () => {
        console.log('CLICK');
        this.handleClick();  // `this` not shadowed
    });
}
```

2.2. Function declaration → const + arrow function

In ECMAScript 5, you have function declarations for normal functions:

```
function foo(arg1, arg2) {
    ...
}
```

In ECMAScript 6, you'll `const`-declare an arrow function:

```
const foo = (arg1, arg2) => {
    ...
};
```

The problem with function declarations is that they shadow `this` inside methods and constructors.

But they also have two advantages: First, a function object created by a function declaration always gets a meaningful name, which is useful for debugging. However, ECMAScript 6 engines will probably also assign names to arrow functions, at least in standard scenarios such as the one above.

Second, function declarations are *hoisted* (moved to the beginning of the current scope). That allows you to call them before they appear in the source code. Here, more discipline is required in ECMAScript 6 and source code will sometimes not look as nice (depending on your taste). However, one important case of calling methods and normal functions that appear later does not change: calling them from other methods and functions (after the callees have been evaluated!).

Ironically, not using function declarations may make things less confusing for newcomers, because they won't need to understand the difference between function expressions and function declarations [3]. In ECMAScript 5, I'm often seeing code like this, using a function expression instead of a function declaration (even though the latter is considered best practice):

```
var foo = function (arg1, arg2) {
    ...
};
```

2.3. IIFE → block + let

ECMAScript 5 does not have block-scoped variables. In order to simulate blocks, you use the IIFE [1] pattern:

```
(function () {  // open IIFE
    var tmp = ...;
    ...
}());  // close IIFE
```

In ECMAScript 6, you can simply use a block and a `let` variable declaration:

```
{  // open block
    let tmp = ...;
    ...
}  // close block
```

2.4. Function in object literal → concise method syntax

In ECMAScript 5, you define a method inside an object literal by providing a property value via a function expression:

```
var obj = {
    myMethod: function (arg1, arg2) {
```

Tweets by @rauschma

Axel Rauschm…
@rauschma

Enjoying "Troll Hunters":
– "Let's call him 'Gnome Chomsky'"
– "Juliet dies in this? Nooo!"

10h

Axel Rauschmayer Retweeted

Jonathan Creamer
@jcreamer898

A nice little shout out to @rauschma... infoworld.com/article/316483… #es2017 #async

JavaScr…
ECMASc…
infoworld…

14h

Axel Rauschmayer
@rauschma

Not a physics book! twitter.com/ManningBooks/s…

02 Feb

Axel Rauschmayer Retweeted

Jordan Harband
@ljharb

Making our React components forbid extra props has caught SO many bugs. I highly recommend it.

npmjs.com/airbnb-prop-ty…

npm: air…
Custom …
npmjs.com

02 Feb

Axel Rauschmayer Retweeted

Seth Petry-Johnson
@spetryjohnson

Open strong. Be bold. Tell a story. Do something to get me interested. Opening w/ the "obligatory 'about me' slide" puts me to sleep :)

02 Feb

Google

java (37)
ios (33)
business (32)
video (32)
clientjs (31)
hci (27)
entertainment (26)
nodejs (26)
society (26)
browser (25)
firefox (25)
html5 (24)
ipad (24)
movie (23)
psychology (22)
2ality (18)
tv (18)
android (17)
social (17)
chrome (16)
fun (16)
jsmodules (16)
tablet (16)
humor (15)
politics (15)
web (15)
cloud (14)
hardware (14)
microsoft (14)
software engineering (13)
blogging (12)
gaming (12)
eclipse (11)
gwt (11)
numbers (11)
programming languages (11)
app store (10)
media (10)
nature (10)
security (10)
semantic web (10)
software (10)
twitter (10)
webos (10)
12quirks (9)
education (9)
jstools (9)
photo (9)
webcomponents (9)
windows 8 (9)
async (8)
idea (8)
iphone (8)

```
            ...
        }
    };
```

In ECMAScript 6, you get more compact syntax for defining a method (internally, the result is the same):

```
let obj = {
    myMethod(arg1, arg2) {
        ...
    }
};
```

2.5. Constructor → class

In ECMAScript 5, you use constructors. Which becomes clumsy if you want to define sub-constructors:

```
// Super-constructor
function Point(x, y) {
    this.x = x;
    this.y = y;
}
Point.prototype.toString = function () {
    return '('+this.x+', '+this.y+')';
};

// Sub-constructor
function ColorPoint(x, y, color) {
    Point.call(this, x, y);
    this.color = color;
}
ColorPoint.prototype = Object.create(Point.prototype);
ColorPoint.prototype.constructor = ColorPoint;
ColorPoint.prototype.toString = function () {
    return this.color+' '+Point.prototype.toString.call(this);
};
```

In ECMAScript 6, you use classes [4] (which have the same method definition syntax as ECMAScript 6 object literals):

```
// Super-class
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    toString() {
        return '('+this.x+', '+this.y+')';
    }
}

// Sub-class
class ColorPoint extends Point {
    constructor(x, y, color) {
        super(x, y);  // same as super.constructor(x, y)
        this.color = color;
    }
    toString() {
        return this.color+' '+super();
    }
}
```

2.6. New in ECMAScript 6: generator functions and generator methods

One construct is completely new in ECMAScript 6: generators [5]. You can create one via a generator function declaration:

```
function *generatorFunction(arg1, arg2) {
    ...
}
```

Or via a generator method definition (which can also be used in classes):

```
let obj = {
    *generatorMethod() {
        ...
    }
};
```

Generator functions are an unfortunate mix of the old and the new. Like function declarations, they have dynamic `this`. And there are no generator function expressions.

In my opinion, a better choice would be to replace generator function declarations with generator arrow functions. Or at least to additionally introduce the latter, with an asterisk somewhere. For example:

```
const generatorFunction = (arg1, arg2) =>* {
    ...
};
```

Alas, this idea has been explicitly rejected for ECMAScript 6, due to syntactic issues.

3. Avoiding function expressions

There are two cases where you may think you need old-school function expressions with dynamic `this`. This section shows you that you don't.

3.1. Functions with `this` as an implicit parameter

Some libraries use `this` as an implicit parameter:

```
var $button = $('#myButton');
$button.on('click', function () {
    this.classList.toggle('clicked');
});
```

If you are using such a library, you have no choice but to use functione expressions. If you are considering using this pattern for your own library then know that you don't have to. You can always introduce an explicit parameter, instead:

```
var $button = $('#myButton');
$button.on('click', target => {
    target.classList.toggle('clicked');
});
```

As an added benefit, the `this` of the surrounding scope remains accessible.

3.2. Adding methods to an object

To add a method to an existing object in ECMAScript 5, you use a function expression:

```
MyClass.prototype.foo = function (arg1, arg2) {
    ...
};
```

In ECMAScript 6, you can use `Object.assign()` and a method definition inside an object literal:

```
Object.assign(MyClass.prototype, {
    foo(arg1, arg2) {
        ...
    }
});
```

4. Conclusion

The large amount of callable entities in ECMAScript 6 can be a bit overwhelming at first. But they do help with the three problems of using functions for everything (#1 confusing, #2 can be used incorrectly, #3 dynamic `this` where you don't want it):

münchen (4)
occupy (4)
pl fundamentals (4)
presenting (4)
publishing (4)
series (4)
textbook (4)
web design (4)
amazon (3)
asmjs (3)
back to the future (3)
bitwise_ops (3)
css (3)
es2016 (3)
flattr (3)
fluentconf (3)
food (3)
foreign languages (3)
house (3)
icloud (3)
info mgmt (3)
jsfuture (3)
jsstyle (3)
linux (3)
mozilla (3)
python (3)
regexp (3)
samsung (3)
tizen (3)
traffic (3)
typedjs (3)
unix (3)
adobe (2)
angry birds (2)
angularjs (2)
astronomy (2)
audio (2)
comic (2)
design (2)
dom (2)
ecommerce (2)
eval (2)
exploring es6 (2)
facebook flow (2)
facets (2)
flash (2)
free (2)
futurama (2)
guide (2)
history (2)
hyena (2)
internet explorer (2)
iteration (2)
journalism (2)

1. The clear separation of concerns makes things less confusing, especially for newcomers: classes replace constructors, blocks replace IIFEs, the keyword `function` does not appear when you define a method, etc.
2. ECMAScript 6 prevents some incorrect uses of functions, but not many: you will get an exception if you invoke a class as a function. Calling extracted methods as functions remains a problem.
3. Arrow functions eliminate the pitfall of inadvertently shadowing `this`.

5. References

[1] JavaScript variable scoping and its pitfalls

[2] ECMAScript.next: arrow functions and method definitions

[3] Expressions versus statements in JavaScript

[4] ECMAScript.next: classes

[5] Iterators and generators in ECMAScript 6

10 Comments The 2ality blog Login

Recommend 1 Share Sort by Best

Join the discussion…

Wil Moore III • 3 years ago

Wasn't Object.mixin dropped from ES6?

https://github.com/rwaldron/tc...

Has it been brought back?

Reply • Share ›

Axel Rauschmayer Mod ➔ Wil Moore III • 3 years ago

Not it hasn't, I just hadn't updated the blog post, yet. Fixed now, thanks.

Reply • Share ›

Donald Pipowitch • 3 years ago

Thank you for the article. Could you please explain me "3.2 Adding methods to an object" a little bit further? Is this just a different syntax or are there technical differences? The "this" isn't dynamic here or am I wrong? It always refers to the current instance which invokes the method (in ES5 and ES6).

Reply • Share ›

avi tshuva • 3 years ago

Very good article. It put some of the features i already read about in a refreshing light.
One comment: i would not give up the usage of "function" for normal functions because i like the "hoisting" feature - it let me write a more readable code sometimes.
Saying this, it is clear that i have no problem (yet, at least) with the generator's syntax. Must admit i never used it yet.

Reply • Share ›

Kuba • 3 years ago

Thanks for that article. I have two questions:
1. You said that blocks are to replace immediately invoked function expressions. But how do they work? Are they, like for example catch block, pushing a new object in front of the scope chain of the active context? If so, I think that in one aspect IIFE are better - they can import variables from the wrapping scope making them local to the called functions (and, as we know, avoiding deep scope chain traversing is considered to be a good practice). Is this also possible with ES6 blocks?

jquery (2)
jsengine (2)
jslib (2)
law (2)
lightning (2)
markdown (2)
math (2)
meego (2)
month (2)
nike (2)
nokia (2)
npm (2)
programming (2)
raffle (2)
repl (2)
servo (2)
sponsor (2)
steve jobs (2)
travel (2)
typescript (2)
usb (2)
winphone (2)
wwdc (2)
airbender (1)
amdefine (1)
aol (1)
app urls (1)
architecture (1)
atscript (1)
basic income (1)
biology (1)
blink (1)
bluetooth (1)
canada (1)
clip (1)
coding (1)
community (1)
cross-platform (1)
deutsch (1)
diaspora (1)
distributed-social-network (1)
dsl (1)
dvd (1)
dzone (1)
emacs (1)
emberjs (1)
energy (1)
esnext news (1)
esprop (1)
example (1)
facetator (1)
feedback (1)
firefly (1)
firefoxos (1)

this also possible with ES6 blocks?
2. You also said that function declarations are considered to be the best practice, or at least to be a better solution than function expressions. Can you tell why do you think so? Because I find myself using always FEs (because I think hoisting can be rather tricky than helpful and see no reasons for desired usage of it... Also FEs looks to me somehow nicer) and though I know all the differences between them (or at least I think I know) still don't know why FDs are considered to be a better solution.

Thanks in advance.

▲ ▼ • Reply • Share ›

Kuba ➚ Kuba • 3 years ago

Well, about the first question... I just thought that we can just declare a block-scoped variable and assign to it a value of some outer-scoped variable and the effect will be pretty much the same as importing it... So we can pretend that this question was not asked at all :)

▲ ▼ • Reply • Share ›

Axel Rauschmayer Mod ➚ Kuba • 3 years ago

Yes, exactly!
Question 2: Then you are well prepared for ES6. The blog post mentions two advantages that function declarations have over function expressions (after “But they also have two advantages: …”).

▲ ▼ • Reply • Share ›

Kuba ➚ Axel Rauschmayer • 3 years ago

Thank you very much for that quick response. I though that there were some other advantages than those you mentioned in the text (as I said I am not really convinced to desired usage of hoisting, I think that it breaks code readability, especially for some JS freshmen who are not aware of it and don't know what it is caused by. But that's only my opinion. And about naming - well, if one needed to, they could always name a function expression too) which make JS programmers to generally consider FDs as a better solution (or just a better practice). For now I think it depends on the programmer which one they want to use, of course if they are aware what are the consequences of using it.

Thanks for the responce again and have a nice day.

▲ ▼ • Reply • Share ›

John Eric Torres Orolfo • 3 years ago

oooooh very informative

▲ ▼ • Reply • Share ›

rwaldron • 3 years ago

Nice coverage :)

▲ ▼ • Reply • Share ›

Subscribe Add Disqus to your site Add Disqus Add Privacy

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

fritzbox (1)

german (1)

git (1)

guest (1)

guice (1)

h.264 (1)

home entertainment (1)

hosting (1)

htc (1)

ical (1)

jsdom (1)

jsmyth (1)

library (1)

location (1)

marketing (1)

mars (1)

meta-data (1)

middle east (1)

mpaa (1)

msl (1)

mssurface (1)

netflix (1)

nsa (1)

obama (1)

openoffice (1)

opinion (1)

oracle (1)

organizing (1)

philosophy (1)

pixar (1)

pnacl (1)

prism (1)

privacy (1)

proxies (1)

puzzle (1)

raspberry pi (1)

read (1)

rodney (1)

rust (1)

safari (1)

sponsoring (1)

star trek (1)

static generation (1)

talk (1)

technique (1)

theora (1)

thunderbird (1)

typography (1)

unicode (1)

v8 (1)

voice control (1)

webassembly (1)

webkit (1)

webm (1)

webpack (1)

yahoo (1)

Powered by Blogger.