sitepoint

PRACTICAL ES6

1. 2. 3.

LEVEL UP YOUR JAVASCRIPT

Practical ES6

Cover Design: Alex Walker

Notice of Rights

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, design, and more.

Preface

There’s no doubt that the JavaScript ecosystem changes fast. Not only are new tools and frameworks introduced and developed at a rapid rate, the language itself has undergone big changes with the introduction of ES2015 (aka ES6). Understandably, many articles have been written complaining about how difficult it is to learn modern JavaScript development these days. We're aiming to minimize that confusion with this set of books on modern JavaScript.

This book provides an introduction to many of the powerful new JavaScript language features that were introduced in ECMAScript 2015, as well as features introduced in ECMAScript 2016 and 2017. It also takes a look at the features planned for ECMAScript 2018 in this rapidly evolving language.

Who Should Read This Book?

This book is for all front-end developers who wish to improve their JavaScript skills. You'll need to be familiar with HTML and CSS and have a reasonable level of understanding of JavaScript in order to follow the discussion.

Conventions Used

Code Samples

Code in this book is displayed using a fixed-width font, like so: `<h1>A Perfect Summer's Day</h1> <p>It was a lovely day for a walk in the park. The birds were singing and the kids were all back at school.</p>`

Where existing code is required for context, rather than repeat all of it, ⋮ will be displayed: `function animate() { ⋮ new_variable = "Hello"; }`

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An ➥ indicates a line break that exists for formatting purposes only, and should be ignored:
`URL.open("http://www.sitepoint.com/responsive-web- ➥design-real-user-testing/?responsive1");`

You'll notice that we've used certain layout styles throughout this book to signify different types of information. Look out for the following items.

Tips, Notes, and Warnings

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Chapter 1: New Keywords: let and const

by Aurelio de Rosa

In this tutorial, I'll introduce `let` and `const`, two new keywords added to JavaScript with the arrival of ES6. They enhance JavaScript by providing a way to define block-scope variables and constants.

`let`

Up to ES5, JavaScript had only two types of scope, function scope and global scope. This caused a lot of frustration and unexpected behaviors for developers coming from other languages such as C, C++ or Java. JavaScript lacked block scope, meaning that a variable is only accessible within the block in which it's defined. A block is everything inside an opening and closing curly bracket. 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 the following output in the console:

```
1
2
1
2
```

What most developers coming from the languages mentioned above 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`.

This situation changed in ES6 with the availability of block scope. The ECMA organization members knew that they could not change the behavior of the keyword `var`, as that would break 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. 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.

Thanks to ES6, we can easily solve this issue by declaring the `i` variable in the `for` loop using `let`:

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

The `let` statement is supported in Node and all modern browsers. There are, however, a couple of gotchas in Internet Explorer 11 which you can read about in the ES6 compatibility table.

A live demo that shows the difference between `var` and `let` is available at JSBin.

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're not hoisted). In case you try to access a constant before it's declared, you'll receive a `ReferenceError`. If you try to assign a different value to a variable declared with `const`, you'll receive a `TypeError`.

Please note, however, that `const` is *not* about immutability. As Mathias Bynens states in his blog post ES2015 const is not about immutability, `const` creates an immutable binding, but does not indicate that a value is immutable, as the following code demonstrates:

```
const foo = {};
foo.bar = 42;
console.log(foo.bar);
// → 42
```

If you want to make an object's values truly immutable, use Object.freeze().

Browser support for `const` is equally good as for `let`. The statement `const` is supported in Node and all modern browsers. But here, too, there are some gotchas in Internet Explorer 11, which you can read about in the ES6 compatability table.

An example usage 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 live demo of the previous code is available at JSBin.

Conclusion

In this tutorial, I've introduced you to `let` and `const`, two new methods for declaring variables that were introduced to the language with ES6. While `var` isn't going away any time soon, I'd encourage you to use `const` and `let` whenever possible to reduce your code's susceptibility to errors. By way of further reading, you might also like our quick tip How to Declare Variables in JavaScript, which delves further into the mechanics of variable declaration.

Chapter 2: Using Map, Set, WeakMap, WeakSet

by Kyle Pennell

This chapter examines four new ES6 collections and the benefits they provide.

Most major programming languages have several types of data collections. Python has lists, tuples, and dictionaries. Java has lists, sets, maps, queues. Ruby has hashes and arrays. JavaScript, up until now, had only arrays. Objects and arrays were the workhorses of JavaScript. ES6 introduces four new data structures that will add power and expressiveness to the language: `Map`, `Set`, `WeakSet`, and `WeakMap`.

Searching for the JavaScript HashMap

HashMaps, dictionaries, and hashes are several ways that various programming languages store key/value pairs, and these data structures are optimized for fast retrieval.

In ES5, JavaScript objects — which are just arbitrary collections of properties with keys and values — can simulate hashes, but there are several downsides to using objects as hashes.

Downside #1: Keys must be strings in ES5

JavaScript object property keys must be strings, which limits their ability to serve as a collection of key/value pairs of varying data types. You can, of course, coerce/stringify other data types into strings, but this adds extra work.

Downside #2: Objects are not inherently iterable

Objects weren't designed to be used as collections, and as a result there's no efficient way to determine how many properties an object has. (See, for example, Object.keys is slow). When you loop over an object's properties, you also get its prototype properties. You could add the `iterable` property to all objects, but not all objects are meant to be used as collections. You could use the `for … in` loop and the `hasOwnProperty()` method, but this is just a workaround. When you loop over an object's properties, the properties won't necessarily be retrieved in the same order they were inserted.

Downside #3: Challenges with built-in method collisions

Objects have built-in methods like `constructor`, `toString`, and `valueOf`. If one of these was added as a property, it could cause collisions. You could use `Object.create(null)` to create a bare object (which doesn't inherit from `object.prototype`), but, again, this is just a workaround.

ES6 includes new collection data types, so there’s no longer a need to use objects and live with their drawbacks.

Using ES6 Map Collections

`Map` is the first data structure/collection we'll examine. Maps are collections of keys and values of any type. It's easy to create new Maps, add/remove values, loop over keys/values and efficiently determine their size. Here are the crucial methods:

Creating a map and using common methods

```
const map = new Map(); // Create a new Map map.set('hobby',
'cycling'); // Sets a key value pair



const foods = { dinner: 'Curry', lunch: 'Sandwich', breakfast:
'Eggs' }; // New Object const normalfoods = {}; // New Object



map.set(normalfoods, foods); // Sets two objects as key value
pair



for (const [key, value] of map) {



  console.log(`${key} = ${value}`); // hobby = cycling  [object
Object] = [object Object]



}









map.forEach((value, key) => {
```

```
  console.log(`${key} = ${value}`); }, map); // hobby = cycling
[object Object] = [object Object]




map.clear(); // Clears key value pairs console.log(map.size ===
0); // True
```

[Run this example on JSBin]

Using the Set Collection

Sets are ordered lists of values that contain no duplicates. Instead of being indexed like arrays are, sets are accessed using keys. Sets already exist in Java, Ruby, Python, and many other languages. One difference between ES6 Sets and those in other languages is that the order matters in ES6 (not so in many other languages). Here are the crucial Set methods: `const planetsOrderFromSun = new Set();`

```
planetsOrderFromSun.add('Mercury');

planetsOrderFromSun.add('Venus').add('Earth').add('Mars'); // Chainable Method console.log(planetsOrderFromSun.has('Earth')); // True

planetsOrderFromSun.delete('Mars');

console.log(planetsOrderFromSun.has('Mars')); // False

for (const x of planetsOrderFromSun) {

console.log(x); // Same order in as out - Mercury Venus Earth }

console.log(planetsOrderFromSun.size); // 3


planetsOrderFromSun.add('Venus'); // Trying to add a duplicate
console.log(planetsOrderFromSun.size); // Still 3, Did not add the duplicate

planetsOrderFromSun.clear();

console.log(planetsOrderFromSun.size); // 0
```

Run this example on JSBin

Weak Collections, Memory, and Garbage Collections

JavaScript Garbage Collection is a form of memory management whereby objects that are no longer referenced are automatically deleted and their resources are reclaimed.

`Map` and `Set`'s references to objects are strongly held and will not allow for garbage collection. This can get expensive if maps/sets reference large objects that are no longer needed, such as DOM elements that have already been removed from the DOM.

To remedy this, ES6 also introduces two new weak collections called `WeakMap` and `WeakSet`. These ES6 collections are 'weak' because they allow for objects which are no longer needed to be cleared from memory.

WeakMap

WeakMap is the third of the new ES6 collections we're covering. `WeakMaps` are similar to normal `Maps`, albeit with fewer methods and the aforementioned difference with regards to garbage collection.

```
const aboutAuthor = new WeakMap(); // Create New WeakMap
const currentAge = {}; // key must be an object
const currentCity = {}; // keys must be an object

aboutAuthor.set(currentAge, 30); // Set Key Values
aboutAuthor.set(currentCity, 'Denver'); // Key Values can be of
different data types

console.log(aboutAuthor.has(currentCity)); // Test if WeakMap
has a key

aboutAuthor.delete(currentAge); // Delete a key
```

Run this example on JSBin

Use cases

WeakMaps have several popular use cases. They can be used to keep an object's private data private, and they can also be used to keep track of DOM nodes/objects.

Private data use case

The following example is from JavaScript expert Nicholas C. Zakas:

```
var Person = (function() {
  var privateData = new WeakMap();

  function Person(name) {
    privateData.set(this, { name: name });
  }

  Person.prototype.getName = function() {
    return privateData.get(this).name;
```

```
  };

  return Person;
}());
```

Using a `WeakMap` here simplifies the process of keeping an object's data private. It’s possible to reference the `Person` object, but access to the `privateDataWeakMap` is disallowed without the specific `Person` instance.

DOM nodes use case

The Google Polymer project uses `WeakMaps` in a piece of code called PositionWalker.

> PositionWalker keeps track of a position within a DOM subtree, as a current node and an offset within that node.

WeakMap is used to keep track of DOM node edits, removals, and changes:

```
_makeClone() {
  this._containerClone = this.container.cloneNode(true);
  this._cloneToNodes = new WeakMap();
  this._nodesToClones = new WeakMap();

  ...

  let n = this.container;
  let c = this._containerClone;

  // find the currentNode's clone
  while (n !== null) {
    if (n === this.currentNode) {
    this._currentNodeClone = c;
    }
    this._cloneToNodes.set(c, n);
    this._nodesToClones.set(n, c);

    n = iterator.nextNode();
    c = cloneIterator.nextNode();
  }
}
```

WeakSet

`WeakSets` are Set Collections whose elements can be garbage collected when objects they're referencing are no longer needed. `WeakSets` don't allow for iteration. Their use cases are rather limited (for now, at least). Most early adopters say that `WeakSets` can be used to tag objects without mutating them. ES6-Features.org has an example of adding and deleting elements from a WeakSet in order to keep track of whether or not the objects have been marked: `let isMarked = new WeakSet() let attachedData = new WeakMap()`

```
export class Node {

constructor (id) { this.id = id }

mark () { isMarked.add(this) }

unmark () { isMarked.delete(this) }

marked () { return isMarked.has(this) }

set data (data) { attachedData.set(this, data) }

get data () { return attachedData.get(this) }

}


let foo = new Node("foo")

JSON.stringify(foo) === '{"id":"foo"}'

foo.mark()

foo.data = "bar"

foo.data === "bar"

JSON.stringify(foo) === '{"id":"foo"}'
```

```
isMarked.has(foo) === true attachedData.has(foo) === true foo =
null /* remove only reference to foo */

attachedData.has(foo) === false isMarked.has(foo) === false
```

Map All Things? Records vs ES6 Collections

Maps and Sets are nifty new ES6 collections of key/value pairs. That said, JavaScript objects still can be used as collections in many situations. No need to switch to the new ES6 collections unless the situation calls for it.

MDN has has a nice list of questions to determine when to use an object or a keyed collection:

- Are keys usually unknown until run time, and do you need to look them up dynamically?
- Do all values have the same type, and can be used interchangeably?
- Do you need keys that aren't strings?
- Are key-value pairs often added or removed?
- Do you have an arbitrary (easily changing) amount of key-value pairs?
- Is the collection iterated?

New ES6 Collections Yield a More Usable JavaScript

JavaScript collections have previously been quite limited, but this has been remedied with ES6. These new ES6 collections will add power and flexibility to the language, as well as simplify the task of JavaScript developers who adopt them.

Chapter 3: New Array.* and Array.prototype.* Methods

by Aurelio De Rosa

In this chapter we'll discuss most of the new methods available in ES6 that work with the `Array` type, using `Array.*` and `Array.prototype.*`.

When discussing them, 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. 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(arrayLike);
```

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;
  });
}

const result = double(1, 2, 3, 4);

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

A live demo of the previous code is available at JSBin.

This method is supported in Node and all modern browsers, with the exception of Internet Explorer. If you need to support older browsers, 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` value) 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's more than one element in the array, that will return a truthy value, and only the first is returned.

An example usage is shown below:

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

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

A live demo of the previous code is available at JSBin.

The method is supported in Node and all modern browsers, with the exception of Internet Explorer. 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:

```
const arr = [1, 2, 3, 4];
const 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 live demo of the previous code is available at JSBin.

The method is supported in Node and all modern browsers, with the exception of Internet Explorer. If you need a polyfill, one 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. If you want to learn more about array iterators, you can refer to the specifications or the MDN page.

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

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

An example of use is the following:

```
const arr = [1, 2, 3, 4];
const iterator = arr.keys();

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

A live demo is available at JSBin.

`Array.prototype.keys()` in Node and all modern browsers, with the exception of Internet Explorer.

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:

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

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

A live demo of the previous code is available at JSBin.

The `Array.prototype.values()` is currently not implemented in most browsers. In order for you to use it you need to transpile it via Babel.

Array.prototype.fill()

If you've worked in the PHP world (like me), you'll recall a function named `array_fill()` that was missing in JavaScript. In ES6 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).

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:

```
const 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 live demo of the previous code is available at JSBin.

The method is supported in Node and all modern browsers, with the exception of Internet Explorer. As polyfills you can employ the one on the method's page on MDN, or the polyfill developed by Addy Osmani.

Conclusion

In this chapter we've discussed several of the new methods introduced in ES6 that work with arrays. With the exception of `Array.prototype.values()`, they enjoy good browser support and can be used today!

Chapter 4: New String Methods — String.prototype.*

by Aurelio de Rosa

In my previous chapter on ES6 array methods, I introduced the new methods available in ECMAScript 6 that work with the `Array` type. In this tutorial, you'll learn about new ES6 methods that work with strings: `String.prototype.*`

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.

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. Before ES6, JavaScript had no such function, meaning 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”.

An example use of this method is shown below:

```
const str = 'hello!';
let 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 live demo of the previous code is available at JSBin.

The method is supported in Node and all modern browsers, with the exception of Internet Explorer. If you need to support older browsers, 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 with `'house'.endsWith('us', 4)`, we obtain `true`, because it's like we actually had the string `hous` (note the missing “e”).

An example use of this method is shown below:

```
const str = 'hello!';
const 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 live demo of the previous snippet is available at JSBin.

The method is supported in Node and all modern browsers, with the exception of Internet Explorer. If you need to support older browsers, 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:

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

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

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

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

You can find a live demo at JSBin.

`String.prototype.includes()` is supported in Node and all modern browsers, with the exception of Internet Explorer. If you need to support older browsers, as with 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:

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

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

result = str.repeat(0);

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

A live demo of the previous code is available at JSBin.

The method is supported in Node and all modern browsers, with the exception of Internet Explorer. If you need to support older browsers, two polyfills are available for this method: 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 it's 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 shortly. 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:

```
const name = 'Aurelio De Rosa';
const 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);
```

A live demo of the previous code is available at JSBin.

The method is supported in Node and all modern browsers, with the exception of Opera and Internet Explorer. If you need to support older browsers, you can employ a polyfill, such as this one available on npm.

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 chapter and that you'll continue to follow our channel to learn more about ECMAScript 6.

Chapter 5: New Number Methods

by Aurelio de Rosa

This chapter covers new and improved number methods in ES6 (ECMAScript 6).

I'll introduce you to the new methods and constants added to the `Number` data type. Some of the number methods covered, as we'll see, aren't new at all, but they've 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 get started.

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;
};
```

Just for 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 shortly. 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 the 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 live demo of the previous code is available at JSBin.

The method is supported in Node and all modern browsers, with the exception of Internet Explorer. If you need to support older browsers, you can employ a polyfill, such as the one available on the Mozilla Developer Network on the methods page. This is also reproduced 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 shouldn't be new to you. For a while now, JavaScript has had a method called `isNaN()` that's exposed through the `window` object. This method tests if a value is equal to `NaN`, in which case it returns `true`, or otherwise `false`. The problem with `window.isNaN()` is that it has an issue in that it also returns `true` 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)
```

Here, `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 live demo of the previous snippet is available at JSBin.

The method is supported in Node and all modern browsers, with the exception of Internet Explorer. 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 this works is because `NaN` is the only non-reflexive value in JavaScript, which means that it’s the only value that isn’t equal to itself.

Number.isFinite()

This method shares the same story as the previous one. In JavaScript there's 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 demonstrated 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's a method called `isFinite()` that belongs to `Number`. Its syntax is the following:

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

Here, `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 live demo of the previous snippet is available at JSBin.

The method is supported in Node and all modern browsers, with the exception of Internet Explorer. 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 ES6. 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 two conditions:

- 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, safe integers are all the integers from $-(2^{53} - 1)$ inclusive to $2^{53} - 1$ inclusive. These values are important and we'll discuss them a little more at the end of this section.

The syntax of this method is:

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

Here, `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 live demo of this code is available at JSBin.

The `Number.isSafeInteger()` is supported in Node and all modern browsers, with the exception of Internet Explorer. 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 also introduced two related constant values: `Number.MAX_SAFE_INTEGER` and `Number.MIN_SAFE_INTEGER`. The former represents the maximum safe integer in JavaScript — that is, $2^{53} - 1$ — while the latter the minimum safe integer, which is $-(2^{53} - 1)$. 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 chapter, they already existed in a 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. The purpose of adding these methods to `Number` is the modularization of globals.

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)
```

Here, `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 live demo of this code is available at JSBin.

These methods are supported in Node and all modern browsers, with the exception of Internet Explorer. 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);
};
```

ES6 Number Methods: Wrapping Up

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 ES6 has also has added another constant that I haven't mentioned so far. This 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.

Chapter 6: ES6 Arrow Functions: Fat and Concise Syntax in JavaScript

by Kyle Pennell

Arrow functions were introduced with ES6 as a new syntax for writing JavaScript functions. They save developers time and simplify function scope. The good news is that many major modern browsers support the use of arrow functions.

This chapter will cover the details of arrow functions — how to use them, common syntaxes, common use cases, and gotchas/pitfalls.

What Are Arrow Functions?

Arrow functions – also called “fat arrow” functions, from CoffeeScript (a transcompiled language) — are a more concise syntax for writing function expressions. They utilize a new token, `=>`, that looks like a fat arrow. Arrow functions are anonymous and change the way `this` binds in functions.

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

Using Arrow Functions

There are a variety of syntaxes available in arrow functions, of which MDN has a thorough list. We'll cover the common ones here to get you started. Let's compare how ES5 code with function expressions can now be written in ES6 using arrow functions.

Basic Syntax with Multiple Parameters (from MDN)

```
// (param1, param2, paramN) => expression

// ES5
var multiplyES5 = function(x, y) {
  return x * y;
};

// ES6
const multiplyES6 = (x, y) => { return x * y };
```

Code Example at JSBin.

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

Curly brackets aren't required if only one expression is present. The preceding example could also be written as:

```
const multiplyES6 = (x, y) => x * y;
```

Basic Syntax with One Parameter

Parentheses are optional when only one parameter is present

```
//ES5
var phraseSplitterEs5 = function phraseSplitter(phrase) {
  return phrase.split(' ');
};

//ES6
const phraseSplitterEs6 = phrase => phrase.split(" ");
```

```
console.log(phraseSplitterEs6("ES6 Awesomeness"));  // ["ES6",
"Awesomeness"]
```

Code Example at JSBin.

No Parameters

Parentheses are required when no parameters are present.

```
//ES5
var docLogEs5 = function docLog() {
    console.log(document);
};

//ES6
var docLogEs6 = () => { console.log(document); };
docLogEs6(); // #document... <html> ….
```

Code Example at JSBin.

Object Literal Syntax

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

```
//ES5
var setNameIdsEs5 = function setNameIds(id, name) {
  return {
    id: id,
    name: name
  };
};

// ES6
var setNameIdsEs6 = (id, name) => ({ id: id, name: name });

console.log(setNameIdsEs6 (4, "Kyle"));   // Object {id: 4,
name: "Kyle"}
```

Code Example at JSBin.

Use Cases for Arrow Functions

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

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

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

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

```
// ES5
var prices = smartPhones.map(function(smartPhone) {
  return smartPhone.price;
});

console.log(prices); // [649, 576, 489]
```

An arrow function is more concise and easier to read:

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

Code Example at JSBin.

Here's another example using the array filter method:

```
const array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15];

// ES5
var divisibleByThrreeES5 = array.filter(function (v){
  return v % 3 === 0;
```

```
});

// ES6
const divisibleByThrreeES6 = array.filter(v => v % 3 === 0);

console.log(divisibleByThrreeES6); // [3, 6, 9, 12, 15]
```

Code Example at JSBin.

Promises and Callbacks

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

```
// ES5
aAsync().then(function() {
  returnbAsync();
}).then(function() {
  returncAsync();
}).done(function() {
  finish();
});
```

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

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

Arrow functions should similarly simplify callback-laden NodeJS code.

What's the meaning of this?!

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

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

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

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

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

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

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

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

Gotchas and Pitfalls of Arrow Functions

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

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

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

More about *this*

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

Constructors

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

Generators

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

Arguments object

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

How Much Use Is There for Arrow Functions?

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

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

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

Arrow functions are here: they're powerful, concise, and developers love them. Perhaps it's time for you to start using them!

Chapter 7: Symbols and Their Uses

by Nilson Jacques

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 — such as symbols.

The symbol is a new primitive type, a unique token that's guaranteed never to clash with another symbol. In this sense, you could think of symbols as a kind of UUID (universally unique identifier). Let's look at how symbols work, and what we can do with them.

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'll get a new and completely unique value.

```
const foo = Symbol();
const bar = Symbol();

foo === bar
// <-- false
```

Symbols can also be created with a label, by passing a string as the first argument. The label doesn't 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 Symbols?

Symbols could 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;

case default:

throw new Error('Invalid application mode: ' + mode); }

}

}
```

```
Application.DEV = Symbol('dev'); Application.PROD =
Symbol('prod');

// Example use

const 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: `const 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.

```
const user = {}; const 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’s worth noting, however, that using symbols as keys doesn’t 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're 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.

`Symbol.iterator` is a well-known symbol that's used to assign a special method to objects, which allows them to be iterated over:

```
const band = ['Freddy', 'Brian', 'John', 'Roger'];
const 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.

`Symbol.for(key)` retrieves the symbol for a given key from the registry. If a symbol doesn't 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 doesn't exist in the registry returns undefined: `const debbie = Symbol.for('user'); const 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 chapter, is when you want to add “hidden” properties to objects that won’t 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 initialized 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 won’t 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 quite good. 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’s 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.

Chapter 8: How to Use Proxies

by Craig Buckler

In computing terms, proxies sit between you and the things you're 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.

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.

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:

```
const 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:

```
const 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:

```
const 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:

```
const 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;
    }
  }
};

const 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 object's 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`.
- **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 descriptor 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.

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) {




  const

    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. For example:

```
const myObject = {
```

```
  h: 'Hello',
  w: 'World'
};



// create a myObject proxy
const 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
const 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;

  const 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
const 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
const 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 may 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. For example, `arr[-1]` returns the last element, `arr[-2]` returns the second-to-last element, and so on.

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 an 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!

Currently, proxy support is implemented in Node and all current browsers, with the exception of Internet Explorer 11. However, please note that not all browsers support all traps. You can get a better idea of what's supported by consulting this browser compatibility table on the MDN Proxy page.

Unfortunately, it's not possible to polyfill or transpile ES6 proxy code using tools such as Babel, because proxies are powerful and have no ES5 equivalent.

Chapter 9: Destructuring Assignment

by Craig Buckler

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!

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:

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

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

You can ignore certain values, e.g.

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

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

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

```
const [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
const {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.

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

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

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

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

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

const {
    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 can’t 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
const { a, b, c } = myObject;
```

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

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

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

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

// THIS WORKS
let 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
const [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:

```
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 &amp;&amp; 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’s 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];
}

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

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

For-of Iteration

Consider an array of book information:

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

Destructuring assignment provides further enhancements, e.g.

```
for (const {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:

```
const [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!

Currently, support for destructuring assignment is good. It's available in Node and all major browsers, with the exception of Internet Explorer. If you need to support older browsers, it's advisable to use a compiler such as Babel or Traceur, which will translate ES6 destructuring assignments to an ES5 equivalent.

Chapter 10: ES6 Generators and Iterators: a Developer’s Guide

by Byron Houwens

ES6 brought a number of new features to the JavaScript language. Two of these features, generators and iterators, have substantially changed how we write specific functions in more complex front-end code.

While they do play nicely with each other, what they actually do can be a little confusing, 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 (const i of foo) {
  // do something with i
}
```

This is arguably way cleaner and easier to work with, and reminds me of 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.

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:

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

for (const 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 the `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.

In case you're interested in this sort of detail, 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 chapter: 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.

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:

```
const nextPrime = getNextPrime();
```

And then call `nextPrime` whenever we want to obtain — 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 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}
```

Here, that `done` key tells you whether or not 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 Generators and Iterators Now?

Although ECMAScript 2015 has been finalized and has been in the wild for some years, browser support for its features — particularly generators — is far from complete. If you really want to use these and other modern 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 can now be written , those are worth a look.

Conclusions

IGenerators and iterators 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.

Support for these generators and iterators is good. They're supported in Node and all modern browsers, with the exception of Internet Explorer. If you need to support older browsers, your best bet is to use a transpiler such as Babel.

Chapter 11: Object-oriented JavaScript: A Deep Dive into ES6 Classes

by Jeff Mott

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. ES6 classes give us a convenient syntax for defining the state and behavior of objects that will represent our concepts.

ES6 classes make our code safer by guaranteeing that 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
const today = {
  month: 24,
  day: 12,
};

const tomorrow = {
  year: today.year,
  month: today.month,
  day: today.day + 1,
};

const dayAfterTomorrow = {
  year: tomorrow.year,
```

```
  month: tomorrow.month,
  day: tomorrow.day + 1 <= 31 ? tomorrow.day + 1 : 1,
};
```

The date `today` isn't valid: there's 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
const today = new SimpleDate(2000, 2, 28);

// Manipulating data only through a fixed set of functions
ensures we maintain valid state
today.addDays(1);
```

Jargon

- 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

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.

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's 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 of JavaScript as of ES6, 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 of JavaScript is `Object.getOwnPropertySymbols`, and it allows the outside to access the symbol keys we tried to keep private: `const SimpleDate = (function() {`

```
const _yearKey = Symbol(); const _monthKey = Symbol(); const
_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 of 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: `const SimpleDate = (function() {`

```
const _years = new WeakMap(); const _months = new WeakMap();
const _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's 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: `// Get a reference to the "getDay" function const 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); const 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, 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**:

```
// This will be our common superclass


class Cache {


  get(key, defaultValue) {


    const 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 superclass interface
```

```
function compute(cache) {

  const cached = cache.get('result');

  if (!cached) {

    const result = // ...

    cache.set('result', result);

  }




  // ...

}




compute(new ArrayCache()); // use array cache through
superclass interface

compute(new LocalStorageCache()); // use local storage cache
through superclass interface
```

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 do in 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 for implementing 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:

```
const transmitter = {
  transmit() {}
};

const receiver = {
  receive() {}
};

// Create a proxy object that intercepts property accesses and
forwards to each parent,
// returning the first defined value it finds
const 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 ES6 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()) {}


const 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

Support for classes is pretty good. Hopefully this chapter has given you an insight into how classes work in ES6 and has demystified some of the jargon surrounding them.

Chapter 12: Understanding ES6 Modules

by Craig Buckler

This chapter explores ES6 modules, showing how they can be used today with the help of a transpiler.

Almost every language has a concept of *modules* — a way to include functionality declared in one file within another. Typically, a developer creates an encapsulated library of code responsible for handling related tasks. That library can be referenced by applications or other modules.

The benefits:

1. Code can be split into smaller files of self-contained functionality.
2. The same modules can be shared across any number of applications.
3. Ideally, modules need never be examined by another developer, because they've has been proven to work.
4. Code referencing a module understands it's a dependency. If the module file is changed or moved, the problem is immediately obvious.
5. Module code (usually) helps eradicate naming conflicts. Function `x()` in module1 cannot clash with function `x()` in module2. Options such as namespacing are employed so calls become `module1.x()` and `module2.x()`.

Where are Modules in JavaScript?

Anyone starting web development a few years ago would have been shocked to discover there was no concept of modules in JavaScript. It was impossible to directly reference or include one JavaScript file in another. Developers therefore resorted to alternative options.

Multiple HTML `<script>` Tags

HTML can load any number JavaScript files using multiple `<script>` tags:

```
<script src="lib1.js"></script>
<script src="lib2.js"></script>
<script src="core.js"></script>
<script>
console.log('inline code');
</script>
```

The average web page in 2018 uses 25 separate scripts, yet it's not a practical solution:

- Each script initiates a new HTTP request, which affects page performance. HTTP/2 alleviates the issue to some extent, but it doesn't help scripts referenced on other domains such as a CDN.
- Every script halts further processing while it's run.
- Dependency management is a manual process. In the code above, if `lib1.js` referenced code in `lib2.js`, the code would fail because it had not been loaded. That could break further JavaScript processing.
- Functions can override others unless appropriate module patterns are used. Early JavaScript libraries were notorious for using global function names or overriding native methods.

Script Concatenation

One solution to problems of multiple `<script>` tags is to concatenate all JavaScript files into a single, large file. This solves some performance and

dependency management issues, but it could incur a manual build and testing step.

Module Loaders

Systems such as RequireJS and SystemJS provide a library for loading and namespacing other JavaScript libraries at runtime. Modules are loaded using Ajax methods when required. The systems help, but could become complicated for larger code bases or sites adding standard `<script>` tags into the mix.

Module Bundlers, Preprocessors and Transpilers

Bundlers introduce a compile step so JavaScript code is generated at build time. Code is processed to include dependencies and produce a single ES5 cross-browser compatible concatenated file. Popular options include Babel, Browserify, webpack and more general task runners such as Grunt and Gulp.

A JavaScript build process requires some effort, but there are benefits:

- Processing is automated so there's less chance of human error.
- Further processing can lint code, remove debugging commands, minify the resulting file, etc.
- Transpiling allows you to use alternative syntaxes such as TypeScript or CoffeeScript.

ES6 Modules

The options above introduced a variety of competing module definition formats. Widely-adopted syntaxes included:

- CommonJS — the `module.exports` and `require` syntax used in Node.js
- Asynchronous Module Definition (AMD)
- Universal Module Definition (UMD).

A single, native module standard was therefore proposed in ES6 (ES2015).

Everything inside an ES6 module is private by default, and runs in strict mode (there's no need for `'use strict'`). Public variables, functions and classes are exposed using `export`. For example:

```
// lib.js
export const PI = 3.1415926;

export function sum(...args) {
  log('sum', args);
  return args.reduce((num, tot) => tot + num);
}

export function mult(...args) {
  log('mult', args);
  return args.reduce((num, tot) => tot * num);
}

// private function
function log(...msg) {
  console.log(...msg);
}
```

Alternatively, a single `export` statement can be used. For example:

```
// lib.js
const PI = 3.1415926;

function sum(...args) {
  log('sum', args);
```

```
  return args.reduce((num, tot) => tot + num);
}

function mult(...args) {
  log('mult', args);
  return args.reduce((num, tot) => tot * num);
}

// private function
function log(...msg) {
  console.log(...msg);
}

export { PI, sum, mult };
```

`import` is then used to pull items from a module into another script or module:

```
// main.js
import { sum } from './lib.js';

console.log( sum(1,2,3,4) ); // 10
```

In this case, `lib.js` is in the same folder as `main.js`. Absolute file references (starting with `/`), relative file references (starting `./` or `../`) or full URLs can be used.

Multiple items can be imported at one time:

```
import { sum, mult } from './lib.js';

console.log( sum(1,2,3,4) );  // 10
console.log( mult(1,2,3,4) ); // 24
```

and imports can be aliased to resolve naming collisions:

```
import { sum as addAll, mult as multiplyAll } from './lib.js';

console.log( addAll(1,2,3,4) );      // 10
console.log( multiplyAll(1,2,3,4) ); // 24
```

Finally, all public items can be imported by providing a namespace:

```
import * as lib from './lib.js';

console.log( lib.PI );            // 3.1415926
console.log( lib.add(1,2,3,4) );  // 10
console.log( lib.mult(1,2,3,4) ); // 24
```

Using ES6 Modules in Browsers

At the time of writing, ES6 modules are supported in Chromium-based browsers (v63+), Safari 11+, and Edge 16+. Firefox support will arrive in version 60 (it's behind an `about:config` flag in v58+).

Scripts which use modules must be loaded by setting a `type="module"` attribute in the `<script>` tag. For example:

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

or inline:

```
<script type="module">
  import { something } from './somewhere.js';
  // ...
</script>
```

Modules are parsed once, regardless of how many times they're referenced in the page or other modules.

Server Considerations

Modules must be served with the MIME type `application/javascript`. Most servers will do this automatically, but be wary of dynamically generated scripts or `.mjs` files (see the Node.js section below).

Regular `<script>` tags can fetch scripts on other domains but modules are fetched using cross-origin resource sharing (CORS). Modules on different domains must therefore set an appropriate HTTP header, such as `Access-Control-Allow-Origin: *`.

Finally, modules won't send cookies or other header credentials unless a `crossorigin="use-credentials"` attribute is added to the `<script>` tag and the response contains the header `Access-Control-Allow-Credentials: true`.

Module Execution is Deferred

The `<script defer>` attribute delays script execution until the document has loaded and parsed. Modules — *including inline scripts* — defer by default. Example:

```
<!-- runs SECOND -->
<script type="module">
  // do something...
</script>

<!-- runs THIRD -->
<script defer src="c.js"></script>

<!-- runs FIRST -->
<script src="a.js"></script>

<!-- runs FOURTH -->
<script type="module" src="b.js"></script>
```

Module Fallbacks

Browsers without module support won't run `type="module"` scripts. A fallback script can be provided with a `nomodule` attribute which module-compatible browsers ignore. For example:

```
<script type="module" src="runs-if-module-supported.js">
</script>
<script nomodule src="runs-if-module-not-supported.js">
</script>
```

Should You Use Modules in the Browser?

Browser support is growing, but it's possibly a little premature to switch to ES6 modules. For the moment, it's probably better to use a module bundler to create a script that works everywhere.

Using ES6 Modules in Node.js

When Node.js was released in 2009, it would have been inconceivable for any runtime not to provide modules. CommonJS was adopted, which meant the Node package manager, npm, could be developed. Usage grew exponentially from that point.

A CommonJS module can be coded in a similar way to an ES2015 module. `module.exports` is used rather than `export`:

```
// lib.js
const PI = 3.1415926;

function sum(...args) {
  log('sum', args);
  return args.reduce((num, tot) => tot + num);
}

function mult(...args) {
  log('mult', args);
  return args.reduce((num, tot) => tot * num);
}

// private function
function log(...msg) {
  console.log(...msg);
}

module.exports = { PI, sum, mult };
```

`require` (rather than `import`) is used to pull this module into another script or module:

```
const { sum, mult } = require('./lib.js');

console.log( sum(1,2,3,4) );  // 10
console.log( mult(1,2,3,4) ); // 24
```

`require` can also import all items:

```
const lib = require('./lib.js');
```

```
console.log( lib.PI );            // 3.1415926
console.log( lib.add(1,2,3,4) );  // 10
console.log( lib.mult(1,2,3,4) ); // 24
```

So ES6 modules were easy to implement in Node.js, right? *Er, no*.

ES6 modules are behind a flag in Node.js 9.8.0+ and will not be fully implemented until at least version 10. While CommonJS and ES6 modules share similar syntax, they work in fundamentally different ways:

- ES6 modules are pre-parsed in order to resolve further imports before code is executed.
- CommonJS modules load dependencies on demand while executing the code.

It would make no difference in the example above, but consider the following ES2015 module code:

```
// ES2015 modules

// --------------------------------
// one.js
console.log('running one.js');
import { hello } from './two.js';
console.log(hello);

// --------------------------------
// two.js
console.log('running two.js');
export const hello = 'Hello from two.js';
```

The output for ES2015:

```
running two.js
running one.js
hello from two.js
```

Similar code written using CommonJS:

```
// CommonJS modules

// --------------------------------
// one.js
```

```
console.log('running one.js');
const hello = require('./two.js');
console.log(hello);

// --------------------------------
// two.js
console.log('running two.js');
module.exports = 'Hello from two.js';
```

The output for CommonJS:

```
running one.js
running two.js
hello from two.js
```

Execution order could be critical in some applications, and what would happen if ES2015 and CommonJS modules were mixed in the same file? To resolve this problem, Node.js will only permit ES6 modules in files with the extension `.mjs`. Files with a `.js` extension will default to CommonJS. It's a simple option which removes much of the complexity and should aid code editors and linters.

Should You Use ES6 Modules in Node.js?

ES6 modules are only practical from Node.js v10 onwards (released in April 2018). Converting an existing project is unlikely to result in any benefit, and would render an application incompatible with earlier versions of Node.js.

For new projects, ES6 modules provide an alternative to CommonJS. The syntax is identical to client-side coding, and may offer an easier route to isomorphic JavaScript, which can run in the either the browser or on a server.

Module Melee

A standardized JavaScript module system took many years to arrive, and even longer to implement, but the problems have been rectified. All mainstream browsers and Node.js from mid 2018 support ES6 modules, although a switch-over lag should be expected while everyone upgrades.

Learn ES6 modules today to benefit your JavaScript development tomorrow.

Chapter 13: An Overview of JavaScript Promises

by Sandeep Panda

This tutorial covers the basics of JavaScript promises, showing how you can leverage them in your JavaScript development.

The concept of promises is not new to web development. Many of us have already used promises in the form of libraries such as Q, when.js, RSVP.js, etc. Even jQuery has something called a Deferred object, which is similar to a promise. But now we have native support for promises in JavaScript, which is really exciting.

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 is that the actual data isn't 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:

```
const promise = new Promise((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:

```
const promise = new Promise((resolve, reject) => {
  const request = new XMLHttpRequest();

  request.open('GET', 'https://api.icndb.com/jokes/random');
  request.onload = () => {
    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 = () => {
    reject(Error('Error fetching data.')); // error occurred,
reject the  Promise
  };

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

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

promise.then((data) => {
  console.log('Got data! Promise fulfilled.');
  document.body.textContent = JSON.parse(data).value.joke;
}, (error) => {
  console.log('Promise rejected.');
  console.log(error.message);
});
```

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 https://api.icndb.com/jokes/random, which returns a random joke. When a JSON response is received from the remote server, it's 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're 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 CodePen example. To view a new random joke, hit the *RERUN* button in the bottom right-hand corner of the embed. 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're not interested in knowing when the promise is settled. We're only concerned with the final outcome of the promise.

Chaining Promises

It’s sometimes desirable to chain promises together. For instance, you might have multiple asynchronous operations to be performed. When one operation gives you data, you’ll 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
}

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

promise.then((result) => {
  //we have our result here
  return getPromise(result); //return a promise here again
}).then((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((result) => {
  console.log('Got data!', result);
}).catch((error) => {
  console.log('Error occurred!', error);
});
```

This is equivalent to:

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

Note that if the promise was rejected and `then()` does't 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's 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's more to the API that has not been covered here. To learn more about Promises, check out my follow-up article A Deeper Dive Into JavaScript Promises, as well as these great resources:

- HTML5Rocks
- Mozilla Developer Network

Chapter 14: JavaScript Decorators: What They Are and When to Use Them

by Graham Cox

With the introduction of ES2015+, and as transpilation has become commonplace, many of you will have come across newer language features, either in real code or tutorials. One of these features that often has people scratching their heads when they first come across them are JavaScript decorators.

Decorators have become popular thanks to their use in Angular 2+. In Angular, decorators are available thanks to TypeScript, but in JavaScript they're currently a stage 2 proposal, meaning they should be part of a future update to the language. Let's take a look at what decorators are, and how they can be used to make your code cleaner and more easily understandable.

What is a Decorator?

In its simplest form, a decorator is simply a way of wrapping one piece of code with another — literally “decorating” it. This is a concept you might well have heard of previously as **functional composition**, or **higher-order functions**.

This is already possible in standard JavaScript for many use cases, simply by calling on one function to wrap another:

```
function doSomething(name) {
  console.log('Hello, ' + name);
}

function loggingDecorator(wrapped) {
  return function() {
    console.log('Starting');
    const result = wrapped.apply(this, arguments);
    console.log('Finished');
    return result;
  }
}

const wrapped = loggingDecorator(doSomething);
```

This example produces a new function — in the variable `wrapped` — that can be called exactly the same way as the `doSomething` function, and will do exactly the same thing. The difference is that it will do some logging before and after the wrapped function is called:

```
doSomething('Graham');
// Hello, Graham

wrapped('Graham');
// Starting
// Hello, Graham
// Finished
```

How to Use JavaScript Decorators

Decorators use a special syntax in JavaScript, whereby they are prefixed with an `@` symbol and placed immediately before the code being decorated.

Note: at the time of writing, the decorators are currently in "Stage 2 Draft" form, meaning that they are mostly finished but still subject to changes.

It's possible to use as many decorators on the same piece of code as you desire, and they'll be applied in the order that you declare them.

For example:

```
@log()
@immutable()
class Example {
  @time('demo')
  doSomething() {
    //
  }
}
```

This defines a class and applies three decorators — two to the class itself, and one to a property of the class:

- `@log` could log all access to the class
- `@immutable` could make the class immutable — maybe it calls `Object.freeze` on new instances
- `@time` will record how long a method takes to execute and log this out with a unique tag.

At present, using decorators requires transpiler support, since no current browser or Node release has support for them yet. If you're using Babel, this is enabled simply by using the transform-decorators-legacy plugin.

Note: the use of the word "legacy" in this plugin is because it supports the Babel 5 way of handling decorators, which might well be different from the final form when they're standardized.

Why Use Decorators?

Whilst functional composition is already possible in JavaScript, it’s significantly more difficult — or even impossible — to apply the same techniques to other pieces of code (e.g. classes and class properties).

The decorator proposal adds support for class and property decorators that can be used to resolve these issues, and future JavaScript versions will probably add decorator support for other troublesome areas of code.

Decorators also allow for a cleaner syntax for applying these wrappers around your code, resulting in something that detracts less from the actual intention of what you’re writing.

Different Types of Decorator

At present, the only types of decorator that are supported are on classes and members of classes. This includes properties, methods, getters, and setters.

Decorators are actually nothing more than functions that return another function, and that are called with the appropriate details of the item being decorated. These decorator functions are evaluated once when the program first runs, and the decorated code is replaced with the return value.

Class member decorators

Property decorators are applied to a single member in a class — whether they are properties, methods, getters, or setters. This decorator function is called with three parameters:

- `target`: the class that the member is on.
- `name`: the name of the member in the class.
- `descriptor`: the member descriptor. This is essentially the object that would have been passed to Object.defineProperty.

The classic example used here is `@readonly`. This is implemented as simply as:

```
function readonly(target, name, descriptor) {
  descriptor.writable = false;
  return descriptor;
}
```

Literally updating the property descriptor to set the “writable” flag to false.

This is then used on a class property as follows:

```
class Example {
  a() {}
  @readonly
  b() {}
}
```

```
const e = new Example();
e.a = 1;
e.b = 2;
// TypeError: Cannot assign to read only property 'b' of object
'#<Example>'
```

But we can do better than this. We can actually replace the decorated function with different behavior. For example, let's log all of the inputs and outputs:

```
function log(target, name, descriptor) {
  const original = descriptor.value;
  if (typeof original === 'function') {
    descriptor.value = function(...args) {
      console.log(`Arguments: ${args}`);
      try {
        const result = original.apply(this, args);
        console.log(`Result: ${result}`);
        return result;
      } catch (e) {
        console.log(`Error: ${e}`);
        throw e;
      }
    }
  }
  return descriptor;
}
```

This replaces the entire method with a new one that logs the arguments, calls the original method and then logs the output.

Note that we've used the spread operator here to automatically build an array from all of the arguments provided, which is the more modern alternative to the old `arguments` value.

We can see this in use as follows:

```
class Example {
  @log
  sum(a, b) {
    return a + b;
  }
}
```

```
const e = new Example();
e.sum(1, 2);
// Arguments: 1,2
// Result: 3
```

You'll notice that we had to use a slightly funny syntax to execute the decorated method. This could cover an entire article of its own, but in brief, the `apply` function allows you to call the function, specifying the `this` value and the arguments to call it with.

Taking it up a notch, we can arrange for our decorator to take some arguments. For example, let's re-write our `log` decorator as follows:

```
function log(name) {
  return function decorator(t, n, descriptor) {
    const original = descriptor.value;
    if (typeof original === 'function') {
      descriptor.value = function(...args) {
        console.log(`Arguments for ${name}: ${args}`);
        try {
          const result = original.apply(this, args);
          console.log(`Result from ${name}: ${result}`);
          return result;
        } catch (e) {
          console.log(`Error from ${name}: ${e}`);
          throw e;
        }
      }
    }
    return descriptor;
  };
}
```

This is getting more complex now, but when we break it down we have this:

- A function, `log`, that takes a single parameter: `name`.
- This function then returns a function that *is itself a decorator*.

This is identical to the earlier `log` decorator, except that it makes use of the `name` parameter from the outer function.

This is then used as follows:

```
class Example {
  @log('some tag')
  sum(a, b) {
    return a + b;
  }
}

const e = new Example();
e.sum(1, 2);
// Arguments for some tag: 1,2
// Result from some tag: 3
```

Straight away we can see that this allows us to distinguish between different log lines using a tag that we've supplied ourselves.

This works because the `log('some tag')` function call is evaluated by the JavaScript runtime straight away, and then the response from that is used as the decorator for the `sum` method.

Class decorators

Class decorators are applied to the entire class definition all in one go. The decorator function is called with a single parameter which is the constructor function being decorated.

Note that this is applied to the constructor function and not to each instance of the class that is created. This means that if you want to manipulate the instances you need to do so yourself by returning a wrapped version of the constructor.

In general, these are less useful than class member decorators, because everything you can do here you can do with a simple function call in exactly the same way. Anything you do with these needs to end up returning a new constructor function to replace the class constructor.

Going back to our logging example, let's write one that logs the constructor parameters:

```
function log(Class) {
  return (...args) => {
    console.log(args);
```

```
    return new Class(...args);
  };
}
```

Here we are accepting a class as our argument, and returning a new function that will act as the constructor. This simply logs the arguments and returns a new instance of the class constructed with those arguments.

For example:

```
@log
class Example {
  constructor(name, age) {
  }
}

const e = new Example('Graham', 34);
// [ 'Graham', 34 ]
console.log(e);
// Example {}
```

We can see that constructing our Example class will log out the arguments provided and that the constructed value is indeed an instance of `Example`. Exactly what we wanted.

Passing parameters into class decorators works exactly the same as for class members:

```
function log(name) {
  return function decorator(Class) {
    return (...args) => {
      console.log(`Arguments for ${name}: args`);
      return new Class(...args);
    };
  }
}

@log('Demo')
class Example {
  constructor(name, age) {}
}

const e = new Example('Graham', 34);
// Arguments for Demo: args
```

```
console.log(e);
// Example {}
```

Real World Examples

Core decorators

There's a fantastic library called Core Decorators that provides some very useful common decorators that are ready to use right now. These generally allow for very useful common functionality (e.g. timing of method calls, deprecation warnings, ensuring that a value is read-only) but utilizing the much cleaner decorator syntax.

React

The React library makes very good use of the concept of Higher-Order Components. These are simply React components that are written as a function, and that wrap around another component.

These are an ideal candidate for using as a decorator, because there's very little you need to change to do so. For example, the react-redux library has a function, `connect`, that's used to connect a React component to a Redux store.

In general, this would be used as follows:

```
class MyReactComponent extends React.Component {}
export default connect(mapStateToProps, mapDispatchToProps)
(MyReactComponent);
```

However, because of how the decorator syntax works, this can be replaced with the following code to achieve the exact same functionality:

```
@connect(mapStateToProps, mapDispatchToProps)
export default class MyReactComponent extends React.Component
{}
```

MobX

The MobX library makes extensive use of decorators, allowing you to easily mark fields as Observable or Computed, and marking classes as Observers.

Summary

Class member decorators provide a very good way of wrapping code inside a class in a very similar way to how you can already do so for freestanding functions. This provides a good way of writing some simple helper code that can be applied to a lot of places in a very clean and easy-to-understand manner.

The only limit to using such a facility is your imagination!

Chapter 15: Enhanced Object Literals

by Craig Buckler

This chapter looks at what's possible with object literals in JavaScript, especially in the light of recent ECMAScript updates.

The ability to create JavaScript objects using literal notation is powerful. New features introduced from ES2015 (ES6) make object handling even easier in all modern browsers (not IE) and Node.js.

Creating objects in some languages can be expensive in terms of development time and processing power when a `class` must be declared before anything can be achieved. In JavaScript, it's easy to create objects on the fly. For example: `// ES5-compatible code var myObject = { prop1: 'hello', prop2: 'world', output: function() { console.log(this.prop1 + ' ' + this.prop2); } }; myObject.output(); // hello world`

Single-use objects are used extensively. Examples include configuration settings, module definitions, method parameters, return values from functions, etc. ES2015 (ES6) added a range of features to enhance object literals.

Object Initialization From Variables

Objects' properties are often created from variables with the same name. For example: `// ES5 code var`

```
a = 1, b = 2, c = 3; obj = {

a: a,

b: b,

c: c

};


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

There's no need for nasty repetition in ES6!…

```
// ES6 code const


  a = 1, b = 2, c = 3; obj = {


    a


    b


    c


  };
```

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

This could be useful for returned objects when using a [revealing module pattern](), which (effectively) namespaces code in order to avoid naming conflicts. For example: `// ES6 code const lib = (() => {`

```
function sum(a, b) { return a + b; }

function mult(a, b) { return a * b; }


return {

sum,

mult

};


}());


console.log( lib.sum(2, 3) ); // 5

console.log( lib.mult(2, 3) ); // 6
```

You've possibly seen it used in ES6 modules: `// lib.js function sum(a, b) { return a + b; }`

```
function mult(a, b) { return a * b; }


export { sum, mult };
```

Object Method Definition Shorthand

Object methods in ES5 require the `function` statement. For example: `// ES5 code var lib = {`

```
sum: function(a, b) { return a + b; }, mult: function(a, b) { return a * b; }
```

```
};
```

```
console.log( lib.sum(2, 3) ); // 5
```

```
console.log( lib.mult(2, 3) ); // 6
```

This is no longer necessary in ES6; it permits the following shorthand syntax: `// ES6 code const lib = {`

```
sum(a, b) { return a + b; }, mult(a, b) { return a * b; }
```

```
};
```

```
console.log( lib.sum(2, 3) ); // 5
```

```
console.log( lib.mult(2, 3) ); // 6
```

It's not possible to use ES6 fat arrow `=>` function syntax here, because the method requires a name. That said, you can use arrow functions if you name each method directly (like ES5). For example: `// ES6 code const lib = {`

```
sum: (a, b) => a + b, mult: (a, b) => a * b };
```

```
console.log( lib.sum(2, 3) ); // 5
```

```
console.log( lib.mult(2, 3) ); // 6
```

Dynamic Property Keys

In ES5, it wasn't possible to use a variable for a key name, although it could be added *after* the object had been created. For example:

```
// ES5 code
var
  key1 = 'one',
  obj = {
    two: 2,
    three: 3
  };

obj[key1] = 1;

// obj.one = 1, obj.two = 2, obj.three = 3
```

Object keys can be dynamically assigned in ES6 by placing an expression in `[` square brackets `]`. For example:

```
// ES6 code
const
  key1 = 'one',
  obj = {
    [key1]: 1,
    two: 2,
    three: 3
  };

// obj.one = 1, obj.two = 2, obj.three = 3
```

Any expression can be used to create a key. For example:

```
// ES6 code
const
  i = 1,
  obj = {
    ['i' + i]: i
  };

console.log(obj.i1); // 1
```

A dynamic key can be used for methods as well as properties. For example:

```
// ES6 code
const
  i = 2,
  obj = {
    ['mult' + i]: x => x * i
  };

console.log( obj.mult2(5) ); // 10
```

Whether you *should* create dynamic properties and methods is another matter. The code can be difficult to read, and it may be preferable to create object factories or classes.

Destructuring (Variables From Object Properties)

It's often necessary to extract a property value from an object into another variable. This had to be explicitly declared in ES5. For example:

```
// ES5 code
var myObject = {
  one:   'a',
  two:   'b',
  three: 'c'
};

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

ES6 supports destructuring: you can create a variable with the same name as an equivalent object property. For example:

```
// ES6 code
const myObject = {
  one:   'a',
  two:   'b',
  three: 'c'
};

const { one, two, three } = myObject;
// one = 'a', two = 'b', three = 'c'
```

It's also possible to assign properties to variables with any name using the notation `{ propertyName: newVariable }`. For example:

```
// ES6 code
const myObject = {
  one:   'a',
  two:   'b',
  three: 'c'
};
```

```
const { one: first, two: second, three: third } = myObject;
// first = 'a', second = 'b', third = 'c'
```

More complex objects with nested arrays and sub-objects can also be referenced in destructuring assignments. For example:

```
// ES6 code
const meta = {
  title: 'Enhanced Object Literals',
  pageinfo: {
    url: 'https://www.sitepoint.com/',
    description: 'How to use object literals in ES2015 (ES6).',
    keywords: 'javascript, object, literal'
  }
};

const {
  title   : doc,
  pageinfo: { keywords: topic }
} = meta;

/*
  doc   = 'Enhanced Object Literals'
  topic = 'javascript, object, literal'
*/
```

This initially appears complicated, but remember that in all destructuring assignments:

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

There are a number of caveats. You can’t start a statement with a curly brace, because it looks like a code block. For example:

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

You must either declare the variables — for example:

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

or use parentheses if variables have already been declared — for example:

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

You should therefore be careful not to mix declared and undeclared variables.

There are a number of situations where object destructuring is useful.

Default Function Parameters

It's often easier to pass a single object to a function than use a long list of arguments. For example:

```
prettyPrint( {
  title: 'Enhanced Object Literals',
  publisher: {
    name: 'SitePoint',
    url: 'https://www.sitepoint.com/'
  }
} );
```

In ES5, it's necessary to parse the object to ensure appropriate defaults are set. For example:

```
// ES5 assign defaults
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. For example:

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

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}`;

}
```

Whether you find this code easier to read is another matter!

Parsing Returned Objects

Functions can only return one value, but that could be an object with hundreds of properties and/or methods. In ES5, it's necessary to obtain the returned object, then extract values accordingly. For example:

```
// ES5 code
var
  obj = getObject(),
  one = obj.one,
  two = obj.two,
  three = obj.three;
```

ES6 destructuring makes this process simpler, and there's no need to retain the object as a variable:

```
// ES6 code
const { one, two, three } = getObject();
```

You may have seen similar assignments in Node.js code. For example, if you only required the File System (`fs`) methods `readFile` and `writeFile`, you could reference them directly. For example:

```
// ES6 Node.js
const { readFile, writeFile } = require('fs');
```

```
readFile('file.txt', (err, data) => {
  console.log(err || data);
});

writeFile('new.txt', 'new content', err => {
  console.log(err || 'file written');
});
```

ES2018 (ES9) Rest/Spread Properties

In ES2015, rest parameter and spread operator three-dot (. . .) notation applied to arrays only. ES2018 enables similar rest/spread functionality for objects. A basic example:

```
const myObject = {
  a: 1,
  b: 2,
  c: 3
};

const { a, ...x } = myObject;
// a = 1
// x = { b: 2, c: 3 }
```

You can use the technique to pass values to a function:

```
restParam({
  a: 1,
  b: 2,
  c: 3
});

function restParam({ a, ...x }) {
  // a = 1
  // x = { b: 2, c: 3 }
}
```

You can only use a single rest property at the end of the declaration. In addition, it only works on the top level of each object and not sub-objects.

The spread operator can be used within other objects. For example:

```
const
  obj1 = { a: 1, b: 2, c: 3 },
  obj2 = { ...obj1, z: 26 };

// obj2 is { a: 1, b: 2, c: 3, z: 26 }
```

You could use the spread operator to clone objects (`obj2 = { ...obj1 };`), but be aware you only get shallow copies. If a property holds another

object, the clone will refer to the same object.

ES2018 (ES9) rest/spread property support is patchy, but it’s available in Chrome, Firefox and Node.js 8.6+.

Object literals have always been useful. The new features introduced from ES2015 did not fundamentally change how JavaScript works, but they save typing effort and lead to clearer, more concise code.

Chapter 16: Introduction to the Fetch API

by Ludovico Fischer

In this chapter, we'll learn what the new Fetch API looks like, what problems it solves, and the most practical way to retrieve remote data inside your web page using the `fetch()` function.

For years, XMLHttpRequest has been web developers' trusted sidekick. Whether directly or under the hood, `XMLHttpRequest` has enabled Ajax and a whole new type of interactive experience, from Gmail to Facebook.

However, `XMLHttpRequest` is slowly being superseded by the Fetch API. Both can be used to make network requests, but the Fetch API is Promise-based, which enables a cleaner, more concise syntax and helps keep you out of callback hell.

The Fetch API

The Fetch API provides a `fetch()` method defined on the `window` object, which you can use to perform requests. This method returns a Promise that you can use to retrieve the response of the request.

The `fetch` method only has one mandatory argument, which is the URL of the resource you wish to fetch. A very basic example would look something like the following. This fetches the top five posts from r/javascript on Reddit: `fetch('https://www.reddit.com/r/javascript/top/.json?limit=5') .then(res => console.log(res));`

If you inspect the response in your browser's console, you should see a `Response` object with several properties:

```
{
  body: ReadableStream
  bodyUsed: false
  headers: Headers {}
  ok: true
  redirected: false
  status: 200
  statusText: ""
  type: "cors"
  url: "https://www.reddit.com/top/.json?count=5"
}
```

It seems that the request was successful, but where are our top five posts? Let's find out.

Loading JSON

We can’t block the user interface waiting until the request finishes. That’s why `fetch()` returns a `Promise`, an object which represents a future result. In the above example, we’re using the `then` method to wait for the server’s response and log it to the console.

Now let’s see how we can extract the JSON payload from that response once the request completes:

```
fetch('https://www.reddit.com/r/javascript/top/.json?limit=5')
.then(res => res.json())
.then(json => console.log(json));
```

We start the request by calling `fetch()`. When the promise is fulfilled, it returns a `Response` object, which exposes a `json` method. Within the first `then()` we can call this `json` method to return the response body as JSON.

However, the `json` method also returns a promise, which means we need to chain on another `then()`, before the JSON response is logged to the console.

And why does `json()` return a promise? Because HTTP allows you to stream content to the client chunk by chunk, so even if the browser receives a response from the server, the content body might not all be there yet!

Async … await

The `.then()` syntax is nice, but a more concise way to process promises in 2018 is using `async … await` — a new syntax introduced by ES2017. Using `async Ú await` means that we can mark a function as `async`, then wait for the promise to complete with the `await` keyword, and access the result as a normal object. Async functions are supported in all modern browsers (not IE or Opera Mini) and Node.js 7.6+.

Here’s what the above example would look like (slightly expanded) using `async … await`:

```
async function fetchTopFive(sub) {
  const URL = `https://www.reddit.com/r/${sub}/top/.json?
limit=5`;
  const fetchResult = fetch(URL)
  const response = await fetchResult;
  const jsonData = await response.json();
  console.log(jsonData);
}

fetchTopFive('javascript');
```

Not much has changed. Apart from the fact that we've created an `async` function, to which we're passing the name of the subreddit, we're now awaiting the result of calling `fetch()`, then using `await` again to retrieve the JSON from the response.

That's the basic workflow, but things involving remote services doesn't always go smoothly.

Handling Errors

Imagine we ask the server for a non-existing resource or a resource that requires authorization. With `fetch()`, you must handle application-level errors, like 404 responses, inside the normal flow. As we saw previously, `fetch()` returns a `Response` object with an `ok` property. If `response.ok` is `true`, the response status code lies within the 200 range:

```
async function fetchTopFive(sub) {
  const URL = `http://httpstat.us/404`; // Will return a 404
  const fetchResult = fetch(URL)
  const response = await fetchResult;
  if (response.ok) {
    const jsonData = await response.json();
    console.log(jsonData);
  } else {
    throw Error(response.statusText);
  }
}

fetchTopFive('javascript');
```

The meaning of a response code from the server varies from API to API, and oftentimes checking `response.ok` might not be enough. For example, some APIs will return a 200 response even if your API key is invalid. Always read the API documentation!

To handle a network failure, use a `try … catch` block:

```
async function fetchTopFive(sub) {
  const URL = `https://www.reddit.com/r/${sub}/top/.json?
limit=5`;
  try {
    const fetchResult = fetch(URL)
    const response = await fetchResult;
    const jsonData = await response.json();
    console.log(jsonData);
  } catch(e){
    throw Error(e);
  }
}
```

```
fetchTopFive('javvascript'); // Notice the incorrect spelling
```

The code inside the `catch` block will run only when a network error occurs.

You’ve learned the basics of making requests and reading responses. Now let’s customize the request further.

Change the Request Method and Headers

Looking at the example above, you might be wondering why you couldn't just use one of the existing XMLHttpRequest wrappers. The reason is that there's more the fetch API offers you than just the `fetch()` method.

While you must use the same instance of `XMLHttpRequest` to perform the request and retrieve the response, the fetch API lets you configure request objects explicitly.

For example, if you need to change how `fetch()` makes a request (e.g. to configure the request method) you can pass a `Request` object to the `fetch()` function. The first argument to the `Request` constructor is the request URL, and the second argument is an option object that configures the request:

```
async function fetchTopFive(sub) {
  const URL = `https://www.reddit.com/r/${sub}/top/.json?
limit=5`;
  try {
    const fetchResult = fetch(
      new Request(URL, { method: 'GET', cache: 'reload' })
    );
    const response = await fetchResult;
    const jsonData = await response.json();
    console.log(jsonData);
  } catch(e){
    throw Error(e);
  }
}

fetchTopFive('javascript');
```

Here, we specified the request method and asked it never to cache the response.

You can change the request headers by assigning a `Headers` object to the request `headers` field. Here's how to ask for JSON content only with the `'Accept'` header:

```
const headers = new Headers();
headers.append('Accept', 'application/json');
const request = new Request(URL, { method: 'GET', cache:
'reload', headers: headers });
```

You can create a new request from an old one to tweak it for a different use case. For example, you can create a POST request from a GET request to the same resource. Here's an example:

```
const postReq = new Request(request, { method: 'POST' });
```

You also can access the response headers, but keep in mind that they're read-only values.

```
fetch(request).then(response => console.log(response.headers));
```

`Request` and `Response` follow the HTTP specification closely; you should recognize them if you've ever used a server-side language. If you're interested in finding out more, you can read all about them on the Fetch API page on MDN.

Bringing it all Together

To round off the chapter, here's a runnable example demonstrating how to fetch the top five posts from a particular subreddit and display their details in a list.

Try entering a few subreddits (e.g. 'javascript', 'node', 'linux', 'lolcats') as well as a couple of non-existent ones.

Where to Go from Here

In this chapter, you've seen what the new Fetch API looks like and what problems it solves. I've demonstrated how to retrieve remote data with the `fetch()` method, how to handle errors and to create `Request` objects to control the request method and headers.

Support for `fetch()` is good. If you need to support older browsers a polyfill is available.

So next time you reach for a library like jQuery to make Ajax requests, take a moment to think if you could use native browser methods instead.

Chapter 17: ES6 (ES2015) and Beyond: Understanding JavaScript Versioning

by James Wright

As programming languages go, JavaScript's development has been positively frantic in the last few years. With each year now seeing a new release of the ECMAScript specification, it's easy to get confused about JavaScript versioning, which version supports what, and how you can future-proof your code.

To better understand the how and why behind this seemingly constant stream of new features, let's take a brief look at the history of the JavaScript and JavaScript versioning, and find out why the standardization process is so important.

The Early History of JavaScript Versioning

The prototype of JavaScript was written in just ten days in May 1995 by Brendan Eich. He was initially recruited to implement a Scheme runtime for Netscape Navigator, but the management team pushed for a C-style language that would complement the then recently released Java.

JavaScript made its debut in version 2 of Netscape Navigator in December 1995. The following year, Microsoft reverse-engineered JavaScript to create their own version, calling it JScript. JScript shipped with version 3 of the Internet Explorer browser, and was almost identical to JavaScript — even including all the same bugs and quirks — but it did have some extra Internet Explorer-only features.

The Birth of ECMAScript

The necessity of ensuring that JScript (and any other variants) remained compatible with JavaScript motivated Netscape and Sun Microsystems to standardize the language. They did this with the help of the European Computer Manufacturers Association, who would host the standard. The standardized language was called ECMAScript to avoid infringing on Sun's Java trademark — a move that caused a fair deal of confusion. Eventually ECMAScript was used to refer to the specification, and JavaScript was (and still is) used to refer to the language itself.

The working group in charge of JavaScript versioning and maintaining ECMAScript is known as Technical Committee 39, or TC39. It's made up of representatives from all the major browser vendors such as Apple, Google, Microsoft and Mozilla, as well as invited experts and delegates from other companies with an interest in the development of the Web. They have regular meetings to decide on how the language will develop.

When JavaScript was standardized by TC39 in 1997, the specification was known as ECMAScript version 1. Subsequent versions of ECMAScript were initially released on an annual basis, but ultimately became sporadic due to the lack of consensus and the unmanageably large feature set surrounding ECMAScript 4. This version was thus terminated and downsized into 3.1, but wasn't finalized under that moniker, instead eventually evolving into ECMAScript 5. This was released in December 2009, 10 years after ECMAScript 3, and introduced a JSON serialization API, Function.prototype.bind, and strict mode, amongst other capabilities. A maintenance release to clarify some of the ambiguity of the latest iteration, 5.1, was released two years later.

ECMAScript 2015 and the Resurgence of Yearly Releases

With the resolution of TC39's disagreement resulting from ECMAScript 4, Brendan Eich stressed the need for nearer-term, smaller releases. The first of these new specifications was *ES2015* (originally named ECMAScript 6, or ES6). This edition was a large but necessary foundation to support the future, annual JavaScript versioning. It includes many features that are well-loved by many developers today, such as:

- Classes
- Promises
- Arrow functions
- ES Modules
- Generators and Iterators

ES2015 was the first offering to follow the *TC39 process*, a proposal-based model for discussing and adopting elements.

The TC39 Process

There are five stages through which a proposal must pass before it can be accepted into an upcoming version of ECMAScript.

Stage 0: Strawman

This is a convenience step to permit the submission of ideas to the specification. Features can be suggested by anyone — namely, TC39 members and non-members who have registered as a contributor.

Stage 1: Proposal

The first stage at which a proposal is formalized. It's necessary that:

- any existing problems rectified by the solution are described
- an API outline is provided, alongside high-level implementation details, as well as polyfills and/or demos
- potential impediments are discussed upfront.

A *champion* must be selected to adopt and advance the proposal. This individual must be a TC39 member.

Stage 2: Draft

This is the milestone at which a feature is likely to be included in a future version of ECMAScript. Here, the proposal's syntax and semantics are detailed using the formal language described by the specification. An experimental implementation should be available at this point.

Stage 3: Candidate

Here, the majority of the proposal and the backing technology have been developed, but it requires further feedback from users and implementers (such as browser vendors). Once this is available and acted upon, the

outline and specification details are finalized and signed off by designated reviewers and the appointed editor. As a compliant implementation is required at this stage, only critical changes are henceforth embraced.

Stage 4: Finished

The proposal has been accepted and can be added to ECMAScript. It's thus inherent that:

- acceptance tests, which are part of the *Test262* suite and are crafted with JavaScript, have been written to prove the conformity and behavior of the feature
- at least two compliant implementations are available, and have shipped, all of which demonstrate robustness and developer usability
- a pull request has been submitted to the official ECMA-262 repo, which has been signed off by the specification editor.

The above repository's contribution document further details the use of GitHub issues and pull requests for managing additions to the language.

Moving Forward

Following the completion of ES2015 and the establishment of the TC39 process of JavaScript versioning and updating, subsequent releases have occurred each June, with the inclusion of proposals being timeboxed to one year. At the time of writing, there have been three new specifications.

ES2016

Also known as ES7, this was the first smaller, incremental version of ECMAScript. Aside from bug fixes, it added just two features.

Array.prototype.includes

This instance method simplifies searching for values in an `Array`:

```
// pre-ES2016:
const hasBob = names.indexOf('bob') > -1;

// ES2016:
const hasBob = names.includes('bob');
```

Exponent Operator

Prior to ES2016, one could perform exponentiation with `Math.pow(base, exponent)`. This version introduces an operator (**) that has its own precedence:

```
// pre-ES2016
Math.pow(5, 3); // => 125

// ES2016
5 ** 3; // => 125
```

ES2017

A slightly larger release, ES2017 (aka ES8) contains a handful of useful methods and syntactical constructs.

Asynchronous Functions

Promises have saved us from callback hell, but their API nonetheless demonstrates verbosity. Asynchronous functions abstract them with a syntax that closely resembles synchronous code:

```
// promises
const getProfile = name => {
  return fetch(`https://some-api/people/${name}`)
    .then(res => res.json())
    .then(({ profile }) => profile); // destructuring `profile`
from parsed object
};

// async/await
const getProfile = async name => {
  const res = await fetch(`https://some-api/people/${name}`);
  const { profile } = await res.json();
  return profile;
};
```

String Padding Methods

String.prototype.padStart(length, padder) and padEnd(length, padder) will respectively prepend and append `padder` (this is optional, defaulting to a space) to a string repeatedly until it reaches `length` characters:

```
'foo'.padStart(6);          // => '   foo';
'foo'.padEnd(6);            // => 'foo   ';
'foo'.padStart(10, 'bar');  // => 'barbarbfoo';
'foo'.padEnd(10, 'bar');    // => 'foobarbarb';
```

Other features include trailing commas, shared memory and atomics, and static `Object` methods (Object.entries(), Object.values(), and Object.getOwnPropertyDescriptors().)

If you'd like to read more about the complete feature set of ES2017, please see the next chapter.

ES2018

This latest iteration, at the time of writing, introduces a small set of powerful additions.

Asynchronous Iterators

While `Promise.all()` allows you to await the resolution of multiple promises, there are cases in which you may need to sequentially iterate over asynchronously-retrieved values. It's now possible to await async iterators along with arrays of promises:

```
(async () => {
  const personRequests = ['bob', 'sarah', 'laura'].map(
    n => fetch(`https://api/people/${n}`)
  );

  for await (const response of personRequests) {
    console.log(await response.json());
  }
})();
```

Object Spread and Rest Properties

Ostensibly, these two syntactical improvements are already popular amongst JavaScript developers thanks to the availability of compilers such as Babel. Object spread and rest properties are similar to array spread and rest properties, and permit the shallow copying and grouped destructuring of object properties:

```
const react = {
  name: 'React',
  vendor: 'Facebook',
  description: 'A JavaScript library for building user
interfaces',
  npm: true,
  cdn: true,
};

/* Use spread syntax inside an object literal to create
 * a shallow copy, while overriding certain properties.
```

```
 */
const vue = {
  ...react,
  vendor: 'Evan You',
  description: 'A JavaScript framework for building UIs',
};

/* Use rest within destructuring syntax to create a
 * label for accessing additional object properties.
 */
const { name, vendor, ...rest } = vue;
console.log(rest.description); // => 'A JavaScript framework
for building UIs'
```

Other accepted proposals are Promise.prototype.finally(), as well as enhancements to regular expressions and template literals.

If you'd like to read more about the complete feature set of ES2018, check out the final chapter of this book.

A Final Word

JavaScript has evolved greatly over a short space of time. While this is attributable to the ECMAScript standard and the brilliant work of TC39, it was initially an arduous journey due to the previous lack of stability and cohesion in JavaScript versioning and development.

Thanks to the relatively mature proposals process, the language can only improve in a pragmatic and manageable manner. It’s a great time to be a web developer!

Chapter 18: What's New in ES2017: Async Functions, Improved Objects, and More

by Craig Buckler

Let's take a look at the most important JavaScript updates that came with ES2017, and also briefly cover how this updating process actually takes place.

The Update Process

JavaScript (ECMAScript) is an ever-evolving standard implemented by many vendors across multiple platforms. ES6 (ECMAScript 2015) was a large release which took six years to finalize. A new annual release process was formulated to streamline the process and rapidly add new features.

The modestly named Technical Committee 39 (TC39) consists of parties including browser vendors who meet to push JavaScript proposals along a strict progression path: **Stage 0: strawman -** An initial submission of ideas for new or improved ECMAScript features.

Stage 1: proposal - A formal proposal document championed by at least one member of TC39, which includes API examples, language semantics, algorithms, potential obstacles, polyfills and demonstrations.

Stage 2: draft - An initial version of the feature specification. Two experimental implementations of the feature are required, although one can be in a transpiler such as Babel.

Stage 3: candidate - The proposal specification is reviewed and feedback is gathered from vendors.

Stage 4: finished - The proposal is ready for inclusion in ECMAScript. A feature should only be considered a standard once it reaches this stage. However, it can take longer to ship in browsers and runtimes such as Node.js.

If ES2015 was too large, ES2016 was purposely tiny to prove the standardization process. Two new features were added:

1. The array `.includes()` method which returns true or false when a value is contained in an array, and
2. The `a ** b` exponentiation operator, which is identical to `Math.pow(a, b)`.

What's New in ES2017

The feature set for ES2017 (or ES8 in old money) is considered to be the first proper amendment to the ECMAScript specification. It delivers the following goods …

Async functions

Unlike most languages, JavaScript is asynchronous by default. Commands which can take any amount of time do not halt execution. That includes operations such as requesting a URL, reading a file, or updating a database. A callback function must be passed, which executes when the result of that operation is known.

This can lead to callback hell when a series of nested asynchronous functions must be executed in order. For example:

```
function doSomething() {
  doSomething1((response1) => {
    doSomething2(response1, (response2) => {
      doSomething3(response2, (response3) => {
        // etc...
      };
    });
  });
}
```

ES2015 (ES6) introduced Promises, which provided a cleaner way to express the same functionality. Once your functions were Promisified, they could be executed using:

```
function doSomething() {
  doSomething1()
  .then(doSomething2)
  .then(doSomething3)
}
```

ES2017 Async functions expand on Promises to make asynchronous calls even clearer:

```
async function doSomething() {
  const
    response1 = await doSomething1(),
    response2 = await doSomething2(response1),
    response3 = await doSomething3(response2);
}
```

`await` effectively makes each call appear as though it's synchronous while not holding up JavaScript's single processing thread.

Async functions are supported in all modern browsers (not IE or Opera Mini) and Node.js 7.6+. They'll change the way you write JavaScript, and a whole article could be dedicated to callbacks, Promises and Async functions. Fortunately, we have one! Refer to Flow Control in Modern JavaScript.

Object.values()

`Object.values()` is a quick and more declarative way to extract an array of values from name–value pairs within an object. For example:

```
const myObject = {
  a: 1,
  b: 'Two',
  c: [3,3,3]
}

const values = Object.values(myObject);
// [ 1, 'Two', [3,3,3] ]
```

You need never write a `for … of` loop again! `Object.values` is natively supported in all modern browsers (not IE or Opera Mini) and Node.js 7.0+.

Object.entries()

`Object.entries()` returns an array from an object containing name–value pairs. Each value in the returned array is a sub-array containing the name (index 0) and value (index 1). For example: `const myObject = { a: 1, b: 'Two', c: [3,3,3] } const entries = Object.entries(myObject); /* [ [ 'a', 1 ], [ 'b', 'Two' ], [ 'c', [3,3,3] ] ] */`

This provides another way to iterate over the properties of an object. It can also be used to define a Map:

```
const map = new Map(Object.entries({
  a: 1,
  b: 2,
  c: 3
}));
```

`Object.values` is natively supported in most modern browsers (but not IE, Opera Mini and iOS Safari) and Node.js 7.0+.

Object.getOwnPropertyDescriptors()

The `Object.getOwnPropertyDescriptors()` method returns another object containing all property descriptors (`.value`, `.writable`, `.get`, `.set`, `.configurable`, `.enumerable`).

The properties are directly present on an object and not in the object's prototype chain. It's similar to Object.getOwnPropertyDescriptor(object, property) — except all properties are returned, rather than just one. For example:

```
const myObject = {
  prop1: 'hello',
  prop2: 'world'
};

const descriptors = Object.getOwnPropertyDescriptors(myObject);

console.log(descriptors.prop1.writable); // true
console.log(descriptors.prop2.value);    // 'world'
```

padStart() and padEnd() String Padding

String padding has been controversial in JavaScript. The popular left-pad library was pulled from npm after it attracted the attention of lawyers representing an instant messaging app with the same name. Unfortunately, it had been used as a dependency in thousands of projects and the internet broke. npm subsequently changed operating procedures and left-pad was un-unpublished.

Native string padding has been added to ES2017, so there's no need to use a third-party module. `.padStart()` and `.padEnd()` add characters to the start or end of a string respectively, until they reach a desired length. Both accept a minimum length and an optional `'fill'` string (space is the default) as parameters. Examples: `'abc'.padStart(5); // ' abc' 'abc'.padStart(5,'-'); // '--abc' 'abc'.padStart(10, '123'); // '1231231abc' 'abc'.padStart(1); // 'abc' 'abc'.padEnd(5); // 'abc ' 'abc'.padEnd(5,'-'); // 'abc--' 'abc'.padEnd(10, '123'); // 'abc1231231' 'abc'.padEnd(1); // 'abc'`

`.padStart()` and `.padEnd()` are supported in all modern browsers (not IE) and Node.js 8.0+.

Trailing Commas are Permitted

A small ES2017 update: trailing commas no longer raise a syntax error in object definitions, array declarations, function parameter lists, and so on:

```
// ES2017 is happy!
const a = [1, 2, 3,];

const b = {
  a: 1,
  b: 2,
  c: 3,
};

function c(one,two,three,) {};
```

Trailing commas are enabled in all browsers and Node.js. However, trailing commas in function parameters are only supported in Chrome 58+ and Firefox 52+ at the time of writing.

SharedArrayBuffer and Atomics

The SharedArrayBuffer object is used to represent a fixed-length raw binary data buffer that can be shared between web workers. The Atomics object provided a predicable way to read from and write to memory locations defined by `SharedArrayBuffer`.

While both objects were implemented in Chrome and Firefox, it was disabled in January 2018 in response to the Spectre vulnerability.

The full ECMAScript 2017 Language Specification is available at the ECMA International website. Are you hungry for more? The new features in ES2018 have been announced, and we'll discuss them next.

Chapter 19: What's New in ES2018

by Craig Buckler

In this chapter, I'll cover the new features of JavaScript introduced via ES2018 (ES9), with examples of what they're for and how to use them.

JavaScript (ECMAScript) is an ever-evolving standard implemented by many vendors across multiple platforms. ES6 (ECMAScript 2015) was a large release which took six years to finalize. A new annual release process has been formulated to streamline the process and add features quicker. ES9 (ES2018) is the latest iteration at the time of writing.

Technical Committee 39 (TC39) consists of parties including browser vendors who meet to push JavaScript proposals along a strict progression path: **Stage 0: strawman -** The initial submission of ideas.

Stage 1: proposal - A formal proposal document championed by at least once member of TC39 which includes API examples.

Stage 2: draft - An initial version of the feature specification with two experimental implementations.

Stage 3: candidate - The proposal specification is reviewed and feedback is gathered from vendors.

Stage 4: finished - The proposal is ready for inclusion in ECMAScript but may take longer to ship in browsers and Node.js.

ES2016

ES2016 proved the standardization process by adding just two small features:

1. The array includes() method, which returns true or false when a value is contained in an array, and
2. The `a ** b` exponentiation operator, which is identical to `Math.pow(a, b)`.

ES2017

ES2017 provided a larger range of new features:

- Async functions for a clearer Promise syntax
- `Object.values()` to extract an array of values from an object containing name–value pairs
- `Object.entries()`, which returns an array of sub-arrays containing the names and values in an object
- `Object.getOwnPropertyDescriptors()` to return an object defining property descriptors for own properties of another object (`.value`, `.writable`, `.get`, `.set`, `.configurable`, `.enumerable`)
- `padStart()` and `padEnd()`, both elements of string padding
- trailing commas on object definitions, array declarations and function parameter lists
- `SharedArrayBuffer` and `Atomics` for reading from and writing to shared memory locations (disabled in response to the Spectre vulnerability).

Refer to What's New in ES2017 for more information.

ES2018

ECMAScript 2018 (or ES9 if you prefer the old notation) is now available. The following features have reached stage 4, although working implementations will be patchy across browsers and runtimes at the time of writing.

Asynchronous Iteration

At some point in your async/await journey, you'll attempt to call an asynchronous function *inside* a synchronous loop. For example: `async function process(array) { for (let i of array) { await doSomething(i); } }`

It won't work. Neither will this:

```
async function process(array) {
  array.forEach(async i => {
    await doSomething(i);
  });
}
```

The loops themselves remain synchronous and will always complete before their inner asynchronous operations.

ES2018 introduces asynchronous iterators, which are just like regular iterators except the `next()` method returns a Promise. Therefore, the `await` keyword can be used with `for … of` loops to run asynchronous operations in series. For example: `async function process(array) { for await (let i of array) { doSomething(i); } }`

Promise.finally()

A Promise chain can either succeed and reach the final `.then()` or fail and trigger a `.catch()` block. In some cases, you want to run the same code regardless of the outcome — for example, to clean up, remove a dialog, close a database connection etc.

The `.finally()` prototype allows you to specify final logic in one place rather than duplicating it within the last `.then()` and `.catch()`:

```
function doSomething() {
  doSomething1()
  .then(doSomething2)
  .then(doSomething3)
  .catch(err => {
    console.log(err);
  })
  .finally(() => {
    // finish here!
  });
}
```

Rest/Spread Properties

ES2015 introduced the rest parameters and spread operators. The three-dot (. . .) notation applied to array operations only. Rest parameters convert the last arguments passed to a function into an array:

```
restParam(1, 2, 3, 4, 5);

function restParam(p1, p2, ...p3) {
  // p1 = 1
  // p2 = 2
  // p3 = [3, 4, 5]
}
```

The spread operator works in the opposite way, and turns an array into separate arguments which can be passed to a function. For example, `Math.max()` returns the highest value, given any number of arguments:

```
const values = [99, 100, -1, 48, 16];
console.log( Math.max(...values) ); // 100
```

ES2018 enables similar rest/spread functionality for object destructuring as well as arrays. A basic example:

```
const myObject = {
  a: 1,
  b: 2,
  c: 3
};

const { a, ...x } = myObject;
// a = 1
// x = { b: 2, c: 3 }
```

Or you can use it to pass values to a function:

```
restParam({
  a: 1,
  b: 2,
  c: 3
});
```

```
function restParam({ a, ...x }) {
  // a = 1
  // x = { b: 2, c: 3 }
}
```

Like arrays, you can only use a single rest parameter at the end of the declaration. In addition, it only works on the top level of each object and not sub-objects.

The spread operator can be used within other objects. For example:

```
const obj1 = { a: 1, b: 2, c: 3 };
const obj2 = { ...obj1, z: 26 };
// obj2 is { a: 1, b: 2, c: 3, z: 26 }
```

You could use the spread operator to clone objects (`obj2 = { ...obj1 };`), but be aware you only get shallow copies. If a property holds another object, the clone will refer to the same object.

Regular Expression Named Capture Groups

JavaScript regular expressions can return a match object — an array-like value containing matched strings. For example, to parse a date in YYYY-MM-DD format:

```
const
  reDate = /([0-9]{4})-([0-9]{2})-([0-9]{2})/,
  match  = reDate.exec('2018-04-30'),
  year   = match[1], // 2018
  month  = match[2], // 04
  day    = match[3]; // 30
```

It's difficult to read, and changing the regular expression is also likely to change the match object indexes.

ES2018 permits groups to be named using the notation `?<name>` immediately after the opening capture bracket `(`. For example:

```
const
  reDate = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]
{2})/,
  match  = reDate.exec('2018-04-30'),
  year   = match.groups.year,  // 2018
  month  = match.groups.month, // 04
  day    = match.groups.day;   // 30
```

Any named group that fails to match has its property set to `undefined`.

Named captures can also be used in `replace()` methods. For example, convert a date to US MM-DD-YYYY format:

```
const
  reDate = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]
{2})/,
  d      = '2018-04-30',
  usDate = d.replace(reDate, '$<month>-$<day>-$<year>');
```

Regular Expression lookbehind Assertions

JavaScript currently supports *lookahead* assertions inside a regular expression. This means a match must occur but nothing is captured, and the assertion isn't included in the overall matched string. For example, to capture the currency symbol from any price: `const reLookahead = /\D(?=\d+)/, match = reLookahead.exec('$123.89'); console.log( match[0] ); // $`

ES2018 introduces *lookbehind* assertions that work in the same way, but for preceding matches. We can therefore capture the price number and ignore the currency character:

```
const
  reLookbehind = /(?<=\D)\d+/,
  match        = reLookbehind.exec('$123.89');

console.log( match[0] ); // 123.89
```

This is a positive lookbehind assertion; a non-digit `\D` must exist. There's also a negative lookbehind assertion, which sets that a value must not exist. For example:

```
const
  reLookbehindNeg = /(?<!\D)\d+/,
  match           = reLookbehind.exec('$123.89');

console.log( match[0] ); // null
```

Regular Expression s (dotAll) Flag

A regular expression dot `.` matches any single character *except* carriage returns. The `s` flag changes this behavior so line terminators are permitted. For example:

```
/hello.world/s.test('hello\nworld'); // true
```

Regular Expression Unicode Property Escapes

Until now, it hasn’t been possible to access Unicode character properties natively in regular expressions. ES2018 adds Unicode property escapes — in the form `\p{...}` and `\P{...}` — in regular expressions that have the `u` (unicode) flag set. For example: `const reGreekSymbol = /\p{Script=Greek}/u; reGreekSymbol.test('π'); // true`

Template Literals Tweak

Finally, all syntactic restrictions related to escape sequences in template literals have been removed.

Previously, a `\u` started a unicode escape, an `\x` started a hex escape, and `\` followed by a digit started an octal escape. This made it impossible to create certain strings such as a Windows file path `C:\uuu\xxx\111`. For more details, refer to the MDN template literals documentation.