sitepoint

#1

ECMAScript 2015

A SitePoint Anthology

February 2016

ECMAScript 2015: A SitePoint Anthology

Editor: James Hibbard
Designer: Alex Walker

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.
48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: books@sitepoint.com

Contents

Preface

Welcome to SitePoint's ES2015 Anthology, a collection of the most useful and interesting articles on ECMAScript 2015 (a.k.a ES6) recently published on sitepoint.com, plus an exclusive article from everyone's favorite developer evangelist, none other than Microsoft's Christian Heilmann.

As Christian states in his introduction, ES6 is now a ratified standard and brings a lot of new and exciting features to the language we love. However, as is often the case, it can take browser vendors a long time to implement these and you might be forgiven for asking yourself if ES6 is ready for the prime time. Well, the good news is that it is. Browser support is surprisingly good and for those features not yet implemented, you can use a polyfill, or a compiler such as Babel. If you want to check the current state of play of a particular feature, we recommend the ECMAScript 6 compatibility table: https://kangax.github.io/compat-table/es6/

I'd also like to touch on naming conventions. Is it ECMAScript 6, ES6, ECMAScript 2015, or ES2015? As the following quote from Douglas Crockford shows, this isn't the first time that naming has been an issue for JavaScript:

> JavaScript, aka Mocha, aka LiveScript, aka JScript, aka ECMAScript, is one of the world's most popular programming languages. ...

Throughout the anthology we have used ECMAScript 6 (ES6) and ECMAScript 2015 (ES2015) interchangeably. This is because up until recently, ECMAScript (the specification that defines the semantics, syntax, and behavior of the JavaScript programming language) was versioned by ordinal number. However, late into the ES6 specification timeline, suggestions surfaced that versioning should switch to a year-based schema (to promote faster rolling release cycles). By this time, many developers were already using the more succinct ES6, so ES2015 struggled to gain a proper hold. Going forward however, I believe that year based versioning will prevail, as this is what the spec will be referred to throughout the entire standardization process.

I hope that you enjoy this anthology and find it useful; this is the second of many such collections that we're planning to publish on a variety of topics. We'd very much welcome your feedback on this book, as it will help us shape the series in the future.

JavaScript: The State of Play

DHTML, Ajax, HTML5… It seems that as a web development community we're doomed to repeat ourselves. We keep giving our current state of play a hot new label and then turn it into hype. Right now is not different. ES6, ES7, ES2015, ES2016 describes the language we all need to write our work in now. Or we'll be yesterday's news. Time to stop for a spot of tea and see what the whole rush is about.

By Christian Heilmann
@codepo8
christianheilmann.com

JavaScript – So Hot Right Now!

Here's what's going on. JavaScript is no longer a freak language but became the most coveted tech on the web. This means more groups are looking at JavaScript. Each group then applies their own views of what it should and shouldn't do. Some see it as not useful yet, as it creates errors in older browsers. Others see it as a chance to change our ways of scripting into a more organized way of architecting our code. Publication on the web is easy and immediate. That's why we end up with a lot of opinionated advice and rushed "best practices". These can be intimidating and make us feel like we're falling behind. That's not necessarily the case.

The uses cases of JavaScript have changed over the last few years. We now write full applications with it instead of just extending them. We even write servers in it using Node.js. Our hardware changed. We need to run our programs on low-end devices with a lot less RAM than our computers.

This means that the language itself had to evolve. As the standardization of the language was slow, we created a lot of interim solutions. We had plugins, superset languages and polyfills.

"One of the amazing and dangerous features of JavaScript is that you can fix almost everything with it. That also includes the language itself."

One of the amazing and dangerous features of JavaScript is that you can fix almost everything with it. That also includes the language itself. For years we simulated features of other languages like classes, templating, type safety and block level scoping with libraries.

ES6 Is Now a Standard - Anything Else Is Experimentation

ES6 now is a ratified standard that brings a lot of these features to the language itself. It has unit tests that anyone who wants to support it can test against. That helps a lot with making it a baseline you can work from.

Anything else around the ES* label is important, but less reliable R&D trying to define the next version of the language. Many things will be proposed, a lot will be implemented in browsers and other tools. A lot of these will be experimental in nature and incompatible with others. Many a great initial concept will be discarded as a bad idea during this process.

That's OK. We need people to rock to boat to make it move forward. It is less useful though, when we get entangled in minutiae. We don't all have to be clairvoyants of the technological future. We also have work to deliver.

Web technology innovation works differently right now. We moved away from the one truth defined by a standards body and then hopefully implemented by the industry. Nowadays we follow a more nuanced and distributed way of innovating. Browsers and libraries try out things, find consensus and the standards bodies then implement them.

For us, as developers, we need to be aware of this. Working with a standard means we don't have to worry about future breakage. If you work with the newest, hottest, browser-prefixed technologies, future breakage is not a maybe — it is by design. Working with things you can trust doesn't make you less of a developer, it takes all kinds of people to move the web forward. Those who deliver products that work beautifully, behave well and lead to happy users have as much impact as those who are lucky enough to make a prediction of the future that does come true.

A Pick 'n' Mix Offering Empowering Different Developers

ES6 has evolved JavaScript into a language that has a lot of new features. These cater to the different needs of a diverse group of developers. This doesn't mean that to be an ES6 user you need to know and use them all. They aren't Pokémon. They are things that should make it easier for us as developers to do our jobs.

Syntactic Sugar

Take for example the syntactic sugar that ES6 has added to the language. These features don't change the language per se, but they enable more people to use it. It feels easier and more familiar and thus makes us feel more confident. Many features came from other languages, libraries and JavaScript helper tools. There was already a demand for them before they got added to the language. Crowd-pleasers, so to say.

I Had Strings but Now I'm Free

Strings in ES6 now also allow for literals. This is something we used libraries such as mustache for. Using backticks instead of quotes, strings now become as powerful as they are in, for example, PHP:

```
var x = 3;
var y = 5;
var result = `${x} and ${y} together are ${x + y}`;
// "3 and 5 together are 8"
```

This doesn't give us all the functionality of templating engines, but it takes the headache out of some string features. For example, strings can now span multiple lines. String concatenation has never been fun:

```
var lyrics = "I've got no strings " +
"To hold me down " +
"To make me fret, or make me frown " +
"I had strings " +
"But now I'm free " +
"There are no strings on me ";
```

With template literals this can now be:

```
var lyrics = `I've got no strings
To hold me down
To make me fret, or make me frown
I had strings
But now I'm free
There are no strings on me `;
```

The white space is part of the string and is kept as is. Another bonus is that you don't have to worry about commenting out quotes in your string with a `\` any longer. There is not much chance that you'll need a backtick in your normal text. Both strings and arrays got a lot more convenience methods in ES6. It is worth taking some time to familiarize yourself with them instead of resorting to your own looping and regular expressions.

What's This? What's This?

Big confusion points in JavaScript are scoping, the keyword `this` and how it behaves with closures. Here's a simple example taken from Kyle Simpson's Arrow This post:

```
function foo() {
  console.log('id:', this.id);
}
foo.call({id: 42});
// id is 42
```

No problems here. But as soon as you create a callback with an anonymous function (with, for example, a timeout), the scope is different and `this` becomes `undefined`. It now refers to the callback function which has no value assigned to `this`:

```
function foo() {
  setTimeout(function() {
```

```
    console.log('id:', this.id);
  }, 100);
}
foo.call({id: 42});
// id is undefined
```

The clumsy workaround for this is to create another variable called `self`, which doesn't get overridden by the anonymous function.

```
function foo() {
  var self = this;
  setTimeout(function() {
    console.log('id:', self.id);
  }, 100);
}
foo.call({id: 42});
// id is 42
```

Another way is to use the bind method:

```
function foo() {
  setTimeout(function() {
    console.log('id:', this.id);
  }.bind(this), 100);
}
foo.call({id: 42});
// id is 42
```

ES6 introduced arrow functions, which automatically do the binding for you:

```
function foo() {
  setTimeout(() => {
    console.log('id:', this.id);
  }, 100);
}
foo.call({id: 42});
// id is 42
```

In addition to fixing `this`, arrow functions also make our scripts much terser. The following creates an array containing the lengths of the strings in another array. In ES5, we needed an anonymous function and a `return` statement for this. In ES6, the arrow does all that for us.

```
var avengers = [
  'Iron Man',
  'Captain America',
  'Black Widow',
  'The Hulk',
  'Thor',
  'Hawkeye'
];

// ES5
var chars = avengers.map( function(s) {
  return s.length
});

// ES6
var chars2 = avengers.map( s => s.length );
```

Shorter and Leaner

ES6 is full of these simplifications. Other examples are spread and rest, destructuring and having default values for function parameters. These help us write less code and avoid having to loop over arrays and test for values by hand. This is handy, but you can also get overboard with it. Let's use this power wisely.

"These days, we build massive applications with hundreds of thousands of lines in JavaScript. Whether that's the right approach or not is irrelevant — it happens."

Building to Scale

These days, we build massive applications with hundreds of thousands of lines in JavaScript. Whether that's the right approach or not is irrelevant — it happens. We need to empower developers to do this in a sensible fashion. Conventional wisdom for large systems is to write them in an object-oriented way. You encapsulate functionality in distinct classes. To keep the memory footprint low, and to avoid security issues, it is prudent to have type safety. Furthermore you need constants and scoped variables. JavaScript had none of that, but ES6 offers some of it now.

A Touch of Class

The prototypical inheritance of JavaScript has always been a thorn in the side of developers coming from an object-oriented world. It didn't feel right. Now that ES6 has classes, this reads much simpler for many developers out there.

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  toString() {
    return `(${this.x}, ${this.y})`;
  }
}

class ColorPoint extends Point {
  constructor(x, y, color) {
    super(x, y);
    this.color = color;
  }
  toString() {
    return super.toString() + ' in ' + this.color;
  }
}
```

Under the hood, nothing changes. Classes are constructor functions. If you run `typeof` on them you get a function as the return.ES6 has not turned JavaScript into a class-based language. But it has made it more accessible to developers who wouldn't have touched it otherwise. Everybody wins. You can now approach large JavaScript based projects in an OO fashion and you don't need to understand how to simulate classes in a prototypical language. Does this mean you now have to write everything as classes? No, but it means that you can without having to resort to libraries.

Constants and Block-level Scoping

ES6 also introduced block-level scoping using the `let` keyword and constants using `const`. These have an obvious technical benefit for JavaScript compilers as they can allocate memory once. It is also

considered a cleaner way of writing code. You tell a future maintainer what is fixed to a value and shall not change. You also describe what is only limited to a certain block of code rather than being ready for re-use and re-assignment.

Working with ES6 Right Now

I hope you are sold that ES6 offers quite a few conveniences. It also solves a lot of issues JavaScript had in the past. But what's the state of play when it comes to using it right now?

Browser Support

The good news is that browsers support a lot of ES6 right now and are all on board with it. You can look up the current state of play on the support grid at http://kangax.github.io/compat-table/es6/. The current numbers are pretty impressive. Even Safari, which is less open than others in its roadmap, added 20% more features between versions 8 and 9.

If you want to be sure, you can feature test for the parts of ES6 you want to support. ES Feature Tests is a library that makes this easy.

Transpiling in a Workflow

If you need to support older browsers, but you want to write your code using all the goodness of ES6, you can use transpilation. This means converting your code into ES5 which more browsers understand. Babel is the big player here. You can use it in a workflow, build process or even on your local machine in your editor of choice. You can write cleaner, terser code without having to worry about browser support.

Superset Languages

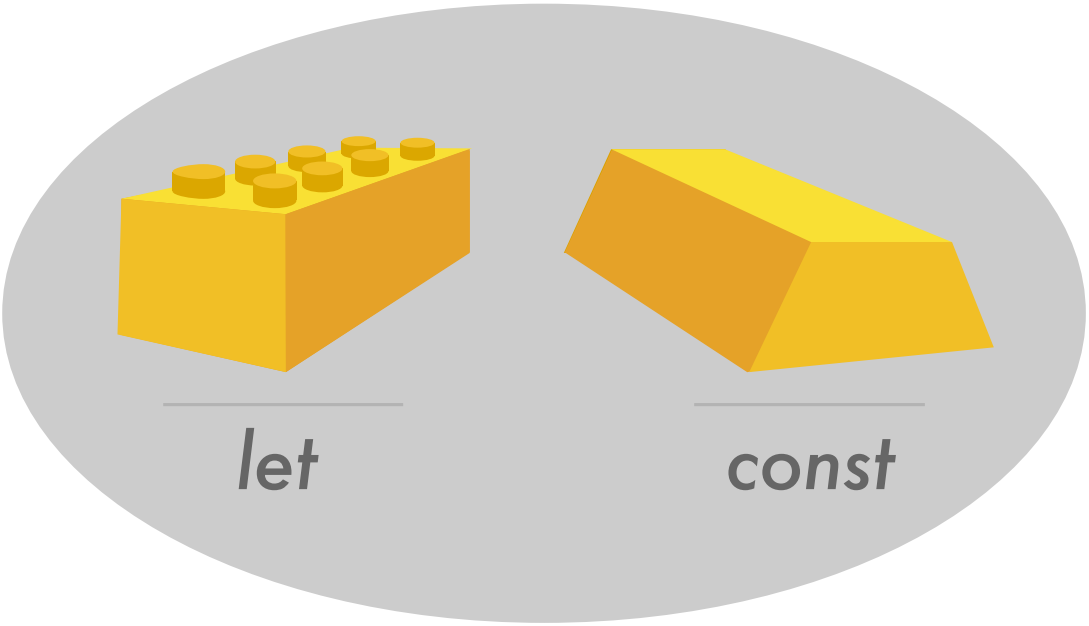
Superset languages have features of the JavaScript of tomorrow. They come with under-the-hood converters that create browser-understandable JavaScript. There were quite a few of those around, but it now seems that the last one standing is TypeScript. Angular 2 uses TypeScript and so do other large frameworks such as Dojo. The main argument for using it is that you can write your code in a type-safe manner and you don't need to worry about transpilation. Behind the scenes TypeScript converts your code into ES5 and, when supported, ES6.

"If you need to support older browsers, but you want to write your code using all the goodness of ES6, you can use transpilation."

ES6 Is Here, Get Used to It!

There is not much doubt that the future of the web belongs to ES6. It's cleaner and terser structure makes it appealing, both to developers coming from other languages and current JavaScript developers. All new APIs defined by browser vendors and standards bodies rely on promises. ES6 modules allow for organized code libraries that load on demand. And these are features not supported in ES5.

This is a good time to draw a line in the sand and re-evaluate your way of writing JavaScript code. This is never a bad thing. The web evolves and you should do so with it. You don't need to be part of the break-neck invention cycle, but you can be the person to keep those who try to predict the future on their toes by giving them data on how useful the different parts of the language are in day-to-day delivery.



Preparing for ECMAScript 6: let and const

If you're a frequent SitePoint reader, especially of the JavaScript channel you've hopefully learned a lot about the new features of ECMAScript 6 lately.

By Aurelio De Rosa
@aurelioderosa
www.audero.it

In this tutorial I'll introduce you to two new keywords: `let` and `const`. They enhance JavaScript even more by filling the gap with other languages and providing us a way to define block-scope variables and constants. If you want to learn more about them, keep reading.

`let`

Up to ECMAScript 5, JavaScript had only two types of scope: function scope and global scope.

This causes a lot of frustration and unexpected behavior for most developers coming from other lan-

guages such as C, C++, or Java. The reason is that JavaScript lacks block scope, which means a variable exists, and thus is only accessible, within the block in which it's defined. A block is everything inside an opening and closing curly bracket.

"Up to ECMAScript 5, JavaScript had only two types of scope: function scope and global scope."

Let's take a look at the following example:

```
function foo() {
  var par = 1;
  if (par >= 0) {
    var bar = 2;
    console.log(par); // prints 1
    console.log(bar); // prints 2
  }
  console.log(par); // prints 1
  console.log(bar); // prints 2
}
foo();
```

After running this code, you'll see on the console the following output:

```
1
2
1
2
```

What most developers coming from cited languages would expect is that outside the `if` block you can't access the `bar` variable. For example, running the equivalent code in C results in the error `'bar' undeclared at line ...` which refers to the use of `bar` outside the `if`.

With ECMAScript 6 the situation will change with the availability of block scope. The ECMA organization members knew that they could not change the behavior of the keyword `var` for the sake of backward compatibility. So, they decided to introduce a new keyword called `let`. The latter can be used to define variables limiting their scope to the block in which they are declared.

"With ECMAScript 6 the situation will change with the availability of block scope"

In addition, unlike `var`, variables declared using `let` aren't hoisted. If you reference a variable in a block before the `let` declaration for that variable is encountered, this results in a `ReferenceError`.

But what does this mean in practice? Is it only good for newbies? Not at all!

To explain you why you'll love `let` consider the following code taken from my article 5 More JavaScript Interview Exercises:

```
var nodes = document.getElementsByTagName('button');
for (var i = 0; i < nodes.length; i++) {
  nodes[i].addEventListener('click', function() {
    console.log('You clicked element #' + i);
  });
}
```

Here you can recognize a well-known issue that comes from variable declaration, their scope, and event handlers. If you don't know what I'm talking about, go check the article I mentioned and than come back, I'll wait here.

Back? Good! Thanks to `let` we can easily solve this issue by simply reassigning the value of `i` to a support variable declared using `let`:

```
var nodes = document.getElementsByTagName('button');
for (var i = 0; i < nodes.length; i++) {
  let j = i;
  nodes[i].addEventListener('click', function() {
    console.log('You clicked element #' + j);
  });
}
```

Cool, isn't it?

Now the bad news. At the time of writing no browsers support this feature by default, which means that this isn't really something you can use today unless you use a transpiler that will convert your code into an equivalent source that is compatible with ECMAScript 5. In Chrome 38 and Opera 25, you can try `let` by activating the "Experimental JavaScript features" flag but only if you run the code in strict mode.

Please note that even if a given browser supports `let` behind a flag, it might implement only a subset of the specifications. For example, the browser might not be able to throw the `ReferenceError` where needed.

A demo that shows the difference between `var` and `let` is shown below and is also available as a JSFiddle:

```
<h1>Var</h1>
<button class="var">Click 1</button>
<button class="var">Click 2</button>
<button class="var">Click 3</button>

<h1>Let</h1>
<button class="let">Click 1</button>
<button class="let">Click 2</button>
<button class="let">Click 3</button>
'use strict';

var varNodes = document.getElementsByClassName('var');
for (var i = 0; i < varNodes.length; i++) {
  varNodes[i].addEventListener('click', function() {
    console.log('You clicked element #' + i);
  });
}

var letNodes = document.getElementsByClassName('let');
for (var i = 0; i < letNodes.length; i++) {
  let j = i;
  letNodes[i].addEventListener('click', function() {
    console.log('You clicked element #' + j);
  });
}
```

const

`const` addresses the common need of developers to associate a mnemonic name with a given value, such that the value can't be changed (or in simpler terms, define a constant). For example, if you're working with math formulas, you may need to create a `Math` object. Inside this object you want to associate the values of π and e with a mnemonic name. `const` allows you to achieve this goal. Using it you can create a constant that can be global or local to the function in which it is declared.

Constants defined with `const` follow the same scope rules as variables but they can't be redeclared. Constants also share a feature with variables declared using `let` in that they are block-scoped instead of function-scoped (and thus they are not hoisted1). In case you try to access a constant before it's declared you'll receive a `ReferenceError`.

In this case too, the support is heavily fractional and most browsers have implemented a subset of the feature. In case you want to know in detail which browsers support `const` and which specific features are implemented, you can take a look at this ECMAScript 6 compatibility table.

An example of use of `const` is shown below:

```
'use strict';

function foo() {
  const con1 = 3.141;
  if (con1 > 3) {
    const con2 = 1.414;
    console.log(con1); // prints 3.141
    console.log(con2); // prints 1.414
  }
  console.log(con1); // prints 3.141
  try {
    console.log(con2);
  } catch (ex) {
    console.log('Cannot access con2 outside its block');
  }
}
foo();
```

A demo of this code is available as a JSFiddle. Remember that different browsers may behave differently based on the features supported.

Conclusion

In this tutorial I've introduced you to two new and very interesting features of JavaScript that are available in the upcoming ECMAScript "Harmony" 6. I bet that most of you have encountered use cases where the use of `let` and `const` would have been beneficial. Unfortunately, due to their poor support

among browsers you have to use a transpiler or you won't be able to take advantage of them. Hopefully this will change soon and, anyway, this is what the future of JavaScript looks like.

Preparing for ECMAScript 6: New Function Syntax

In the book ***Secrets of the JavaScript Ninja*** by John Resig and Bear Bibeault, the authors describe functions as the most important concept of the language because in JavaScript everything pivots around them. Functions are very important indeed.

By Aurelio De Rosa
@aurelioderosa
www.audero.it

The new version of JavaScript adds even more features to them that you'll love to use. In this tutorial you'll learn more about new features that work with functions and that will enable you to write even more powerful code.

If you want to know more about ECMAScript 6, I suggest you check out my articles about the String and Array data types, and newly introduced Map and WeakMap data types.

Arrow Functions

The first feature I want to cover is the arrow function. As the name suggests, to use it we'll use an arrow (`=>`) that you might recognize if you've ever worked with PHP. This feature comes in two forms that integrate with the current function syntax. Keep in mind that by using an arrow function you can replace anonymous functions only. The two possible syntaxes are shown below:

> "Keep in mind that by using an arrow function you can replace anonymous functions only."

```
// First syntax
([param] [, param]) => { statements }

// Second syntax
param => expression
```

The `param` placeholder represents the parameters of the function, while `statements` represents the body of the function. `expression` represents any valid expression and it's a replacement for the right part of the first syntax (`{ statements }`). In the first form you can place any number of parameters between the parentheses, while the second is limited to one.

To understand what they look like, let's say that we have a set of numbers that we need to test for even or oddness. In addition, we need to perform this test only once, so we don't need to define an `isEven()` function and can use an anonymous function. Our function will return `true` if a number is even, and `false` otherwise. This results in the code below:

```
var numbers = [10, 21, 15, 8];

// prints "[true, false, false, true]"
console.log(
  numbers.map(function(number) {
    return number % 2 === 0;
  })
);
```

This is a perfect example of why arrow functions are useful. As you can see, the anonymous function is pretty simple and its body is made of a single statement. Nonetheless, to adhere to the current JavaScript syntax, we have to type a lot of additional characters. Thanks to arrow functions we can avoid them and write code like this:

```
var numbers = [10, 21, 15, 8];

// prints "[true, false, false, true]"
console.log(
  numbers.map(number => number % 2 === 0)
);
```

Much shorter isn't it? In this case we can use the second, shorter syntax because the body of the function is made of only one statement. In addition, the only statement we had is a `return`. In fact, with the second syntax the result of the right expression is used as a return value.

If we want to use the arrow functions to replace functions with more statements, we can use the first form. To see it in action, let's enhance our function to verify that the given parameter is actually a number and that it's an integer. The resulting code is the following:

```
var numbers = [10, 15, false, 'test', {}];

// prints "[true, false, false, false, false]"
console.log(
  numbers.map(function(number) {
    // The parameter is a number and it's an integer
    if (typeof number !== 'number' || number % 1 !== 0) {
      return false;
    }
    return number % 2 === 0;
  })
);
```

We can make it shorter using the first form of arrow functions as shown below:

```
var numbers = [10, 15, false, 'test', {}];

// prints "[true, false, false, false, false]"
console.log(
  numbers.map(number => {
    // The parameter is a number and it's an integer
    if (typeof number !== 'number' || number % 1 !== 0) {
      return false;
```

```
    }
    return number % 2 === 0;
  })
);
```

A demo of this code is available as a JSFiddle.

Arrow functions aren't only great because they allow us to save a few keystrokes. Another important feature is that they implicitly bind the `this` value of a function. Imagine that a given page has some buttons and that the page has the following code:

```
var Utility = {
  fullname: 'Aurelio De Rosa',
  handler: function(elements) {
    for (var i = 0; i < elements.length; i++) {
      elements[i].addEventListener('click', function() {
        console.log(this.fullname);
      });
    }
  }
};

var buttons = document.getElementsByTagName('button');
Utility.handler(buttons);
```

Any time a button is pressed the string "Aurelio De Rosa" is printed on the console. However, when the handler is attached in the `for` loop, the value of `this` won't be the `Utility` object anymore but it'll be `window`. Therefore, the console will display `undefined`.

This is a common problem in JavaScript that we can solve in a lot of different ways. For example you could store the reference to the `Utility` object using a variable (the classic `var that = this` approach) or use the `bind()` function. However, thanks to the introduction of the arrow function, we can fix the issue like this:

```
var Utility = {
  fullname: 'Aurelio De Rosa',
  handler: function(elements) {
    for (var i = 0; i < elements.length; i++) {
      elements[i].addEventListener('click', () => {
```

```
        console.log(this.fullname);
      });
    }
  }
};

var buttons = document.getElementsByTagName('button');
Utility.handler(buttons);
```

A demo of this code is available as a JSFiddle.

This feature is currently only supported by Firefox 22+.

Default Values for Parameters

The new version of JavaScript has introduced another feature that PHP developers use extensively: the ability to set a default value for parameters. The JavaScript version is even more powerful than the PHP one because a default value used for a given parameter is available to the next parameters in the list. Also, in JavaScript, parameters with default values can be followed by parameters without. In PHP this is not possible. To assign a default value you have to place an equals sign right after the parameter name, followed by the default value you want to assign, as you'd do in a classic assignment.

Default values are something that we have used several times in the past but in a different way. Ponder the following code:

```
function Person(name, surname, gender) {
  // Set default values
  name = name || 'Aurelio';
  surname = surname || 'De Rosa';
  gender = gender || 'male';

  this.toString = function() {
    return 'My name is ' + name + ' ' + surname + ' and I am a ' + gender;
  }
};

// prints "My name is John Doe and I am a male"
console.log(new Person('John', 'Doe').toString());
```

```
// prints "My name is Aurelio De Rosa and I am a male"
console.log(new Person().toString());
```

Running this snippet of code works as expected, but the manual management of default values is really boring. Furthermore, the approach used may have unexpected results because it tests for a *falsy* value and then assigns a default. Let's say that one of the parameters was a number and an ideal default value is 10 but zero is acceptable. Using the technique of the code above we could write:

```
param = param || 10;
```

In this case, because zero is a *falsy* value, 10 will be assigned to `param` which is really not what we want. Of course it's possible to use other techniques but they require us to write even more code. Thanks to the new features of ECMAScript 6 we can avoid issues like this and shorten the code as shown below:

```
function Person(name = 'Aurelio', surname = 'De Rosa', gender = 'male') {
  this.toString = function() {
    return 'My name is ' + name + ' ' + surname + ' and I am a ' + gender;
  }
};

// prints "My name is John Doe and I am a male"
console.log(new Person('John', 'Doe').toString());

// prints "My name is Aurelio De Rosa and I am a male"
console.log(new Person().toString());
```

This version is not only more concise but also more readable because the default values are set close to the parameters. It worth noting that the default value is used also if the argument passed in is `undefined`.

A demo of this code is available as a JSFiddle.

In the previous example we've only seen the basic use of this new feature. The next example shows how we can have parameters without a default value after one that has a default one:

```
function prod(number1 = 1, number2) {
  return number1 * number2;
}
```

The final example shows a parameter whose default value depends on a previous parameter with a default value:

```
function Person(name, surname, username = name + ' ' + surname) {
}
```

Like the previous feature this default value for a parameter is currently only supported by Firefox 15+.

Rest Parameter

The rest parameter is a special parameter that enables us to express an arbitrary number of parameters in a function. It'll include all the passed arguments that don't match a named parameter as elements of an array (so not an array-like element). To define this parameter, you have to place it as the last in the function's signature and prepend three dots to it. The syntax for this parameter is reported below:

"The rest parameter is a special parameter that enables us to express an arbitrary number of parameters in a function."

```
function (...paramName) {
}
```

`paramName` can be any arbitrary name you want to assign to this special parameter.

Developers have simulated this feature for a long time using `arguments` and removing the named parameters. To visualize the difference between the old approach and the new one, let's say that we have a function that accepts some data about a person and prints it on the console. This function has two mandatory parameters that represent the name and the surname of the person, and then any number of additional parameters. In the old approach, such a function can be written as follows:

```
function presentation(name, surname) {
  var otherInfo = [].slice.call(arguments, 2);
  console.log('My name is ' + name + ' ' + surname);
  if (otherInfo.length > 0) {
    console.log('Other info: ' + otherInfo.join(', '));
  }
}

// prints "My name is John Doe"
```

```
presentation('John', 'Doe');

// Prints "My name is Aurelio De Rosa"
// "Other info: male, Italian, Web developer"
presentation('Aurelio', 'De Rosa', 'male', 'Italian', 'Web developer');
```

Using the rest parameter we can get rid of the first statement of the function resulting in the code listed below:

```
function presentation(name, surname, ...otherInfo) {
  console.log('My name is ' + name + ' ' + surname);
  if (otherInfo.length > 0) {
    console.log('Other info: ' + otherInfo.join(', '));
  }
}

// prints "My name is John Doe"
presentation('John', 'Doe');

// Prints "My name is Aurelio De Rosa"
// "Other info: male, Italian, Web developer"
presentation('Aurelio', 'De Rosa', 'male', 'Italian', 'Web developer');
```

A demo of this code is available as a JSFiddle.

This feature is currently only supported by Firefox 15+.

Conclusion

In this tutorial we've covered the new features introduced in ECMAScript 6 that work with functions. They will allow us to write even more powerful and concise code. Due to the poor support of browsers they aren't something that can you really use today, but you should be prepared because this is what the future of JavaScript looks like.

Preparing for ECMAScript 6: New Number Methods

In this series about the new features of ECMAScript 6, we've discussed new methods available for the String and Array data types, but also new types of data like Map and WeakMap.

By Aurelio De Rosa
@aurelioderosa
www.audero.it

In this article I'm going to introduce you to the new methods and constants added to the `Number` data type. Some of the methods covered, as we'll see, aren't new at all but they have been improved and/or moved under the right object (for example `isNaN()`). As always, we'll also put the new knowledge acquired into action with some examples. So, without further ado, let's start.

Number.isInteger()

The first method I want to cover is `Number.isInteger()`. It's a new addition to JavaScript and this is something you may have defined and used by yourself in the past. It determines whether the value passed to the function is an integer or not. This method returns `true` if the passed value is an integer, and `false` otherwise. The implementation of this method was pretty easy but it's still good to have it natively. One of the possible solutions to recreate this function is:

```
Number.isInteger = Number.isInteger || function (number) {
  return typeof number === 'number' && number % 1 === 0;
};
```

For my fun, I tried to recreate this function and I ended up with a different approach:

```
Number.isInteger = Number.isInteger || function (number) {
  return typeof number === 'number' && Math.floor(number) === number;
};
```

Both these functions are good and useful but they don't respect the ECMAScript 6 specifications. So, if you want to polyfill this method you need something a little bit more complex as we'll see in a few moments. For the moment, let's start by discovering the syntax of `Number.isInteger()`:

```
Number.isInteger(number)
```

The `number` argument represents the value you want to test.

Some examples of use of this method are shown below:

```
// prints "true"
console.log(Number.isInteger(19));

// prints "false"
console.log(Number.isInteger(3.5));

// prints "false"
console.log(Number.isInteger([1, 2, 3]));
```

A demo of this code is available as a JSFiddle.

The method is supported by almost any modern browser and specifically by Firefox, Chrome, Opera, and Safari. If you want to support Internet Explorer and some other browsers, you need a polyfill. One that you can employ is available on the Mozilla Developer Network on the method's page and also reported below for your convenience:

```
if (!Number.isInteger) {
  Number.isInteger = function isInteger (nVal) {
    return typeof nVal === "number" &&
           isFinite(nVal) && nVal > -9007199254740992 &&
           nVal < 9007199254740992 &&
           Math.floor(nVal) === nVal;
  };
}
```

`Number.isNaN()`

If you've written any JavaScript code in the past, this method should not be new to you. For a while now, JavaScript has had a method called `isNaN()` that is exposed through the `window` object. This method tests if a value is equal to `NaN`, in which case it returns `true`, or not, in which case `false` is returned. The problem with `window.isNaN()` is that it has an issue in that it returns `true` also for values that *converted* to a number will be `NaN`. To give you a concrete idea of this issue, all the following statements return `true`:

```
// prints "true"
console.log(window.isNaN(0/0));

// prints "true"
console.log(window.isNaN('test'));

// prints "true"
console.log(window.isNaN(undefined));

// prints "true"
console.log(window.isNaN({prop: 'value'}));
```

What you might need is a method that returns `true` only if the `NaN` value is passed. That's why ECMAScript 6 has introduced the `Number.isNaN()` method. Its syntax is pretty much what you'd expect:

```
Number.isNaN(value)
```

Where `value` is the value you want to test. Some example uses of this method are shown below:

```
// prints "true"
console.log(Number.isNaN(0/0));

// prints "true"
console.log(Number.isNaN(NaN));

// prints "false"
console.log(Number.isNaN(undefined));

// prints "false"
console.log(Number.isNaN({prop: 'value'}));
```

As you can see, testing the same values we obtain different results.

A demo of this code is available as a JSFiddle.

The method is currently only supported by Firefox, Chrome and Opera. If you want to support other browsers, a very simple polyfill for this method is the following:

```
Number.isNaN = Number.isNaN || function (value) {
  return value !== value;
};
```

The reason why it works is because `NaN` is the only non-reflexive value in JavaScript, which means that it is the only value that isn't equal to itself.

"The reason why it works is because `NaN` is the only non-reflexive value in JavaScript, which means that it is the only value that isn't equal to itself."

`Number.isFinite()`

This method shares the same story as the previous one. In JavaScript there is a method called `window.isFinite()` that tests if a value

passed is a finite number or not. Unfortunately, it also returns `true` for values that *converted* to a number will be a finite number. Examples of this issue are reported below:

```
// prints "true"
console.log(window.isFinite(10));

// prints "true"
console.log(window.isFinite(Number.MAX_VALUE));

// prints "true"
console.log(window.isFinite(null));

// prints "true"
console.log(window.isFinite([]));
```

For this reason, in ECMAScript 6 there is a method called `isFinite()` that belongs to `Number`. Its syntax is the following:

```
Number.isFinite(value)
```

Where `value` is the value you want to test. If you test the same values from the previous snippet, you can see that the results are different:

```
// prints "true"
console.log(Number.isFinite(10));

// prints "true"
console.log(Number.isFinite(Number.MAX_VALUE));

// prints "false"
console.log(Number.isFinite(null));

// prints "false"
console.log(Number.isFinite([]));
```

A demo of this code is available as a JSFiddle.

The method is currently only supported by Firefox, Chrome and Opera. You can find a polyfill for it on the method's page on MDN.

`Number.isSafeInteger()`

The `Number.isSafeInteger()` method is a completely new addition to the next version of JavaScript. It tests whether the value passed is a number that is a safe integer, in which case it returns `true`. A safe integer is defined as an integer that satisfies both the following conditions:

"The Number.isSafeInteger() method is a completely new addition to the next version of JavaScript."

- The number can be exactly represented as an IEEE-754 double precision number
- The IEEE-754 representation of the number can't be the result of rounding any other integer to fit the IEEE-754 representation

Based on this definition, the safe integers are all the integers from -(253 - 1) inclusive to 253 - 1 inclusive. These values are important and we'll discuss them a bit more at the end of this section.

The syntax of this method is:

```
Number.isSafeInteger(value)
```

Where `value` is the value you want to test. A few example uses of this method are shown below:

```
// prints "true"
console.log(Number.isSafeInteger(5));

// prints "false"
console.log(Number.isSafeInteger('19'));

// prints "false"
console.log(Number.isSafeInteger(Math.pow(2, 53)));

// prints "true"
console.log(Number.isSafeInteger(Math.pow(2, 53) - 1));
```

A demo of this code is available as a JSFiddle.

The `Number.isSafeInteger()` method is supported by Firefox, Chrome and Opera. A polyfill for this method, extracted from es6-shim by Paul Miller, is:

```
Number.isSafeInteger = Number.isSafeInteger || function (value) {
  return Number.isInteger(value) && Math.abs(value) <= Number.MAX_SAFE_INTEGER;
};
```

Note that this polyfill relies on the `Number.isInteger()` method discussed before, so you need to polyfill the latter as well to use this one.

ECMAScript 6 "Harmony" also introduces two related constant values: `Number.MAX_SAFE_INTEGER` and `Number.MIN_SAFE_INTEGER`. The former represents the maximum safe integer in JavaScript, that is 253 - 1, while the latter the minimum safe integer which is -(253 - 1). As you might note, these are the same values I cited earlier.

`Number.parseInt()` and `Number.parseFloat()`

The `Number.parseInt()` and `Number.parseFloat()` methods are covered in the same section because, unlike other similar methods mentioned in this article, they already existed in previous version of ECMAScript but aren't different from their old global version. So, you can use them in the same way you've done so far and you can expect the same results. They have been added to `Number` because they actually belonged to it from the very beginning.

For the sake of completeness I'm reporting their syntax:

```
// Signature of Number.parseInt
Number.parseInt(string, radix)

// Signature of Number.parseFloat
Number.parseFloat(string)
```

Where `string` represents the value you want to parse and `radix` is the radix you want to use to convert `string`.

The following snippet shows a few example uses:

```
// Prints "-3"
console.log(Number.parseInt('-3'));

// Prints "4"
console.log(Number.parseInt('100', 2));
```

```
// Prints "NaN"
console.log(Number.parseInt('test'));

// Prints "NaN"
console.log(Number.parseInt({}));

// Prints "42.1"
console.log(Number.parseFloat('42.1'));

// Prints "NaN"
console.log(Number.parseFloat('test'));

// Prints "NaN"
console.log(Number.parseFloat({}));
```

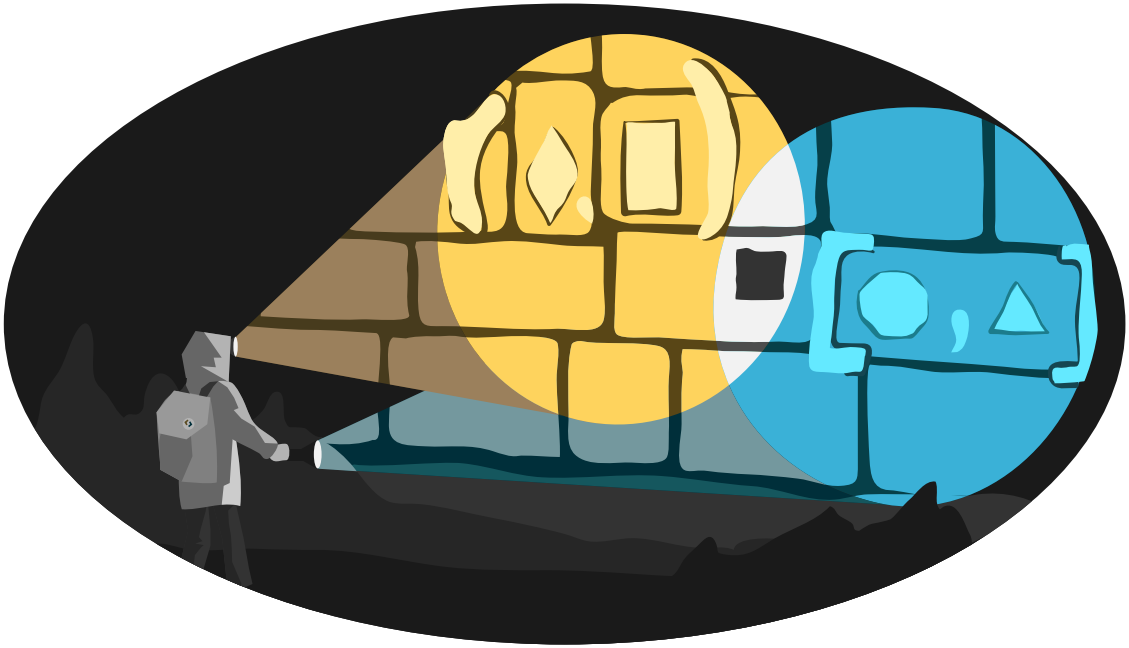
A demo of this code is available as a JSFiddle.

These methods are currently implemented in Firefox, Chrome, and Opera. In case you want to polyfill them, you can simply call their related global method as listed below:

```
// Polyfill Number.parseInt
Number.parseInt = Number.parseInt || function () {
  return window.parseInt.apply(window, arguments);
};

// Polyfill Number.parseFloat
Number.parseFloat = Number.parseFloat || function () {
  return window.parseFloat.apply(window, arguments);
};
```

Conclusion

In this tutorial we've covered the new methods and constants added in ECMAScript 6 that work with the `Number` data type. It's worth noting that the new version of JavaScript also has added another constant that I didn't mention so far. This constant is `Number.EPSILON` and represents the difference between one and the smallest value greater than one that can be represented as a `Number`. With this last note, we've concluded our journey for the `Number` data type.



Preparing for ECMAScript 6: Set and WeakSet

In one of my recent articles titled Preparing for ECMAScript 6: Map and WeakMap, I introduced you to two new data types available in ECMAScript 6: Map and its weak counterpart WeakMap.

By Aurelio De Rosa
@aurelioderosa
www.audero.it

In this tutorial we're going to cover another duo of similar data types called `Set` and `WeakSet`. They share a lot of similarities with `Map` and `WeakMap`, especially when it comes to the methods available. However, as we'll discuss here, they have different scopes.

As I've pointed out in all the previous articles discussing ECMAScript 6, if you want to polyfill what we'll cover, you can employ es6-shim by Paul Miller.

Set

Like the name says, the `Set` data type represents a set of elements (a collection). As mathematical notion suggests, this means that a set lets you store the same elements only once (e.g. the string "test" can't be stored twice). Like other JavaScript data types, it isn't mandatory to store elements of the same type, so in the same set you can store arrays, numbers, strings, and so on.

It's also worth noting that a single element in a set cannot be retrieved, for example using a `get()` method. The reason is that an element has neither a key nor an index you can refer to in order to retrieve it. But because you can verify that an element is contained in a given `Set` instance, you don't need a `get()` method. For example, if you know the string "test" is contained in a set you don't need to retrieve it, because you already have that value. It's still possible to retrieve *all* the elements stored, as you'll learn in this tutorial.

"But when is this data type a good fit?" you may ask. Well, let's say that you need to store the IDs of some elements. When it comes to these situations, you don't want duplicates. Under these circumstances and in ECMAScript 5, most of you have probably used arrays or objects to store the elements. The problem is that every time a new element comes in, you have to check that it hasn't been already added to avoid duplicates. If you used an array, you'd have code like this:

```
var collection = [1, 2, 3, 4, 5];
var newElements = [4, 8, 10];

for (var i = 0; i < newElements.length; i++) {
  if (collection.indexOf(newElements[i]) === -1) {
    collection.push(newElements[i]);
  }
}
```

Using the `Set` data type, you can simplify the previous code as shown below:

```
var collection = new Set([1, 2, 3, 4, 5]);
var newElements = [4, 8, 10];

for (var i = 0; i < newElements.length; i++) {
  collection.add(newElements[i]);
}
```

Now that you know what `Set` is and when to use it, let's discuss the properties and the methods exposed.

`Set.prototype.size`

The `size` property returns the number of elements in a `Set` instance. This is similar to the `length` of the `Array` data type.

`Set.prototype.constructor()`

The constructor, as you might know, is used to instantiate new objects. It accepts an optional argument called `iterable` that is an array or an iterable object whose elements will be added to the new set. A basic example of use is shown below:

```
var array = [1, 2, "test", {a: 10}];
var set = new Set(array);
```

`Set.prototype.add()`

The `add()` method adds a new element to the set if it isn't already present; otherwise the element isn't added. The signature of this method is the following:

```
Set.prototype.add(value)
```

where `value` is the element you want to store. This method modifies the set it's called upon but also returns the new set, allowing for chaining. An example of how to use such feature is shown below:

```
var set = new Set();
set.add("test").add(1).add({});
```

This method is currently implemented in Firefox, Internet Explorer 11, Chrome 38 and Opera 25. In versions of Chrome prior to 38 and Opera prior to 25 this method is supported behind the activation of the flag "Enable Experimental JavaScript".

`Set.prototype.delete()`

In the same way we can add elements, we can also delete them from a set. To do that we can use the

`delete()` method. It accepts the value to delete and returns `true` if the element is successfully removed or `false` otherwise. The signature of this method is shown below:

```
Set.prototype.delete(value)
```

`value` represents the element you want to delete.

This method is currently implemented in Firefox, Internet Explorer 11, Chrome 38 and Opera 25. In versions of Chrome prior to 38 and Opera prior to 25 you have to activate the usual flag.

`Set.prototype.has()`

The `has()` method is one of the methods that the `Set` data type has in common with `Map`. It allows us to verify if an element exists or not in the set. It returns `true` if the value is found or `false` otherwise. The signature of this method is as follows:

```
Set.prototype.has(value)
```

where `value` is the value you want to search for.

This method is currently implemented in Firefox, Internet Explorer 11, Chrome 38 and Opera 25. In versions of Chrome prior to 38 and Opera prior to 25 this method is supported behind the activation of the flag "Enable Experimental JavaScript".

`Set.prototype.clear()`

The `clear()` method, like the one defined on `Map`, is a convenient way to remove all the elements from a `Set` instance. The method doesn't have a return value (which means it returns `undefined`). The signature of `clear()` is shown below:

```
Set.prototype.clear()
```

`clear()` is currently implemented in Firefox, Internet Explorer 11, Chrome 38 and Opera 25. In versions of Chrome prior to 38 and Opera prior to 25 you have to activate the usual flag.

`Set.prototype.forEach()`

Another method in common with `Map` is `forEach()`. We can use it to iterate over the elements stored in the set in insertion order. The signature of `forEach()` is the following:

```
Set.prototype.forEach(callback[, thisArg])
```

`callback` is a function to run on each of the elements in the set. The `thisArg` parameter is used to set the context (`this`) of the callback. `callback` receives three parameters:

- `value`: the value of the element processed
- `value`: the value of the element processed
- `set`: the `Set` object processed

As you can see, the value being processed is passed twice. The reason is to keep the method consistent with the `forEach()` implemented in `Map` and `Array`.

"As you can see, the value being processed is passed twice. The reason is to keep the method consistent with the forEach() implemented in Map and Array."

This method is supported by Firefox, Internet Explorer 11, Chrome 38 and Opera 25. In versions of Chrome prior to 38 and Opera prior to 25 you have to activate the usual flag.

`Set.prototype.entries()`

The `entries()` method enables us to obtain an `Iterator` to loop though the set's elements. The `Iterator` contains an array of `value`-`value` pairs for each element in the set, in insertion order. The reason for this duplication is the same as before: to keep it consistent with the method of `Map`. The signature of this method is:

```
Set.prototype.entries()
```

This method is currently supported by Firefox, Chrome 38 and Opera 25. In versions of Chrome prior to 38 and Opera prior to 25 you have to activate the usual flag.

`Set.prototype.values()`

Another method that belongs to this data type is `values()`. It returns an `Iterator` object containing the values of the elements of the set, in insertion order. Its signature is the following:

```
Set.prototype.values()
```

This method is currently supported by Firefox, Chrome 38 and Opera 25. In versions of Chrome prior to 38 and Opera prior to 25 this method is supported behind the activation of the flag "Enable Experimental JavaScript".

`Set.prototype.keys()`

Curiously enough, `Set` has also a `keys()` method. It performs the same operation as `values()`, so I won't describe it.

`WeakSet`

`WeakSet` is the weak counterpart to the `Set` data type. A `WeakSet` only accepts objects as its values. This means that `{}`, `function(){}` (functions inherit from `Object`), and instances of your own classes are allowed, but `"test"`, `1`, and other primitive data types are not.

The other important difference is that `WeakSet` objects don't prevent garbage collection if there aren't any other references to an object stored (the reference is *weak*). Due to this difference, there aren't any methods to retrieve values or more than one element at once such as `Set.prototype.values()` and `Set.prototype.entries()`. In addition, similarly to `WeakMap`, there isn't a `size` property available.

As a final note, I want to highlight that Chrome 37 and Opera 24 support `WeakSet` and its methods without a flag, while the same isn't true for `Set`. The newer version Chrome 38 and Opera 25 support `Set` and its methods by default.

Putting it all together

Now that you've seen all the methods and properties of the `Set` and the `WeakSet` data types, it's time to put them into action. In this section I've developed two demos so that you can play with these methods and have a better idea of their power. As you'll note, I haven't used the `Set.prototype.keys()` method because I think it's only good at confusing developers.

> "As you'll note, I haven't used the Set.prototype.keys() method because I think it's only good at confusing developers"

In the first demo I'll use a `Set` object and its methods except `Set.prototype.keys()`.

```
// Creates a new Set object
var set = new Set();
// Defines an array will be stored in the set
var arr = [4, 1, 9];

// Adds a new Number to the set
```

```
set.add(1);
// Adds a new String to the set
set.add('Aurelio De Rosa');
// Adds a new Object to the set
set.add({name: 'John Doe'});
// Adds a new Array element to the set
set.add(arr);

// Checks whether the string "test" is stored in the set. Prints "false"
console.log(set.has('test'));

// Checks whether the number "1" is stored in the set. Prints "true"
console.log(set.has(1));

// Retrieves the set size. Prints "4"
console.log(set.size);

// Deletes the object {name: 'Aurelio De Rosa'}. Prints "false" because even if it has
the same values and properties, it's a different object
console.log(set.delete({name: 'Aurelio De Rosa'}));

// Retrieves the set size. Prints "4"
console.log(set.size);

// Deletes the array arr. Prints "true" because it's the same array
console.log(set.delete(arr));

// Retrieves the set size. Prints "3"
console.log(set.size);

// Loops over each element of the set
set.forEach(function(value, samevalue, set) {
  // Prints the value twice
  console.log('Value ' + value + ' is the same as ' + samevalue);
});

var entries = set.entries();
var entry = entries.next();
// Loops over each element of the set
while (!entry.done) {
```

```
  // Prints both the value and the key
  console.log('Value ' + entry.value[1] + ' is the same as ' + entry.value[0]);
  entry = entries.next();
}

var values = set.values();
var value = values.next();
// Loops over each value of the set
while (!value.done) {
  // Prints the value
  console.log('Value: ' + value.value);
  value = values.next();
}

// Deletes all the elements in the set
set.clear();

// Retrieves the set size. Prints "0"
console.log(set.size);
```

A demo of this code is available as a JSFiddle.

In this second demo we'll see how we can work with a `WeakSet` object.

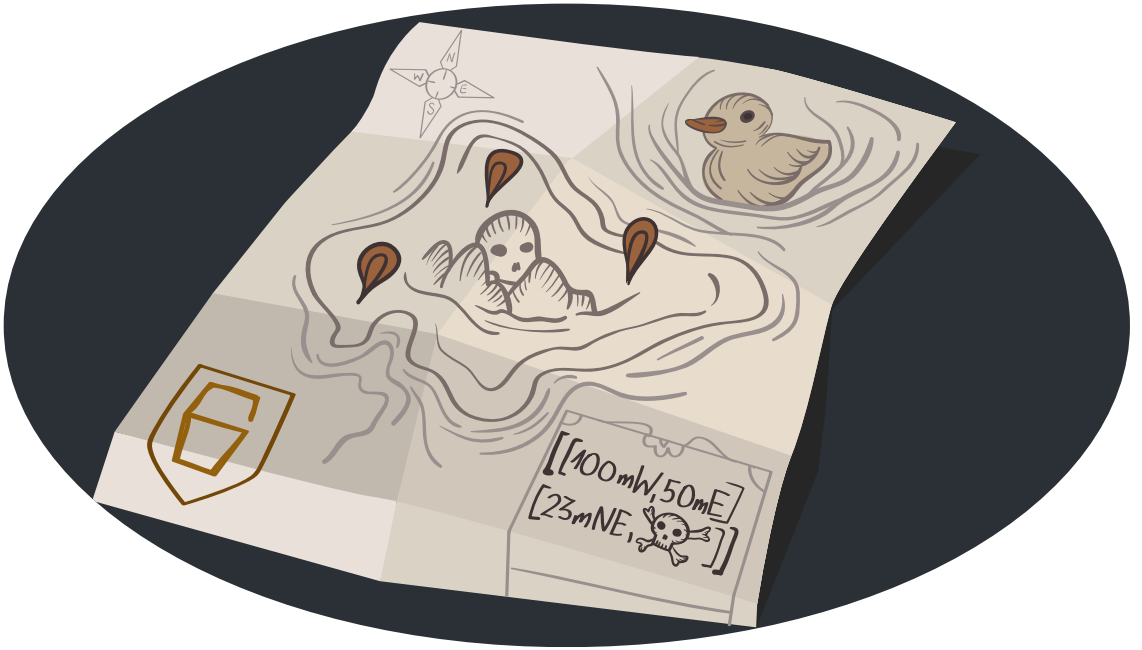
```
// Creates a new WeakSet object
var weakset = new WeakSet();
// Defines an object that will be stored in the set
var obj = {name: 'Aurelio De Rosa'};

// Adds an object to the set
weakset.add(obj);
// Adds a function to the set
weakset.add(function(){});
// Adds another object to the set
weakset.add({name: 'John Doe'});

// Checks whether the Object {name: 'John Doe'} exists in the weak set. Prints "false"
because despite the fact that the passed object and the stored one have the same values
and properties, they are different objects
```

```
console.log(weakset.has({name: 'John Doe'}));

// Checks whether the Object obj exists in the weak set. Prints "true" because it's the
same object
console.log(weakset.has(obj));

// Deletes the obj element. Prints "true"
console.log(weakset.delete(obj));

// Deletes the function(){} element. Prints "false" because the passed function and the
stored one they are different functions (objects)
console.log(weakset.delete(function(){}));

// Deletes all the elements of the weak set
weakset.clear();
```

A demo of the previous code is available as a JSFiddle.

Conclusion

In this tutorial I covered the new `Set` and `WeakSet` data types. In addition to `Map` and `WeakMap` they are the most interesting new types available in ECMAScript 6. I hope you enjoyed the article and learned something interesting.



Preparing for ECMAScript 6: Map and WeakMap

If you're following this series about ECMAScript 6, you've learned about some of the new methods available for the String and Array types. The new version of JavaScript introduces several new data types too. In this article we'll discuss `Map` and its weak counterpart `WeakMap`.

By Aurelio De Rosa
@aurelioderosa
www.audero.it

Remember that if you want to polyfill what we'll cover in this tutorial, you can employ es6-shim by Paul Miller.

`Map`

Maps are one of the most used data structures in programming. Maps are objects that associate a key to a value, regardless of the type of the value (number, string, object, and so on). For those of you who are not aware of maps, let's discuss a brief example. In a typical structured database table you associate an

ID with each entry (a row of the table). So, you have something like:

```
ID 1 -> Aurelio De Rosa, Italy
ID 2 -> Colin Ihrig, USA
ID 3 -> John Doe, USA
```

In languages like Java and C# you have a class that allows you to instantiate maps. In other languages like PHP you can create a map using an associative array. Prior to ECMAScript 6, JavaScript was one of the languages to lack this data structure. Now, this data type exists and it's called `Map`.

JavaScript maps are really powerful because they allow the use of any value (both objects and primitive values) either as a key or a value. This is one of the most important differences compared to maps created using the `Object` type. In fact, maps created using an object literal only allow strings as their keys. In addition, as we'll see in a moment, the `Map` type has a method to easily retrieve the number of elements contained within it, while with objects you have to loop over them manually, checking that the element belongs to the object itself and it isn't inherited (using the good old `hasOwnProperty()`).

"JavaScript maps are really powerful because they allow the use of any value (both objects and primitive values) either as a key or a value."

Now that I've introduced you to this new data type, let's discover what are the properties and the methods available.

`Map.prototype.size`

The `size` property returns the number of elements in the `Map` object. This is a nice addition, that I mentioned in the previous section, because thanks to this, you don't have to count the elements by yourself.

`Map.prototype.constructor()`

The `Map` object's constructor is used to instantiate new objects and accepts an optional argument called `iterable`. The latter is an array or an iterable object whose elements are key/value pairs (two-element arrays). Each of these elements will be added to the new map. For example, you could write:

```
var array = [['key1', 'value1'], ['key2', 100]];
var map = new Map(array);
```

Map.prototype.set()

The `set()` method is used to add a new element (key/value pair) to a map. If the key used already exists, the value associated is replaced by the new one. Its signature is the following:

```
Map.prototype.set(key, value)
```

where `key` is the key you want to use and `value` is the value to store. This method modifies the map it's called upon but also returns the new map.

This method is currently implemented in Firefox, Internet Explorer 11, and Chrome and Opera behind a flag ("Enable Experimental JavaScript").

Map.prototype.get()

The `get()` method returns the value associated with the key provided. If the key isn't found, the method returns `undefined`. The signature of the method is shown below, where `key` is the key you want to use.

```
Map.prototype.get(key)
```

This method is currently implemented in Firefox, Internet Explorer 11, and Chrome and Opera behind a flag ("Enable Experimental JavaScript").

Map.prototype.delete()

The `delete()` method removes the element associated with the provided key from the map. It returns `true` if the element is successfully removed or `false` otherwise. The signature of this method is shown below:

```
Map.prototype.delete(key)
```

`key` represents the key of the element you want to delete.

This method is currently implemented in Firefox, Internet Explorer 11, and Chrome and Opera (you have to activate the usual flag).

Map.prototype.has()

`has()` is a method to verify if an element with the given key exists or not. It returns `true` if the key is found or `false` otherwise. The signature of this method is shown below:

```
Map.prototype.has(key)
```

where `key` is the key you want to search.

This method is currently implemented in Firefox, Internet Explorer 11, and Chrome and Opera behind a flag ("Enable Experimental JavaScript").

Map.prototype.clear()

The `clear()` method is a convenient way to remove all the elements from a `Map` object. The method doesn't have a return value (which means it returns `undefined`). The signature of `clear()` is shown below:

```
Map.prototype.clear()
```

`clear()` is currently implemented in Firefox, Internet Explorer 11, and Chrome and Opera behind the usual flag.

Map.prototype.forEach()

Just as we can loop over arrays, executing a callback function using the `forEach()` method, the same is possible with maps. The signature of `forEach()` is shown below:

```
Map.prototype.forEach(callback[, thisArg])
```

`callback` is the callback function to execute for each of the elements in the map, and `thisArg` is used to set the context (`this`) of the callback. The method doesn't have a return value (which means it returns `undefined`). `callback` receives three parameters that are:

- `value`: the value of the element processed
- `key`: the key of the element processed
- `map`: the `Map` object being processed

This method is supported by Firefox, Internet Explorer 11, and Chrome and Opera behind a flag.

Map.prototype.entries()

`entries()` is a method of obtaining an `Iterator` object to iterate though the elements of the map. I've already mentioned this type of object when talking about the new keys() method of the Array type. The signature of this method is:

```
Map.prototype.entries()
```

This method is currently supported by Firefox, and Chrome and Opera behind a flag.

Map.prototype.keys()

The `keys()` method is very similar to `entries()` but it returns only the keys of the elements. Its signature is the following:

```
Map.prototype.keys()
```

This method is currently supported by Firefox, and Chrome and Opera behind a flag.

Map.prototype.values()

Similar to `keys()` we have `values()`. It returns an `Iterator` object containing the values of the elements of the map. Its signature is the following:

```
Map.prototype.values()
```

This method is currently supported by Firefox, and Chrome and Opera behind a flag.

WeakMap

`WeakMap` is very similar to `Map` but has few important differences. The first is that a `WeakMap` only accepts objects as keys. This means that `{}`, `function(){}` (remember that functions inherit from `Object`), and instances of your own classes are allowed, but `'key'`, `10`, and other primitive data types are not.

"WeakMap is very similar to Map but has few important differences. The first is that a WeakMap only accepts objects as keys."

The other important difference is that `WeakMap` objects don't prevent garbage collection if there aren't any other references to

an object which is acting as a key (the reference is *weak*). Due to this difference, there is no method to retrieve keys (for example the `Map.prototype.keys()` method for `Map`) or more than one element at once (like `Map.prototype.values()` and `Map.prototype.entries()`).

The reason is well explained by the Mozilla developer network (MDN):

> *WeakMap keys are not enumerable (i.e. there is no method giving you a list of the keys). If they were, the list would depend on the state of garbage collection, introducing non-determinism.*

As a further consequence of the previous point, there is no `size` property available.

It's also worth noting that Chrome 37 and Opera 24 (the latest stables at the time of writing) support `WeakMap` and its methods without a flag, while the same isn't true for `Map`.

Putting it all together

So far you've learned all about the `Map` and the `WeakMap` data type and their methods. In this section we'll put them in action so that you can have a better understanding of their power. In addition to showing you some code, we'll also provide you with demos so that you can play with them live.

In the first demo we'll see a `Map` object and its methods in action.

```
// Creates a new Map object
var mapObj = new Map();
// Defines an object that will be used a key in the map
var objKey = {third: 'c'};

// Adds a new element having a String as its key and a String as its value
mapObj.set('first', 'a');
// Adds a new element having a Number as its key and an Array as its value
mapObj.set(2, ['b']);
// Adds a new element having an Object as its key and a Number as its value
mapObj.set(objKey, 3);
// Adds a new element having an Array as its key and a String as its value
mapObj.set(['crazy', 'stuff'], 'd');

// Checks whether an element having a key of "2" exists in the map. Prints "true"
```

```
console.log(mapObj.has(2));

// Checks whether an element having a key of "test" exists in the map. Prints "false"
console.log(mapObj.has('test'));

// Retrieves the element having key of "first". Prints "a"
console.log(mapObj.get('first'));

// Retrieves the element having key of "['crazy', 'stuff']". Prints "undefined" because
even if the value of this array are identical to the one used to set a value, they are
not the same array
console.log(mapObj.get(['crazy', 'stuff']));

// Retrieves the element having as a key the value of objKey. Prints "3" because it's
exactly the same object using to set the element
console.log(mapObj.get(objKey));

// Retrieves the element having key of "empty". Prints "undefined"
console.log(mapObj.get('empty'));

// Retrieves the map size. Prints "4"
console.log(mapObj.size);

// Deletes the element having key of "first". Prints "true"
console.log(mapObj.delete('first'));

// Retrieves the map size. Prints "3"
console.log(mapObj.size);

// Loops over each element of the map
mapObj.forEach(function(value, key, map) {
  // Prints both the value and the key
  console.log('Value ' + value + ' is associated to key ' + key);
});

var entries = mapObj.entries();
var entry = entries.next();
// Loops over each element of the map
while (!entry.done) {
```

```
  // Prints both the value and the key
  console.log('Value ' + entry.value[1] + ' is associated to key ' + entry.value[0]);
  entry = entries.next();
}

var values = mapObj.values();
var value = values.next();
// Loops over each value of the map
while (!value.done) {
  // Prints the value
  console.log('Value: ' + value.value);
  value = values.next();
}

var keys = mapObj.keys();
var key = keys.next();
// Loops over each key of the map
while(!key.done) {
  // Prints the key
  console.log('Key: ' + key.value);
  key = keys.next();
}

// Deletes all the elements of the map
mapObj.clear();

// Retrieves the map size. Prints "0"
console.log(mapObj.size);
```

A demo of this code is available as a JSFiddle.

In this second demo we'll see how we can work with a `WeakMap` object.

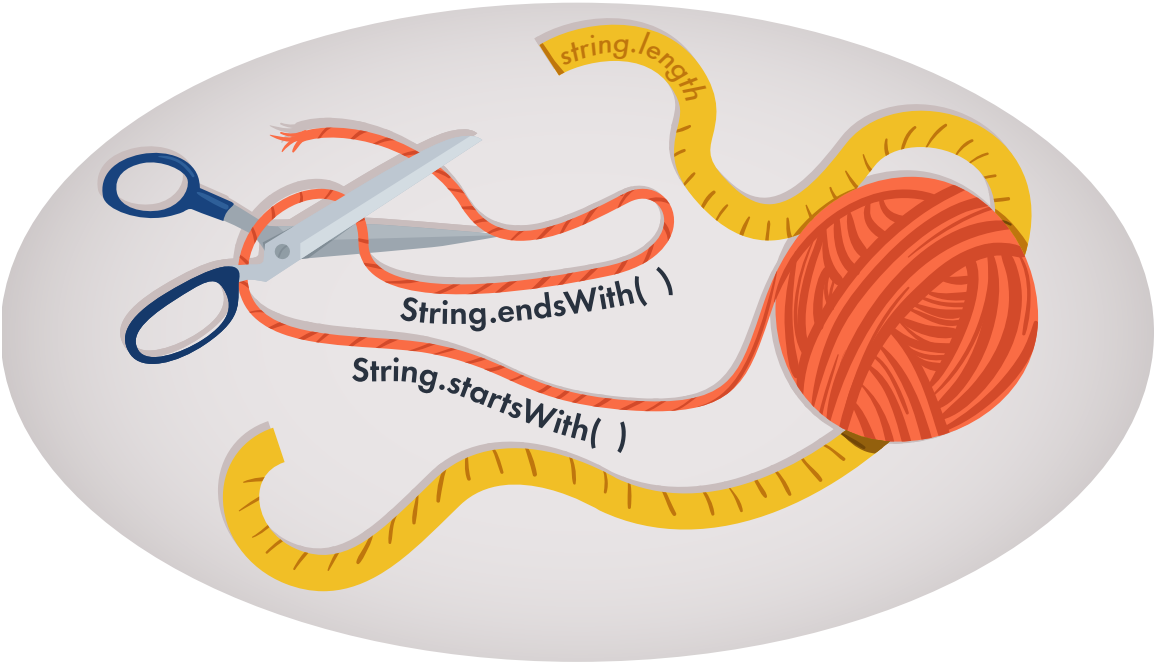
```
// Creates a new WeakMap object
var weakMapObj = new WeakMap();
// Defines an object that will be used a key in the map
var objKey1 = {a: 1};
// Defines another object that will be used a key in the map
var objKey2 = {b: 2};
```

```
// Adds a new element having an Object as its key and a String as its value
weakMapObj.set(objKey1, 'first');
// Adds a new element having an Object as its key and a String as its value
weakMapObj.set(objKey2, 'second');
// Adds a new element having a Function as its key and a Number as its value
weakMapObj.set(function(){}, 3);

// Checks whether an element having as its key the value of objKey1 exists in the weak
map. Prints "true"
console.log(weakMapObj.has(objKey1));

// Retrieve the value of element associated with the key having the value of objKey1.
Prints "first"
console.log(weakMapObj.get(objKey1));

// Deletes the element having key of objKey1. Prints "true"
console.log(weakMapObj.delete(objKey1));

// Deletes all the elements of the weak map
weakMapObj.clear();
```

A demo of this code is available as a JSFiddle.

Conclusion

In this tutorial I covered the new `Map` and `WeakMap` data types. The former is a nice addition to the language because most developers have simulated maps for a long time. Its weak counterpart isn't really something you'll use a lot in your day job, but there are surely situations where it might be a good fit. To reinforce the concepts discussed, I strongly encourage you to play with the demos provided. Have fun!



Preparing for ECMAScript 6: New String Methods

In my previous article, I introduced you to most of the new methods available in ECMAScript 6 "Harmony" that work with the `Array` type. In this tutorial you'll learn about those that work with strings. We'll develop several examples and mention the polyfills available for them. Remember that if you want to polyfill them all using a single library, you can employ es6-shim by Paul Miller.

By Aurelio De Rosa
@aurelioderosa
www.audero.it

`String.prototype.startsWith()`

One of the most used functions in every modern programming language is the one to verify if a string starts with a given substring. So far, JavaScript hasn't had such a function, so you had to write it yourself. The following code shows how developers usually polyfilled it:

```
if (typeof String.prototype.startsWith !== 'function') {
  String.prototype.startsWith = function(str) {
    return this.indexOf(str) === 0;
  };
}
```

Or, alternatively:

```
if (typeof String.prototype.startsWith !== 'function') {
  String.prototype.startsWith = function(str) {
    return this.substring(0, str.length) === str;
  };
}
```

These snippets are still valid, but they don't reproduce exactly what the newly available `String.prototype.startsWith()` method does. The new method has the following syntax:

```
String.prototype.startsWith(searchString[, position]);
```

You can see that in addition to a substring, it accepts a second argument. The `searchString` parameter specifies the substring you want to verify is the start of the string. `position` indicates the position at which to start the search.

The default value of `position` is 0. The methods returns `true` if the string starts with the provided substring, and `false` otherwise. Remember that the method is case sensitive, so "Hello" is different from "hello".

> "One of the most used functions in every modern programming language is the one to verify if a string starts with a given substring. So far, JavaScript hasn't had such a function, so you had to write it yourself."

An example usage of this method is shown below:

```
var str = 'hello!';
var result = str.startsWith('he');

// prints "true"
console.log(result);
```

```
// verify starting from the third character
result = str.startsWith('ll', 2);

// prints "true"
console.log(result);
```

A demo of this code is available as a JSFiddle.

The method is supported by Firefox 17+, and Chrome 35+ and Opera 22+ behind a flag ("Enable Experimental JavaScript"). A polyfill for this method can be found in the method's page on MDN. Another polyfill has also been developed by Mathias Bynens.

`String.prototype.endsWith()`

In addition to `String.prototype.startsWith()`, ECMAScript 6 introduces the `String.prototype.endsWith()` method. It verifies that a string terminates with a given substring. The syntax of this method, shown below, is very similar to `String.prototype.startsWith()`:

```
String.prototype.endsWith(searchString[, position]);
```

As you can see, this method accepts the same parameters as `String.prototype.startsWith()`, and also returns the same type of values.

A difference is that the `position` parameter lets you search within the string as if the string were only this long. In other words, if we have the string `house` and we call the method in this way `'house'.endsWith('us', 4)`, we obtain `true` because it's like we actually had the string `hous` (note the missing "e").

"A difference is that the `position` parameter lets you search within the string as if the string were only this long."

An example use of this method is shown below:

```
var str = 'hello!';
var result = str.endsWith('lo!');

// prints "true"
console.log(result);

// verify as if the string was "hell"
```

```
result = str.endsWith('lo!', 5);

// prints "false"
console.log(result);
```

A demo of this code is available as a JSFiddle.

The method is supported by Firefox 17+, and Chrome 35+ and Opera 22+ behind a flag ("Enable Experimental JavaScript"). A polyfill for this method can be found in the method's page on MDN. Another polyfill has been developed by Mathias Bynens.

`String.prototype.includes()`

While we're talking about verifying if one string is contained in another, let me introduce you to the `String.prototype.includes()` method. It returns `true` if a string is contained in another, no matter where, and `false` otherwise.

Its syntax is shown below:

```
String.prototype.includes(searchString[, position]);
```

The meaning of the parameters is the same as for `String.prototype.startsWith()`, so I won't repeat them. An example use of this method is shown below:

```
var str = 'Hello everybody, my name is Aurelio De Rosa.';
var result = str.includes('Aurelio');

// prints "true"
console.log(result);

result = str.includes('Hello', 10);

// prints "false"
console.log(result);
```

A demo of this code is available as a JSFiddle.

`String.prototype.includes()` is currently supported by Firefox 17+, and Chrome 35+ and Opera

22+ behind the usual flag. Just like the other methods discussed in this tutorial, you can find a polyfill provided by Mathias Bynens (this guy knows how to do his job!) and another on the Mozilla Developer Network.

String.prototype.repeat()

Let's now move on to another type of method. `String.prototype.repeat()` is a method that returns a new string containing the same string it was called upon but repeated a specified number of times. The syntax of this method is the following:

```
String.prototype.repeat(times);
```

The `times` parameter indicates the number of times the string must be repeated. If you pass zero you'll obtain an empty string, while if you pass a negative number or infinity you'll obtain a `RangeError`.

An example use of this method is shown below:

```
var str = 'hello';
var result = str.repeat(3);

// prints "hellohellohello"
console.log(result);

result = str.repeat(0);

// prints ""
console.log(result);
```

A demo of this code is available as a JSFiddle.

This method is supported by Firefox 24+, and Chrome 35+ and Opera 22+ behind the usual flag. Two polyfills for this method are the one developed by Mathias Bynens and another on the Mozilla Developer Network.

String.raw

The last method I want to cover in this tutorial is `String.raw`. It's defined as a tag function of template

strings. It's interesting because its kind of a replacement for templating libraries, although I'm not 100% sure it can scale enough to actually replace those libraries. However, the idea is basically the same as we'll see in a moment. What it does is to compile a string and replace every placeholder with a provided value.

Its syntax is the following (note the backticks):

```
String.raw`templateString`
```

The `templateString` parameter represents the string containing the template to process.

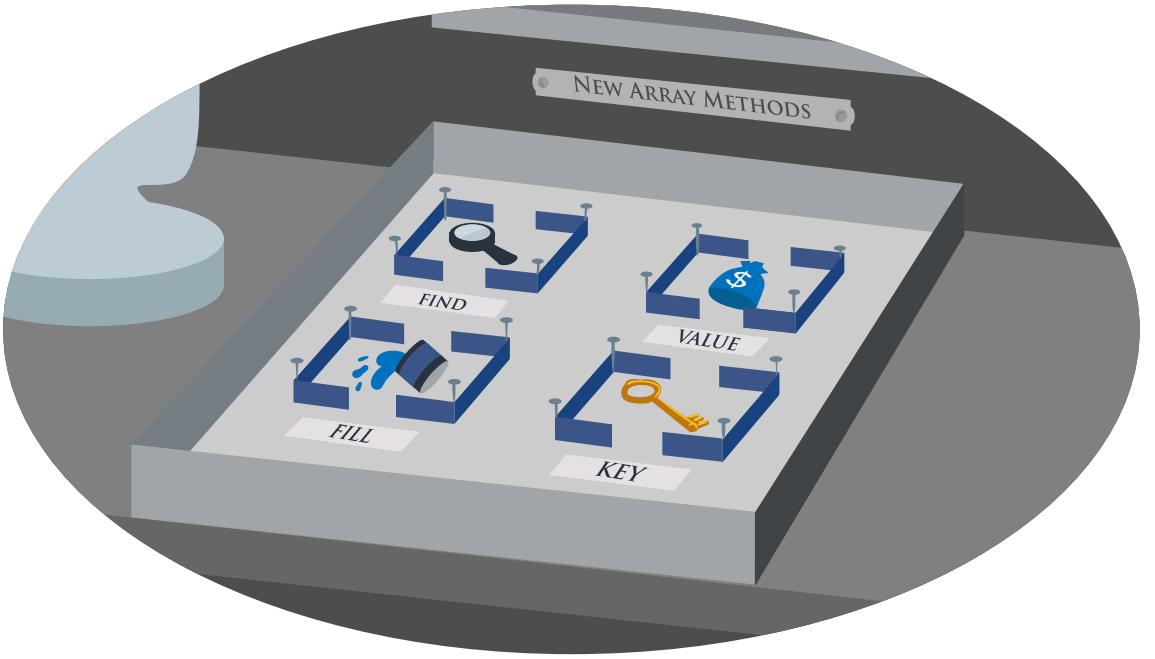
To better understand this concept, let's see a concrete example:

```
var name = 'Aurelio De Rosa';
var result = String.raw`Hello, my name is ${name}`;

// prints "Hello, my name is Aurelio De Rosa" because ${name}
// has been replaced with the value of the name variable
console.log(result);
```

Unfortunately at the time of this writing, `String.raw` isn't supported by any browser. For this reason, I'm not including a demo for it.

Conclusion

In this tutorial you've learned about several new methods introduced in ECMAScript 6 that work with strings. Other methods that we haven't covered are String.fromCodePoint(), String.prototype.codePointAt(), and String.prototype.normalize(). I hope you enjoyed the article and that you'll continue to follow our channel to learn more about ECMAScript 6.



Preparing for ECMAScript 6: New Array Methods

As developers, we constantly need to keep up with the latest technologies (that's why you read SitePoint, right?). If you're a JavaScript developer, it's your responsibility to understand the features introduced in new versions of the language.

By Aurelio De Rosa
@aurelioderosa
www.audero.it

The next version, known as ECMAScript 6, or Harmony, has been under development for a few years. As of August 2014, it has been stabilized. This means that no new features will be added to the specification. It will be completed around the end of 2014, and will start to go into the official publication process starting in March 2015, but things are stable enough to start learning the new methods right now.

In this article we'll discuss most of the new methods available in ECMAScript 6 that work with the `Array` type. When dis-

"If you're a JavaScript developer, it's your responsibility to understand the features introduced in new versions of the language."

cussing them, you'll note that I'll write `Array.method()` when I describe a "class" method and `Array.prototype.method()` when I outline an "instance" method.

We'll also see some example uses and mention several polyfills for them. Keep in mind that because these methods are very recent, the examples will work only in a few browsers. If you need a polyfill-them-all library, you can use es6-shim by Paul Miller.

`Array.from()`

The first method I want to mention is `Array.from()`. It creates a new `Array` instance from an array-like or an iterable object. This method can be used to solve an old problem with array-like objects that most developers solve using this code:

```
// typically arrayLike is arguments
var arr = [].slice.call(arryLike);
```

The syntax of `Array.from()` is shown below:

```
Array.from(arrayLike[, mapFn[, thisArg]])
```

The meaning of its parameters are:

- `arrayLike`: An array-like or an iterable object
- `mapFn`: A function to call on every element contained
- `thisArg`: A value to use as the context (`this`) of the `mapFn` function

Now that we know its syntax and its parameters, let's see this method in action. In the code below we're going to create a function that accepts a variable number of arguments, and returns an array containing these elements doubled:

```
function double(arr) {
  return Array.from(arguments, function(elem) {
    return elem * 2;
  });
}

var result = double(1, 2, 3, 4);
```

```
// prints [2, 4, 6, 8]
console.log(result);
```

A demo of this code is available as a JSFiddle.

This method is currently only supported in Firefox 32+, so if you want to employ it you need a polyfill. There are a couple of polyfills to choose from: one is available on the method's page on MDN, while the other has been written by Mathias Bynens and is called Array.from.

`Array.prototype.find()`

Another of the methods introduced is `Array.prototype.find()`. The syntax of this method is:

```
Array.prototype.find(callback[, thisArg])
```

As you can see, it accepts a callback function used to test the elements of the array and an optional argument to set the context (`this`) of the callback function. The callback function receives three parameters:

- `element`: The current element
- `index`: The index of the current element
- `array`: The array you used to invoke the method

This method returns a value in the array if it satisfies the provided callback function, or `undefined` otherwise. The callback is executed once for each element in the array until it finds one where a truthy value is returned. If there is more than one element in the array that will return a truthy value, only the first is returned.

An example usage is shown below:

```
var arr = [1, 2, 3, 4];
var result = arr.find(function(elem) {return elem > 2;});

// prints "3" because it's the first
// element greater than 2
console.log(result);
```

A demo of this code is available as a JSFiddle.

This method has slightly better support, as it's implemented in Firefox 25+, and Chrome 35+ and Opera 22+ behind a flag ("Enable Experimental JavaScript"). If you need a polyfill, one is provided on the method's page on MDN.

`Array.prototype.findIndex()`

A method that is very similar to the previous one is `Array.prototype.findIndex()`. It accepts the same arguments but instead of returning the first element that satisfies the callback function, it returns its index. If none of the elements return a truthy value, `-1` is returned. An example usage of this method is shown below.

```
var arr = [1, 2, 3, 4];
var result = arr.findIndex(function(elem) {return elem > 2;});

// prints "2" because is the index of the
// first element greater than 2
console.log(result);
```

A demo of this code is available as a JSFiddle.

This method is supported by Firefox 25+, and Chrome 35+ and Opera 22+ behind a flag ("Enable Experimental JavaScript"). A polyfill for this method can be found on the method's page on MDN.

`Array.prototype.keys()`

Yet another method introduced in this new version of JavaScript is `Array.prototype.keys()`. This method returns a new `Array Iterator` (not an array) containing the keys of the array's values. We'll cover array iterators in an upcoming article, but if you want to learn more about them now, you can refer to the specification or the MDN page.

The syntax of `Array.prototype.keys()` is shown below:

```
Array.prototype.keys()
```

An example of use is the following:

```
var arr = [1, 2, 3, 4];
```

```
var iterator = arr.keys();

// prints "0, 1, 2, 3", one at a time, because the
// array contains four elements and these are their indexes
var index = iterator.next();
while(!index.done) {
  console.log(index.value);
  index = iterator.next();
}
```

A demo of this code is available as a JSFiddle.

`Array.prototype.keys()` is supported by Firefox 28+, and Chrome 35+ and Opera 22+ behind a flag.

Array.prototype.values()

In the same way we can retrieve the keys of an array, we can retrieve its values using `Array.prototype.values()`. This method is similar to `Array.prototype.keys()` but the difference is that it returns an `Array Iterator` containing the values of the array.

The syntax of this method is shown below:

```
Array.prototype.values()
```

An example use is shown below:

```
var arr = [1, 2, 3, 4];
var iterator = arr.values();

// prints "1, 2, 3, 4", one at a time, because the
// array contains these four elements
var index = iterator.next();
while(!index.done) {
  console.log(index.value);
  index = iterator.next();
}
```

A demo of this code is available as a JSFiddle.

The `Array.prototype.values()` method is supported by Chrome 35+ and Opera 22+ behind the usual flag.

`Array.prototype.fill()`

If you have worked in the PHP world (like me), you will recall a function named `array_fill()` that was missing in JavaScript. In ECMAScript 6 this method is no longer missing. `Array.prototype.fill()`, fills an array with a specified value optionally from a start index to an end index (not included).

"If you have worked in the PHP world (like me), you will recall a function named `array_fill()` that was missing in JavaScript. In ECMAScript 6 this method is no longer missing."

The syntax of this method is the following:

```
Array.prototype.fill(value[, start[, end]])
```

The default values for `start` and `end` are respectively `0` and the `length` of the array. These parameters can also be negative. If `start` or `end` are negative, the positions are calculated starting from the end of the array.

An example of use of this method is shown below:

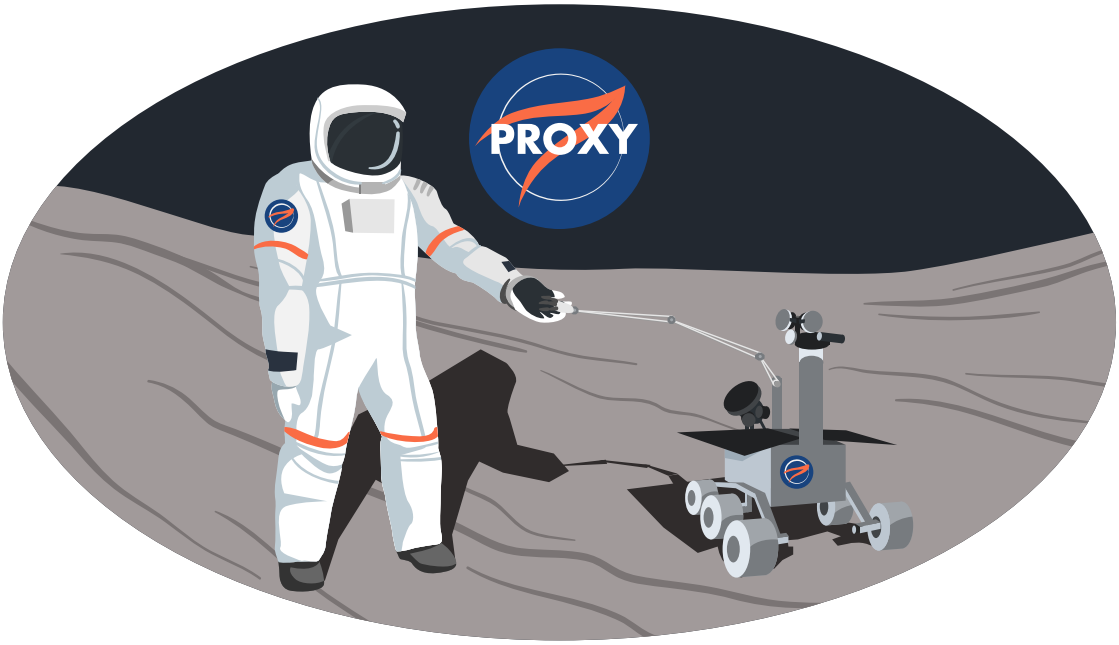
```
var arr = new Array(6);
// This statement fills positions from 0 to 2
arr.fill(1, 0, 3);
// This statement fills positions from 3 up to the end of the array
arr.fill(2, 3);

// prints [1, 1, 1, 2, 2, 2]
console.log(arr);
```

A demo of this code is available as a JSFiddle.

This method is currently supported in Firefox 31+, and Chrome 37+ and Opera 24+ behind the usual flag. As polyfills you can employ the one on the method's page on MDN, or the polyfill developed by Addy Osmani.

Conclusion

In this article we've discussed several of the new methods introduced in ECMAScript 6 that work with arrays. Whether you find them useful or not, they are on their way. It's clear that these methods aren't really ready for the prime time, but you still should keep an eye on them. Of course there is more in ECMAScript 6 than what we've described in this article as we'll discover in the upcoming ones.



Preparing for ECMAScript 6: Proxies

In computing terms, a proxy sits between you and the thing you are communicating with. The term is most often applied to a proxy server -- a device between the web browser (Chrome, Firefox, Safari, Edge etc.) and the web server (Apache, NGINX, IIS etc.) where a page is located. The proxy server can modify requests and responses. For example, it can increase efficiency by caching regularly-accessed assets and serving them to multiple users.

By Craig Buckler
@craigbuckler
craigbuckler.com

ES6 proxies sit between your code and an object. A proxy allows you to perform meta-programming operations such as intercepting a call to inspect or change an object's property.

"In computing terms, a proxy sits between you and the thing you are communicating with."

The following terminology is used in relation to ES6 proxies:

`target` : The original object the proxy will virtualize. This could be a JavaScript object such as the jQuery library or native objects such as arrays or even another proxies.

`handler`: An object which implements the proxy's behavior using…

`traps`: Functions defined in the handler which provide access to the target when specific properties or methods are called.

It's best explained with a simple example. We'll create a target object named `target` which has three properties:

```
var target = {
  a: 1,
  b: 2,
  c: 3
};
```

We'll now create a handler object which intercepts all get operations. This returns the target's property when it's available or 42 otherwise:

```
var handler = {

  get: function(target, name) {
    return (
      name in target ? target[name] : 42
    );
  }

};
```

We now create a new Proxy by passing the target and handler objects. Our code can interact with the proxy rather than accessing the `target` object directly:

```
var proxy = new Proxy(target, handler);

console.log(proxy.a);  // 1
console.log(proxy.b);  // 2
console.log(proxy.c);  // 3
console.log(proxy.meaningOfLife);  // 42
```

Let's expand the proxy handler further so it only permits single-character properties from `a` to `z` to be set:

```
var handler = {

  get: function(target, name) {
    return (name in target ? target[name] : 42);
  },

  set: function(target, prop, value) {
    if (prop.length == 1 && prop >= 'a' && prop <= 'z') {
      target[prop] = value;
      return true;
    }
    else {
      throw new ReferenceError(prop + ' cannot be set');
      return false;
    }
  }

};

var proxy = new Proxy(target, handler);

proxy.a = 10;
proxy.b = 20;
proxy.ABC = 30;
// Exception: ReferenceError: ABC cannot be set
```

Proxy Trap Types

We've seen the `get` and `set` in action which are likely to be the most useful traps. However, there are several other trap types you can use to supplement proxy handler code:

- **construct(target, argList)** Traps the creation of a new object with the `new` operator.
- **get(target, property)** Traps `Object.get()` and must return the property's value.
- **set(target, property, value)** Traps `Object.set()` and must set the property value. Return

`true` if successful. In strict mode, returning `false` will throw a TypeError exception.

- **deleteProperty(target, property)** Traps a `delete` operation on an objects property. Must return either `true` or `false`.
- **apply(target, thisArg, argList)** Traps object function calls.
- **has(target, property)** Traps `in` operators and must return either `true` or `false`.
- **enumerate(target)** Traps `for ... in` statements and must return an iterator object.
- **ownKeys(target)** Traps `Object.getOwnPropertyNames()` and must return an enumerable object.
- **getPrototypeOf(target)** Traps `Object.getPrototypeOf()` and must return the prototype's object or null.
- **setPrototypeOf(target, prototype)** Traps `Object.setPrototypeOf()` to set the prototype object -- no value is returned.
- **isExtensible(target)** Traps `Object.isExtensible()` which determines whether an object can have new properties added. Must return either `true` or `false`.
- **preventExtensions(target)** Traps `Object.preventExtensions()` which prevents new properties from being added to an object. Must return either `true` or `false`.
- **getOwnPropertyDescriptor(target, property)** Traps `Object.getOwnPropertyDescriptor()` which returns undefined or a property descrip tor object with attributes for `value`, `writable`, `get`, `set`, `configurable` and `enumerable`.
- **defineProperty(target, property, descriptor)** Traps `Object.defineProperty()` which defines or modifies an object property. Must return `true` if the target property was successfully defined or `false` if not.

Proxy Example 1: Profiling

Proxies allow you to create generic wrappers for any object without having to change the code within the target objects themselves.

"Proxies allow you to create generic wrappers for any object without having to change the code within the target objects themselves."

In this example we'll create a profiling proxy which counts the number of times a property is accessed. First, we require a `makeProfiler` factory function which returns the Proxy object and retains the count state:

```
// create a profiling Proxy
function makeProfiler(target) {

  var
    count = {},
    handler = {
```

```
      get: function(target, name) {

        if (name in target) {
          count[name] = (count[name] || 0) + 1;
          return target[name];
        }
      }
    };

  return {
    proxy: new Proxy(target, handler),
    count: count
  }

};
```

We can now apply this proxy wrapper to any object or another proxy, e.g.

```
var myObject = {
  h: 'Hello',
  w: 'World'
};

// create a myObject proxy
var pObj = makeProfiler(myObject);

// access properties
console.log(pObj.proxy.h); // Hello
console.log(pObj.proxy.h); // Hello
console.log(pObj.proxy.w); // World
console.log(pObj.count.h); // 2
console.log(pObj.count.w); // 1
```

While this is a trivial example, imagine the effort involved if you had to perform property access counts in several different objects without using proxies.

Proxy Example 2: Two-Way Data Binding

Data binding synchronizes objects. It's typically used in JavaScript MVC libraries to update an internal object when the DOM changes and vice versa.

Presume we have an input field with an id of `inputname`:

```
<input type="text" id="inputname" value="" />
```

We also have a JavaScript object named `myUser` with an `id` property which references this input:

```
// internal state for #inputname field
var myUser = {
  id: 'inputname',
  name: ''
};
```

Our first objective is to update `myUser.name` when a user changes the input value. This can be achieved with an `onchange` event handler on the field:

```
inputChange(myUser);

// bind input to object
function inputChange(myObject) {

  if (!myObject || !myObject.id) return;

  var input = document.getElementById(myObject.id);
  input.addEventListener('onchange', function(e) {
    myObject.name = input.value;
  });
}
```

Our next objective is to update the input field when we modify `myUser.name` within JavaScript code. This is not as simple but proxies offer a solution:

```
// proxy handler
var inputHandler = {

  set: function(target, prop, newValue) {

    if (prop == 'name' && target.id) {

      // update object property
      target[prop] = newValue;

      // update input field value
      document.getElementById(target.id).value = newValue;

      return true;
    }
    else return false;

  }

}

// create proxy
var myUserProxy = new Proxy(myUser, inputHandler);

// set a new name
myUserProxy.name = 'Craig';
console.log(myUserProxy.name); // Craig
console.log(document.getElementById('inputname').value); // Craig
```

This is not be the most efficient data-binding option but proxies allow you to alter the behavior of many existing objects without changing their code.

Further Examples

Hemanth.HM's article Negative Array Index in JavaScript suggests using proxies to implement negative array indexes, e.g. `arr[-1]` returns the last element, `arr[-2]` returns the second-to-last element, etc.

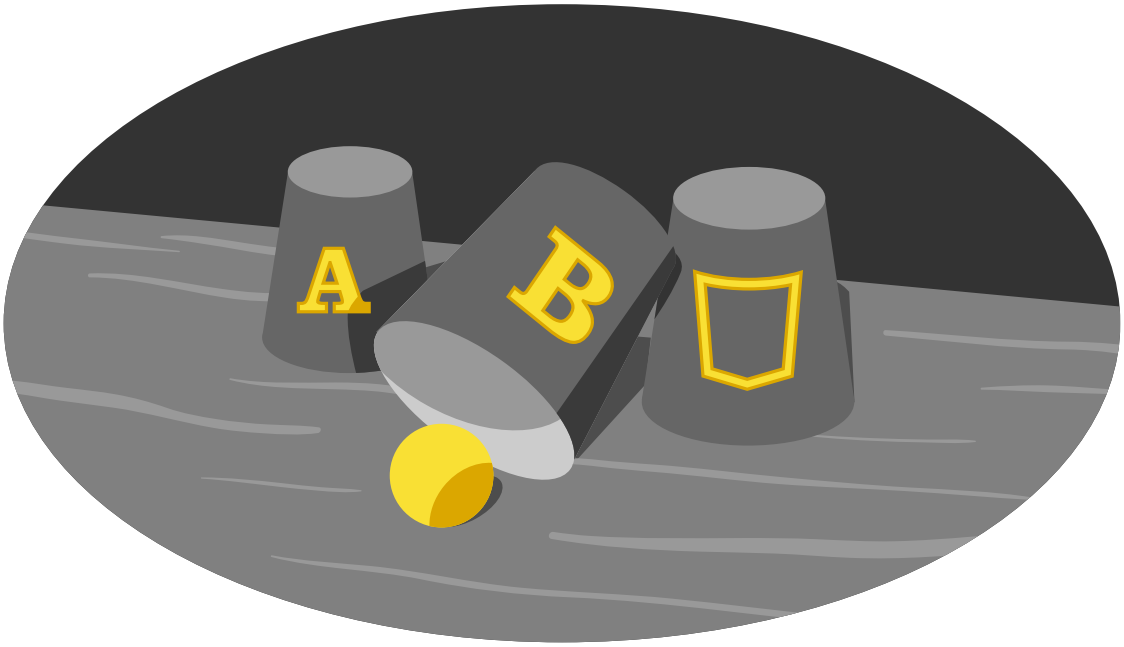
Nicholas C. Zakas' article Creating type-safe properties with ECMAScript 6 proxies illustrates how proxies can be used to implement type safety by validating new values. In the example above, we could verify `myUserProxy.name` was always set to a string and throw and error otherwise.

Proxy Support

The power of proxies may not be immediately obvious but they offer powerful meta-programming opportunities. Brendan Eich, the creator of JavaScript, thinks Proxies are Awesome!

As of end 2015, basic proxy support is implemented in Edge and Firefox 18+ although not all traps can be used. Experimental support is available in Node 4.0+ if you run Node with the `--harmony-proxies` flag but use it at your own risk.

Unfortunately, it's not possible to polyfill or transpile ES6 proxy code using tools such as Babel because they're powerful and have no ES5 equivalent. A little more waiting may be necessary



Preparing for ECMAScript 6: Destructuring Assignment

Destructuring assignment sounds complex. It reminds me of object oriented terms such as *encapsulation* and *polymorphism* — I'm convinced they were chosen to make simple concepts appear more sophisticated!

By Craig Buckler
@craigbuckler
craigbuckler.com

In essence, ECMAScript 6 (ES2015) destructuring assignment allows you to extract individual items from arrays or objects and place them into variables using a shorthand syntax. Those coming from PHP may have encountered the list() function which extracts arrays into variables in one operation. ES6 takes it to another level.

Presume we have an array:

```
var myArray = ['a', 'b', 'c'];
```

We can extract these values by index in ES5:

```
var
  one   = myArray[0],
  two   = myArray[1],
  three = myArray[2];

// one = 'a', two = 'b', three = 'c'
```

ES6 destructuring permits a simpler and less error-prone alternative:

```
var [one, two, three] = myArray;

// one = 'a', two = 'b', three = 'c'
```

You can ignore certain values, e.g.

```
var [one, , three] = myArray;

// one = 'a', three = 'c'
```

or use the spread operator (`...`) to extract remaining elements:

```
var [one, ...two] = myArray;

// one = 'a', two = ['b, 'c']
```

Destructuring also works on objects, e.g.

```
var myObject = {
  one:   'a',
  two:   'b',
  three: 'c'
};

// ES5 example
var
  one   = myObject.one,
  two   = myObject.two,
  three = myObject.three;
```

```
// one = 'a', two = 'b', three = 'c'

// ES6 destructuring example
var {one, two, three} = myObject;

// one = 'a', two = 'b', three = 'c'
```

In this example, the variable names `one`, `two` and `three` matched the object property names. We can also assign properties to variables with any name, e.g.

```
var myObject = {
  one:   'a',
  two:   'b',
  three: 'c'
};

// ES6 destructuring example
var {one: first, two: second, three: third} = myObject;

// first = 'a', second = 'b', third = 'c'
```

More complex nested objects can also be referenced, e.g.

```
var meta = {
  title: 'Destructuring Assignment',
  authors: [
    {
      firstname: 'Craig',
      lastname: 'Buckler'
    }
  ],
  publisher: {
    name: 'SitePoint',
    url: 'http://www.sitepoint.com/'
  }
};

var {
    title: doc,
```

```
    authors: [{ firstname: name }],
    publisher: { url: web }
  } = meta;

/*
  doc   = 'Destructuring Assignment'
  name  = 'Craig'
  web   = 'http://www.sitepoint.com/'
*/
```

This appears a little complicated but remember that in all destructuring assignments:

- the left-hand side of the assignment is the **destructuring target**; the pattern which defines the variables being assigned
- the right-hand side of the assignment is the **destructuring source**; the array or object which holds the data being extracted

There are a number of other caveats. First, you cannot start a statement with a curly brace because it looks like a code block, e.g.

```
// THIS FAILS
{ a, b, c } = myObject;
```

You must either declare the variables, e.g.

```
// THIS WORKS
var { a, b, c } = myObject;
```

or use parenthesis if variables are already declared, e.g.

```
// THIS WORKS
({ a, b, c }) = myObject;

// SO DOES THIS!
({ a, b, c } = myObject);
```

You should also be wary of mixing declared and undeclared variables, e.g.

```
// THIS FAILS
var a;
var { a, b, c } = myObject;

// THIS WORKS
var a, b, c;
({ a, b, c }) = myObject;
```

That's the basics of destructuring. So when would it be useful? I'm glad you asked…

Easier Declaration

Variables can be declared without explicitly defining each value, e.g.

```
// ES5
var a = 'one', b = 'two', c = 'three';

// ES6
var [a, b, c] = ['one', 'two', 'three'];
```

Admittedly, the destructured version is longer. It's a little easier to read although that may not be the case with more items.

Variable Value Swapping

Swapping values in ES5 requires a temporary third variable but it's far simpler with destructuring:

"Swapping values in ES5 requires a temporary third variable but it's far simpler with destructuring"

```
var a = 1, b = 2;

// ES5 swap
var temp = a;
a = b;
b = temp;

// a = 2, b = 1
```

```
// ES6 swap back
[a, b] = [b, a];

// a = 1, b = 2
```

You're not limited to two variables — any number of items can be rearranged, e.g.

```
// rotate left
[b, c, d, e, a] = [a, b, c, d, e];
```

Default Function Parameters

Presume we had a function to output our `meta` object:

```
var meta = {
  title: 'Destructuring Assignment',
  authors: [
    {
      firstname: 'Craig',
      lastname: 'Buckler'
    }
  ],
  publisher: {
    name: 'SitePoint',
    url: 'http://www.sitepoint.com/'
  }
};

prettyPrint(meta);
```

In ES5, it's necessary to parse this object to ensure appropriate defaults are available, e.g.

```
// ES5 default values
function prettyPrint(param) {

  param = param || {};
  var
```

```
    pubTitle = param.title || 'No title',
    pubName = (param.publisher && param.publisher.name) || 'No publisher';

  return pubTitle + ', ' + pubName;
}
```

In ES6 we can assign a default value to any parameter, e.g.

```
// ES6 default value
function prettyPrint(param = {}) {
```

but we can then use destructuring to extract values and assign defaults where necessary:

```
// ES6 destructured default value
function prettyPrint(
  {
    title: pubTitle = 'No title',
    publisher: { name: pubName = 'No publisher' }
  } = {}
) {

  return pubTitle + ', ' + pubName;
}
```

I'm not convinced this is easier to read but it is significantly shorter.

Returning Multiple Values from a Function

Functions can only return one value but that can be a complex object or multi-dimensional array. Destructuring assignment makes this more practical, e.g.

```
function f() {
  return [1, 2, 3];
}

var [a, b, c] = f();

// a = 1, b = 2, c = 3
```

For-of Iteration

Consider an array of book information:

```
var books = [
  {
    title: 'Full Stack JavaScript',
    author: 'Colin Ihrig and Adam Bretz',
    url: 'http://www.sitepoint.com/store/full-stack-javascript-development-mean/'
  },
  {
    title: 'JavaScript: Novice to Ninja',
    author: 'Darren Jones',
    url: 'http://www.sitepoint.com/store/leaern-javascript-novice-to-ninja/'
  },
  {
    title: 'Jump Start CSS',
    author: 'Louis Lazaris',
    url: 'http://www.sitepoint.com/store/jump-start-css/'
  },
];
```

The ES6 for-of is similar to `for-in` except that it extracts each value rather than the index/key, e.g.

```
for (let b of books) {
  console.log(b.title + ' by ' + b.author + ': ' + b.url);
}
```

Destructuring assignment provides further enhancements, e.g.

```
for (let {title, author, url} of books) {
  console.log(title + ' by ' + author + ': ' + url);
}
```

Regular Expression Handling

Regular expressions functions such as match return an array of matched items which can form the source of a destructuring assignment:

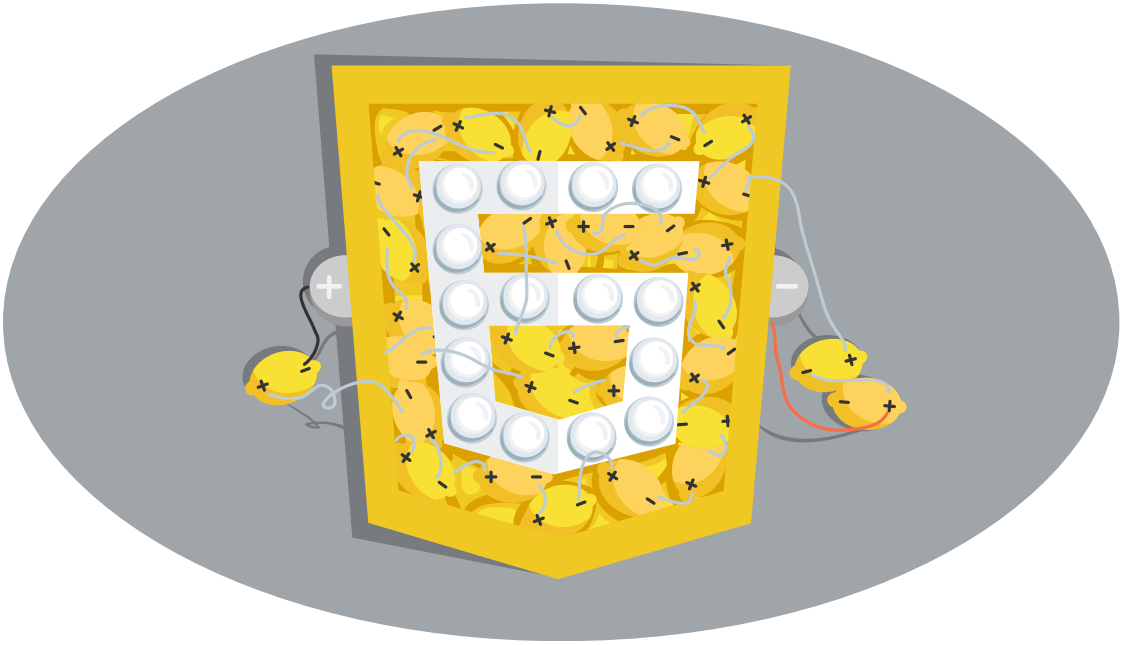
```
var [a, b, c, d] = 'one two three'.match(/\w+/g);

// a = 'one', b = 'two', c = 'three', d = undefined
```

Destructuring Assignment Support

Destructuring assignment may not revolutionize your development life but it could save some considerable typing effort!

As of end 2015, support for destructuring assignment is still incomplete. It's available in Firefox 34+ with basic features in Safari 7.1+. Experimental support is available in Node 4.0+ if you run Node with the `--harmony-destructuring` flag but it's best not to depend on it.

Until ES6 is generally available, compilers such as Babel and Traceur will translate ES6 destructuring assignments to an ES5 equivalent.



ECMAScript 2015: Generators and Iterators

A few days ago ECMAScript 2015, also known as ECMAScript 6 or ES6, was accepted as an official standard. Even though this event happened very recently, there are already a number of features supported in the latest versions of Chrome, Firefox, Safari, and Opera. If you're really itching for the ES6 goodness, you can even use a number of well-supported transpilers right now.

By Byron Houwens
@bhouwen

Two of the new features, generators and iterators, are set to change some of the ways we write specific functions when it comes to some more complex front-end code. While they do play nicely with one another, what they actually do can be a little confusing to some, so let's check them out.

Iterators

Iteration is a common practice in programming and is usually used to loop over a set of values, either transforming each value, or using or saving it in some way for later.

In JavaScript we've always had `for` loops that look like this:

```
for (var i = 0; i < foo.length; i++) {
  // do something with i
}
```

But ES6 gives us an alternative:

```
for (var i of foo) {
  // do something with i
}
```

This is arguably way cleaner and easier to work with, and reminds me languages like Python and Ruby. But there's something else that's pretty important to note about this new kind of iteration: it allows you to interact with elements of a data set directly.

"This is arguably way cleaner and easier to work with, and reminds me languages like Python and Ruby."

Imagine that we want to find out if each number in an array is prime or not. We could do this by coming up with a function that does exactly that. It might look like this:

```
function isPrime(number) {
  if (number < 2) {
    return false;
  } else if (number === 2) {
    return true;
  }

  for (var i = 2; i < number; i++) {
    if (number % i === 0) {
      return false;
      break;
    }
  }
```

```
  return true;
}
```

Not the best in the world, but it works. The next step would be to loop over our list of numbers and check whether each one is prime with our shiny new function. It's pretty straightforward:

```
var possiblePrimes = [73, 6, 90, 19, 15];
var confirmedPrimes = [];

for (var i = 0; i < possiblePrimes.length; i++) {
  if (isPrime(possiblePrimes[i])) {
    confirmedPrimes.push(possiblePrimes[i]);
  }
}

// confirmedPrimes is now [73, 19]
```

Again, it works, but it's clunky and that clunkiness is largely down to the way JavaScript handles `for` loops. With ES6, though, we're given an almost Pythonic option in the new iterator. So the previous `for` loop could be written like this:

```
let possiblePrimes = [73, 6, 90, 19, 15];
let confirmedPrimes = [];

for (let i of possiblePrimes) {
  if (isPrime(i)) {
    confirmedPrimes.push(i);
  }
}

// confirmedPrimes is now [73, 19]
```

This is far cleaner, but the most striking bit of this is that `for` loop. The variable `i` now represents the actual item in the array called `possiblePrimes`. So, we don't have to call it by index anymore. This means that instead of calling `possiblePrimes[i]` in the loop, we can just call `i`.

Behind the scenes, this kind of iteration is making use of ES6's bright and shiny Symbol.iterator()

method. This bad boy is in charge of describing the iteration and, when called, returns a JavaScript object containing the next value in the loop and a `done` key that is either `true` or `false` depending on whether or not the loop is finished.

"Behind the scenes, this kind of iteration is making use of ES6's bright and shiny Symbol.iterator() method."

In case you're interested in these sort of details, you can read more about it on this fantastic blog post titled Iterators gonna iterate by Jake Archibald. It'll also give you a good idea of what's going on under the hood when we dive into the other side of this article: generators.

Generators

Generators, also called "iterator factories", are a new type of JavaScript function that creates specific iterations. They give you special, self-defined ways to loop over stuff.

Okay, so what does all that mean? Let's look at an example. Let's say that we want a function that will give us the next prime number every time we call it. Again, we'll use our `isPrime` function from before to check if a number is prime:

```
function* getNextPrime() {
  let nextNumber = 2;

  while (true) {
    if (isPrime(nextNumber)) {
      yield nextNumber;
    }
    nextNumber++;
  }
}
```

If you're used to JavaScript, some of this stuff will look a bit like voodoo, but it's actually not too bad. /We have that strange asterisk after the keyword `function`, but all this does is to tell JavaScript that we're defining a generator.

"If you're used to JavaScript, some of this stuff will look a bit like voodoo, but it's actually not too bad."

The other funky bit would be the `yield` keyword. This is actually what a generator spits out when you call it. It's roughly equivalent to `return` but it keeps the state of the function instead of rerunning

everything whenever you call it. It "remembers" its place while running so the next time you call it, it carries on where it left off.

This means that we can do this:

```
var nextPrime = getNextPrime();
```

And then call `nextPrime` whenever we want, you guessed it, the next prime:

```
console.log(nextPrime.next().value); // 2
console.log(nextPrime.next().value); // 3
console.log(nextPrime.next().value); // 5
console.log(nextPrime.next().value); // 7
```

You get the picture. You can also just call `nextPrime.next()`, which is useful in situations where your generator isn't infinite, because it returns an object like this:

```
console.log(nextPrime.next());
// {value: 2, done: false}
```

Where that `done` key tells you whether the function has completed its task. In our case our function will never finish, and could theoretically give us all prime numbers up to infinity (if we had that much computer memory, of course).

Cool, So Can I Use This Now?

Although ECMAScript 2015 has been finalized, browser support for its features, particularly generators, is not excellent. If you really want to use these and other new features, you can check out transpilers like Babel and Traceur, which will convert your ECMAScript 2015 code into its equivalent (where possible) ECMAScript 5 code.

There are also many online editors with support for ECMAScript 2015 or that specifically focus on it, particularly Facebook's Regenerator and JS Bin. If you're just looking to play around and get a feel for how JavaScript will be written in the near future, those are worth a look.

Conclusions

Iterator and generators give us quite a lot of new flexibility in our approach to JavaScript problems. Iterators allow us a more Pythonic way of writing `for` loops, which means our code will look cleaner and be easier to read.

Generator functions give us the ability to write functions that remember where they were when you last saw them, and can pick up where they left off. They can also be infinite in terms of how much they actually remember, which can come in really handy in certain situations.

Although browser support for these features isn't great yet, there are a number of transpilers and online editors with enough support for you to get your hands dirty if you're keen. Anyway, I think we, as developers using these new features, will really love what they have to offer.

An Overview of JavaScript Promises

By Sandeep Panda
@Sandeepg33k
Hashnode

Well, this has come like a Christmas gift to all JavaScript developers. You will be glad to know that promises are now a part of standard JavaScript. Chrome 32 beta has already implemented the basic promise API. The concept of promises is not new to web development.

Many of us have already used promises in the form of several JS libraries such as Q, when, RSVP.js, etc. Even jQuery has something called a Deferred object which is similar to a promise. But having native support for promises in JavaScript is really amazing. This tutorial will cover the basics of promises and show how you can leverage them in your JS development.

Note: This is still an experimental feature. Only Chrome 32 beta and the latest Firefox nightly currently support it.

"A Promise object represents a value that may not be available yet, but will be resolved at some point in the future. It allows you to write asynchronous code in a more synchronous fashion."

Overview

A `Promise` object represents a value that may not be available yet, but will be resolved at some point in the future. It allows you to write asynchronous code in a more synchronous fashion. For example, if you use the promise API to make an asynchronous call to a remote web service you will create a `Promise` object which represents the data that will be returned by the web service in future. The caveat being that the actual data is not available yet. It will become available when the request completes and a response comes back from the web service.

In the meantime the `Promise` object acts like a proxy to the actual data. Furthermore, you can attach callbacks to the `Promise` object which will be called once the actual data is available.

The API

To get started, let's examine the following code which creates a new `Promise` object.

```
if (window.Promise) { // Check if the browser supports Promises
  var promise = new Promise(function(resolve, reject) {
    //asynchronous code goes here
  });
}
```

We start by instantiating a new `Promise` object and passing it a callback function. The callback takes two arguments, `resolve` and `reject`, which are both functions. All your asynchronous code goes inside that callback. If everything is successful, the promise is fulfilled by calling `resolve()`. In case of an error, `reject()` is called with an `Error` object. This indicates that the promise is rejected.

Now let's build something simple which shows how promises are used. The following code makes an asynchronous request to a web service that returns a random joke in JSON format. Let's examine how promises are used here.

```
if (window.Promise) {
  console.log('Promise found');

  var promise = new Promise(function(resolve, reject) {
    var request = new XMLHttpRequest();
```

```
    request.open('GET', 'http://api.icndb.com/jokes/random');
    request.onload = function() {
      if (request.status == 200) {
        resolve(request.response); // we got data here, so resolve the Promise
      } else {
        reject(Error(request.statusText)); // status is not 200 OK, so reject
      }
    };

    request.onerror = function() {
      reject(Error('Error fetching data.')); // error occurred, reject the  Promise
    };

    request.send(); //send the request
  });

  console.log('Asynchronous request made.');

  promise.then(function(data) {
    console.log('Got data! Promise fulfilled.');
    document.getElementsByTagName('body')[0].textContent = JSON.parse(data).value.joke;
  }, function(error) {
    console.log('Promise rejected.');
    console.log(error.message);
  });
} else {
  console.log('Promise not available');
}
```

In the previous code, the `Promise` constructor callback contains the asynchronous code used to get data the from remote service. Here, we just create an Ajax request to http://api.icndb.com/jokes/random which returns a random joke. When a JSON response is received from the remote server, it is passed to `resolve()`. In case of any error, `reject()` is called with an `Error` object.

When we instantiate a `Promise` object we get a proxy to the data that will be available in future. In our case we are expecting some data to be returned from the remote service at some point in future. So, how do we know when the data becomes available? This is where the `Promise.then()` function is used. This function takes two arguments: a success callback and a failure callback. These callbacks are called when the `Promise` is settled (i.e. either fulfilled or rejected). If the promise was fulfilled, the success

callback will be fired with the actual data you passed to `resolve()`. If the promise was rejected, the failure callback will be called. Whatever you passed to `reject()` will be passed as an argument to this callback.

Try this Plunkr example. Simply refresh the page to view a new random joke. Also, open up your browser console so that you can see the order in which the different parts of the code are executed. Note that a promise can have three states:

- pending (not fulfilled or rejected)
- fulfilled
- rejected

The `Promise.status` property, which is code-inaccessible and private, gives information about these states. Once a promise is rejected or fulfilled, this status gets permanently associated with it. This means a promise can succeed or fail only once. If the promise has already been fulfilled and later you attach a `then()` to it with two callbacks, the success callback will be correctly called. So, in the world of promises, we are not interested in knowing when the promise is settled. We are only concerned with the final outcome of the promise.

"It is sometimes desirable to chain promises together. For instance, you might have multiple asynchronous operations to be performed."

Chaining Promises

It is sometimes desirable to chain promises together. For instance, you might have multiple asynchronous operations to be performed. When one operation gives you data, you will start doing some other operation on that piece of data and so on. Promises can be chained together as demonstrated in the following example.

```
function getPromise(url) {
  // return a Promise here
  // send an async request to the url as a part of promise
  // after getting the result, resolve the promise with it
}

var promise = getPromise('some url here');

promise.then(function(result) {
```

```
  //we have our result here
  return getPromise(result); //return a promise here again
}).then(function(result) {
  //handle the final result
});
```

The tricky part is that when you return a simple value inside `then()`, the next `then()` is called with that return value. But if you return a promise inside `then()`, the next `then()` waits on it and gets called when that promise is settled.

Handling Errors

You already know the `then()` function takes two callbacks as arguments. The second one will be called if the promise was rejected. But, we also have a `catch()` function which can be used to handle promise rejection. Have a look at the following code:

```
promise.then(function(result) {
  console.log('Got data!', result);
}).catch(function(error) {
  console.log('Error occurred!', error);
});
```

This is equivalent to:

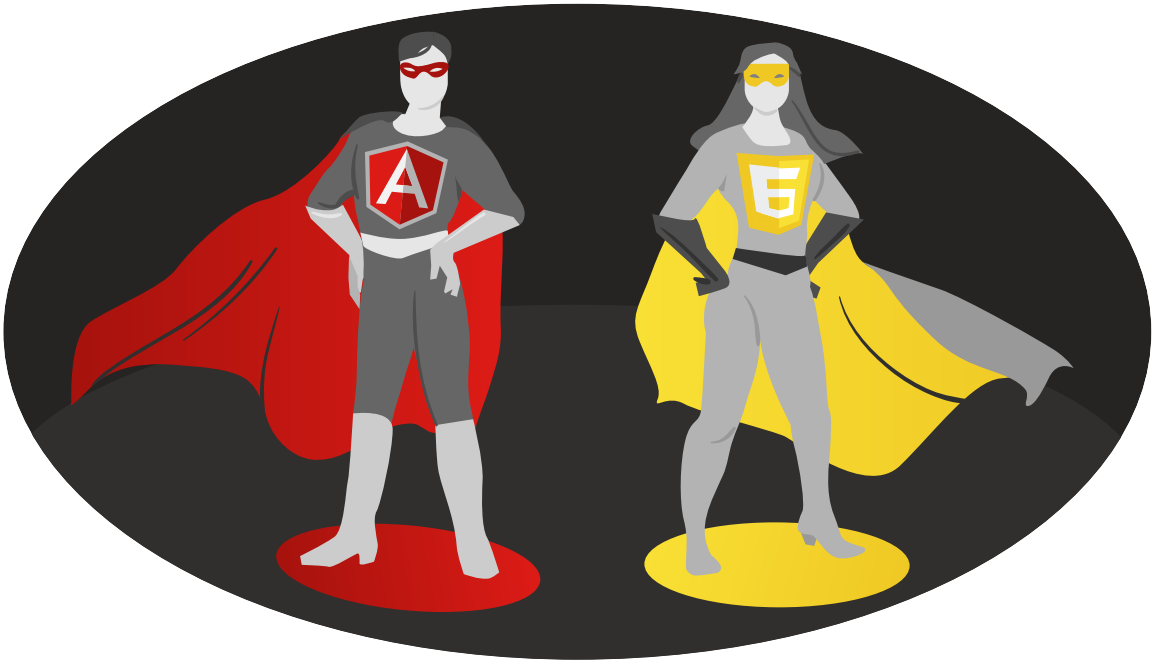
```
promise.then(function(result) {
  console.log('Got data!', result);
}).then(undefined, function(error) {
  console.log('Error occurred!', error);
});
```

Note that if the promise was rejected and `then()` does not have a failure callback, the control will move forward to the next `then()` with a failure callback or the next `catch()`. Apart from explicit promise rejection, `catch()` is also called when any exception is thrown from the `Promise` constructor callback. So, you can also use `catch()` for logging purposes. Note that we could use `try...catch` to handle errors, but that is not necessary with promises as any asynchronous or synchronous error is always caught by `catch()`.

Conclusion

This was just a brief introduction to JavaScript's new Promises API. Clearly it lets us write asynchronous code very easily. We can proceed as usual without knowing what value is going to be returned from the asynchronous code in the future. There is more to the API, which has not been covered here. To learn more about Promises, browse the following resources, and stay tuned to SitePoint!

- HTML5Rocks
- Mozilla Developer Network



Writing AngularJS Apps Using ES6

By Rabi Kiran
@SRavi_Kiran

As many of you are aware, ECMAScript 6 is in its draft state now and is expected to be finalized some time this year. But it has already caught a lot of attention in the community and browsers have already started implementing it.

We also have a number of transpilers like Traceur, 6to5, and many others that convert ES6 code to ES5 compatible code. Community members have started playing around with ES6 and many of them are blogging about what they learn. SitePoint's JavaScript channel also has a good number of articles describing the different features of ES6.

It is possible to write any piece of everyday JavaScript using ES6. To do this, we need to be aware of the key features of ES6 and know which piece fits where. In this article, we will see how we can use features of ES6 to build different pieces of an AngularJS application and load them using ES6 modules. We will do this by building a simple online book shelf application and we will see how it is structured and written.

As ever, code for this application can be found on our GitHub repository.

A Note on the Bookshelf Application

The sample BookShelf application contains following views:

- **Home page:** Shows a list of active books. Books can be marked as read and moved to the archive from this page
- **Add book page:** Adds a new book to the shelf by accepting the title of the book and name of the author. It doesn't allow a duplicate title
- **Archive page:** Lists all archived books

Setting up the Application for ES6

As we will be using ES6 to write the front-end part of the application, we need a transpiler to make the ES6 features understandable for all the browsers. We will be using the Traceur client-side library to compile our ES6 script on the fly and run it in the browser. This library is available on bower. The sample code has an entry for this library in `bower.json`.

On the home page of the application, we need to add a reference to this library and the following script:

```
traceur.options.experimental = true;
new traceur.WebPageTranscoder(document.location.href).run();
```

The app's JavaScript code is divided into multiple files. These files are loaded into the main file using the ES6 module loader. As today's browsers can't understand ES6 modules, Traceur polyfills this feature for us.

In the sample code, the `bootstrap.js` file is responsible for loading the main AngularJS module and manually bootstraping the Angular app. We cannot use `ng-app` to bootstrap the application as the modules are loaded asynchronously. This is the code contained in that file:

```
import { default as bookShelfModule} from './ES6/bookShelf.main';
angular.bootstrap(document, [bookShelfModule]);
```

Here, `bookShelfModule` is name of the AngularJS module containing all the pieces. We will see the content of the `bookShelf.main.js` file later. The `bootstrap.js` file is loaded in the `index.html` file using the following script tag:

```
<script type="module" src="ES6/bootstrap.js"></script>
```

Defining Controllers

AngularJS controllers can be defined in two ways:

1. Controllers using `$scope`
2. Using the *controller as* syntax

The second approach fits better with ES6, as we can define a class and register it as a controller. The properties associated with an instance of the class will be visible through the controller's alias. In addition, the *controller as* syntax is comparatively less coupled with `$scope`. If you are not aware, `$scope` will be removed from the framework in Angular 2, so we can train our brains to be less dependent on `$scope` from now on by using the *controller as* syntax.

"Though classes in ES6 keep us away from the difficulty of dealing with prototypes, they don't support a direct way of creating private fields."

Though classes in ES6 keep us away from the difficulty of dealing with prototypes, they don't support a direct way of creating private fields. There are some indirect ways to create private fields in ES6. One of them is to store the values using variables at module level and not including them in the export object.

We will use a WeakMap to store the private fields. The Reason behind choosing WeakMap is that those entries that have objects as keys are removed once the object is garbage collected.

As stated above, the home page of the application loads and displays a list of active books. It depends on a service to fetch data and to mark a book as read, or to move it to the archive. We will create this service in the next section. So that the dependencies injected into controller's constructor are available in instance methods, we need to store them in the WeakMaps. The home page's controller has two dependencies: the service performing the Ajax operations and `$timeout` (used to show success messages and hide them after a certain time). We also need a private `init` method to fetch all active books as soon as the controller loads. So, we need three WeakMaps. Let's declare the WeakMaps as constants to prevent any accidental re-assignment.

The following snippet creates these WeakMaps and the class `HomeController`:

```
const INIT = new WeakMap();
const SERVICE = new WeakMap();
```

```
const TIMEOUT = new WeakMap();

class HomeController{
  constructor($timeout, bookShelfSvc){
    SERVICE.set(this, bookShelfSvc);
    TIMEOUT.set(this, $timeout);
    INIT.set(this, () => {
      SERVICE.get(this).getActiveBooks().then(books => {
        this.books = books;
      });
    });

    INIT.get(this)();
  }

  markBookAsRead(bookId, isBookRead){
    return SERVICE.get(this).markBookRead(bookId, isBookRead)
      .then(() => {
        INIT.get(this)();
        this.readSuccess = true;
        this.readSuccessMessage = isBookRead ? "Book marked as read." : "Book marked as
unread.";
        TIMEOUT.get(this)(() => {
          this.readSuccess = false;
        }, 2500);
      });
  }

  addToArchive(bookId){
    return SERVICE.get(this).addToArchive(bookId)
      .then(() => {
        INIT.get(this)();
        this.archiveSuccess = true;
        TIMEOUT.get(this)(() => {
          this.archiveSuccess = false;
        }, 2500);
      });
  }
}
```

The above snippet uses following ES6 features:

- Classes and WeakMaps, as already mentioned
- The arrow function syntax to register callbacks. The `this` reference inside the arrow functions is same as the `this` reference outside, which is the current instance of the class
- The new syntax for creating a method and attaching it to an object without using the `function` keyword

Let's apply dependency injection and register this class as a controller:

```
HomeController.$inject = ['$timeout', 'bookShelfSvc'];

export default HomeController;
```

As you see, there is no difference in the way that we applied dependency injection — it is same as the way we do in ES5. We are exporting the `HomeController` class from this module.

Check the code of `AddBookController` and `ArchiveController`. They follow a similar structure. The file `bookShelf.controllers.js` imports these controllers and registers them to a module. This is the code from this file:

```
import HomeController from './HomeController';
import AddBookController from './AddBookController';
import ArchiveController from './ArchiveController';

var moduleName='bookShelf.controllers';

angular.module(moduleName, [])
  .controller('bookShelf.homeController', HomeController)
  .controller('bookShelf.addBookController', AddBookController)
  .controller('bookShelf.archiveController', ArchiveController);

export default moduleName;
```

The `bookShelf.controllers` module exports the name of the AngularJS module it created, so that this can be imported into another module to create to create the main module.

Defining Services

"Service" is an overloaded term in general and in Angular as well! The three types of services used are: *providers*, *services* and *factories*. Out of these, providers and services are created as instances of types, so we can create classes for them. Factories are functions that return objects. I can think of two approaches for creating a factory:

"Service" is an overloaded term in general and in Angular as well! The three types of services used are: providers, services and factories.

- The same as in ES5, create a function which returns an object
- A class with a static method which returns an instance of the same class. This class would contain the fields that have to be exposed from the factory object

Let's use the second approach to define a factory. This factory is responsible for interacting with the Express API and serving data to the controllers. The factory depends on Angular's `$http` service to perform Ajax operations. As it has to be a private field in the class, we will define a WeakMap for it.

The following snippet creates the factory class and registers the static method as a factory:

```
var moduleName='bookShelf.services';

const HTTP = new WeakMap();

class BookShelfService
{
  constructor($http)
  {
    HTTP.set(this, $http);
  }

  getActiveBooks(){
    return HTTP.get(this).get('/api/activeBooks').then(result => result.data );
  }

  getArchivedBooks(){
    return HTTP.get(this).get('/api/archivedBooks').then(result => result.data );
  }
```

```
  markBookRead(bookId, isBookRead){
    return HTTP.get(this).put(`/api/markRead/${bookId}`, {bookId: bookId, read: is-
BookRead});
  }

  addToArchive(bookId){
    return HTTP.get(this).put(`/api/addToArchive/${bookId}`,{});
  }

  checkIfBookExists(title){
    return HTTP.get(this).get(`/api/bookExists/${title}`).then(result =>  result.data );
  }

  addBook(book){
    return HTTP.get(this).post('/api/books', book);
  }

  static bookShelfFactory($http){
    return new BookShelfService($http);
  }
}

BookShelfService.bookShelfFactory.$inject = ['$http'];

angular.module(moduleName, [])
  .factory('bookShelfSvc', BookShelfService.bookShelfFactory);

export default moduleName;
```

This snippet uses the following additional features of ES6 (in addition to classes and arrow functions):

- A static member in the class
- String templates to concatenate the values of variables into strings

Defining Directives

Defining a directive is similar to defining a factory, with one exception — we have to make an instance of the directive available for later use inside the `link` function, because the `link` function is not called in the context of the directive object. This means that the `this` reference inside the `link` function is not the

same as the directive object. We can make the object available through a static field.

We will be creating an attribute directive that validates the title of the book entered in the text box. It has to call an API to check if the title exists already and invalidate the field if the title is found. For this task, it needs the service we created in the previous section and `$q` for promises.

The following snippet creates a directive which it registers with a module.

```
var moduleName='bookShelf.directives';

const Q = new WeakMap();
const SERVICE = new WeakMap();

class UniqueBookTitle
{
  constructor($q, bookShelfSvc){
    this.require='ngModel';  //Properties of DDO have to be attached to the instance
through this reference
    this.restrict='A';

    Q.set(this, $q);
    SERVICE.set(this, bookShelfSvc);
  }

  link(scope, elem, attrs, ngModelController){
    ngModelController.$asyncValidators.uniqueBookTitle = function(value){

      return Q.get(UniqueBookTitle.instance)((resolve, reject) => {
        SERVICE.get(UniqueBookTitle.instance).checkIfBookExists(value).then( result => {
          if(result){
            reject();
          }
          else{
            resolve();
          }
        });
      });
    };
  }
```

```
  static directiveFactory($q, bookShelfSvc){
    UniqueBookTitle.instance =new UniqueBookTitle($q, bookShelfSvc);
    return UniqueBookTitle.instance;
  }
}

UniqueBookTitle.directiveFactory.$inject = ['$q', 'bookShelfSvc'];

angular.module(moduleName, [])
  .directive('uniqueBookTitle', UniqueBookTitle.directiveFactory);

export default moduleName;
```

Here, we could have used ES6's promise API, but that would involve calling `$rootScope.$apply` after the promise produces a result. The good thing is that promise API in AngularJS 1.3 supports a syntax similar to the ES6 promises.

Defining the Main Module and Config block

Now that we have modules containing the directives, controllers and services, let's load them into one file and create the main module of the application. Let's begin by importing the modules.

```
import { default as controllersModuleName } from './bookShelf.controllers';
import { default as servicesModuleName } from './bookShelf.services';
import { default as directivesModuleName } from './bookShelf.directives';
```

The config block defines routes for the application. This can be a simple function as it doesn't have to return any value.

```
function config($routeProvider){
  $routeProvider
    .when('/',{
      templateUrl:'templates/home.html',
      controller:'bookShelf.homeController',
      controllerAs:'vm'
    })
    .when('/addBook',{
      templateUrl:'templates/addBook.html',
```

```
      controller:'bookShelf.addBookController',
      controllerAs:'vm'
    })
    .when('/archive', {
      templateUrl:'templates/archive.html',
      controller:'bookShelf.archiveController',
      controllerAs:'vm'
    })
    .otherwise({redirectTo:'/'});
}

config.$inject = ['$routeProvider'];
```

Finally, let's define the main module and export its name. If you remember, this name is used in the `bootstrap.js` file for manual bootstrapping.

```
var moduleName = 'bookShelf';

var app = angular.module(moduleName, ['ngRoute','ngMessages', servicesModuleName, con-
trollersModuleName, directivesModuleName])
  .config(config);

export default moduleName;
```

Conclusion

Hopefully this gives you an insight into using ES6 to write AngularJS apps. AngularJS 2.0 is being written completely using ES6 and as web developers we need to be aware of the way we have to write our code in the near future. ES6 solves many problems that have been bugging JavaScript programmers for years and using it with AngularJS is a lot of fun!

And please remember, the sample code for this application can be found on our GitHub repository.

Setting up an ES6 Project Using Babel and Browserify

The JavaScript world is changing and ES6 is rapidly taking over. Many famous frameworks like AngularJS 2 and React Native have already started supporting ES6. It's important that we are prepared for this change. To do so, we need to start writing code that uses ES6 even before the support for it lands in all browsers.

By Ritesh Kumar
@ritz078

In this article, I'll show you how to set up a project that integrates Babel and Browserify to write modern code that can be executed by older browsers as well. Babel compiles ES6 code into ES5 which is supported by many browsers, including old ones like Internet Explorer 9. Browserify is a tool for writing code that follows the CommonJS pattern and packaging it to be used in the browser.

Creating the package.json File

First of all, let's see the folder structure of the demo we are going to make

```
/
|--dist/
  |----modules.js
|--modules/
  |----import.js
  |----index.js
|--Gruntfile.js
|--package.json
```

In the project's root folder, there are two files `Gruntfile.js` and `package.json` and two folders `modules` and `dist`. The `modules` folder contains all the modules written in ES6 and the `dist` folder contains the bundled and compiled ES5 JavaScript file. I have excluded `.gitignore` as it is just a utility file which in no way affects the project.

Now, let's start by creating the `package.json` file. There are many fields in a typical `package.json` file like `description`, `version`, `author`, and others, but in this project we're only using the important ones.

The following is the content of the `package.json` file we'll use:

```
{
  "name": "browserify-babel-demo",
  "main": "dist/module.js",
  "devDependencies": {
    "grunt": "^0.4.5",
    "babelify": "^6.1.0",
    "grunt-browserify": "^3.8.0",
    "grunt-contrib-watch": "^0.6.1"
  }
}
```

As you can see from the file above, the modules used for this project are:

- Grunt: A JavaScript task runner
- grunt-browserify: The Browserify Grunt task
- babelify: Babel transformer for Browserify
- grunt-contrib-watch: A Grunt task to watch the JavaScript files for every change and then optionally execute tasks. In our case, we'll run the browserify task on each change

All these modules are `devDependencies` as they are only needed in the development environment and not when the client-side code is executed. The versions of the modules can be set according to need.

Now run `npm install` in the root folder of the project to install all the dependencies listed in the `package.json` file. In case you're not familiar with npm, I suggest you to read this article to get started with it.

Set up the Gruntfile.js

In this article I assume that you know what Grunt and a `Gruntfile.js` file are and how to work with Grunt. In case you need a refresher, I suggest you to go through this article before moving forward.

JavaScript files containing code written in ES6 can have either `.js` or `.es6` extension. Here, for simplification purposes, we are using the `.js` extension for all the JavaScript files (even for those written in ES6). The code written in `Gruntfile.js` is as shown below:

```
module.exports = function(grunt) {
  grunt.initConfig({
    browserify: {
      dist: {
        options: {
          transform: [
            ["babelify", {
              loose: "all"
            }]
          ]
        },
        files: {
          // if the source file has an extension of es6 then
          // we change the name of the source file accordingly.
          // The result file's extension is always .js
          "./dist/module.js": ["./modules/index.js"]
```

```
        }
      }
    },
    watch: {
      scripts: {
        files: ["./modules/*.js"],
        tasks: ["browserify"]
      }
    }
  });

  grunt.loadNpmTasks("grunt-browserify");
  grunt.loadNpmTasks("grunt-contrib-watch");

  grunt.registerTask("default", ["watch"]);
  grunt.registerTask("build", ["browserify"]);
};
```

We have defined two Grunt tasks:

1. `grunt default` / `grunt`: When we run this command on the terminal inside the project folder, this task starts watching all the JavaScript files included in the `modules` folder. For any change detected, Grunt will execute the browserify task. The watch task keeps running until the task is terminated. To terminate the task press Ctrl + C on the terminal.
2. `grunt build`: This task executes the browserify task once and stops.

Every time the browserify task executes, all the JavaScript code present inside the `modules` folder is bundled into a single JavaScript file. Then the code goes through babelify (Babel transformer for Browserify) which compiles that bundled ES6 code into ES5 code.

As seen in the above code, we have set `loose: 'all'` as an option for babelify as we want the ES5 code to be as close to the ES6 code that we are writing as possible. We don't want it to adhere strictly to the specification because this will be more difficult for an ES6 beginner to debug. All the other options provided by Babel can be found here.

Let's Write Some ES6 Code

This demo uses only a few features of ES6, such as `import` and `export`. So, if you want to have a deep dive into ES6, I suggest you to go through the ES6 tutorials published here on SitePoint. You'll get an

idea of all the new and exciting features that ES6 will bring to the table.

In our demo we'll create two files, `index.js` and `import.js`, inside the `modules` folder. The former is the main file of the project, while the latter contains all the functions and variables that are part of a module. In other words, `index.js` will import all the functions and variables from the `import.js` file.

The code of the `import.js` file is listed below:

```
var sum = (a, b = 6) => (a + b);

var square = (b) => {
  return b * b;
};

var variable = 8;

class MyClass {
  constructor(credentials) {
       this.name = credentials.name;
       this.enrollmentNo = credentials.enrollmentNo
  }
  getName() {
    return this.name;
  }
}

export { sum, square, variable, MyClass };
```

The `import.js` file is a module that contains a variable, a class, and function expressions (written using the arrow function). The functions and the variables defined in a module are not visible outside of the module unless we explicitly export them. You can do that by using the `export` keyword. In the last line of `import.js`, we have exported `sum`, `square`, `variable` and `MyClass`.

"Since we are using Browserify, we can also import modules using the CommonJS pattern by using the require() method."

In the `index.js` file we import all the variables of the module by using the `import` keyword. So, all these imported variables from `import.js` file become accessible in the main file, i.e. `index.js`. In the code below, which lists the contents of the file `index.js`, you can see how we are able to use the `square()` function or `MyClass` by importing the module that exports it.

We can import functions, variables, and classes from as many files as we want.

```
import {sum, square, variable, MyClass} from './import';

// 25
console.log(square(5));

var cred = {
  name: 'Ritesh Kumar',
  enrollmentNo: 11115078
}

var x = new MyClass(cred);

//Ritesh Kumar
console.log(x.getName());
```

In case we are importing from a file that has a `.es6` extension then we have to write the filename with the extension in `import`. An example is shown in in the code snippet below:

```
// if file extension of the importing file is .js
// both of the following methods work
import { sum, square, variable, MyClass } from './import';
import { sum, square, variable, MyClass } from './import.js'

// if file extension of the importing file is .es6
// it's mandatory to add the extension
import { sum, square, variable, MyClass } from './import.es6';
```

Since we are using Browserify, we can also import modules using the CommonJS pattern by using the `require()` method. For example, if we want to import jQuery as a module, we can use the following code:

```
var $ = require('path/to/jquery');
$(window).click(function(){
  //do something
});
```

Babel can convert ES6 code to ES5, but it can't bundle the modules. So, we are using Browserify for bundling the modules.

The power of ES6's `import` and `export` combined with the `require()` method, gives us the freedom to organize all of the client-side code into modules and at the same time write the code using all the power of the new version of JavaScript.

As soon as we run the `grunt` command on the terminal a few things will happen:

- Browserify will bundle all the files into one
- The bundled file is passed through babelify to transform the code into ES5
- A file named `module.js` that can be executed in all modern browsers, including Internet Explorer 9, is generated

To give you an idea of what the generated `module.js` file looks like, I'm including the resulting code below:

```
(function e(t,n,r){function s(o,u){if(!n[o]){if(!t[o]){var a=typeof require=="func-
tion"&&require;if(!u&&a)return a(o,!0);if(i)return i(o,!0);var f=new Error("Cannot find
module '"+o+"'");throw f.code="MODULE_NOT_FOUND",f}var l=n[o]={exports:{}};t[o][0].
call(l.exports,function(e){var n=t[o][1][e];return s(n?n:e)},l,l.exports,e,t,n,r)}return
n[o].exports}var i=typeof require=="function"&&require;for(var o=0;o<r.length;o++)
s(r[o]);return s})({1:[function(require,module,exports){
"use strict";

exports.__esModule = true;

function _classCallCheck(instance, Constructor) { if (!(instance instanceof Construc-
tor)) { throw new TypeError("Cannot call a class as a function"); } }

var sum = function sum(a) {
  var b = arguments.length <= 1 || arguments[1] === undefined ? 6 : arguments[1];
  return a + b;
};

var square = function square(b) {
  return b * b;
};
```

```
var variable = 8;

var MyClass = (function () {
  function MyClass(credentials) {
    _classCallCheck(this, MyClass);

    this.name = credentials.name;
    this.enrollmentNo = credentials.enrollmentNo;
  }

  MyClass.prototype.getName = function getName() {
    return this.name;
  };

  return MyClass;
})();

exports.sum = sum;
exports.square = square;
exports.variable = variable;
exports.MyClass = MyClass;

},{}],2:[function(require,module,exports){
'use strict';

var _import = require('./import');

console.log(_import.square(5));

var cred = {
  name: 'Ritesh Kumar',
  enrollmentNo: 11115078
};

var x = new _import.MyClass(cred);

console.log(x.getName());

},{"./import":1}]},{},[2]);
```

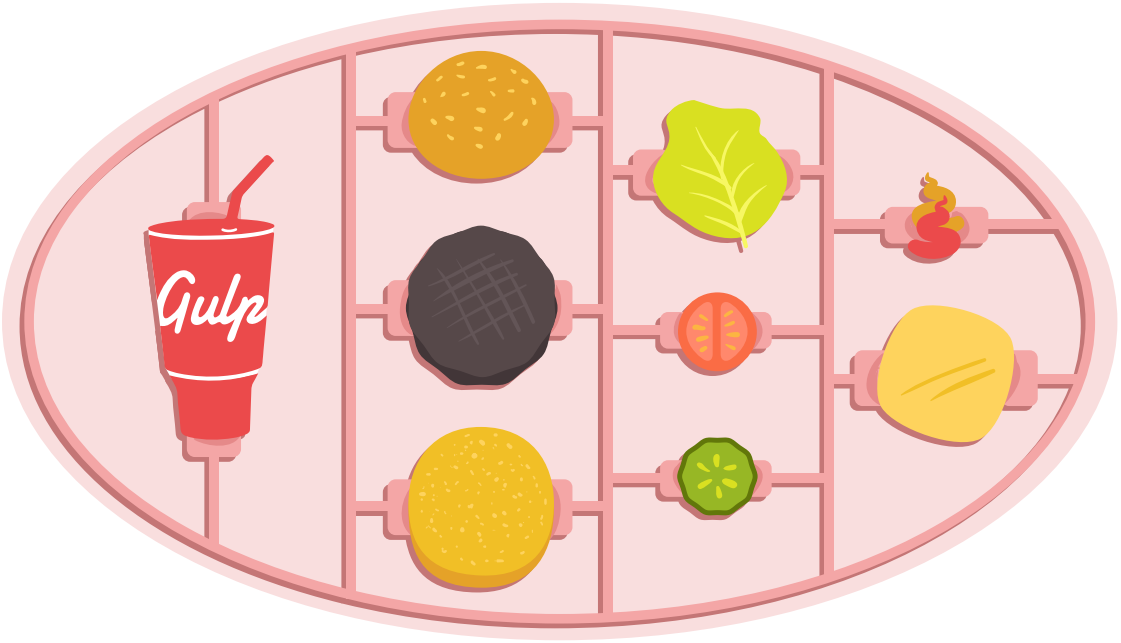
This file can be then be used in your web pages just as a normal JavaScript file. If you want, you can also use other Grunt tasks like grunt-uglify, grunt-rev, and many others on `module.js`. Once done, you're ready to include `module.js` in the HTML page and the browser will load it.

```
<!-- Usage of the final bundled file in html -->
<script src="path/to/module.js"></script>
```

Conclusion

In this article we've seen how to write a project that has its JavaScript code written using the features introduced by ES6. In addition, I covered how to configure Browserify and Babel as Grunt tasks using grunt-browserify and babelify respectively. We created a demo project which demonstrated how this setup works and the way in which ES6 code is compiled into ES5.

I hope that you have enjoyed the article. In case you want to play with this project, the code of the demo project is available here.



Transpiling ES6 Modules to AMD & CommonJS Using Babel & Gulp

ECMAScript 6 (a.k.a ECMAScript 2015 or ES6), the specification for next version of JavaScript has been approved and browser vendors are hard at work implementing it. Unlike the previous versions of ECMAScript, ES6 comes with a huge set of changes to the language to make it a good fit for the scale at which it is used today. SitePoint has a number of articles covering these features.

By Rabi Kiran
@SRavi_Kiran

Although browsers haven't implemented all of the features yet, we can already take advantage of ES6 during development and convert it to a version that browser understands before shipping the application. Babel and Traceur are two of the leading transpilers used for this purpose. Microsoft's typed

superset of JavaScript, TypeScript can also be used as an ES6 transpiler.

I covered how ES6 can be used today to write Angular 1.x applications in one of my previous articles. In that article I used Traceur's on-the-fly transpiler to run the application. Although it works, it is always better to transpile beforehand and reduce the amount of work to be done in the browser. In this article, we will see how the same sample application can be transpiled to ES5 and the modules into either CommonJS or, AMD using Babel to make it run on today's browsers. Though the sample is based on Angular, the techniques of transpilation can be used with any valid ES6 code.

As ever, you can find the code to accompany this article on our GitHub repo.

The Importance of Modules

One of the key features in any language used to write large applications, is the ability to load different pieces of the application in the form of modules. Modules not only help us keep the code cleaner but they also play a role in reducing the usage of global scope. The contents of a module are not made available to any other module unless the other module explicitly loads it.

The importance of modules is not limited to applications. Even large JavaScript libraries can take advantage of the module system to export their objects as modules and the applications using the libraries import these modules as required. Angular 2 and Aurelia have started using this feature.

If you'd like a quick primer on using modules in ES6, please read: Understanding ES6 Modules

"One of the key features in any language used to write large applications, is the ability to load different pieces of the application in the form of modules."

About the Sample Application

The subject of our sample application is a virtual book shelf. It consists of the following pages:

1. Home page: shows a list of active books that can be marked as read, or moved to the archive.
2. Add book page: adds a new book to the shelf by accepting title of the book and name of author. It doesn't allow a duplicate titles.
3. Archive page: lists all archived books.

The application is built using AngularJS 1.3 and ES6. If you look at any of the files in the `app` folder, you will see the keywords `export` and `import` used to export objects from the current module and to import objects from other modules. Now, our job is to use Babel's Gulp tasks to convert these modules to one of the existing module systems.

But I'm Not Using Angular. I Just Want to Convert ES6 Modules to CommonJS/AMD

No worries! We got you covered. With a minor amount of tweaking the recipes demonstrated below can be used in any project involving ES6 modules. Angular is quite unimportant here.

Converting to CommonJS

CommonJS is a module system defined by the CommonJS group. It is a synchronous module system, in which the modules are loaded using the `require` function and exported using the `exports` property of the `module` object. The `module` object is expected to be available in all modules by default.

Node.js uses this module system, so it defines the `module` object natively and makes it available to your application. As browsers don't have this object defined, we need to use a utility called Browserify to fill the gap.

Before we start, we will also need to install a few npm packages. These will enable us to use Babel and Browserify in conjunction with Gulp to convert our ES6 modules to one of the common module formats and package the application as a single file for the browser to consume.

- gulp-babel — converts ES6 code into vanilla ES5
- Browserify — lets you `require('modules')` in the browser by bundling up all of your dependencies
- vinyl-source-stream — handles the Browserify module directly, avoiding need for gulp-browserify wrapper
- vinyl-buffer — converts stream to a buffer (necessary for gulp-uglify which doesn't support streams)
- gulp-uglify — minifies files
- del — lets you delete files and folders
- gulp-rename — a plugin to let you rename files

You can get this lot by typing:

```
npm install gulp-babel browserify gulp-browserify vinyl-source-stream vinyl-buffer
gulp-uglify del gulp-rename --save-dev
```

Now let's start using these packages in our `gulpfile.js`. We need to write a task to take all ES6 files and pass them to Babel. The default module system in Babel is CommonJS, so we don't need to send any options to the babel function.

```
var babel = require('gulp-babel'),
    browserify = require('browserify'),
    source = require('vinyl-source-stream'),
    buffer = require('vinyl-buffer'),
    rename = require('gulp-rename'),
    uglify = require('gulp-uglify'),
    del = require('del');

gulp.task('clean-temp', function(){
  return del(['dest']);
});

gulp.task('es6-commonjs',['clean-temp'], function(){
  return gulp.src(['app/*.js','app/**/*.js'])
    .pipe(babel())
    .pipe(gulp.dest('dest/temp'));
});
```

Hopefully there is nothing too confusing here. We are declaring a task named `es6-commonjs` which grabs any JavaScript files in the app directory and any of its sub directories. It then pipes them through Babel, which in turn converts the individual files to ES5 and CommonJS modules and copies the converted files into the `dest/temp` folder. The `es6-commonjs` task has a dependency named `clean-temp`, which will remove the `dest` directory and any files in it, before the `es6-commonjs` task runs.

"Now we can create a single bundled file from these individual files by applying Browserify and then minifying the output using the uglify package."

If you want to make the code more explicit and specify the module system, you may modify usage of Babel as:

```
.pipe(babel({
  modules:"common"
}))
```

Now we can create a single bundled file from these individual files by applying Browserify and then minifying the output using the uglify package. The following snippet shows this:

```
gulp.task('bundle-commonjs-clean', function(){
  return del(['es5/commonjs']);
});

gulp.task('commonjs-bundle',['bundle-commonjs-clean','es6-commonjs'], function(){
  return browserify(['dest/temp/bootstrap.js']).bundle()
    .pipe(source('app.js'))
    .pipe(buffer())
    .pipe(uglify())
    .pipe(rename('app.js'))
    .pipe(gulp.dest("es5/commonjs"));
});
```

The above task has two dependencies: the first is the `bundle-commonjs-clean` task, which will delete the directory `es5/commonjs`, the second is the previously discussed `es6-commonjs` task. Once these have run, the task places the combined and minified file `app.js` in the folder `es5/commonjs`. This file can be referenced directly in `index.html` and the page can be viewed in a browser.

Finally, we can add a task to kick things off:

```
gulp.task('commonjs', ['commonjs-bundle']);
```

Converting to AMD

The Asynchronous Module Definition (AMD) system is, as the name suggests, an asynchronous module loading system. It allows multiple dependent modules to load in parallel and it doesn't wait for one module to be completely loaded before attempting to load other modules.

Require.js is the library used to work with AMD. RequireJS is available through Bower:

```
bower install requirejs --save
```

We also need the Gulp plugin for require.js to bundle the application. Install the `gulp-requirejs` npm package for this.

```
npm install gulp-requirejs --save-dev
```

Now we need to write the tasks for converting the ES6 code to ES5 and AMD and then to bundle it using RequireJS. The tasks are pretty much similar to the tasks created in the CommonJS section.

```
var requirejs = require('gulp-requirejs');

gulp.task('es6-amd',['clean-temp'], function(){
    return gulp.src(['app/*.js','app/**/*.js'])
    .pipe(babel({ modules:"amd" }))
    .pipe(gulp.dest('dest/temp'));
});

gulp.task('bundle-amd-clean', function(){
  return del(['es5/amd']);
});

gulp.task('amd-bundle',['bundle-amd-clean','es6-amd'], function(){
  return requirejs({
    name: 'bootstrap',
    baseUrl: 'dest/temp',
    out: 'app.js'
  })
  .pipe(uglify())
  .pipe(gulp.dest("es5/amd"));
});

gulp.task('amd', ['amd-bundle']);
```

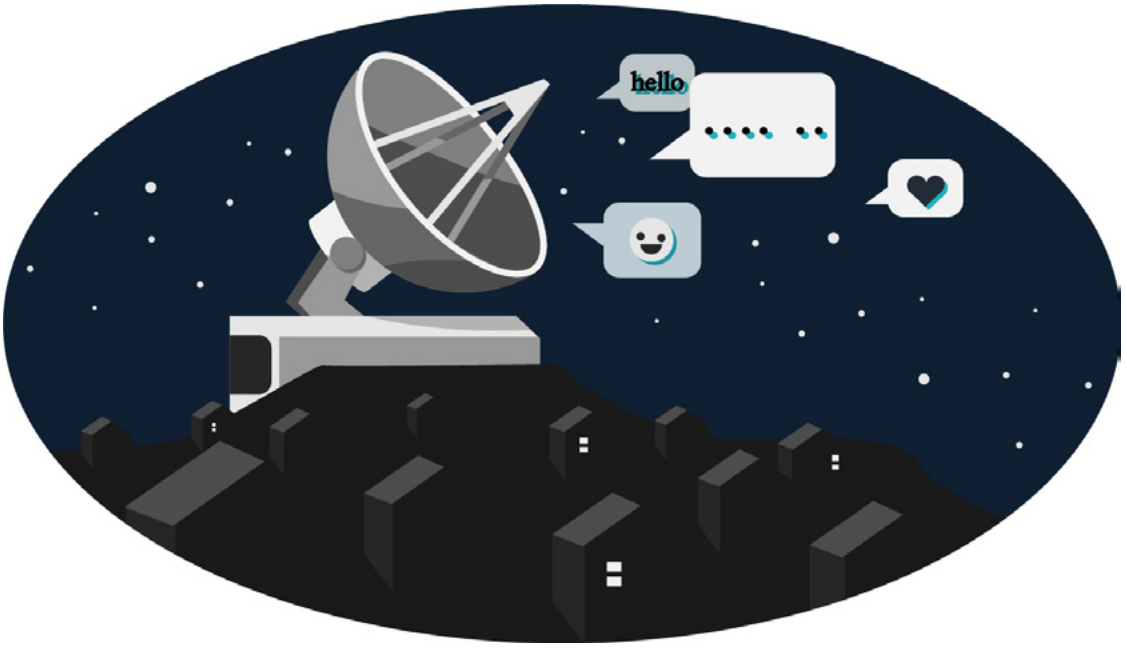
To use the final script on index.html page, we need to add a reference to RequireJS, the generated script and then load the `bootstrap` module. The `bootstrap.js` file inside `app` folder bootstraps the AngularJS application, so we need to load it to kick start the AngularJS application.

```
<script src="bower_components/requirejs/require.js" ></script>
<script src="es5/amd/app.js"></script>
<script>
  (function(){
    require(['bootstrap']);
  }());
</script>
```

Conclusion

Modules are a long overdue feature in JavaScript. They will be arriving in ES6, but unfortunately, their native browser support is currently poor. That does not however, mean that you cannot use them today. In this tutorial I have demonstrated how to use Gulp, Babel and a variety of plugins to convert ES6 modules to the CommonJS and AMD format that you can run in your browser.

And as for ES6? ES6 has gained a lot of attention in the community since it was announced. It is already used by several JavaScript libraries or, frameworks including Bootstrap's JavaScript plugins, Aurelia, Angular 2 and several others. TypeScript has also added support for a handful number of ES6 features including modules. Learning about and using ES6 today, will reduce the effort required to convert the code in future.



Asynchronous APIs Using the Fetch API and ES6 Generators

ECMAScript 6 (a.k.a. ECMAScript 2015 or ES6) brings a number of new features to JavaScript which will make the language a good fit for large applications. One of these features is better support for asynchronous programming using promises and generators. Another is the addition of the Fetch API which aims to replace `XMLHttpRequest` as the foundation of communication with remote resources.

By Rabi Kiran
@SRavi_Kiran

The Fetch API's methods return ES6 `Promise` objects, which can be used in conjunction with generators to form the basis of complex asynchronous operations. This could be anything from a chain of asynchronous operations, where each operation depends on the value returned by the previous one, to an asynchronous call that has to be made repeatedly to a server to get the latest update.

In this article we will see how the Fetch API can be used in conjunction with generators to build asyn-

chronous APIs. The Fetch API is currently supported in Chrome, Opera, Firefox and Android browsers. We have a polyfill available from GitHub for unsupported browsers.

As ever, the code for this article can be found on our GitHub repository and there is a demo of the final technique on CodePen.

Generators for Asynchronous Operations

Tip: If you need a refresher on what generators are and how they work, check out: ECMAScript 2015: Generators and Iterators.

So how can we use generators to perform async operations? Well, if we analyze the way generators work we will find the answer.

> "The yield keyword is responsible for returning a result and halting execution of the iterator function until it is next invoked."

A generator function implementing an iterator has the following structure:

```
function *myIterator(){
  while(condition){
    //calculate next value to return
    yield value;
  }
}
```

The `yield` keyword is responsible for returning a result and halting execution of the iterator function until it is next invoked. It also keeps the state of the function instead of rerunning everything when next you call it, effectively remembering the last place it left off.

We can re-imagine the above function without while loop as follows:

```
function *myIterator(){
  //calculate value 1
  yield value1;

  //calculate value 2
  yield value2;
  ...
```

```
  //calculate value n
  yield valuen;
}
```

The behavior of the function will be identical in both of the above cases. The only reason for using the yield keyword is to pause the execution of the function until the next iteration (which in itself seems kind of asynchronous). And as the `yield` statement can return any value, we can also return promises and make the function run multiple asynchronous calls.

Using Generators with the Fetch API

Tip: For a refresher on the Fetch API, check out: Introduction to the Fetch API

As mentioned earlier the Fetch API is intended to replace `XMLHttpRequest`. This new API provides control over every part of an HTTP request and returns a promise that either resolves or rejects based on the response from the server.

Long Polling

One of the use cases where the Fetch API and generators can be used together is long polling. Long polling is a technique in which a client keeps sending requests to a server until it gets a response. Generators can be used in such a case to keep yielding responses until the response contains data.

"One of the use cases where the Fetch API and generators can be used together is long polling."

To mimic long polling, I included an Express REST API in the sample code that responds with weather information of a city after five attempts. The following is the REST API:

```
var polls=0;

app.get('/api/currentWeather', function(request, response){
  console.log(polls, polls<5);
  if(polls < 5){
    console.log("sending...empty");
    polls++;
    response.send({});
  }
  else{
```

```
    console.log("sending...object");
    response.send({
      temperature: 25,
      sky: "Partly cloudy",
      humid: true
    });
    polls = 0;
  }
});
```

Now, let's write a generator function that calls this API multiple times and returns a promise on every iteration. Being on the client side, we don't know after how many iterations we will get data from the server. So, this method will have an infinite loop pinging the server on every iteration and returning the promise on every occasion. Following is the implementation of this method:

```
function *pollForWeatherInfo(){
  while(true){
    yield fetch('/api/currentWeather',{
      method: 'get'
    }).then(function(d){
      var json = d.json();
      return json;
    });
  }
}
```

We need a function to keep calling this function and checking if the value exists after the promise resolves. It will be a recursive function that invokes the next iteration of the generator and only stops the process when it finds a value returned from the generator. The following snippet shows the implementation of this method and a statement that calls this method:

```
function runPolling(generator){
  if(!generator){
    generator = pollForWeatherInfo();
  }

  var p = generator.next();
  p.value.then(function(d){
    if(!d.temperature){
      runPolling(generator);
```

```
    } else {
      console.log(d);
    }
  });
}

runPolling();
```

As we see here, the first call to the function `runPolling` creates the `generator` object. The `next` method returns an object with a `value` property which in our case contains a promise returned by the `fetch` method. When this promise resolves, it will either contain an empty object (returned if the `polls` variable is below 5), or an object containing the desired information.

Next, we check for the `temperature` property of this object (which would indicate success). If it's not present we pass the `generator` object back to the next function call (so as not to lose the state of the generator) or we print the value of the object to the console.

To see this in action, grab the code from our repo, install the dependencies, start the server, then navigate to http://localhost:8000. You should see the following results in the shell:

```
0 true
sending...empty
1 true
sending...empty
2 true
sending...empty
3 true
sending...empty
4 true
sending...empty
5 false
sending...object
```

And the object itself logged to the browser console.

Multiple Dependent Asynchronous Calls

Quite often, we need to implement multiple dependent asynchronous calls, where each successive asynchronous operation depends on the value returned by the the preceding asynchronous operation.

If we have a group of such operations and they have to be called multiple times, we can put them together in a generator function and execute it whenever we need it.

To demonstrate this, I will be using GitHub's API. This API provides us access to basic information on users, organizations and repos. We will use this API to get the list of contributors to a random repo of an organization and display the fetched data on the screen.

For this we need to make calls to three different endpoints. These are the tasks to be performed:

- Get details of the organization
- If the organization exists, get the organization's repos
- Get contributors to one of the organization's repos (selected at random)

Let's create a wrapper function around Fetch API to avoid repeating the code to create the headers and build the request object.

```
function wrapperOnFetch(url){
  var headers = new Headers();
  headers.append('Accept', 'application/vnd.github.v3+json');
  var request = new Request(url, {headers: headers});

  return fetch(request).then(function(res){
    return res.json();
  });
}
```

The following function consumes the above function and yields a promise for each invocation:

```
function* gitHubDetails(orgName) {
  var baseUrl = "https://api.github.com/orgs/";
  var url = baseUrl + orgName;

  var reposUrl = yield wrapperOnFetch(url);
  var repoFullName = yield wrapperOnFetch(reposUrl);
  yield wrapperOnFetch(`https://api.github.com/repos/${repoFullName}/contributors`);
}
```

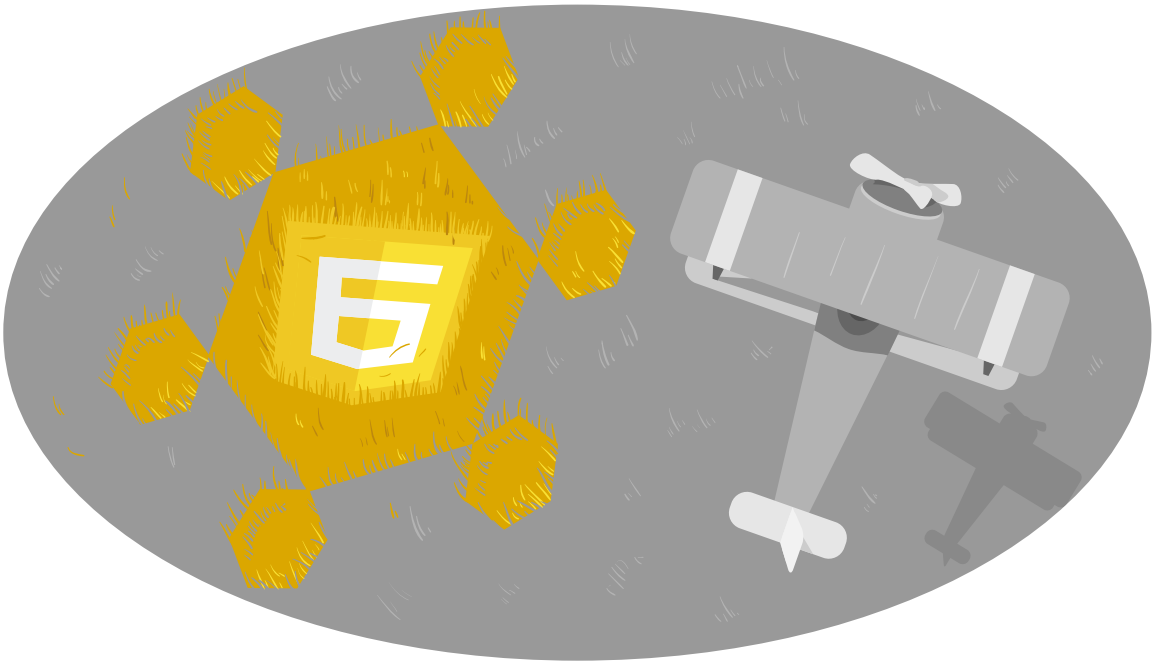
Now, let's write a piece of logic to call the above function to get the generator and then use the values obtained from the server to populate the UI. As every call to the generator's `next` method returns a promise, we will have to chain these promises. The following is the skeleton of the code using the generator returned by the above function:

```
var generator = gitHubDetails("aspnet");

generator.next().value.then(function (userData) {
  //Update UI

  return generator.next(userData.repos_url).value.then(function (reposData) {
    return reposData;
  });
}).then(function (reposData) {
  //Update UI

  return generator.next(reposData[randomIndex].full_name).value.then(function (selecte-
dRepoCommits) {
    //Update UI
 });
});
```

To see this in action, as detailed above, grab the code from our repo, install the dependencies, start the server, then navigate to http://localhost:8000. Or just check out the demo on CodePen (try rerunning it).

Conclusion

In this article I have demonstrated how the Fetch API can be used in conjunction with generators to build asynchronous APIs. ECMAScript 6 will bring a slew of new features to the language and looking for inventive ways to combine them and harness their power can often bring outstanding results.



Preparing for ECMAScript 6: Symbols and Their Uses

While ES2015 has introduced many language features that have been on developers' wish lists for some time, there are some new features that are less well known and understood, and the benefits of which are much less clear.

By Nilson Jacques
@nilsonjacques

One such feature is the symbol. The symbol is a new primitive type, a unique token that is guaranteed never to clash with another symbol.

In this sense, you could think of them as a kind of UUID (Universally Unique Identifier). Let's look at how they work, and what we can do with them (Note: all examples in this article have been tested in the latest versions of Chrome and Firefox for Linux, but the behavior may vary from browser to browser or when using tools such as Firebug).

Creating New Symbols

Creating new symbols is very straightforward and is simply a case of calling the Symbol function. Note that this is a just a standard function and not an object constructor. Trying to call it with the `new` operator will result in a `TypeError`. Every time you call the `Symbol` function, you will get a new and completely unique value.

```
let foo = Symbol();
let bar = Symbol();

foo === bar
// <-- false
```

Symbols can also be created with a label, by passing a string as the first argument. The label does not affect the value of the symbol, but is useful for debugging, and is shown if the symbol's `toString()` method is called. It's possible to create multiple symbols with the same label, but there's no advantage to doing so and this would probably just lead to confusion.

```
let foo = Symbol('baz');
let bar = Symbol('baz');

foo === bar
// <-- false
console.log(foo);
// <-- Symbol(baz)
```

What Can I Do With Them?

Symbols could often be a good replacement for strings or integers as class/module constants:

```
class Application {
  constructor(mode) {
    switch (mode) {
      case Application.DEV:
        // Set up app for development environment
```

```
        break;
      case Application.PROD:
        // Set up app for production environment
        break;
        default:
        throw new Error('Invalid application mode: ' + mode);
    }
  }
}

Application.DEV = Symbol('dev');
Application.PROD = Symbol('prod');

// Example use
let app = new Application(Application.DEV);
```

String and integers are not unique values; values such as the number 2 or the string 'development', for example, could also be in use elsewhere in the program for different purposes. Using symbols means we can be more confident about the value being supplied.

Another interesting use of symbols is as object property keys. If you've ever used a JavaScript object as a hashmap (an associative array in PHP terms, or dictionary in Python) you'll be familiar with getting/setting properties using the bracket notation:

```
let data = [];

data['name'] = 'Ted Mosby';
data['nickname'] = 'Teddy Westside';
data['city'] = 'New York';
```

Using the bracket notation, we can also use a symbol as a property key. There are a couple of advantages to doing so: First, you can be sure that symbol-based keys will never clash, unlike string keys, which might conflict with keys for existing properties or methods of an object. Second, they won't be enumerated in `for..in` loops, and are ignored by functions such as `Object.keys()`, `Object.getOwnPropertyNames()` and `JSON.stringify()`. This makes them ideal for properties that you don't want to be included when serializing an object.

```
let user = {};
let email = Symbol();

user.name = 'Fred';
user.age = 30;
user[email] = 'fred@example.com';

Object.keys(user);
// <-- Array [ "name", "age" ]

Object.getOwnPropertyNames(user);
// <-- Array [ "name", "age" ]

JSON.stringify(user);
// <-- "{"name":"Fred","age":30}"
```

It is worth noting, however, that using symbols as keys does not guarantee privacy. There are some new tools provided to allow you to access symbol-based property keys.

`Object.getOwnPropertySymbols()` returns an array of any symbol-based keys, while `Reflect.ownKeys()` will return an array of all keys, including symbols.

```
Object.getOwnPropertySymbols(user);
// <-- Array [ Symbol() ]

Reflect.ownKeys(user)
// <-- Array [ "name", "age", Symbol() ]
```

Well-known Symbols

Because symbol-keyed properties are effectively invisible to pre-ES6 code, they are ideal for adding new functionality to JavaScript's existing types without breaking backwards compatibility. The so-called 'well-known' symbols are predefined properties of the `Symbol` function that are used to customize the behavior of certain language features, and are used to implement new functionality such as iterators.

"Because symbol-keyed properties are effectively invisible to pre-ES6 code, they are ideal for adding new functionality to JavaScript's existing types without breaking backwards compatibility."

`Symbol.iterator` is a well-known symbol which is used to assign a special method to objects which allows them to be iterated over.

```
let band = ['Freddy', 'Brian', 'John', 'Roger'];
let iterator = band[Symbol.iterator]();

iterator.next().value;
// <-- { value: "Freddy", done: false }
iterator.next().value;
// <-- { value: "Brian", done: false }
iterator.next().value;
// <-- { value: "John", done: false }
iterator.next().value;
// <-- { value: "Roger", done: false }
iterator.next().value;
// <-- { value: undefined, done: true }
```

The built-in types `String`, `Array`, `TypedArray`, `Map` and `Set` all have a default `Symbol.iterator` method which is called when an instance of one of these types is used in a `for...of` loop, or with the spread operator. Browsers are also starting to use the `Symbol.iterator` key to allow DOM structures such as `NodeList` and `HTMLCollection` to be iterated over in the same way.

The Global Registry

The specification also defines a runtime-wide symbol registry, which means that you can store and retrieve symbols across different execution contexts, such as between a document and an embedded iframe or service worker.

"The specification also defines a runtime-wide symbol registry, which means that you can store and retrieve symbols across different execution contexts"

`Symbol.for(key)` retrieves the symbol for a given key from the registry. If a symbol does not exist for the key, a new one is returned. As you might expect, subsequent calls for the same key will return the same symbol.

`Symbol.keyFor(symbol)` allows you to retrieve the key for a given symbol. Calling the method with a symbol that does not exist in the registry returns undefined.

```
let debbie = Symbol.for('user');
let mike   = Symbol.for('user');

debbie === mike
// <-- true

Symbol.keyFor(debbie);
// <-- "user"
```

Use Cases

There are a couple of use cases where using symbols provides an advantage. One, which I touched on earlier in the article, is when you want to add 'hidden' properties to objects that will not be included when the object is serialized.

Library authors could also use symbols to safely augment client objects with properties or methods without having to worry about overwriting existing keys (or having their keys overwritten by other code). For example, widget components (such as date pickers) are often intialized with various options and state that needs to be stored somewhere. Assigning the widget instance to a property of the DOM element object is not ideal because that property could potentially clash with another key. Using a symbol-based key neatly side-steps this issue and ensures that your widget instance will not be overwritten. See the Mozilla Hacks blog post ES6 in Depth: Symbols for a more detailed exploration of this idea.

Browser Support

If you want to experiment with symbols, mainstream browser support is already quite good: https://kangax.github.io/compat-table/es6/. As you can see, the current versions of Chrome, Firefox, Microsoft Edge and Opera support the Symbol type natively, along with Android 5.1 and iOS 9 on mobile devices. There are also polyfills available if you need to support older browsers.

Conclusion

Although the primary reason for the introduction of symbols seems to have been to facilitate adding new functionality to the language without breaking existing code, they do have some interesting uses. It is worthwhile for all developers to have at least a basic knowledge of them, and be familiar with the most commonly used well-known symbols and their purpose.

Object-Oriented JavaScript — A Deep Dive into ES6 Classes

Often we need to represent an idea or concept in our programs—maybe a car engine, a computer file, a router, or a temperature reading. Representing these concepts directly in code comes in two parts: data to represent the state and functions to represent the behavior. Classes give us a convenient syntax to define the state and behavior of objects that will represent our concepts.

By Jeffrey Mott

They make our code safer by guaranteeing an initialization function will be called, and they make it easier to define a fixed set of functions that operate on that data and maintain valid state. If you can think of something as a separate entity, it's likely you should define a class to represent that "thing" in your program.

Consider this non-class code. How many errors can you find? How would you fix them?

```
// set today to December 24
let today = {
  day: 12,
  month: 24,
};

let tomorrow = {
  year: today.year,
  month: today.month,
  day: today.day + 1,
};

let dayAfterTomorrow = {
  year: tomorrow.year,
  month: tomorrow.month,
  day: tomorrow.day + 1 <= 31 ? tomorrow.day + 1 : 1,
};
```

The date `today` isn't valid; there is no month 24. Also, `today` isn't fully initialized; it's missing the year. It would be better if we had an initialization function that couldn't be forgotten. Notice also, that when adding a day, we checked in one place if we went beyond 31 but missed that check in another place. It would be better if we interacted with the data only through a small and fixed set of functions that each maintain valid state.

Here's the corrected version that uses classes.

```
class SimpleDate {
  constructor(year, month, day) {
    // Check that (year, month, day) is a valid date
    // ...

    // If it is, use it to initialize "this" date
    this._year = year;
    this._month = month;
    this._day = day;
  }

  addDays(nDays) {
    // Increase "this" date by n days
```

```
    // ...
  }

  getDay() {
    return this._day;
  }
}

// "today" is guaranteed to be valid and fully initialized
let today = new SimpleDate(2000, 2, 28);

// Manipulating data only through a fixed set of functions ensures we maintain valid
state
today.addDays(1);
```

> JARGON TIP:
>
> - When a function is associated with a class or object, we call it a "method"
> - When an object is created from a class, that object is said to be an "instance" of the class.

Constructors

The `constructor` method is special, and it solves the first problem. Its job is to initialize an instance to a valid state, and it will be called automatically so we can't forget to initialize our objects.

Keep Data Private

We try to design our classes so that their state is guaranteed to be valid. We provide a constructor that creates only valid values, and we design methods that also always leave behind only valid values. But as long as we leave the data of our classes accessible to everyone, someone will mess it up. We protect against this by keeping the data inaccessible except through the functions we supply.

> JARGON TIP: Keeping data private to protect it is called "encapsulation".

Privacy with Conventions

Unfortunately, private object properties don't exist in JavaScript. We have to fake them. The most common way to do that is to adhere to a simple convention: If a property name is prefixed with an underscore (or, less commonly, suffixed with an underscore), then it should be treated as non-public. We used this approach in the earlier code example. Generally this simple convention works, but the data is technically still accessible to everyone, so we have to rely on our own discipline to do the right thing.

"Unfortunately, private object properties don't exist in JavaScript. We have to fake them"

Privacy with Privileged Methods

The next most common way to fake private object properties is to use ordinary variables in the constructor, and capture them in closures. This trick gives us truly private data that is inaccessible to the outside. But to make it work, our class's methods would themselves need to be defined in the constructor and attached to the instance.

```
class SimpleDate {
  constructor(year, month, day) {
    // Check that (year, month, day) is a valid date
    // ...

    // If it is, use it to initialize "this" date's ordinary variables
    let _year = year;
    let _month = month;
    let _day = day;

    // Methods defined in the constructor capture variables in a closure
    this.addDays = function(nDays) {
      // Increase "this" date by n days
      // ...
    }

    this.getDay = function() {
      return _day;
    }
  }
}
```

Privacy with Symbols

Symbols are a new feature to JavaScript, and they give us another way to fake private object properties. Instead of underscore property names, we could use unique symbol object keys, and our class can capture those keys in a closure. But there's a leak. Another new feature to JavaScript is `Object.getOwnPropertySymbols`, and it allows the outside to access the symbol keys we tried to keep private.

```
let SimpleDate = (function() {
  let _yearKey = Symbol();
  let _monthKey = Symbol();
  let _dayKey = Symbol();

  class SimpleDate {
    constructor(year, month, day) {
      // Check that (year, month, day) is a valid date
      // ...

      // If it is, use it to initialize "this" date
      this[_yearKey] = year;
      this[_monthKey] = month;
      this[_dayKey] = day;
    }

    addDays(nDays) {
      // Increase "this" date by n days
      // ...
    }

    getDay() {
      return this[_dayKey];
    }
  }

  return SimpleDate;
}());
```

Privacy with Weak Maps

Weak maps are also a new feature to JavaScript. We can store private object properties in key/value pairs using our instance as the key, and our class can capture those key/value maps in a closure.

```
let SimpleDate = (function() {
  let _years = new WeakMap();
  let _months = new WeakMap();
  let _days = new WeakMap();

  class SimpleDate {
    constructor(year, month, day) {
      // Check that (year, month, day) is a valid date
      // ...
      // If it is, use it to initialize "this" date
      _years.set(this, year);
      _months.set(this, month);
      _days.set(this, day);
    }

    addDays(nDays) {
      // Increase "this" date by n days
      // ...
    }

    getDay() {
      return _days.get(this);
    }
  }
  return SimpleDate;
}());
```

Other Access Modifiers

There are other levels of visibility besides "private" that you'll find in other languages, such as "protected", "internal", "package private", or "friend". JavaScript still doesn't give us a way to enforce those other levels of visibility. If you need them, you'll have to rely on conventions and self discipline.

Referring to the Current Object

Look again at `getDay()`. It doesn't specify any parameters, so how does it know the object for which it was called? When a function is called as a method using the `object.function` notation, there is an implicit argument that it uses to identify the object, and that implicit argument is assigned to an implicit parameter named `this`. To illustrate, here's how we would send the object argument explicitly rather than implicitly.

"We have the option to define data and functions that are part of the class but not part of any instance of that class. We call these static properties and static methods, respectively."

```
// Get a reference to the "getDay" function
let getDay = SimpleDate.prototype.getDay;

getDay.call(today); // "this" will be "today"
getDay.call(tomorrow); // "this" will be "tomorrow"

tomorrow.getDay(); // same as last line, but "tomorrow" is passed implicitly
```

Static Properties and Methods

We have the option to define data and functions that are part of the class but not part of any instance of that class. We call these static properties and static methods, respectively. There will only be one copy of a static property rather than a new copy per instance.

```
class SimpleDate {
  static setDefaultDate(year, month, day) {
    // A static property can be referred to without mentioning an instance
    // Instead, it's defined on the class
    SimpleDate._defaultDate = new SimpleDate(year, month, day);
  }

  constructor(year, month, day) {
    // If constructing without arguments,
    // then initialize "this" date by copying the static default date
    if (arguments.length === 0) {
      this._year = SimpleDate._defaultDate._year;
```

```
      this._month = SimpleDate._defaultDate._month;
      this._day = SimpleDate._defaultDate._day;

      return;
    }

    // Check that (year, month, day) is a valid date
    // ...

    // If it is, use it to initialize "this" date
    this._year = year;
    this._month = month;
    this._day = day;
  }

  addDays(nDays) {
    // Increase "this" date by n days
    // ...
  }

  getDay() {
    return this._day;
  }
}

SimpleDate.setDefaultDate(1970, 1, 1);

let defaultDate = new SimpleDate();
```

Subclasses

Often we find commonality between our classes—repeated code that we'd like to consolidate. Subclasses let us incorporate another class's state and behavior into our own. This process is often called "inheritance," and our subclass is said to "inherit" from a parent class, also called a superclass. Inheritance can avoid duplication and simplify the implementation of a class that needs the same data and functions as another class. Inheritance also allows us to substitute subclasses, relying only on the interface provided by a common superclass.

Inherit to Avoid Duplication

Consider this non-inheritance code.

```
class Employee {
  constructor(firstName, familyName) {
    this._firstName = firstName;
    this._familyName = familyName;
  }

  getFullName() {
    return `${this._firstName} ${this._familyName}`;
  }
}

class Manager {
  constructor(firstName, familyName) {
    this._firstName = firstName;
    this._familyName = familyName;
    this._managedEmployees = [];
  }

  getFullName() {
    return `${this._firstName} ${this._familyName}`;
  }

  addEmployee(employee) {
    this._managedEmployees.push(employee);
  }
}
```

The data properties `_firstName` and `_familyName`, and the method `getFullName`, are repeated between our classes. We could eliminate that repetition by having our `Manager` class inherit from the `Employee` class. When we do, the state and behavior of the `Employee` class—its data and functions—will be incorporated into our `Manager` class.

Here's a version that uses inheritance. Notice the use of super.

```
// Manager still works same as before but without repeated code
class Manager extends Employee {
  constructor(firstName, familyName) {
    super(firstName, familyName);
    this._managedEmployees = [];
  }

  addEmployee(employee) {
    this._managedEmployees.push(employee);
  }
}
```

IS-A and WORKS-LIKE-A

There are design principles to help you decide when inheritance is appropriate. Inheritance should always model an IS-A and WORKS-LIKE-A relationship. That is, a manager "is a" and "works like a" specific kind of employee, such that anywhere we operate on a superclass instance, we should be able to substitute in a subclass instance, and everything should still just work. The difference between violating and adhering to this principle can sometimes be subtle. A classic example of a subtle violation is a `Rectangle` superclass and a `Square` subclass.

```
class Rectangle {
  set width(w) {
    this._width = w;
  }

  get width() {
    return this._width;
  }

  set height(h) {
    this._height = h;
  }

  get height() {
    return this._height;
  }
}
```

```
// A function that operates on an instance of Rectangle
function f(rectangle) {
  rectangle.width = 5;
  rectangle.height = 4;

  // Verify expected result
  if (rectangle.width * rectangle.height !== 20) {
    throw new Error("Expected the rectangle's area (width * height) to be 20");
  }
}

// A square IS-A rectangle... right?
class Square extends Rectangle {
  set width(w) {
    super.width = w;

    // Maintain square-ness
    super.height = w;
  }

  set height(h) {
    super.height = h;

    // Maintain square-ness
    super.width = h;
  }
}

// But can a rectangle be substituted by a square?
f(new Square()); // error
```

A square may be a rectangle *mathematically*, but a square doesn't *work like a* rectangle behaviorally.

This rule that any use of a superclass instance should be substitutable by a subclass instance is called the Liskov Substitution Principle, and it's an important part of object oriented class design.

Beware Overuse

It's easy to find commonality everywhere, and the prospect of having a class that offers complete

functionality can be alluring, even for experienced developers. But there are disadvantages to inheritance too. Recall that we ensure valid state by manipulating data only through a small and fixed set of functions. But when we inherit, we increase the list of functions that can directly manipulate the data, and those additional functions are then also responsible for maintaining valid state. If too many functions can directly manipulate the data, then that data becomes nearly as bad as global variables. Too much inheritance creates monolithic classes that dilute encapsulation, are harder to make correct, and harder to reuse. Instead, prefer to design minimal classes that embody just one concept.

Let's revisit the code duplication problem. Could we solve it without inheritance? An alternative approach is to connect objects through references to represent a part-whole relationship. We call this "composition".

Here's a version of the manager-employee relationship using composition rather than inheritance.

```
class Employee {
  constructor(firstName, familyName) {
    this._firstName = firstName;
    this._familyName = familyName;
  }

  getFullName() {
    return `${this._firstName} ${this._familyName}`;
  }
}

class Group {
  constructor(manager /* : Employee */ ) {
    this._manager = manager;
    this._managedEmployees = [];
  }

  addEmployee(employee) {
    this._managedEmployees.push(employee);
  }
}
```

Here, a manager isn't a separate class. Instead, a manager is an ordinary `Employee` instance that a `Group` instance holds a reference to. If inheritance models the IS-A relationship, then composition models the HAS-A relationship. That is, a group "has a" manager.

If either inheritance or composition can reasonably express our program concepts and relationships, then prefer composition.

Inherit to substitute subclasses

Inheritance also allows different subclasses to be used interchangeably through the interface provided by a common superclass. A function that expects a superclass instance as an argument can also be passed a subclass instance without the function having to know about any of the subclasses. Substituting classes that have a common superclass is often called "polymorphism".

“Inheritance also allows different subclasses to be used interchangeably through the interface provided by a common superclass.”

```
// This will be our common superclass
class Cache {
  get(key, defaultValue) {
    let value = this._doGet(key);
    if (value === undefined || value === null) {
      return defaultValue;
    }

    return value;
  }

  set(key, value) {
    if (key === undefined || key === null) {
      throw new Error('Invalid argument');
    }

    this._doSet(key, value);
  }

  // Must be overridden
  // _doGet()
  // _doSet()
}

// Subclasses define no new public methods
```

```
// The public interface is defined entirely in the superclass
class ArrayCache extends Cache {
  _doGet() {
    // ...
  }

  _doSet() {
    // ...
  }
}

class LocalStorageCache extends Cache {
  _doGet() {
    // ...
  }

  _doSet() {
    // ...
  }
}

// Functions can polymorphically operate on any cache by interacting through the super-
class interface
function compute(cache) {
  let cached = cache.get('result');
  if (!cached) {
    let result = // ...
    cache.set('result', result);
  }

  // ...
}

compute(new ArrayCache()); // use array cache through superclass interface
compute(new LocalStorageCache()); // use local storage cache through superclass inter-
face
```

More than Sugar

JavaScript's class syntax is often said to be syntactic sugar, and in a lot of ways it is, but there are also real differences—things we can do with ES6 classes that we couldn't with ES5.

Static Properties Are Inherited

ES5 didn't let us create true inheritance between constructor functions. `Object.create` could create an ordinary object but not a function object. We faked inheritance of static properties by manually copying them. Now with ES6 classes, we get a real prototype link between a subclass constructor function and the superclass constructor.

```
// ES5
function B() {}
B.f = function () {};

function D() {}
D.prototype = Object.create(B.prototype);

D.f(); // error
// ES6
class B {
  static f() {}
}

class D extends B {}

D.f(); // ok
```

Built-in Constructors Can Be Subclassed

Some objects are "exotic" and don't behave like ordinary objects. Arrays, for example, adjust their `length` property to be greater than the largest integer index. In ES5, when we tried to subclass `Array`, the `new` operator would allocate an ordinary object for our subclass, not the exotic object of our superclass.

```
// ES5
function D() {
```

```
  Array.apply(this, arguments);
}
D.prototype = Object.create(Array.prototype);

var d = new D();
d[0] = 42;

d.length; // 0 - bad, no array exotic behavior
```

ES6 classes fixed this by changing when and by whom objects are allocated. In ES5, objects were allocated before invoking the subclass constructor, and the subclass would pass that object to the superclass constructor. Now with ES6 classes, objects are allocated before invoking the *superclass* constructor, and the superclass makes that object available to the subclass constructor. This lets `Array` allocate an exotic object even when we invoke `new` on our subclass.

```
// ES6
class D extends Array {}

let d = new D();
d[0] = 42;

d.length; // 1 - good, array exotic behavior
```

Miscellaneous

There's a small assortment of other, probably less significant differences. Class constructors can't be function-called. This protects against forgetting to invoke constructors with `new`. Also, a class constructor's `prototype` property can't be reassigned. This may help JavaScript engines optimize class objects. And finally, class methods don't have a `prototype` property. This may save memory by eliminating unnecessary objects.

Using New Features in Imaginative Ways

Many of the features described here and in other SitePoint articles are new to JavaScript, and the community is experimenting right now to use those features in new and imaginative ways.

Multiple Inheritance with Proxies

One such experiment uses proxies, a new feature to JavaScript, to implement multiple inheritance. JavaScript's prototype chain allows only single inheritance. Objects can delegate to only one other object. Proxies give us a way to delegate property accesses to multiple other objects.

```
let transmitter = {
  transmit() {}
};

let receiver = {
  receive() {}
};

// Create a proxy object that intercepts property accesses and forwards to each parent,
// returning the first defined value it finds
let inheritsFromMultiple = new Proxy([transmitter, receiver], {
  get: function(proxyTarget, propertyKey) {
    const foundParent = proxyTarget.find(parent => parent[propertyKey] !== undefined);
    return foundParent && foundParent[propertyKey];
  }
});

inheritsFromMultiple.transmit(); // works
inheritsFromMultiple.receive(); // works
```

Can we expand this to work with classes? A class's `prototype` could be a proxy that forwards property access to multiple other prototypes. The JavaScript community is working on this right now. Can you figure it out? Join the discussion and share your ideas.

Multiple Inheritance with Class Factories

Another approach the JavaScript community has been experimenting with is generating classes on demand that extend a variable superclass. Each class still has only a single parent, but we can chain those parents in interesting ways.

```
function makeTransmitterClass(Superclass = Object) {
  return class Transmitter extends Superclass {
    transmit() {}
  };
}

function makeReceiverClass(Superclass = Object) {
  return class Receiver extends Superclass {
    receive() {}
  };
}

class InheritsFromMultiple extends makeTransmitterClass(makeReceiverClass()) {}

let inheritsFromMultiple = new InheritsFromMultiple();

inheritsFromMultiple.transmit(); // works
inheritsFromMultiple.receive(); // works
```

Are there other imaginative ways to use these features? Now's the time to leave your footprint in the JavaScript world.

Conclusion

Hopefully this article has given you an insight into how classes work in ES6 and has demystified some of the jargon surrounding them. Unfortunately, at the time of writing, support for classes isn't very good, so you'll need to use a transpiler such as Babel if you want to give them a try.