ECMAScript 6

Succinctly®

by Matthew Duffield

Syncfusion | Technology Resource Portal

ECMAScript 6 Succinctly

By
Matthew Duffield

Foreword by Daniel Jebaraj

Suite 200
Morrisville, NC 27560
USA

Important licensing information. Please read.

Technical Reviewer: James McCaffrey
Copy Editor: Darren West, content producer, Syncfusion, Inc.
Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Matt Duffield is an author, consultant, and trainer at Duffield Consulting, Inc. He has more than 20 years of development experience and has received multiple Microsoft MVP awards for Windows Development. He has worked considerably in JavaScript, .NET, Data Warehouse/Data Mining, and Natural Language Processing.

Matt is very active in the community and was the leader of the Charlotte ALT.NET User Group and founder/leader of the Charlotte Unity Game Developer Group. He recently moved back to his stomping grounds in Portland, OR. You can find him on Twitter as @mattduffield or visit his blog on WordPress.

Chapter 1 Introduction

Introducing ES6

ECMAScript 6 (ES6), also known as ECMAScript 2015, brings new functionality and features to the table that developers have been wanting for a long time. The wait is over, and you can now develop all of these features. The really good news is that you can also target browsers that don't even support ES6 yet using a transpiler. A compiler translates one language to another, such as C# to MSIL, while a transpiler converts one version of a language to another, such as ES6 to ES5.

Getting started

In order to follow along with the code samples in this book, you will need to have the following on your development machine:

- Sublime Text – we will be setting up a custom build command so that you can simply transpile and execute your scripts using the default build command. You can get Sublime Text from the following link: http://www.sublimetext.com.

- NodeJS – please ensure that you have Node.js installed on your machine. You can get Node.js from the following link: https://nodejs.org.

- Babel – we will be using Babel as our transpiler. Babel allows us to write our code using ES6 syntax and transpiles it down to ES5 syntax. If you go to their website, you can see your code get transpiled live. You can get BabelJS from the following link: https://babeljs.io. On the Babel website, you can also click on the "Try it out" link and test most of the code presented here.

It is possible to run most of the code examples in the browser using your favorite transpiler, but for the sake of consistency we will be using previously stated dependencies. It is our goal to also provide you with a quick and easy tool to help you get the most out of understanding the new features in ES6.

Note: Unless otherwise indicated, we are going to use Sublime Text along with Babel and Node.js.

The following are the steps to get your machine ready for using the code examples in this book:

1. Have the latest version of Sublime Text installed on your machine. You can use any editor you like to follow along, as we are simply going to be providing a sample command that will allow you to test the scripts easily.

2. Ensure that you have the latest version Node.js installed on your machine. At the time of writing, we are using v4.2.4.

3. Download and unzip the following Git repository into a folder of your choice: https://github.com/mattduffield/es6-succinctly.git.

4. Open a command shell, navigate to the directory containing the file package.json from the Git repository and enter the command **`npm install`**.

5. In Sublime, go to **`Tools|Build System|New Build System`**.

    a. You will be presented with a new tab. Save the tab and give it the name "babel-node.sublime-build."

    b. If you are using a Mac or Linux, you should be able to use the following code:

Code Listing 1

```
{
      "cmd": ["babel-node", "$file"],
      "path": "/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin"
}
```

    c. If you are using Windows, you should be able to use the following:

Code Listing 2

```
{
      "shell_cmd": "babel-node $file",
      "path": "$PATH"
}
```

    d. For either environment, you can use the command **`which node`** and **`which babel`** to get the paths. I am using Git Bash to perform these commands on Windows.

    e. Next, you need to let Sublime Text know which build system to use. In Sublime Text, ensure that **`babel-node`** is checked from the menu **`Tools|Build System`**. This will execute the new build script we created and allow you to transpile and execute the examples.

6. You should be ready to use Sublime to play with scripts. Take note that Sublime will try and build based on whatever tab is active. This may or may not be what you want if you are dealing with configuration files.

7. You can test this out if you create a file called "test.js." In this file add the following code:

Code Listing 3

```
let greeting = "hello world";
console.log(greeting);
```

Note: You may need to place a 'use strict' statement as the first line in file test.js in order to get the script to run.

8. Save the file and build. You should see something like the following in Sublime:

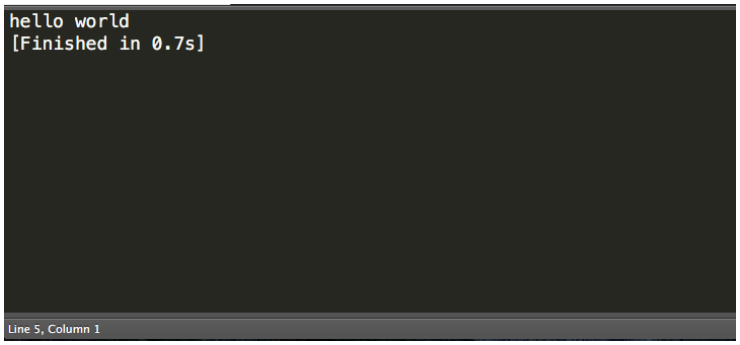

Figure 1: Sublime Text output

If you find that it is not recognizing the **`babel-node`** command, I would suggest installing the following globally so that path environment is updated correctly:

Code Listing 4

```
npm install –g babel
npm install –g babel-cli
```

If you don't wish to use Sublime Text as your editor and build system, you can simply use any text editor (such as Notepad), save your source code with a .js extension, navigate to the directory containing your script, and then execute from a command line by typing:

```
> babel-node myscript.js
```

Another great tool for testing ES6 capabilities is ScratchJS. It is an extension for Chrome and even allows you to select which transpiler to use. Again, you can use your own tool of choice, but when we get to Modules we want to load modules from the file system, and this makes it hard when using a browser testing tool.

Transpilers

The following is a short list of popular transpilers:

- Babel – This is by far the most popular transpiler for ES6. It comes with a 2015 preset that you can use that supports a majority of the 2015 specification. You can find more about Babel.js here: https://babeljs.io. One thing to note is that Babel 6 was recently released, yet the REPL online has not yet been updated.

- Traceur – This is another popular transpiler for ES6. It is not as complete in compliance to specification with ES6 as Babel, but it is still a good option. You can find more about Traceur here: https://github.com/google/traceur-compiler.

- TypeScript – This is probably the most popular transpiler for the Microsoft stack developer. TypeScript isn't strictly ES6, as it adds typing and interfaces into the mix. As far as ES6 compliance, it is behind both Babel and Traceur. However, it is a great tool, and if you like to get strongly typed compile time exceptions, this is the tool for you. You can find more about TypeScript here: http://www.typescriptlang.org/.

Browser support

Desktop browser support is coming along pretty well. You will be surprised to know that Microsoft Edge is right on the tail of Firefox as leading the way in standards compliance. Chrome is close behind but still has some ways to go.

As of writing this book, it is recommended to use a 'transpiler' such as Babel.

Code listings

All code listings in this book have been generated using Sublime Text 3. Most of the code examples and screenshots were created using Scratch JS in Chrome. Scratch JS is an extension for Chrome; it allows you to pick which transpiler you wish to use. For the remainder of this book, except where noted, we will be using Babel as the transpiler. Scratch JS also allows you to toggle the output of your ES6 code down to ES5 code by clicking Toggle Output. This can be beneficial if you are curious as to what the syntax of the code would look like in classic ES5 code. Because ES6 is still under development, you may run into some minor problems getting some of the examples to work, especially if you call babel-node from a command line. If this happens, you can go to the Babel website, and paste ES6 code into their interactive transpiler page. This will generate ES5 code. You can take this output, save it as a .js file, and run it using node. This approach usually solves most issues. Finally, you can also execute your ES6 code from the Scratch JS tab by clicking Run.

Chapter 2 Constants and Scoping

Constants

Most modern languages provide some method of defining a **`constant`**. This ensures that the value of the variable never changes and is a great optimization usage for values you know will never change in your applications.

Let's take a look at several examples of **`constants`**:

Code Listing 5

```
const PI = 3.141593;

console.log(PI);
```

The code produces the following output in Sublime using Ctrl+B for Windows or Cmd+B for Mac:

Code Listing 6

```
3.141593
```

Note: Unless indicated otherwise, we will use Sublime as the editor of choice for code snippets. In the Git repository, you will find the corresponding snippets in their own chapter.

Now, let's try and re-assign a value to the **`PI`** variable. We can do this by adding the following lines to the editor:

Code Listing 7

```
PI = 22;

console.log(PI);
```

Trying to run the previous example will produce a syntax error. The transpiler is smart enough to give us the reason why. Take a look at the following image:

```
1 const PI = 3.141593;
2 console.log(PI);
3
4 PI = 22;
  ! (4:0) SyntaxError: unknown: Line 4: "PI" is read-only
    |PI = 22;
    |^
5 console.log(PI);
```

Figure 2: Constant read-only

You can see in Figure 2 that since **PI** was declared as a **constant**, it is considered read-only. This is good and the error description is very clear.

> ***Note: The const keyword can only be used in block scope areas. We will look at block scopes in the next section.***

Scoping

Previously, JavaScript developers had to deal with hoisting when using the **var** keyword. At runtime, any variable that is declared will be hoisted up to the top of the executing context. Let's look at several examples of how the **var** keyword is handled at runtime:

Code Listing 8

```
var submit = function() {
  var x = "foo";

  if (x == "foo") {
    var y = "bar";
  }
  console.log(x);
  console.log(y);
}

submit();
```

The code produces the following output:

Code Listing 9

```
foo

bar
```

As you can see from the output, both **x** and **y** variables are available at the function level. This is known as hoisting regardless of the logical blocks.

With the **let** keyword, this is now different. Let's modify the previous code listing, but use the **let** keyword when declaring the **y** variable:

Code Listing 10

```
var submit = function() {
  var x = "foo";


  if (x == "foo") {
    let y = "bar";
  }
  console.log(x);
  console.log(y);
}


submit();
```

The code produces the following output in Chrome after clicking Run:

Code Listing 11

```
"ReferenceError: y is not defined
```

This output is actually pretty good in that it lets us know that we are trying to reference a variable that is out of scope.

The following example demonstrates how the **let** operator works inside **for-loops**:

Code Listing 12

```
let shoppingCart = [
```

```
  {id: 0, product: "DVD", price: 21.99},
  {id: 1, product: "CD", price: 11.99}
];

for (let i = 0; i < shoppingCart.length; i++) {
  let item = shoppingCart[i];
  console.log("Item: " + item.product + " - Price: " + item.price);
}
```

Let us look at the output that is produced:

Code Listing 13

```
Item: DVD - Price: 21.99
Item: CD - Price: 11.99
```

As you can see, we still get the expected behavior, but the item variable has a scope boundary of the **`for-loop`**.

When using a transpiler, this alone will help mitigate many scoping issues that were error-prone in ES5 with hoisting.

Enhanced regular expressions

Two new flags are introduced in ES6 for regular expressions:

- /y
- /u

Sticky flag

The new /y sticky flag allows you to create sticky searches; it anchors each match of a regular expression to the end of the previous match.

Let's look at the following example:

Code Listing 14

```
var text = 'First line\nsecond line';
var regex = /^(\S+) line\n?/y;

var match = regex.exec(text);
console.log(match[1]);        // logs 'First'
console.log(regex.lastIndex); // logs '11'

var match2 = regex.exec(text);
console.log(match2[1]);       // logs 'Second'
console.log(regex.lastIndex); // logs '22'

var match3 = regex.exec(text);
console.log(match3 === null); // logs 'true'
```

Note: This example will only work in Microsoft Edge or Firefox.

We start out with a simple string in the text variable. Next, we define our regular expression. As you can see, we are looking for matches that end with “line” and possibly a new line. We then execute the regular expression and grab the first match, which contains the value **`First`** and also logs the **`lastIndex`** position.

We run the **`exec`** method on the text variable again and it continues from the **`lastIndex`** position and finds the “Second” match. The **`lastIndex`** position is updated and logged.

Finally, we run the **`exec`** method a third time, and it returns **`null`,** indicating no more matches were found.

The main use case for this new flag is tokenizing, where you want each match to immediately follow its predecessor.

The following is an example of tokenizing a string:

Code Listing 15

```
const TOKEN = /\s*(\+|[0-9]+)\s*/y;
```

```
function tokenize(TOKEN_REGEX, str) {
  let result = [];
  let match;
  while (match = TOKEN_REGEX.exec(str)) {
    result.push(match[1]);
  }
  return result;
}

let result = tokenize(TOKEN, '3 + 4');
console.log(JSON.stringify(result));
```

Note: This example will only work in Microsoft Edge or Firefox.

Here is the output it produces:

Code Listing 16

```
["3","+","4"]
```

Although this is a contrived example, you can see where this could come in very handy when wanting to tokenize strings.

Unicode

The /u flag puts the regular expression into a special Unicode mode. This mode has two features:

- You can use Unicode code point escape sequences, such as **`\u{1F42A}`**, for specifying characters via code points. Normal Unicode escapes, such as **`\u03B1`**, only have a range of four hexadecimal digits.
- “Characters” in the regular expression pattern and the string are code points (not UTF-16 code units).

Let’s take a look at some examples:

Code Listing 17

```
console.log('\u{1F680}' === '\uD83D\uDE80');

let codepoint = "\u{1F680}";

console.log(codepoint);
```

Executing this produces the following result:

Code Listing 18

```
True

🚀
```

The first line of code demonstrates comparing a Unicode code point against two normal Unicode escapes. The second and third lines simply demonstrate how you can render the Unicode code point.

Note: Neither the /y nor /u flags are supported in all browsers. These samples were used in Firefox but could have also been run in Microsoft Edge.

Chapter 3 Enhanced Object Properties

Property shorthand

Creating object literals has never been easier. Let's first look at an ES5 syntax for creating an object literal:

Code Listing 19

```
let first = "john";
let last = "doe";
let obj = {first: first, last: last};
```

Now, let's look at the same thing using ES6 syntax:

Code Listing 20

```
let first = "john";
let last = "doe";
let obj = {first, last};
```

This makes for very clean and succinct code when dealing with object literals. Let's look at another example of what you can do:

Code Listing 21

```
function createDog(name, ability) {
  return { type: "animal", name, ability };
}
let a = createDog("wolf", "bark");
console.log(JSON.stringify(a));
```

Here is the output of the newly created object:

Code Listing 22

```
{"type":"animal","name":"wolf","ability":"bark"}
```

This shorthand won't change your code drastically, but it can make it a little bit cleaner and more succinct.

Computed property names

It is now possible to define computed property names in object property definitions. You can do this using variables as shown below:

Code Listing 23

```
var prop = "foo";
var o = {
  [prop]: "hey",
  ["b" + "ar"]: "there",
};
console.log(o.foo);
console.log(o.bar);
```

Here is the output:

Code Listing 24

```
hey
there
```

You could create your property names in the following way as well:

Code Listing 25

```
let i = 0;
let a = {
  ["foo" + ++i]: i,
  ["foo" + ++i]: i,
  ["foo" + ++i]: i
};

console.log(a.foo1);
console.log(a.foo2);
console.log(a.foo3);
```

Here is the output:

Code Listing 26

```
1

2

3
```

It is also possible to use functions for more advanced property names, as shown in the following example:

Code Listing 27

```
function foo() {
  return "firstname";
}
let obj = {
    foo: "bar",
    [ "prop_" + foo() ]: 42
}
console.log(JSON.stringify(obj));
```

Let's look at the output:

Code Listing 28

```
{"foo":"bar","prop_firstname":42}
```

Method properties

Method properties are another shortcut to what we could already do in ES5. Let's look at what they looked like in ES5:

Code Listing 29

```
let myMath = {
  add: function(a, b) { return a + b; },
  subtract: function(a, b) { return a - b; },
```

```
  multiply: function(a, b) { return a * b; },
  divide: function(a, b) { return a / b; }
}
```

Now let's see what the same code would look like in ES6:

Code Listing 30

```
let myMath = {
  add(a, b) { return a + b; },
  subtract(a, b) { return a - b; },
  multiply(a, b) { return a * b; },
  divide(a, b) { return a / b; }
}
```

As you can see when comparing the two, the new ES6 version is simply nicer and more succinct.

object.assign

The **`object.assign()`** method is used to copy property values from one or more source objects to a given target object. It will return the target object.

We can use it to create a simple clone of an object, as shown below:

Code Listing 31

```
var obj = { firstname: "matt", lastname: "duffield" };
var copy = Object.assign({}, obj);
console.log(copy);
```

When we run this, we get a new copy of the object:

Code Listing 32

```
Object {firstname: "matt", lastname: "duffield"}
```

Here we see that we provide an empty object literal as our target object and then provide another object literal as a given source. We get back the target with the values from the source as well.

Let's look at another example. This time, let's provide multiple sources as well as a target with values. We will see that the target will receive the values from the sources and return with the values merged. We will also see that the original target variable is also changed as well, **o1** being the target with **o2** and **o3** the two sources:

Code Listing 33

```
var o1 = { a: 1 };
var o2 = { b: 2 };
var o3 = { c: 3 };

var obj = Object.assign(o1, o2, o3);
console.log(obj);
console.log(o1);
```

When we run this, we see that both the new object **obj** and **o1** have the same values.

Code Listing 34

```
Object {a: 1, b: 2, c: 3}
Object {a: 1, b: 2, c: 3}
```

Let's do one more; we'll copy an object literal with accessors:

Code Listing 35

```
var obj = {
  foo: 1,
  get bar() {
    return 2;
  }
};

var copy = Object.assign({}, obj);
console.log(copy);
```

Running the preceding code will produce the following result:

Code Listing 36

```
{foo: 1, bar: 2}
```

What is happening here is that when **`Object.assign()`** performs the merge, it actually executes the getter **`bar()`** method and places the result into a property called **`bar`** instead.

If you really want to copy accessors, consider the following code:

Code Listing 37

```
function myAssign(target, ...sources) {
  sources.forEach(source => {
    Object.defineProperties(target,
Object.keys(source).reduce((descriptors, key) => {
      descriptors[key] = Object.getOwnPropertyDescriptor(source,
key);
      return descriptors;
    }, {}));
  });
  return target;
}


var copy = myAssign({}, obj);
console.log(copy);
```

Don't be alarmed if you see some new features in the preceding code that you don't completely understand; we will address them later in the book. When we run the preceding code, we will get the following result:

Code Listing 38

```
{foo: 1, bar: [Getter]}
```

This time we get what we are expecting, a copy of the properties and accessors.

Math

The global object **`Math`** has several new methods in ES6. The following will provide their names and a short description for each:

Math.sign(x)

Returns the sign of **x** as -1 or +1. Unless **x** is either **`NaN`** or zero; then **x** is returned.

Math.trunc(x)

Removes the decimal part of **x** and returns the integral part.

Math.cbrt(x)

Returns the cube root of **x**.

Math.expm1(x)

Returns **`Math.exp(x) – 1`**. The inverse of **`Math.log1p()`**.

Math.log1p(x)

Returns **`Math.log(1 + x)`**. The inverse of **`Math.expm1()`**.

Math.log2(x)

Computes the logarithm to base 2.

Math.log10(x)

Computes the logarithm to base 10.

Math.fround(x)

Rounds **x** to the nearest single-precision floating point value, which uses 32 bits.

Math.imul(x, y)

Multiplies the two 32 bit integers **x** and **y** and returns the lower 32 bits of the result. This is the only 32 bit basic math operation that can't be simulated by using a JavaScript operator and coercing the result back to 32 bits.

Math.clz32(x)

Counts the leading zero bits in the 32 bit integer **x**.

Math.sinh(x)

Computes the hyperbolic sine of **x**.

Math.cosh(x)

Computes the hyperbolic cosine of **x**.

Math.tanh(x)

Computes the hyperbolic tangent of **x**.

Math.asinh(x)

Computes the inverse hyperbolic sine of **x**.

Math.acosh(x)

Computes the inverse hyperbolic cosine of **x**.

Math.atanh(x)

Computes the inverse hyperbolic tangent of **x**.

Math.hypot(…values)

Computes the square root of the sum of squares of its argument.

These new methods provide added convenience when performing computations and complex operations.

Number

The **`Number`** object has several new methods and properties. The following will provide their names and a short description:

Number.EPSILON

Used for comparing floating point numbers with a tolerance for rounding errors. The property represents the difference between one and the smallest value greater than one that can be represented as a **`Number`**, which is approximately 2.2e-16.

Number.MAX_SAFE_INTEGER

Constant to represent the maximum safe integer in JavaScript ($2^{53} - 1$).

Number.MIN_SAFE_INTEGER

Constant to represent the minimum safe integer in JavaScript ($-(2^{53} - 1)$).

Number.isNaN()

This method determines whether the **passed** value is **NaN**. This is a more robust version of the original global **isNaN()**.

Number.isFinite()

This method determines whether the **passed** value is a finite number.

Number.isInteger()

This method determines whether the **passed** value is an integer.

Number.isSafeInteger()

This method determines whether the provided value is a number that is a safe integer.

Number.parseFloat()

This method parses a string argument and returns a floating point number. This method behaves identically to the global function **parseFloat()**, but its purpose is modularization of globals.

Number.parseInt()

This method parses a string argument and returns an integer of the specified radix or base.

String

The string object has several new methods. The following will provide the names and a short description along with some examples:

String.fromCodePoint()

This method returns a string created by using the specified sequence of code points. This is a static method.

Code Listing 39

```
String.fromCodePoint(42);       // *
String.fromCodePoint(65, 90);   // AZ
```

String.codePointAt()

This method returns a non-negative integer that is the UTF-16 encoded code point value.

Code Listing 40

```
'ABC'.codePointAt(1);          // 66
'\uD800\uDC00'.codePointAt(0); // 65536
```

String.startsWith()

This method determines whether a string begins with the characters of another string, returning true or false as appropriate.

Code Listing 41

```
var str = 'If you dream it, you can do it.';
console.log(str.startsWith('If you'));         // true
console.log(str.startsWith('you can do'));     // false
console.log(str.startsWith('you can do', 17)); // true
```

String.endsWith()

This method determines whether a string ends with the characters of another string, returning true or false as appropriate. The last example allows you to specify the length of the string, thus clamping the actual string to the new value.

Code Listing 42

```
var str = 'If you can dream it, you can do it.';
console.log(str.endsWith('do it.')); // true
console.log(str.endsWith('you can'));     // false
console.log(str.endsWith('you can', 28)); // true
```

String.includes()

This method determines whether one string may be found within another string, returning true or false as appropriate.

Code Listing 43

```
var str = 'If you can dream it, you can do it.';
console.log(str.includes('If you can'));       // true
console.log(str.includes('it.'));    // true
console.log(str.includes('nonexistent')); // false
```

```
console.log(str.includes('If you can', 1));    // false
console.log(str.includes('IF YOU'));       // false
```

String.normalize()

This method returns the Unicode Normalization Form of a given string. If the value isn't a string, it will be converted to one first.

Code Listing 44

```
// U+1E9B: Latin small letter long s with dot above
// U+0323: Combining dot below
var str = '\u1E9B\u0323';

// Canonically-composed form (NFC)
// U+1E9B: Latin small letter long s with dot above
// U+0323: Combining dot below
str.normalize('NFC'); // '\u1E9B\u0323'
str.normalize();      // same as above

// Canonically-decomposed form (NFD)
// U+017F: Latin small letter long s
// U+0323: Combining dot below
// U+0307: Combining dot above
str.normalize('NFD'); // '\u017F\u0323\u0307'

// Compatibly-composed (NFKC)
// U+1E69: Latin small letter s with dot below and dot above
str.normalize('NFKC'); // '\u1E69'

// Compatibly-decomposed (NFKD)
```

```
// U+0073: Latin small letter s
// U+0323: Combining dot below
// U+0307: Combining dot above
str.normalize('NFKD'); // '\u0073\u0323\u0307'
```

String.repeat()

This method constructs and returns a new string that contains the specified number of copies of the string on which it was called, concatenated together.

Code Listing 45

```
'foo'.repeat(-1);   // RangeError
'foo'.repeat(0);    // ''
'foo'.repeat(1);    // 'foo'
'foo'.repeat(2);    // 'foofoo'
'foo'.repeat(2.5);  // 'foofoo' (count will be converted to integer)
'foo'.repeat(1/0);  // RangeError
```

[@@iterator]()

This method returns a new **`Iterator`** object that iterates over the code points of a **`String`** value, returning each code point as a **`String`** value.

Code Listing 46

```
var string = 'A\uD835\uDC68';
var strIter = string[Symbol.iterator]();
console.log(strIter.next().value); // "A"
console.log(strIter.next().value); // "\uD835\uDC68"
```

Array

The **`Array`** object has several new methods. The following will provide the name and a short description and example of each:

Array.of()

This method creates a new **`Array`** instance with a variable number of arguments, regardless of the number or type of the arguments.

Code Listing 47

```
Array.of(1);             // [1]
Array.of(1, 2, 3);       // [1, 2, 3]
Array.of("a", 7, 12.5);  // ["a", 7, 12.5]
Array.of(undefined);     // [undefined]
```

Array.copyWithin(target, start[, end = this.length])

This method copies the sequence of array elements within the array to the position starting at the **`target`**. The copy is taken from the index positions of the second and third arguments, **`start`** and **`end`**. The **`end`** argument is optional and defaults to the length of the array.

Code Listing 48

```
[1, 2, 3, 4, 5].copyWithin(0, 3);      // [4, 5, 3, 4, 5]
[1, 2, 3, 4, 5].copyWithin(0, 3, 4);   // [4, 2, 3, 4, 5]
[1, 2, 3, 4, 5].copyWithin(0, -2, -1); // [4, 2, 3, 4, 5]
```

Array.entries()

This method returns a new **`Array Iterator`** object that contains the key/value pairs for each index in the array.

Code Listing 49

```
var arr = ['a', 'b', 'c'];
var eArr = arr.entries();
console.log(eArr.next().value); // [0, 'a']
console.log(eArr.next().value); // [1, 'b']
console.log(eArr.next().value); // [2, 'c']
```

Array.fill(value[, start = 0[, end = this.length]])

This method fills all the elements of an array from a start index to an end index with a static value.

Code Listing 50

```
[1, 2, 3].fill(4);               // [4, 4, 4]
[1, 2, 3].fill(4, 1);            // [1, 4, 4]
[1, 2, 3].fill(4, 1, 2);         // [1, 4, 3]
[1, 2, 3].fill(4, 1, 1);         // [1, 2, 3]
[1, 2, 3].fill(4, -3, -2);       // [4, 2, 3]
[1, 2, 3].fill(4, NaN, NaN);     // [1, 2, 3]
Array(3).fill(4);                // [4, 4, 4]
[].fill.call({ length: 3 }, 4);  // {0: 4, 1: 4, 2: 4, length: 3}
```

Array.find(callback[, thisArg])

This method returns a value in the array if an element in the array satisfies the provided testing function. Otherwise, **undefined** is returned.

Code Listing 51

```
function isPrime(value, index, array) {
  for (var i = 2; i < value; i++) {
    if (value % i === 0) {
      return false;
    }
  }
  return value > 1;
}
console.log([4, 6, 8, 12].find(isPrime)); // undefined, not found
console.log([4, 5, 8, 12].find(isPrime)); // 5
```

Array.findIndex(callback[, thisArg])

This method returns an index in the array if an element in the array satisfies the provided testing function. Otherwise, **-1** is returned.

Code Listing 52

```
function isPrime(value, index, array) {
  for (var i = 2; i < value; i++) {
    if (value % i === 0) {
      return false;
    }
  }
  return value > 1;
}
console.log([4, 6, 8, 12].findIndex(isPrime)); // -1, not found
console.log([4, 6, 7, 12].findIndex(isPrime)); // 2
```

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

This method creates a new **Array** instance from an array-like or iterable object. This method is not new but has some cool benefits.

Code Listing 53

```
let nodes = document.querySelectorAll(".business");
nodes.forEach((item, index, arr) => {
  console.log(item.name);
});
```

Note: This example will only work in a browser.

Array.includes(searchElement[, fromIndex])

This method determines whether an array includes a certain element, returning **true** or **false** as appropriate.

Code Listing 54

```
[1, 2, 3].includes(2);     // true
[1, 2, 3].includes(4);     // false
[1, 2, 3].includes(3, 3);  // false
```

```
[1, 2, 3].includes(3, -1); // true
[1, 2, NaN].includes(NaN); // true
```

Array.keys()

This method returns a new **`Array Iterator`** that contains the keys for each index in the array.

Code Listing 55

```
var arr = ["a", "b", "c"];
var iterator = arr.keys();
console.log(iterator.next()); // { value: 0, done: false }
console.log(iterator.next()); // { value: 1, done: false }
console.log(iterator.next()); // { value: 2, done: false }
console.log(iterator.next()); // { value: undefined, done: true }
```

Array.values()

This method returns a new **`Array Iterator`** object that contains the values for each index in the array.

Code Listing 56

```
var arr = ['w', 'y', 'k', 'o', 'p'];
var eArr = arr.values();
// your browser must support for..of loop
// and let-scoped variables in for loops
for (let letter of eArr) {
  console.log(letter);
}
```

Array[@@iterator]()

The initial value of the **`@@iterator`** property is the same function object as the initial value of the **`values()`** property.

Code Listing 57

```
var arr = ['w', 'y', 'k', 'o', 'p'];
```

```
var eArr = arr[Symbol.iterator]();

console.log(eArr.next().value); // w

console.log(eArr.next().value); // y

console.log(eArr.next().value); // k

console.log(eArr.next().value); // o

console.log(eArr.next().value); // p
```

Typed arrays

Typed arrays are useful in handling binary data. They describe an array-like view of an underlying binary data buffer. Be aware that there is no global property named **`TypedArray`**, and neither is there a directly visible constructor. Instead, there are a number of different global properties whose values are typed array constructors for specific element types, as listed in Table 1. When dealing with typed arrays, you will deal with two main classes: **`ArrayBuffer`** and **`DataView`**. We will go into more detail on each of them later. You may be asking, "Why would I ever want to use typed arrays?" They are JavaScript's answer for dealing with raw binary data. Typical examples of this could be streaming audio or video data.

The following is a list of the typed array constructors available:

Table 1: Typed Array Objects

Type	Size in bytes	Description	Web IDL type	C type
Int8Array	1	8-bit two's complement signed integer	Byte	int8_t
Uint8Array	1	8-bit unsigned integer	octet	uint8_t
Uint8ClampedArray	1	8-bit unsigned integer (clamped)	octet	uint8_t
Int16Array	2	16-bit two's complement signed integer	short	int16_t
Uint16Array	2	16-bit unsigned integer	unsigned short	uint16_t

Type	Size in bytes	Description	Web IDL type	C type
Int32Array	4	32-bit two's complement signed integer	long	int32_t
Uint32Array	4	32-bit unsigned integer	unsigned long	uint32_t
Float32Array	4	32-bit IEEE floating point number	unrestricted float	float
Float64Array	8	64-bit IEEE floating point number	unrestricted double	double

Let's look at an example below:

Code Listing 58

```
var buffer = new ArrayBuffer(8);
console.log(buffer.byteLength);
```

We get the following output from the preceding code sample:

Code Listing 59

```
8
```

Let's take this a step further and create a **DataView** in order to manipulate the data within our buffer:

Code Listing 60

```
var buffer = new ArrayBuffer(8);

var view = new DataView(buffer);
view.setInt8(0, 3);

console.log(view.getInt8(0));
```

Here, we are creating the buffer like before. Next, we create a **DataView** that takes in the buffer as an argument. We now have the ability to manipulate the buffer by setting specific typed values. We used the **setInt8** function to put the value "3" into the 0-byte offset position of the buffer. If we run this code, we get the following output as expected:

Code Listing 61

```
3
```

Let's try another example:

Code Listing 62

```
var buffer = new ArrayBuffer(8);

var view = new DataView(buffer);
view.setInt32(0, 3);

console.log(view.getInt8(0));
console.log(view.getInt8(1));
console.log(view.getInt8(2));
console.log(view.getInt8(3));

console.log(view.getInt32(0));
```

In this example, we set the value into our **`ArrayBuffer`** using **`setInt32`**. Since **`setInt32`** takes 4 bytes (32 bits = 4 bytes), instead of **`setInt8`**, which takes 1 byte (8 bits = 1 byte), our **`getInt8`** calls don't find the number until our offset is 4 bytes into the buffer.

If we run this code, we get the following output as expected:

Code Listing 63

```
0

0

0

3

3
```

Rather than creating a generic **`DataView`**, there are typed array view classes with more specific names that we can create and access like regular arrays. These are the arrays listed in Table 1.

Consider the following example:

Code Listing 64

```
var buffer = new ArrayBuffer(8);

// 32-bit View
var bigView = new Int32Array(buffer);
```

```
bigView[0] = 98;
bigView[1] = 128;

for(let value of bigView)
{
  console.log(value);
  // 98
  // 128
}

// 16-bit View
var mediumView = new Int16Array(buffer);

for(let value of mediumView)
{
  console.log(value);
  // 98
  // 0
  // 128
  // 0
}

// 8-bit View
var smallView = new Int8Array(buffer);

for(let value of smallView)
{
  console.log(value);
  // 98
  // 0
  // 0
  // 0
  // -128
  // 0
  // 0
  // 0
}

// 8-bit Unsigned View
var unsignedView = new Uint8Array(buffer);

for(let value of unsignedView)
{
  console.log(value);
  // 98
```

```
  // 0
  // 0
  // 0
  // 128
  // 0
  // 0
  // 0
}
```

As you can see from the preceding code, we are able to treat these typed arrays just like regular arrays. The biggest difference here is that these are truly “typed” arrays.

The following is a small representation of browser supported APIs for Typed Arrays:

- File API
- XML HttpRequest
- Fetch API
- Canvas
- WebSockets

Let’s take a look at how we would use each one of these APIs in code.

File API

The **`File`** API lets you access local files. The following code demonstrates how to get the bytes of a submitted local file in an **`ArrayBuffer`**:

Code Listing 65

```
let fileInput = document.getElementById('fileInput');
let file = fileInput.files[0];
let reader = new FileReader();
reader.readAsArrayBuffer(file);
reader.onload = function () {
  let arrayBuffer = reader.result;
  // process the buffer...
};
```

Note: This example will only work in a browser.

XMLHttpRequest API

In the newer versions of the **`XMLHttpRequest`** API, you can have the results delivered as an **`ArrayBuffer`**:

Code Listing 66

```
let xhr = new XMLHttpRequest();
xhr.open('GET', someUrl);
xhr.responseType = 'arraybuffer';

xhr.onload = function () {
  let arrayBuffer = xhr.response;
  // process the buffer...
};

xhr.send();
```

Note: This example will only work in a browser.

Fetch API

Like the **`XMLHttpRequest`** API, the **`Fetch`** API lets you request resources. However, most will consider it to be an improvement on **`XMLHttpRequest`** in terms of API design. It also takes advantage of Promises, which we will look at later on in the book. The following example demonstrates how to download the content pointed to by the URL as an **`ArrayBuffer`**:

Code Listing 67

```
fetch(url)
  .then(request => request.arrayBuffer())
  .then(arrayBuffer => /* process buffer */);
```

Note: This example will only work in a browser.

Canvas

The 2-D context of the **`canvas`** lets you retrieve the bitmap data as an instance of **`Uint8ClampedArray`**:

Code Listing 68

```
let canvas = document.getElementById('my_canvas');
let context = canvas.getContext('2d');
let imageData = context.getImageData(0, 0, canvas.width,
canvas.height);
let uint8ClampedArray = imageData.data;
```

Note: This example will only work in a browser.

WebSockets

`WebSockets` let you send and receive binary data via **`ArrayBuffers`**:

Code Listing 69

```
let socket = new WebSocket('ws://127.0.0.1:8081');
socket.binaryType = 'arraybuffer';

// Wait until socket is open
socket.addEventListener('open', function (event) {
  // Send binary data
  let typedArray = new Uint8Array(4);
  socket.send(typedArray.buffer);
});

// Receive binary data
socket.addEventListener('message', function (event) {
  let arrayBuffer = event.data;
  // process the buffer...
});
```

Note: This example will only work in a browser.

Chapter 4 Arrow Functions

Arrows are a function shorthand using the **=>** syntax. They are syntactically similar to the fat arrow syntax in C#, Java, and CoffeeScript. They support both expression bodies and statement block bodies that return the value of the expression. Unlike functions, arrows share the same lexical **this** as their surrounding code.

The following is what a classic function would look like maintaining the same lexical **this**:

Code Listing 70

```
var square = (function(num) {
  return num * num;
}).bind(this);
```

Now, we can simply write it as follows:

Code Listing 71

```
let square = (num) => {
  return num * num;
};
```

Expression bodies

Expression bodies are a single line expression with the **=>** token and an implied return value.

Code Listing 72

```
let evens = [2, 4, 6, 8, 10];
let odds = evens.map(v => v + 1);
let nums = evens.map((v, i) => v + i);

console.log(odds);
console.log(nums);
```

Running the preceding code produces the following result:

Code Listing 73

```
[3, 5, 7, 9, 11]

[2, 5, 8, 11, 14]
```

As you can see from the example, this new syntax is very clean and removes needless boilerplate code.

Statement bodies

Statement bodies are multiline statements that allow for more complex logic. You also have the option to return a value from the statement or not.

Code Listing 74

```
let fives = [];

let nums = [1, 2, 5, 15, 25, 32];

nums.forEach(v => {

  if (v % 5 === 0)

    fives.push(v);

});


console.log(fives);
```

Running the preceding code produces the following result:

Code Listing 75

```
[5, 15, 25]
```

Lexical (this)

With this new fat arrow syntax, we can safely access the lexical **`this`** without worrying about it changing on us.

Let’s look at an example:

Code Listing 76

```
let matt = {
  name: "Matt",
  friends: ["Mark", "Lyle", "Rian"],
  printFriends() {
    this.friends.forEach(f =>
      console.log(this.name + " knows " + f));
  }
}
matt.printFriends();
```

Running the preceding code produces the following result:

Code Listing 77

```
Matt knows Mark
Matt knows Lyle
Matt knows Rian
```

Previously, we would have had to create a separate **_this** closure variable to enable accessing the correct **this**. That actually is exactly what happens for us automatically when our code gets transpiled. Take a look at the same code snippet after it has been transpiled to ES5:

Code Listing 78

```
"use strict";

var matt = {
  name: "Matt",
  friends: ["Mark", "Lyle", "Rian"],
  printFriends: function printFriends() {
    var _this = this;

    this.friends.forEach(function (f) {
```

```
      return console.log(_this.name + " knows " + f);
    });
  }
};
matt.printFriends();
```

As you can see, a new variable **`_this`** was created for us within the **`forEach`** function.

Chapter 5 Extended Parameter Handling

Another nice set of features introduced in ES6 is the extended parameter handling. These provide a very nice set of idioms that bring us very close to languages like C# or Java.

Default parameter values and optional parameters

Default parameters allow your functions to have optional arguments without needing to check **`arguments.length`** or check for **`undefined`**.

Let's take a look at what that might look like:

Code Listing 79

```
let greet = (msg = 'hello', name = 'world') => {
  console.log(msg,name);
}
greet();
greet('hey');
```

Looking at the preceding code, you can see that we are using the new fat arrow (**`=>`**) syntax as well as the new **`let`** keyword. Neither of those are necessary for this example, but I added it to give you more exposure to the new syntax.

Running the preceding code produces the following result:

Code Listing 80

```
hello world
hey world
```

Rest parameter

Handling a function with a variable number of arguments is always tricky in JavaScript. Rest parameters (…), indicated by three consecutive dot characters, allow your functions to have a variable number of arguments without using the **`arguments`** object. The **`rest`** parameter is an instance of **`Array,`** so all array methods work.

Code Listing 81

```
function f(x, ...y) {

  console.log(y);

  // y is an Array

  return x * y.length;

}

console.log(f(3, 'hello', true) === 6);
```

Running the preceding code produces the following result:

Code Listing 82

```
["hello", true]

true
```

You can have as many regular arguments as you'd like, but the **`rest`** parameter must always be the last. This may be one of the new features that seems a little foreign and weird, but over time you will find that this is a very nice feature and will come in very handy.

Spread operator

The **`spread`** operator is like the reverse of rest parameters. It allows you to expand an array into multiple formal parameters.

Code Listing 83

```
function add(a, b) {

  return a + b;

}

let nums = [5, 4];


console.log(add(...nums));
```

Running the preceding code produces the following result:

Code Listing 84

```
9
```

Again, this may seem strange, but once you start using both of these you will find them useful.

Let's change this slightly to have both a regular parameter and a **spread** operator in the same call:

Code Listing 85

```
function add(a, b) {
  return a + b;
}
let nums = [4];
console.log(add(5, ...nums));
```

As with the previous example, we get the same result:

Code Listing 86

```
9
```

Let's look at one more example when creating array literals:

Code Listing 87

```
let a = [2, 3, 4];
let b = [1, ...a, 5];
console.log(b);
```

Running the preceding code produces the following output:

Code Listing 88

```
[1, 2, 3, 4, 5]
```

As you can see, you can get very creative with the usage of the spread operator!

Chapter 6 Template Literals

Template literals are indicated by enclosing strings in backtick characters and provide syntactic sugar for constructing single and multi-line strings. This is similar to string interpolation features in Perl, Python, and more. Optionally, a tag can be added to allow the string construction to be customized, avoiding injection attacks or constructing higher level data structures from string contents.

The following are some examples of template literals:

Code Listing 89

```
`In JavaScript '\n' is a line-feed.`

`Now I can do multi-lines

 with template literals.`
```

String interpolation

With string interpolation, we can now compose very powerful strings in a clean, succinct manner. Consider the following example:

Code Listing 90

```
var customer = { name: "Matt" };

var product = { name: "Halo 5: Guardians" };

let gift = { timelimit: '4 hours', amount: 50.00 };

let message = `Dear ${customer.name},\n

Thank you for making a recent purchase of '${product.name}' on
Amazon.com.

We would love to get your feedback on your experience.

If you respond in the next ${gift.timelimit}, we will give you a gift

certificate of $${gift.amount} dollars!


We look forward to hearing from you!
```

```
Best Regards,


Amazon Customer Relations`;
console.log(message);
```

Running the preceding code produces the following output:

Code Listing 91

```
Dear Matt,


Thank you for making a recent purchase of 'Halo 5: Guardians' on
Amazon.com.
We would love to get your feedback on your experience.
If you respond in the next 4 hours, we will give you a gift
certificate of $50 dollars!


We look forward to hearing from you!


Best Regards,


Amazon Customer Relations
```

Now that is pretty nice! I can see a lot of cool string interpolation that no longer is constrained to the old string concatenation approach.

Let's see what else we can do. Consider the following example:

Code Listing 92

```
let getTotal = (qty, amount) => {
  return qty * amount;
};
let message = `Shopping cart total: $${getTotal(2, 2.99)}`;
```

```
console.log(message);
```

Running the preceding code produces the following output:

Code Listing 93

```
Shopping cart total: $5.98
```

As you can see, we can call functions inside our string interpolation.

You can even do operations inside the string interpolation expression as shown below:

Code Listing 94

```
let message = `Shopping cart total: $${2 * 2.99}`;


console.log(message);
```

Running the preceding code produces the following output:

Code Listing 95

```
Shopping cart total: $5.98
```

Custom interpolation

Perhaps you would like to construct a dynamic URL? This can be accomplished quite easily. Consider the following example:

Code Listing 96

```
let credentials = "private-user=admin&private-pw=p@$$w0rd";
let a = "one";
let b = "two";
let url = `http://myapp.com/login?a=${a}&b=${b}
   Content-Type: application/json
   X-Credentials: ${credentials}
`;
console.log(url);
```

```
let post = `POST ${url}`;
```

As you can see, we can use this new feature in quite a lot of different scenarios.

Dynamic template literals

There may come a time that you wish you could create a template literal dynamically, perhaps based on a database call or some other process. This requires more work, but you can support dynamic template literals with a little trickery. Consider the following example:

Code Listing 97

```
let processMarkup = (markup, data) => {
  let fields = markup.match(/{(.+?)}/g);
  let args = [];
  let params = [];
  fields.forEach((field,index,list) => {
    field = field.replace(/{/g, '');
    field = field.replace(/}/g, '');
    args.push(data[field]);
    params.push(field);
  });
  let template = markup.replace(/{/g, '${');
  let fn = assemble(template, params);
  let render = fn.apply(null, args);
  return render;
}
let assemble = (template, params) => {
  return new Function(params, "return `" + template +"`;");
}
let markup = `Hello {fname} {lname}, how are you?`;
let data = { fname: "Matthew", lname: "Duffield" };
```

```
let template = processMarkup(markup, data);

console.log(template);
```

Here is the output from running the preceding code:

Code Listing 98

```
Hello Matthew Duffield, how are you?
```

This may not be the most recommended thing to do, but it does show you that you can create dynamic templates. Note that we could not use the same symbols that are used in the string interpolation as that would have been resolved immediately. This is where the **`Function()`** constructor comes into play and allows us to "assemble" our template with the new parameters. Again, this is a naïve approach for making dynamic templates, but it does provide a lot of power if you have the need to generate templates on the fly.

As you can see, we can use this new feature in many different ways.

Chapter 7 Destructuring Assignment

The destructuring assignment syntax is a JavaScript expression that makes it possible to extract data from arrays or objects using a syntax that mirrors the construction of array and object literals.

Array matching

Array matching comes in very handy when you simply want to pull the values out of the array and use them as stand-alone variables. Let's look at the following example:

Code Listing 99

```
let [a, , b] = [1,2,3];

console.log("a:", a, "b:", b);
```

We now have two variables, **a** and **b**, that hold the values 1 and 3, respectively. Notice also that the array matching needs to be of the same length or structure as the original array. Here is the output from the preceding code:

Code Listing 100

```
a: 1 b: 3
```

Object matching, shorthand notation

With what we know about array matching, let's try our hand at object matching. Consider the following example:

Code Listing 101

```
var {foo, bar} = {foo: 'lorem', bar: 'ipsum', choo: 'uhoh'};

console.log("foo:", foo, "bar:", bar);
```

As you can see, we are grabbing the properties **foo** and **bar** from the object literal and creating new variables from them. This is very nice and keeps things clean. It is also nice that you can pick whichever property you want and ignore the ones you don't care about. Here is the output from the preceding code:

Code Listing 102

```
foo: lorem bar: ipsum
```

Object matching, deep matching

Let's go a bit further and see how we can apply this to more complex objects. Consider the following example:

Code Listing 103

```
let cust = {
  name: "Acme Corp.",
  address: {
    street: "1001 Oak Drive",
    city: "Summerville",
    state: "OR",
    zip: "97123"
  }
};
let {address: {city: city}, address: {state: state}} = cust;
console.log("City:", city, "\nState:", state);
```

This is a little more work to pull out the exact properties that you want, but it can come in very handy when you know exactly what you want to pull out into a separate variable. Here is the output from the preceding code:

Code Listing 104

```
City: Summerville
State: OR
```

Object matching, parameter context

With our understanding of array and object matching, let's apply what we've learned toward function parameters. Consider the following example:

Code Listing 105

```
function f ([ name, val ]) {
  console.log(name, val);
}
function g ({ name: n, val: v }) {
  console.log(n, v);
}
function h ({ name, val }) {
  console.log(name, val);
}
f([ "bar", 42 ]);
g({ name: "foo", val:  7 });
h({ name: "bar", val: 42 });
```

As you can see from the preceding functions, we are using the same pattern but this time for parameters for a given function. The first function expects an array as input with a length of two. It extracts those two values as individual parameter variables. The next function defines an object literal with the values for the given properties to be extracted when called. Finally, the last function is the same as the previous function but simply uses the shorthand notation. As you can see, it is possible to make some pretty interesting assignments with very little code now.

Here is the output from the three functions:

Code Listing 106

```
bar 42
foo 7
bar 42
```

Fail-soft destructuring

Fail-soft destructuring allows us to continue to do the same things that we have been doing up until now but also provides optional default values. Consider the following example:

Code Listing 107

```
let list = [ 7, 42 ];

let [a = 1, b = 2, c = 3, d] = list;

console.log("a:", a, "\nb:", b, "\nc:", c, "\nd:", d);
```

You can see that we can now optionally provide default values when trying to pull the values from the list. Let's look at the output from the preceding code:

Code Listing 108

```
a: 7

b: 42

c: 3

d: undefined
```

We can observe that if the value was available from the list, it was provided; otherwise, we saw our default value or **`undefined`**.

Chapter 8 Modules

Modules provide support for exporting and importing values without polluting the global namespace. We have had this capability in JavaScript through third-party libraries like AMD and CommonJS, but ECMAScript 6 now has this ability as well.

Note: This chapter deals with actual files to demonstrate export and import features of ECMAScript 6. All "file" references are to a file with the same name as the snippet title in the following format: code-listing-xxx.js.

Export

The best way of understanding export is by looking at it in action. Consider the following file:

Code Listing 109

```
export function sum(x, y) {
  return x + y;
}
export var pi = 3.141593;
```

We are exporting a function named **sum** as well as a variable, **pi**.

Let's move onto the next section and see how we can import.

Import

Now that we have created our own module, let's take advantage and use it in our own code. Consider the following file:

Code Listing 110

```
import {sum, pi} from './code-listing-109';


console.log('2 pi = ' + sum(pi, pi));
```

In this example, we are importing named exports from the module.

Code Listing 111

```
2 pi = 6.283186
```

As you can see, we were able to access both the function and the variable that we exported.

We can provide aliases for our named exports as well as our imports. Let's look at another example:

Code Listing 112

```
function sum(x, y) {
  return x + y;
}
var pi = 3.141593;

export { sum as add, pi}
```

Here, we are exporting both **sum** and **pi**, but this time we have chosen to alias **sum** to **add**.

Let's see how we use this in our file:

Code Listing 113

```
import {add, pi} from './code-listing-112';

console.log("2 pi = " + add(pi, pi));
```

As you can see, we are simply referencing the named exports, which includes the alias for **sum**.

How about we flip this around? Let's look at the following file:

Code Listing 114

```
function sum(x, y) {
  return x + y;
}
var pi = 3.141593;

export { sum, pi}
```

Now let's import this module as well as provide an alias in our file:

Code Listing 115

```
import {sum as add, pi} from './code-listing-114';


console.log("2 pi = " + add(pi, pi));
```

This gives us a lot of flexibility when dealing with named exports. If we know that we already have a previous import that uses the same name, we can simply alias our import and avoid any type of conflicts. It is also possible to export another module inside your own after you have imported the other module.

Consider the following syntax:

Code Listing 116

```
import {sum as add, pi} from './code-listing-113';
export {sum as add, pi} from './code-listing-113';


console.log("2 pi = " + add(pi, pi));
```

It will produce exactly the same output as well as export **`add`** and **`pi`** as named exports.

Default

When authoring your modules, you have the ability to define a **`default export`**. This allows you to perform a basic **`import`** and receive the **`default`** functionality. Let's first look at what we need to do to our file:

Code Listing 117

```
export default function sum(x, y) {
  return x + y;
}
function notAvailable() {
      console.log("You can't call me...");
}
export var pi = 3.141593;
```

As you can see, the only difference here was that we added the **`default`** keyword to the first function.

Now, let's look at the following file:

Code Listing 118

```
import sum from './code-listing-117';


console.log('2 + 3 = ' + sum(2, 3));
```

If you know that you are only looking for the **`default export`**, this can be a handy feature. If you are authoring your own libraries and have a **`default export`**, this could become a nice, consistent feature. So even though we exported both **`sum`** and **`pi`**, all we had to do was name the reference **`sum`** and use it accordingly. The following is the output from running the preceding code:

Code Listing 119

```
2 + 3 = 5
```

Note: You can only define one default export per module.

Wildcard

Wildcards allow us to load the entire module and refer to the named exports via property notation. Look at the following example:

Code Listing 120

```
import * as math from './code-listing-113';


console.log('2 pi = ' + math.sum(math.pi, math.pi));
```

When we execute this code, we get the same result as the first import example:

Code Listing 121

```
2 pi = 6.283186
```

Some of you may be wondering: is it possible to access functions or variables when we load the entire module using the preceding syntax? Consider the following code:

Code Listing 122

```
export function sum(x, y) {
  return x + y;
}
function notAvailable() {
      console.log("You can't call me...");
}
export var pi = 3.141593;
```

How about the **notAvailable** function? Let's see what happens when we try to access it.

Code Listing 123

```
import * as math from './code-listing-122';


math.notAvailable();
```

This is what makes me really love modules! You have the ability to decide what you wish to expose to the outside and what you want kept internal to your module. Observe what happens when we try to run the preceding code:

Code Listing 124

```
TypeError: math.notAvailable is not a function
```

Now that is pretty slick. The module loader recognizes that you have not identified **notAvailable** as a function you wish to export and make public, and a scoping error is thrown the moment you try to run the code.

Module loaders

This section addresses loader handle resolving module specifiers (the string IDs at the end of **import…from**), loading modules, and the like. The constructor is **Reflect.Loader**. Each browser or transpiler that implements ECMAScript 6 modules keeps a customized instance in the global variable **System** (the system loader), which implements its specific style of module loading.

In addition to the declarative syntax for working with modules that we have been examining, there is also a programmatic API. It allows you to do the following:

- Programmatically work with modules and scripts.
- Configure module loading.

Importing modules and loading scripts

You can programmatically import a module via an API based on ES6 Promises, which we will be discussing in a later chapter. For now, consider the following:

Code Listing 125

```
System.import('some_module')
  .then(some_module => {
    // Use some_module
  })
  .catch(error => {
     // Handle the error
  });
```

`System.import()` enables you to do the following:

- Use modules inside **`<script>`** tags where module syntax is not supported.
- Load modules conditionally.

`System.import()` retrieves a single module, but you can use **`Promise.all()`** to import several modules:

Code Listing 126

```
Promise.all(
  ['module1', 'module2', 'module3']
  .map(x => System.import(x)))
    .then(([module1, module2, module3]) => {
      // Use module1, module2, module3
    });
```

We will cover how promises work in a later chapter.

More loader methods:

- **`System.module(source, [options])`** – evaluates the JavaScript code in source to a module (which is delivered asynchronously via a promise).

- **`System.set(name, module)`** – is for registering a module (e.g. one you have created via **`System.module()`**).

- **`System.define(name, source, [options])`** – both evaluates the module code in source and registers the result.

Configuring module loading

The module loader API has various hooks for configuration. It is still a work in progress.

The loader API will permit many customizations of the loading process. For example:

1. Lint modules on import, e.g., via JSLint or JSHint.

2. Automatically translate modules on import, even if the modules contain CoffeeScript or TypeScript code.

3. Use legacy modules like AMD and Node.js.

Configurable module loading is an area where Node.js and CommonJS are limited.

Chapter 9 Classes

Classes in JavaScript provide a great way to reuse code. Up until now, there has never been an easier way to support both classes and inheritance in a clean manner. We will take a look at both of these now.

Class definition

Consider the following **`Animal`** class in the code-listing-127.js file:

Code Listing 127

```
export class Animal {
  constructor(name) {
    this.name = name;
  }
  greeting(sound) {
    return `A ${this.name} ${sound}`;
  }
  static echo(msg) {
    console.log(msg);
  }
}
```

Just like functions, we can export classes to be loaded in another module. We are declaring the **`Animal`** class. We have a constructor that takes in a single parameter with the name **`name`**. Finally, we have a single function called **`greeting`** that takes in a single parameter called **`sound`**. The **`greeting`** function uses the new string interpolation syntax to create a custom message when called.

I am sure you have noticed the static function **`echo`**. The **`static`** keyword allows us to mark a function as static. This allows us to reference the function via the class name instead of an instance.

Let's take a look at the **`AnimalClient`** class in the code-listing-128.js file:

Code Listing 128

```
import {Animal} from './code-listing-127';


export class AnimalClient {
  constructor() {
    this.animal = new Animal("Dog");
    console.log(this.animal.greeting("barks"));
  }
}


let ac = new AnimalClient();
Animal.echo("roof, roof");
```

As we have seen before, we are importing the **`animal`** class from the code-listing-126.js file. Next, we are constructing an **`AnimalClient`** class that creates a new **`Animal`** instance inside its constructor and also renders the **`greeting`** function out to the console. Since we are testing this code, we have a single line after the class to kick everything off and get things going as well as a call to the **`static`** function **`echo`** on the **`Animal`** class. Even though we are using **`AnimalClient`** from the same file, we **`export`** the class in order for it to be accessed from another file.

The following is the output from the preceding code:

Code Listing 129

```
A Dog barks
roof, roof
```

Class inheritance/base class access

Now that we understand what the class syntax looks like, let's extend the **`Animal`** class and create another one that inherits from it. Consider the following code from the code-listing-129.js file:

Code Listing 130

```
export class Animal {
```

```
  constructor(name) {
    this.name = name;
  }
  greeting(sound) {
    return `A ${this.name} ${sound}`;
  }
  static echo(input) {
    console.log(input);
  }
}
export class Canine extends Animal {
  constructor() {
    super("canine");
  }
  static echo() {
    super.echo("bow wow");
  }
}
```

The **Animal** class hasn't changed, but we have added another class to the file called **Canine**. As you might expect, this class **extends** the **Animal** class. This passes in the string **canine** to the base constructor using the **super** keyword. We can also use **super** to access functions and properties off of the base class. This is illustrated in the example of the **echo** function calling the base implementation and passing the string **bow wow**.

Getter/setter

Now let's look at getters and setters. Consider the **Animal** class in the code-listing-131.js file:

Code Listing 131

```
export class Animal {
  constructor(name) {
```

```
    this.name = name;
    this.color = "brown";
  }
  get color() {
    return this._color;
  }
  set color(value) {
    this._color = value;
  }
  toString() {
    return console.log(`I am a ${this.name}.  I am ${this.color} in
color.`);
  }
  static echo(input) {
    console.log(input);
  }
}
```

As you can see, we are using two new keywords: **get** and **set**. This allows us to provide getters and setters by wrapping other variables or performing other operations. This can be very powerful and allows you to centralize all access to your variables.

Let's look at how this is used in the **AnimalClient** class in the code-listing-132.js file:

Code Listing 132

```
import {Animal} from './code-listing-131';


export class AnimalClient {
  constructor() {
    this.animal = new Animal("dog");
    this.animal.toString();
```

```
  }
}


let ac = new AnimalClient();
```

Finally, here is the output from the preceding code example:

Code Listing 133

```
I am a dog.  I am brown in color.
```

Subclassing built-ins

Until ES6, subclassing built-ins has been extremely difficult. Consider the following example that does not use built-ins:

Code Listing 134

```
function SuperCtor(arg1) {
  ...
}
function SubCtor(arg1, arg2) {
  SuperCtor.call(this, arg1);
}

SubCtor.prototype = Object.create(SuperCtor.prototype);
```

This is the prototypical way to subclass another object. However, when we deal with built-ins like Array, Date, and DOM Element(s), this is nearly impossible. In JavaScript, if you invoke a built-in constructor, two steps happen internally with every function:

1. Allocation – creating an instance **`inst`**, an object whose prototype is **`C.prototype`** (if the value is not an object, use **`Object.prototype`**).

2. Initialization – initializes **`inst`** via **`C.call(inst, arg1, arg2, …)`**. If the result of that call is an object, return it. Otherwise, return **`inst`**.

If we were to use the same subclass pattern previously shown, it simply wouldn't work. However, with ES6, we now have the ability to subclass built-ins.

Object construction for a function named **`Ctor`** now uses two phases, both of which are virtually dispatched:

- Call **`Ctor [@@create]`** to allocate the object, installing any special behavior.
- Invoke **`constructor`** on the new instance to initialize.

The known **`@@create`** symbol is available via **`Symbol.create`**. Built-ins now expose **`@@create`** explicitly.

Consider the following code example:

Code Listing 135

```
// Pseudo-code of Array
class Array {
  constructor(...args) { /* ... */ }
  static [Symbol.create]() {
      // Install special [[DefineOwnProperty]]
      // to magically update 'length'
  }
}

// User code of Array subclass
class MyArray extends Array {
  constructor(...args) { super(...args); }
}
// Two-phase 'new':
// 1) Call @@create to allocate object
// 2) Invoke constructor on new instance
var arr = new MyArray();
arr[1] = 12;
console.log(arr.length == 2);
console.log(arr);
```

The first class is an example of what the native JavaScript **`Array`** class would look like. What is important here is understanding that by using the **`@@create`** symbol we can extend the built-in class. The code required to create your own **`Array`** is trivial, although it does not do anything different.

The following is the output of the preceding code:

Code Listing 136

```
true

[ , 12 ]
```

The comma in the output indicates that the length was indeed 2 by returning the result of evaluating the length in a Boolean expression. It also renders out to the console the array where we can easily see that the first index position has nothing assigned.

Chapter 10 Iterators

Iterators are objects that can traverse a collection. They enable custom iteration like the .NET CLR **IEnumerable** or Java **Iterable** objects. They can generalize **for..in** to custom iterator-based iteration with **for..of**. They also don't require realizing an array, thus enabling lazy design patterns like LINQ.

Iterators

The following is an example of an iterator definition for producing Fibonacci numbers:

Code Listing 137

```
let fibonacci = {
  [Symbol.iterator]() {
    let pre = 0, cur = 1;
    return {
      next() {
        [pre, cur] = [cur, pre + cur];
        return { done: false, value: cur }
      }
    }
  }
}
```

The first thing to notice is the usage of the **Symbol.iterator**. This keyword is necessary when defining iterators. Based on the interface, you will notice that the iterator returns a single function, **next**. It is also interesting to note that internal to the **next** function, we see examples of destructuring. This is more convenient than writing the equivalent in multiple statements. Finally, we see that the **next** function returns an object literal with two properties.

Let's move to the next section and see how this iterator is used.

for-of operator

Now that we have an iterator defined, let's see how we can use it. Consider the following code example:

Code Listing 138

```
for (var n of fibonacci) {
  // truncate the sequence at 100
  if (n > 100)
    break;
  console.log(n);
}
```

As you can see, the syntax of **`for..of`** is the same as **`for..in`**. We shortened the sequence so that we could show the output without filling a whole page. The following is the output when we execute the preceding code:

Code Listing 139

```
1
2
3
5
8
13
21
34
55
89
```

Let's look at another example using **`for..of`** with an array:

Code Listing 140

```
for (let element of [1, 2, 3]) {
  console.log(element);
```

```
}
```

The preceding code could have been written as follows as well:

Code Listing 141

```
let elements = [1, 2, 3];
for (let element of elements) {
  console.log(element);
}
```

The following is the output for the preceding code:

Code Listing 142

```
1
2
3
```

When we compare **for..of** and **for..in**, we will see that **for..in** iterates over property names while **for..of** iterates over property values.

Let's take one more look at iterators by looking at the built-in iterable for **String**. Consider the following code:

Code Listing 143

```
let someString = "hi";
var iterator = someString[Symbol.iterator]();
console.log(iterator + "");

console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
```

Here, we are creating a string variable, **someString**, that contains the string **hi**. We next create an iterator off of the **someString** variable using the **Symbol.iterator** syntax. Finally, we log out the iterator. We are forcing it to use its **toString** function by concatenating an empty string. The sequence of calls for **iterator.next()** shows us exactly what is contained from the returned object literals.

The following is the output for the preceding code:

Code Listing 144

```
[object String Iterator]
{ value: 'h', done: false }
{ value: 'i', done: false }
{ value: undefined, done: true }
```

As you can see, this is identical to the signature that we implemented in our custom Fibonacci iterator. Hopefully this gives you a better understanding as to how iterators work.

Chapter 11 Generators

Generators are functions that can be paused and resumed. They simplify iterator-authoring using the **`function*`** and **`yield`** keywords. A function declared as **`function*`** returns a **`Generator`** instance. Generators are subtypes of iterators that include additional **`next`** and **`throw`** functions. These enable values to flow back into the generator, so **`yield`** is an expression form which returns or throws a value.

Generator definition

As stated in the previous paragraph, generators are functions that can be paused and resumed. Let's take a look at the following simple example to get a better understanding of generators:

Code Listing 145

```
function* sampleFunc() {
  console.log('First');
  yield;
  console.log('Second');
}

let gen = sampleFunc();
console.log(gen.next());
console.log(gen.next());
```

Here, we have created a generator named **`sampleFunc`**. Notice that we are pausing in the middle of the function using the **`yield`** keyword. Also notice that when we call the function, it is not executed. It won't be executed until we iterate over the function, and so here we iterate over it using the **`next`** function. The **`next`** function will execute until it reaches a **`yield`** statement. Here it will pause until another **`next`** function is called. This in turn will resume and continue executing until the end of the function.

Let's look at the output for further illustration:

Code Listing 146

```
First

{ value: undefined, done: false }

Second

{ value: undefined, done: true }
```

Now let's move onto the next section and look at generator iteration.

Generator iteration

Let's look at an example of creating a generator and then iterating using it. Consider the following code example:

Code Listing 147

```
let fibonacci = {
  [Symbol.iterator]: function*() {
    let pre = 0, cur = 1;
    for (;;) {
      let temp = pre;
      pre = cur;
      cur += temp;
      yield cur;
    }
  }
}

for (let n of fibonacci) {
  // truncate the sequence at 100
  if (n > 100)
    break;
  console.log(n);
}
```

As you can see from the preceding code, generators are indeed a subtype of iterators. The syntax is very similar, with just a few changes. Notably, you see the **`function*`** and **`yield`** keywords.

The following is the output for the preceding code:

Code Listing 148

```
1

2

3

5

8

13

21
```

```
34

55

89
```

As you can see, we get the same output as we did when we wrote our own iterator.

Let's write this one more time using some more ES6-style coding this time:

Code Listing 149

```
let fibonacci = {
  *[Symbol.iterator]() {
    let pre = 0, cur = 1;
    for (;;) {
      [ pre, cur ] = [ cur, pre + cur ];
      yield cur;
    }
  }
}

for (let n of fibonacci) {
  if (n > 100)
    break;
  console.log(n);
}
```

As you can see, the asterisk (*****) is now on the **`[Symbol.iterator]`** definition. Also note the usage of destructuring of objects. This makes for even terser and cleaner code.

The following is the output for the preceding code:

Code Listing 150

```
1

2

3

5

8

13

21
```

```
34

55

89
```

Let's consider another example where we use the generator directly without the iterator:

Code Listing 151

```
function* range (start, end, step) {
  while (start < end) {
    yield start;
    start += step;
  }
}

for (let i of range(0, 10, 2)) {
  console.log(i);
}
```

Here, we are defining a generator that takes **`start`**, **`end`**, and **`step`** as arguments. We pause when we reach **`yield`**. Next, we perform addition to **`start`** in the amount of **`step`**. This continues until we reach **`end`**.

The following is the output for the preceding code:

Code Listing 152

```
0

2

4

6

8
```

Generator matching

It is possible to use generators and support destructuring via array matching. Let's look at the following example:

Code Listing 153

```
let fibonacci = function* (numbers) {
  let pre = 0, cur = 1;
```

```
  while (numbers-- > 0) {
    [ pre, cur ] = [ cur, pre + cur ];
    yield cur;
  }
}

for (let n of fibonacci(5))
  console.log(n);

let numbers = [ ...fibonacci(5) ];
console.log(numbers);;

let [ n1, n2, n3, ...others ] = fibonacci(5);
console.log(others[0]);
```

In the preceding code, we are first creating a generator. Next, we perform a simple iteration over the generator for five iterations. Next, we use the **spread** operator to assign values to the numbers array. Finally, we use pattern matching in the array and assign others the values from the generator function.

Here is the output for the preceding code:

Code Listing 154

```
1

2

3

5

8

[ 1, 2, 3, 5, 8 ]

5
```

Generator control flow

One of the promises of generators is control flow. This becomes important when dealing with asynchronous programming. We see this a lot with promises, which we will be looking at in another chapter. Let's take a look at how we can handle control flow while using generators. Take a look at the following example:

Code Listing 155

```
function async (proc, ...params) {
  var iterator = proc(...params);
  return new Promise((resolve, reject) => {
    let loop = (value) => {
      let result;
      try {
        result = iterator.next(value);
      }
      catch (err) {
        reject(err);
      }
      if (result.done) {
        resolve(result.value);
      }
      else if (typeof result.value === "object"
        && typeof result.value.then === "function")
        result.value.then((value) => {
          loop(value);
        }, (err) => {
          reject(err);
        })
      else {
        loop(result.value);
      }
    };
    loop();
  })
}

//  application-specific asynchronous builder
function makeAsync (text, after) {
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(text), after);
  })
}

//  application-specific asynchronous procedure
async(function* (greeting) {
  let foo = yield makeAsync("foo", 300);
  let bar = yield makeAsync("bar", 200);
  let baz = yield makeAsync("baz", 100);
  return `${greeting} ${foo} ${bar} ${baz}`;
}, "Hello").then((msg) => {
  console.log(msg);
```

```
});
```

Okay, let's break this down. The first function, **async**, receives a generator as the first argument and any other arguments that follow. In this case, it receives **Hello**. A new promise is created that will loop through all the iterations. In this case, it will loop exactly four times. The first time, the value is undefined and we call **iterator.next()** in passing in the value. All subsequent times, value represents **foo**, **bar**, and **baz**. We complete looping once we have exhausted all of our iterations.

The next function, **makeAsync**, simply takes some text and waits a set period of time. It uses the **setTimeout** function to simulate a real world scenario.

The last code segment is our execution of the **async** function. We pass in a generator function that takes in a greeting as an argument as well as contains three **yield** statements and a **return**. It makes the call to **async** and executes the iterations while adhering to the various **setTimout** specifications. The nice thing here is that regardless of when each of the iteration calls **return**, we still have code that looks very synchronous in manner. We do not display our message until the iteration has been completed.

Here is the output from the preceding code:

Code Listing 156

```
Hello foo bar baz
```

Generator methods

Generator methods are the support for methods in a class and on objects based on generator functions. Consider the following example:

Code Listing 157

```
class GeneratorClass {
  * sampleFunc() {
    console.log('First');
    yield;
    console.log('Second');
  }
}
let gc = new GeneratorClass();
let gen = gc.sampleFunc();
console.log(gen.next());
console.log(gen.next());
```

Just like our first example, we are now using generators inside our ES6 classes. Also like the first example, we get the same output:

Code Listing 158

```
First

{ value: undefined, done: false }

Second

{ value: undefined, done: true }
```

Chapter 12 Map, Set

Maps and **Sets** are a welcome to JavaScript. **Sets** and **Maps** provide an efficient data structure for common algorithms.

Map

In JavaScript, it has always been a pain mapping objects or primitive values from/to other objects or primitive values. In ES5, the best you could do was map from strings to arbitrary values. The **Map** data structure in ES6 lets you use arbitrary values as keys.

Let's look at a quick example of this:

Code Listing 159

```
let map = new Map();

map.set('foo', 123);
console.log(map.get('foo'));
console.log(map.has('foo'));

console.log(map.delete('foo'));
console.log(map.has('foo'));
```

Here is the output from the preceding code:

Code Listing 160

```
123

true

true

false
```

Let's look at another example:

Code Listing 161

```
let map = new Map();
map.set('foo', true);
map.set('bar', false);
```

```
console.log(map.size);
map.clear();
console.log(map.size);
```

Here is the output from the preceding code:

Code Listing 162

```
2

0
```

Setting up a map

You can set up a map via an iterable over key-value pair. One way is to use an array as follows:

Code Listing 163

```
let map = new Map([
  [1, 'one'],
  [2, 'two'],
  [3, 'three']
]);
```

Another way is to chain the set method like so:

Code Listing 164

```
let map = new Map()
  .set(1, 'one')
  .set(2, 'two')
  .set(3, 'three');
```

Iterating over a map

We have several options when iterating over a map. First, consider the following map definition:

Code Listing 165

```
let map = new Map([
  [1, 'one'],
  [2, 'two'],
  [3, 'three']
```

```
]);
```

Now, let's take a look at how we can iterate over this map. The first example will iterate over the keys:

Code Listing 166

```
for (let key of map.keys()) {
  console.log(key);
}
```

We can also iterate over the values:

Code Listing 167

```
for (let value of map.values()) {
  console.log(value);
}
```

We can iterate over the entries:

Code Listing 168

```
for (let entry of map.entries()) {
  console.log(entry[0], entry[1]);
}
```

We can obtain the key/value pair of the entries:

Code Listing 169

```
for (let [key, value] of map.entries()) {
  console.log(key, value);
}
```

Finally, because entries is an iterator itself, we can rewrite the previous in a more terse manner:

Code Listing 170

```
for (let [key, value] of map) {
  console.log(key, value);
}
```

Spreading a map

The **spread** operator (…) turns an iterable into the arguments of a function or parameter call. We have several options when iterating over a map. Thus, we can easily spread our map like the following:

Code Listing 171

```
let map = new Map([
  [1, 'one'],
  [2, 'two'],
  [3, 'three']
]);
let arr = [...map.keys()];
console.log(arr);
```

The preceding code generates the following output:

Code Listing 172

```
[ 1, 2, 3 ]
```

WeakMap

A **WeakMap** is a map that doesn't prevent its keys from being garbage-collected. That means that you can associate data with objects without worrying about memory leaks.

A **WeakMap** is a data structure whose keys must be objects and whose values can be arbitrary values. It has the same API as **Map**, with one significant difference: you can't iterate over the contents. You can't iterate over the keys, nor the values, nor the entries. You also can't clear a **WeakMap**.

Let's look at the following example:

Code Listing 173

```
let _counter = new WeakMap();
let _action = new WeakMap();

class Countdown {
  constructor(counter, action) {
    _counter.set(this, counter);
    _action.set(this, action);
  }
  dec() {
```

```
    let counter = _counter.get(this);
    if (counter < 1) return;
    counter--;
    _counter.set(this, counter);
    if (counter === 0) {
      _action.get(this)();
    }
  }
}

let c = new Countdown(2, () => console.log('DONE'));
c.dec();
c.dec();
```

The preceding code uses **`WeakMaps`** **`_counter`** and **`_action`** to store private data. This is very elegant in that these **`WeakMaps`** are "holding" a reference to the **`Countdown`** class, yet they are doing so in a weak manner and, thus, are memory leak safe.

The preceding code generates the following output:

Code Listing 174

```
DONE
```

Set

`Set` works with arbitrary values, is fast, and even handles **`NaN`** correctly.

Let's take a look at an example:

Code Listing 175

```
let set = new Set();
set.add('red');

console.log(set.has('red'));
set.delete('red');
console.log(set.has('red'));
```

Here is the output from the preceding code:

Code Listing 176

```
true

false
```

Let's look at another example:

Code Listing 177

```
let set = new Set();
set.add('red');
set.add('green');

console.log(set.size);
set.clear();
console.log(set.size);
```

Here is the output from the preceding code:

Code Listing 178

```
2

0
```

Setting up a set

You can set up a set via an iterable over the elements that make up the set. One way is to use an array as follows:

Code Listing 179

```
let set = new Set(['red', 'green', 'blue']);
```

Another way is to chain the set method like so:

Code Listing 180

```
let set = new Set()
      .add('red')
      .add('green')
      .add('blue');
```

Values

Like maps, elements are compared similarly to **===**, with the exception of **NaN** being like any other value. Consider the following:

Code Listing 181

```
let set = new Set([NaN]);
console.log(set.size);
console.log(set.has(NaN));
```

Executing the preceding code produces the following output:

Code Listing 182

```
1

true
```

Iterating over a set

Sets are iterable, and the **for-of** loop works exactly as you would expect. Consider the following example:

Code Listing 183

```
let set = new Set(['red', 'green', 'blue']);
for (let x of set) {
  console.log(x);
}
```

Executing the preceding code produces the following output:

Code Listing 184

```
red

green

blue
```

Sets preserve iteration order. That is, elements are always iterated over in the order in which they were inserted.

Now, let's look at an example using the spread operator (**…**). As you know it works with iterables and thus allows you to convert a set into an array:

Code Listing 185

```
let set = new Set(['red', 'green', 'blue']);
let arr = [...set];
console.log(arr);
```

Executing the preceding code produces the following output:

Code Listing 186

```
[ 'red', 'green', 'blue' ]
```

What about going from array to set? Take a look at the following example:

Code Listing 187

```
let arr = [3, 5, 2, 2, 5, 5];
let unique = [...new Set(arr)];
for (let x of unique) {
  console.log(x);
}
```

The preceding code produces the following output:

Code Listing 188

```
3

5

2
```

As you may have noticed, a set does not have duplicate values in it.

Mapping and filtering a set

Compared to arrays, sets don't have the methods **`map()`** and **`filter()`**. However, you can do a simple trick of converting them to arrays and then back again. Consider the following example:

Code Listing 189

```
let set = new Set([1, 2, 3]);
set = new Set([...set].map(x => x * 2));
// Resulting set: {2, 4, 6}
```

Here is an example using filter:

Code Listing 190

```
let set = new Set([1, 2, 3, 4, 5]);
set = new Set([...set].filter(x => (x % 2) == 0));
// Resulting set: {2, 4}
```

WeakSet

A **`WeakSet`** is a set that doesn't prevent its elements from being garbage-collected. A **`WeakSet`** doesn't allow for iteration, looping, or clearing.

Given that you can't iterate over their elements, there are not very many use cases for **`WeakSet`**s. They enable you to mark objects and to associate them with Boolean values.

WeakSet API

WeakSets only have three methods, and all of them work the same as the **`set`** methods:

- **`add(value)`**
- **`has(value)`**
- **`delete(value)`**

Chapter 13 Symbols

Symbols are a new primitive type in ES6. **`Symbols`** are tokens that serve as unique IDs. You create symbols via the factory function **`Symbol()`**.

Symbol type

Let's take a look at an example using symbols:

Code Listing 191

```
Symbol("foo") !== Symbol("foo")
const foo = Symbol('test1');
const bar = Symbol('test2');
typeof foo === "symbol";
typeof bar === "symbol";
let obj = {};
obj[foo] = "foo";
obj[bar] = "bar";
console.log(JSON.stringify(obj));
console.log(Object.keys(obj));
console.log(Object.getOwnPropertyNames(obj));
console.log(Object.getOwnPropertySymbols(obj));
```

As you can see, symbols are unique and no two symbols will equal each other. Also note how we are assigning symbols to object properties. We use the symbol as the key for the object and assign a value to it. Also observe that you can pass a string into the constructor of a symbol, but it is used only for debugging purposes.

Let's take a look at the output from the preceding code:

Code Listing 192

```
{}

[]

[]

[ Symbol(test1), Symbol(test2) ]
```

Given that we now have a new kind of value that can become the key of a property, the following terminology is used for ES6:

- Property keys – are either strings or symbols
- Property names – are strings

The name of **`Object.keys()`** doesn't really work. It only considers property keys that are strings. This explains why our output was only an empty array (**`[]`**).

Likewise, the name **`Object.getOwnPropertyName()`** only returns property keys that are strings as well.

Global symbols

If you want a symbol to be the same in all realms, you need to create it via the global symbol registry. Let's look at the following example:

Code Listing 193

```
Symbol.for("app.foo") === Symbol.for("app.foo")
const foo = Symbol.for("app.foo")
const bar = Symbol.for("app.bar")
Symbol.keyFor(foo) === "app.foo"
Symbol.keyFor(bar) === "app.bar"
typeof foo === "symbol"
typeof bar === "symbol"
let obj = {}
obj[foo] = "foo"
obj[bar] = "bar"
console.log(JSON.stringify(obj));
console.log(Object.keys(obj));
console.log(Object.getOwnPropertyNames(obj));
console.log(Object.getOwnPropertySymbols(obj));
```

Notice that we now have the **`Symbol.for`** function. This is how we place the symbol in the global registry. Also notice how we assign a key for a symbol using the **`Symbol.keyFor`** function.

Global symbols behave the same as normal symbols, as we can see from the output from the preceding code:

Code Listing 194

```
{}

[]

[]

[ Symbol(app.foo), Symbol(app.bar) ]
```

Chapter 14 Promises

Promises provide a standard implementation of handling asynchronous programming in JavaScript without using callbacks. One of the biggest side effects of using the callback pattern is how dreadfully ugly your code can get with only a few levels of callbacks.

Promises allow you to develop asynchronous scripts more easily than using callbacks. A promise represents a value that we can handle at some point in the future. Promises give us two main advantages over callbacks:

- No other registered handlers of that value can change it. A promise contract is immutable.
- We are guaranteed to receive the value, regardless of when we register a handler for it, even if it's already resolved. This contrasts with events since once an event is fired you can't access its value at a later time.

Let's look at the following example:

Code Listing 195

```
var p2 = Promise.resolve("foo");
p2.then((res) => console.log(res));

var p = new Promise(function(resolve, reject) {
  setTimeout(() => resolve(4), 2000);
});

p.then((res) => {
  res += 2;
  console.log(res);
});

p.then((res) => console.log(res));
```

In the first line, we create a promise and resolve it immediately. This demonstrates the second advantage. We can still call on the promise and get the value from it, unlike events. Next, we define a standard promise and have it resolve after two seconds. Our handler receives the value from the promise but cannot change the value of the promise itself. This is good in that our promise is immutable and always returns the same result when called multiple times.

The following is the output from the preceding example:

Code Listing 196

```
foo

6

4
```

Up until now, we have only resolved promises. What about when we reject a promise? Consider the following code:

Code Listing 197

```
var p = new Promise(function(resolve, reject) {
  setTimeout(() => reject("Timed out!"), 2000);
});

p.then((res) => console.log(res),
      (err) => console.log(err));
```

This time, we are calling **`reject`** after our two second delay. We see that **`then()`**can also take in a second handler for errors.

The following is the output from the preceding example:

Code Listing 198

```
Timed out!
```

But wait, you can write this even more differently! Consider the following code:

Code Listing 199

```
var p = new Promise(function(resolve, reject) {
  setTimeout(() => reject("Timed out!"), 2000);
});

p.then((res) => console.log("Response:", res))
  .catch((err) => console.log("Error:", err));
```

You can use the catch function off of the promise as well. Our output will be exactly the same:

Code Listing 200

```
Error: Timed out!
```

Let's consider another example, when an exception happens. Consider the following code:

Code Listing 201

```
var p = new Promise(function(resolve, reject) {
  setTimeout(() => {throw new Error("Error encountered!");}, 2000);
});

p.then((res) => console.log("Response:", res))
  .catch((err) => console.log("Error:", err));
```

Throwing an **`Error`** is the same as calling **`reject()`**. You are able to catch the error and handle it accordingly.

The following is the output from the preceding example:

Code Listing 202

```
Error encountered!
```

Promise.all

One nice thing about promises is that many synchronous tools still work, because promise-based functions return results.

Consider the following example:

Code Listing 203

```
var fileUrls = [
  'http://example.com/file1.txt',
  'http://example.com/file2.txt'
];
var httpGet = function(item) {
  return new Promise(function(resolve, reject) {
    setTimeout(() => resolve(item), 2000);
  });
};
var promisedTexts = fileUrls.map(httpGet);

Promise.all(promisedTexts)
  .then(function (texts) {
    texts.forEach(function (text) {
      console.log(text);
    });
  })
```

```
  .catch(function (reason) {
    // Receives first rejection among the promises
  });
```

In this example, we are using an asynchronous call to load files. The elegance of this code is that **`then()`** will not be called until all of the promises have completed. However, if any of the promises fail, the **`catch()`** handler will be called.

Promise.race

Sometimes we don't want to wait until all of the promises have completed; rather, we want to get the results of the first promise in the array to fulfill. We can do this with **`Promise.race()`** which, like **`Promise.all()`**, takes an array of promises. However, unlike **`Promise.all()`**, it will fulfill its returned promise as soon as the first promise in that array fulfills.

Code Listing 204

```
function delay(ms) {
  return new Promise((resolve, reject) => {
    setTimeout(resolve, ms);
  });
}

Promise.race([
  delay(3000).then(() => "I finished second."),
  delay(2000).then(() => "I finished first.")
])
.then(function(txt) {
   console.log(txt);
})
.catch(function(err) {
  console.log("error:", err);
});
```

Here, we are using a helper function, **`delay()`**, that will timeout after a certain amount of time. We create our promise passing in an array. Next, we simply dump out what happens.

The following is the output from the preceding example:

Code Listing 205

```
I finished first.
```

As you can see, the first promise that fulfills is what we get in our **`then()`** handler.

Chapter 15 Proxies

Proxies enable the creation of objects with the full range of behaviors available to host objects. They can be used for interception, object virtualization, logging, profiling, and more. Proxies are considered a meta programming feature.

Note: This is one of the areas where not all browsers or implementations are equal. The following examples can be run and tested under the latest version of Firefox or possibly Microsoft Edge.

Proxying

Consider the following example of proxying a normal object:

Code Listing 206

```
var target = {};
var handler = {
  get: function (receiver, name) {
    return `Hello, ${name}!`;
  }
};

var p = new Proxy(target, handler);
p.world === 'Hello, world!';
```

As you can see from the preceding code, our target is a simple object literal. We perform a proxy on getters where it intercepts the get and returns the string **`Hello,`** plus the name of the property. We also test to see if the world property is the same as the string, **`Hello, world!`**.

Let's also look at proxying a function object:

Code Listing 207

```
var target = function () { return 'I am the target'; };
var handler = {
  apply: function (receiver, ...args) {
    return 'I am the proxy';
  }
};

var p = new Proxy(target, handler);
p() === 'I am the proxy';
```

This is very similar to the previous example with the exception that our target is now a function. You will notice that the handler takes in a receiver as well as a spread operator (…) to handle for any number of arguments.

Proxy traps

Proxy traps enable you to intercept and customize operations performed on objects. The following is a list of the traps available for all of the runtime-level meta-operations:

- **get**
- **set**
- **has**
- **deleteProperty**
- **apply**
- **construct**
- **getOwnPropertyDescriptor**
- **defineProperty**
- **getPropertyOf**
- **setPropertyOf**
- **enumerate**
- **ownKeys**
- **preventExtensions**
- **isExtensible**

Chapter 16 Reflect API

The Reflect API provides a runtime-level meta-operations on objects. This is effectively the inverse of the Proxy API, and allows making calls corresponding to the same meta-operations as the proxy traps. The **`Reflect`** object is a static object. You cannot create an instance of it. Likewise, all of its methods are static. You may be thinking that we already had some reflection-like methods available in JavaScript, but the biggest difference is that **`Reflect`** provides more meaningful return values.

Consider the following example:

Code Listing 208

```
let obj = {}, name = "matt", desc = "here we go";
try {
  Object.defineProperty(obj, name, desc);
  // worked.
} catch (e) {
  // error.
}

if (Reflect.defineProperty(obj, name, desc)) {
  // worked
} else {
  // error.
}
```

In the first part, we have a try/catch block, and we are using the **`Object.defineProperty`** method. In this example, this method only returns the first argument passed into it. However, looking at the **`Reflect.defineProperty`** method, we see that it returns a Boolean value, which is much more meaningful.

Let's consider another example:

Code Listing 209

```
let obj = { a: 1 };
Object.defineProperty(obj, "b", { value: 2 });
obj[Symbol("c")] = 3;
console.log(Reflect.ownKeys(obj)); // [ "a", "b", Symbol(c) ]
```

As you can see in this example, **`Reflect.ownKeys`** gives us both string and symbol-based keys.

The following are all the methods the **`Reflect`** object contains:

- **Reflect.get(target, name, [receiver])**
- **Reflect.set(target, name, value, [receiver])**
- **Reflect.has(target, name)**
- **Reflect.apply(target, receiver, args)**
- **Reflect.construct(target, args)**
- **Reflect.getOwnPropertyDescriptor(target, name)**
- **Reflect.defineProperty(target, name, desc)**
- **Reflect.getPrototypeOf(target)**
- **Reflect.setPrototypeOf(target, newProto)**
- **Reflect.deleteProperty(target, name)**
- **Reflect.enumerate(target)**
- **Reflect.preventExtensions(target)**
- **Reflect.isExtensible(target)**
- **Reflect.ownKeys(target)**

Chapter 17 Tail Recursion Calls

ES6 offers tail call optimization (TCO), where you can make some function calls without growing the stack. This is extremely useful in recursive and functional programming usage in JavaScript.

Let's look at an example:

Code Listing 210

```
function factorial(n, acc = 1) {
  'use strict';
  if (n <= 1) return acc;
  return factorial(n - 1, n * acc);
}

// Stack overflow in most implementations today,
// but safe on arbitrary inputs in ES6
factorial(100000);
```

It is important to note that you must be in strict mode for this optimization to work.

Also notice that the last thing to happen in the factorial function is that it returns a value using a call to itself. This is one of the key precepts of functional programming, and most functional programming languages do not incur stack overflow when dealing with recursion.

To summarize, if the last thing that happens before the return statement is the invocation of a function that does not need to access any of the current local variables, the interpreter specified by ES6 will optimize that call by reusing the stack frame.

Note: This will not likely receive optimization until our browsers have an ES6 interpreter in place.