What's new in ECMAScript 2015 and 2016

This document starts with an introduction to the ECMAScript 2015 and 2016 specifications that shows you how to support the new features in older browsers. Then, it presents all of the significant new features. It ends by showing how to use the Internationalization API for formatting numbers and dates.

This document assumes that you already have a solid set of JavaScript skills, like the ones you develop with a book like *Murach's JavaScript and jQuery.* If so, you should be able to grasp what the new features do and how they work with little trouble. Then, you can decide which ones you want to add to your programming skillset.

For a few of the new features, though, you might need to refresh your memory about how the related old feature works in order to understand what the new feature does. That's likely to be true for Promises, the new syntax for object literals and IIFEs, and the new syntax for creating, using, and inheriting objects. In those cases, *Murach's JavaScript and jQuery* is the ideal reference book.

MIKE MURACH & ASSOCIATES, INC.
1-800-221-5528 • (559) 440-9071 • Fax: (559) 440-0963
murachbooks@murach.com • **www.murach.com**

What's new in ECMAScript 2015 and 2016

This document presents all of the significant new features in the last two releases of the ECMAScript specification, ECMAScript 2015 and ECMAScript 2016. This document starts with an introduction to the ECMAScript specification and ends by showing how to use the Internationalization API.

An introduction to ECMAScript

The *ECMAScript specification* details the standards that scripting languages like JavaScript should meet. The history of this specification, and some changes to how new features are rolled out, are described in the topics that follow.

A history of the ECMAScript standard

JavaScript was invented by NetScape in 1995 and released as part of the Netscape Navigator web browser in early 1996. In response, Microsoft developed a similar language called JScript and released it as part of the Internet Explorer web browser in late 1996.

Since there were differences between the two scripting languages, Netscape gave JavaScript to the *European Computer Manufacturers Association (ECMA)* to develop a standard. The standard is called the *ECMAScript specification,* and the first version was released in June 1997. Since then, several versions have been released, as shown in the table in figure 1-1.

In June 2015, the sixth version of the ECMAScript, or ES, specification was released. This version was officially called *ECMAScript 2015* (ES2015), but you'll often see it called ECMAScript 6, or ES6. Then, in June 2016, the seventh version was released. Like the sixth version, this version is officially called *ECMAScript 2016* (ES2016), but you'll often see it called ES7.

ES5, ES2015, and ES2016 added several important features to JavaScript, as described in this figure. The features of ES5 are fully supported by all modern browsers. However, a few of these features won't work in older browsers like Internet Explorer 7, 8, and 9 (which can be referred to as IE7, IE8, and IE9).

The features of ES2015 and ES2016, because they're newer, aren't as fully supported by modern browsers, although they'll become more supported with time. Also, like ES5, the features of ES2015 and ES2016 won't work in older browsers.

You can go to the URL in this figure to see which browsers support which features. Even though this URL is for ES6/2015, you'll see that it also provides links for ES5 and ES2016.

How new versions are now released

When the sixth version of the ES specification was released, the committee in charge of the specification changed how it would release new versions going forward. Instead of having a set specification that they would release when all the features were completed, they moved to yearly releases of features that had been approved to that point. That's why the ECMAScript versions starting with ES6 are named by year rather than by version number.

The versions and release dates of the ECMAScript specification

Version	Release date
`1`	June 1997
`2`	June 1998
`3`	December 1999
`4`	Abandoned, never released
`5`	December 2009
`5.1`	June 2011
`2015`	June 2015
`2016`	June 2016

The URL for a browser compatibility table

`http://kangax.github.io/compat-table/es6/`

ECMAScript 5 enhancements

- Allows you to run in strict mode.
- Adds several methods that make it easier to work with arrays.
- Adds a safer way to create an object and more control over an object's properties.
- Adds a built-in way to work with JavaScript Object Notation (JSON).

ECMAScript 2015 (ES6) enhancements

- Adds Promises, which is a simpler syntax for callback functions.
- Adds several syntax improvements that make code easier to read and understand.
- Adds block scope and easier ways to work with classes.
- Adds several built-in methods for working with strings, numbers, objects, and arrays.

ECMAScript 2016 (ES7) enhancements

- Adds a simpler syntax for computation with powers.
- Adds an array method to check if an array includes a specified element.

Description

- Netscape invented JavaScript in 1995 and turned it over to the *European Computer Manufacturers Association (ECMA)* for standardization in 1996.
- All of the ES5 features are supported by all modern browsers. In contrast, the features of ES2015 and ES2016 are less consistently supported. You can use the table at the URL shown above to see which features are supported in which browsers.
- Starting with the sixth version of the ES specification (ES2015), new versions are released each year. These versions will contain whatever new features are approved at that point.

Figure 1-1 An introduction to ECMAScript

How to use new features with older browsers

If you want to use some of the new features of ECMAScript but still support a wide range of browsers, you can use polyfill files and transpilers. Polyfill files are JavaScript libraries that you include with your applications. Transpilers convert new JavaScript code to older versions of JavaScript code.

How to use polyfill files

A *polyfill file* (also called a *shim file*) is a JavaScript library that implements features in browsers that don't support them. A *sham file* is a JavaScript library that prevents a new feature from throwing an error in a browser that doesn't support it. However, the sham file doesn't implement the feature so it still won't work.

The shim files support features that are either additions to existing objects or new objects themselves. Some examples of these are the Promise object type, and new methods of the String, Number, Math, Array, and Object object types. You'll learn how to use them later on in this document.

Figure 1-2 presents a series of the shim and sham files that make some (but not all) of the ES5, ES2015, and ES2016 features work with browsers that only support ES3. To implement this workaround, you include the URLs shown in this figure in the script elements for the page.

The shim file for ES5 implements the ES5 features using ES3code. In contrast, the shim files for ES2015 (ES6) and ES2016 (ES7) implement the ES2015 and ES2016 features using ES5 code. Because of this, you should always include the ES5 shim file when you use the ES6 or ES7 shim file.

The sham files shown here are optional, and contain ECMAScript features that can't be implemented with ES3. Thus, all these files do is keep your application from throwing errors when you use unsupported features, which may not be what you want. Since the sham files depend on the shim files, you must include a sham file after its corresponding shim file.

The URLS for the shim and sham files for ECMAScript compatibility

```
https://cdnjs.cloudflare.com/ajax/libs/es5-shim/4.5.7/es5-shim.min.js
https://cdnjs.cloudflare.com/ajax/libs/es5-shim/4.5.7/es5-sham.min.js

https://cdnjs.cloudflare.com/ajax/libs/es6-shim/0.34.2/es6-shim.min.js
https://cdnjs.cloudflare.com/ajax/libs/es6-shim/0.34.2/es6-sham.min.js

https://wzrd.in/standalone/es7-shim@latest
```

The difference between the shim and sham files

- The shim.js files are libraries that implement features in browsers that don't support them. A shim.js file can run without its associated sham.js file.
- The sham.js files are libraries that make sure new features don't throw errors in browsers that don't support them, even though the features can't be implemented in these older browsers and won't run as expected.
- A sham.js file requires an associated shim.js file and must be included after it.

Description

- A *polyfill file* is a JavaScript library that implements features in browsers that don't support them. This type of file is also called a *shim file*.
- The polyfill files for ES2015 and ES2016 only work for features, like Promises, that involve new objects or additions to existing objects.
- To make sure that new ECMAScript features work in older browsers, you can use the shim.js and sham.js files shown above.

Figure 1-2 How to use polyfill files for browser compatibility

How to use a transpiler

A *transpiler* is a program that takes code written in one language and converts it to another language. Transpilers for ES2015 or later (ES2015+) convert the ES2015+ code you write to its ES5 equivalent.

A transpiler lets you work with new ECMAScript features that are syntax changes, such as new operators or keywords. Some examples of these are the Spread/Rest operator, the let keyword, arrow functions, and some new shortcut syntax.

Babel is a popular ES2015+ transpiler. It also has a live transpiler page that lets you type or paste ES2015+ code and immediately see the ES5 code it converts to. The URLs for the Babel website and the Babel live transpiler page are shown at the top of figure 1-3.

Below these URLs is an example of the Babel live transpiler page. Here, an ES2015 arrow function is entered in the pane on the left side of the page, and its ES5 equivalent appears in the pane on the right side. When you use this live transpiler, the ES2015+ code that you enter is converted to ES5 as you type.

The live transpiler page is useful for prototyping small amounts of code or getting an idea of how some converted code will look. However, it isn't useful for converting large amounts of code. That's because you would need to repeatedly write code in ES2015+, copy and paste it into the live transpiler, and copy and paste the converted code into your application.

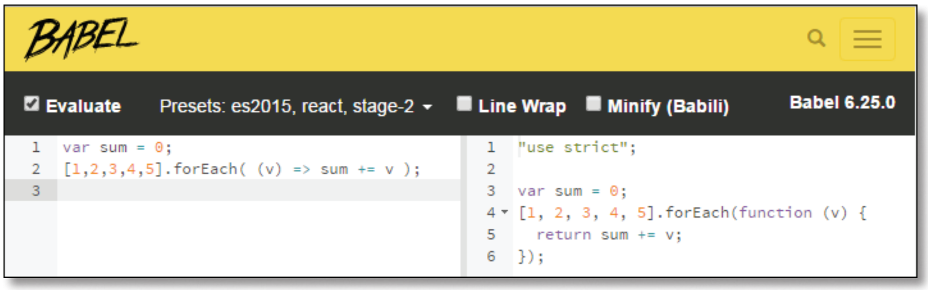
As a result, most programmers use Babel on their desktops. This, however, requires Node.js and the Node Package Manager (npm). To install these items, go to the URL shown in this figure. Then, go to the URL for the Babel CLI and install the Babel *command line interface*. You can also use other desktop tools with Babel. To learn more about them, go to the last URL in this figure.

Note that most transpilers, Babel included, convert code to ES5. As a result, you should always include the ES5 shim file when you use a transpiler.

The URLs for the Babel transpiler website and its live transpiler page

https://babeljs.io/
https://babeljs.io/repl

The Babel live transpiler page with some translated ES2015 code



```
var sum = 0;
[1,2,3,4,5].forEach( (v) => sum += v );
```

```
"use strict";

var sum = 0;
[1, 2, 3, 4, 5].forEach(function (v) {
  return sum += v;
});
```

The URL for installing Node.js and the Node Package Manager (npm)

https://docs.npmjs.com/getting-started/what-is-npm

The URL for installing the Babel CLI (Command Line Interface)

http://babeljs.io/docs/usage/cli/

The URL for installing and using other tools with Babel

http://babeljs.io/docs/setup/

Description

- A *transpiler* takes source code from one language and converts it to another language. This lets you use new syntax features, like operators and keywords.
- Transpilers that work with ES2015 or later (ES2015+) convert ES2015+ code to ES5 code.
- Babel is a popular transpiler with a live transpiler page that lets you see how your code will be translated. The example above shows how an ES2015 arrow function (see figure 1-11) is translated to ES5 code.
- Most of the time, you'll use Babel on your desktop. This requires the installation of Node.js and the Node Package Manager (npm).
- Many programmers also use other desktop tools to automate the process of using Babel to convert their ES2015+ code to ES5.

Figure 1-3 How to use a transpiler for browser compatibility

ECMAScript 2015 (ES6) object changes

ECMAScript 2015 (also known as ES2015 or ES6) was released in June 2015. Because it was the first release in several years, it had many new features. In fact, it had so many new features that not all of them are presented in this document. Rather, we focus on those we think are most relevant or commonly used. For a complete list of all the new features in ES2015, you can visit the URL at the top of figure 1-4.

The topics that follow present the ES2015 features that provide new objects or additions to existing objects. As a result, you need to use the polyfill files in figure 1-2 if you want them to work in browsers that don't support them.

Properties and methods of the Number and Math objects

The table in figure 1-4 summarizes four new methods of the Number object. Since these are static methods, you must use Number as the object name when you're using these methods.

The first group of examples shows how to use the Number.isNaN() method. It is similar to the global isNaN() method since both check whether a value is NaN. They can return different results, however, because Number.isNaN() doesn't convert the argument to a number before it does the checking. It just checks to make sure that the argument is a Number type.

The second group of examples shows how to use the Number.isFinite() method, which is similar to the global isFinite() method. However, it can return different results than the isFinite() method. Although both methods check to make sure that the value it receives is of the Number type, Number.isFinite() doesn't convert the value it's sent to a number before it does the checking.

The third group of examples shows how to use the isInteger() and isSafeInteger() methods. Since the Number data type can be used to represent either decimal or integer values, you can use Number.isInteger() method to make sure a Number value is an integer. The Number.isSafeInteger() method is similar, but it also makes sure the argument is within a specified range.

Next is a table that presents a new property of the Number object named EPSILON. This property returns the difference between 1 and the smallest value that can be represented as a Number. Since EPSILON is a static property, you always start by coding Number when you use it.

You can use the EPSILON value to correct for imprecise floating point results. If, for example, you add 0.1 and 0.2, the result is 0.30000000000000004. Then, if the difference between that result and the expected result (0.3) is less than Number.EPSILON, you can consider the values equal.

The last table in this figure and the examples that follow present two new methods of the Math object. The trunc() method removes the fractional part of a number, but unlike the Math.abs() method, it returns a negative value for a negative number. The sign() method indicates whether a number is positive, negative, or zero.

The URL for a complete list of the new features in ECMAScript 2015

```
http://es6-features.org/
```

Four methods of the Number object

Method	Description
`Number.isNaN(x)`	Returns a Boolean that indicates whether a value is NaN. Similar to the global isNaN() method but returns some different results.
`Number.isFinite(x)`	Returns a Boolean that indicates whether a value is finite. Similar to the global isFinite() method but returns some different results.
`Number.isInteger(x)`	Returns a Boolean that indicates whether a value is an integer.
`Number.isSafeInteger(x)`	Returns a Boolean that indicates whether a value is an integer in a range from $(2^{53} - 1)$ to $-(2^{53} - 1)$.

Example 1: The isNaN() method

```
var result_1a = Number.isNaN( NaN );        // true
var result_1b = Number.isNaN( null );       // false
var result_1c = Number.isNaN( "NaN" );      // false (isNaN returns true)
var result_1d = Number.isNaN( undefined );  // false (isNaN returns true)
```

Example 2: The isFinite() method

```
var result_2a = Number.isFinite( 123 );     // true
var result_2b = Number.isFinite( NaN );     // false
var result_2c = Number.isFinite( null );    // false (isFinite returns true)
var result_2d = Number.isFinite( "0" );     // false (isFinite returns true)
```

Example 3: The isInteger() and isSafeInteger() methods

```
var result_3a = Number.isInteger( 123 );                      // true
var result_3b = Number.isInteger( 123.45 );                   // false
var result_3c = Number.isSafeInteger( Math.pow(2,53) );       // false
var result_3d = Number.isSafeInteger( Math.pow(2,53) - 1 );   // true
```

One property of the Number object

Property	Description
`Number.EPSILON`	Returns 2.220446049250313e-16, which is the difference between 1 and the smallest value that can be represented as a Number.

Two methods of the Math object

Method / Operator	Description
`Math.trunc(x)`	Returns an integer with the fractional digits removed. If x is a positive number, equivalent to Math.floor(); otherwise, equivalent to Math.ceil().
`Math.sign(x)`	Returns 1, -1, 0, -0, or NaN, which indicates whether x is a positive number, negative number, positive zero, negative zero, or NaN.

Example 4: The trunc() and sign() methods

```
var result_4a = Math.trunc( 12.3 );         // result_4a is 12
var result_4b = Math.trunc( -12.3 );        // result_4b is -12
var result_4c = Math.sign( 6 );             // result_4c is 1
var result_4d = Math.sign( -3 );            // result_4d is -1
```

Figure 1-4 ES2015: Properties and methods of the Number and Math objects

Methods of the String object

Figure 1-5 summarizes four new methods of the String object. The first example shows how to use the repeat() method. This method accepts an integer between 0 and Infinity and returns a new string that is the product of concatenating the original string the number of times in the parameter. In this example, you can see that the string is repeated three times because 3 was passed to the repeat method.

The second example shows how to use the startsWith() method. This method accepts a string value and checks to see whether the string it's called on starts with that value. For instance, the first statement in the example checks to see if the string "Hello" starts with the string "H", while the second checks to see if that same string starts with the string "h". As you can see, the check is case-sensitive, so the first statement returns true while the second returns false.

The startsWith() method has an optional parameter that indicates the position in the string to start searching from. Otherwise, the default value is 0. In the third statement, the search starts with the character at position 3, which is "l". So in this statement, the method checks to see if the string "lo" starts with "lo". Since it does, the method returns true.

The third example shows how to use the endsWith() method. This method is similar to the startsWith() method, except it checks to see whether the string it's called on ends with the string it accepts. For instance, the first statement in the example checks to see if the string "Hello" ends with the string "o", while the second statement checks if that same string ends with the string "O". Since endsWith() is also case-sensitive, the first statement returns true while the second returns false. Then, the third statement checks to see if "Hello" ends with the string "lo". As you can see, this check return true.

The endWith() method also accepts an optional parameter that indicates where to search in the string. In this case, though, the number of the second parameter doesn't indicate the position of the character where the search should start. Rather, it indicates the length of the string to search, starting at the first character. Thus, in the third statement, the method is checking to see if the string "Hel" ends with "lo". Since it doesn't, the method returns false.

The fourth example shows how to use the includes() method. Like the startsWith() and endsWith() methods, this one accepts a string value to use in the search. This time, though, the method checks to see if that string exists anywhere within the string the method was called on. For instance, the first statement in this example checks to see if the string "ell" exists within the string "Hello". Since it does, this method returns true.

Like the startsWith() and endsWith() methods, the includes() method is case-sensitive. Thus, the second statement, which searches for the string "ELL" within the string "Hello", returns false.

The includes() method also has an optional parameter that indicates the position in the string to start searching from, with a default value of 0. So in the first and second statements, the entire string is searched. But the third statement checks to see whether the string "ello" includes the string "ell", and the fourth statement checks whether the string "llo" includes the string "ell".

Four methods of the String object

Method	Description
`repeat(n)`	Returns a new string that is *n* copies of the string concatenated together.
`startsWith(str, [start])`	Returns a Boolean indicating whether the string starts with the characters of the string in its first parameter. The second optional parameter indicates the position in the string to start searching from. The default is 0.
`endsWith(str, [length])`	Returns a Boolean indicating whether the string ends with the characters of the string in its first parameter. The second optional parameter indicates the length of the string to search, starting from the first character. The default is the string's actual length.
`includes(str, [start])`	Returns a Boolean indicating whether the characters of the string in its first parameter are found within the string. The second optional parameter indicates the position in the string to start searching from. The default is 0.

Example 1: The repeat() method

```
var hey = "Hey! ";
var result_1a = hey.repeat(3);    // result_1a is "Hey! Hey! Hey! "
```

Example 2: The startsWith() method

```
var result_2a = "Hello".startsWith("H");       // true
var result_2b = "Hello".startsWith("h");       // false
var result_2c = "Hello".startsWith("lo",3);    // true
```

Example 3: The endsWith() method

```
var result_3a = "Hello".endsWith("o");         // true
var result_3a = "Hello".endsWith("O");         // false
var result_3b = "Hello".endsWith("lo");        // true
var result_3c = "Hello".endsWith("lo", 3)      // false
```

Example 4: The includes() method

```
var result_4a = "Hello".includes("ell");       // true
var result_4b = "Hello".includes("ELL");       // false
var result_4c = "Hello".includes("ell", 1);    // true
var result_4d = "Hello".includes("ell", 2);    // false
```

Description

- The startsWith(), endsWith(), and includes() methods are case-sensitive.

Figure 1-5 ES2015: Methods of the String object

The find() method of the Array object

Figure 1-6 summarizes the find() method of the Array object. This method accepts a function, executes it once for each element in the array, and returns the first element that causes the function to return true. This method is similar to the filter() method of the Array object, but that method returns an array of all the values that return true, while the find() method only returns the first value that returns true. The find() method also has an optional second parameter that can be used to set the value of the function's *this* keyword.

You can see how this works in the first example in this figure. Here, a function named isOdd() accepts a value and returns true if the value is an odd number and false if it isn't. Then the find() method is called on several arrays with the isOdd() function as its parameter. In the first three arrays, there's more than one odd number, but the find() method only returns the first one it finds, working left to right. The last array doesn't contain any odd numbers, so the find() method returns undefined.

The assign() method of the Object object

Figure 1-6 also summarizes the assign() method of the Object object. This method accepts a target object and one or more source objects. It copies the properties of the source object or objects to the target object, and then returns the target object. If the target object and a source object have properties with the same names, the values of the source properties will override the values of the target properties. Similarly, if the source objects have properties with the same names, the later source properties will override the earlier ones.

The last example in this figure shows how this works. Here, a target object and two source objects are passed to the Object.assign() method, and the value returned by the assign() method is stored in a variable named newObj.

As you can see, both the target object and the source1 object have a property named one. Thus, the value of the one property in source1 overrides the value in target.

Similarly, both the source1 object and the source2 object have a property named five. Thus, the value of the five property in source2 overrides the value in source1. That's because source2 is passed to the assign() method after source1. If this order were reversed, the source1 value of the five property would override source2.

The find() method of the Array object

Method	Description
`find(function, [this])`	Accepts a function and executes it once for each element in an array until the function returns true. Then, it returns the value of that element. If no elements return true, it returns undefined. Accepts an optional second parameter that sets the value of the this keyword for the function.

How to use the find() method to retrieve a specific value from an array

An isOdd function that determines whether a value is an odd number

```
var isOdd = function(num) {
    return (num % 2 === 0) ? false: true;
};
```

Examples that use the find() method with the isOdd() function

```
var value = [1, 2, 3, 4].find(isOdd);    // value = 1
var value = [2, 3, 4, 5].find(isOdd);    // value = 3
var value = [5, 4, 3, 2].find(isOdd);    // value = 5
var value = [2, 4, 6, 8].find(isOdd);    // value = undefined
```

The assign() method of the Object object

Method	Description
`assign(target, sources...)`	Accepts a target object and one or more source objects, and copies the properties in the source objects to the target object. Then, it returns the target object.

How to assign properties from one or more objects to a target object

```
var target = { one: 1, two: 2 };
var source1 = { one: "one", three: 3, five: "five" };
var source2 = { five: 5 };
var newObj = Object.assign( target, source1, source2 );

console.log( newObj );    // Object {one: "one", two: 2, three: 3, five: 5}
```

Description

- The find() method of an Array object lets you retrieve an element in an array whose value meets specific criteria. If more than one element meets the criteria, the first one is returned. If no elements meet the criteria, undefined is returned.
- The assign() method of an Object object lets you copy the properties of one or more source objects to a target object. If there are duplicate property names, source properties will override target properties and later source properties will override earlier ones.

Figure 1-6 ES2015: New methods of the Array object and Object object

Promises

ECMAScript 2015 introduces a new feature called Promises, which allows you to designate functions that should run in response to the eventual success or failure of asynchronous code. Since this includes a lot of functionality and can be difficult to implement and understand, figure 1-7 provides just some basic information about using Promises. Then, if you want to learn more, you can consult the URLs listed at the bottom of this figure.

The first table in this figure details the three states that a Promise object (or *promise*) can be in. A Promise is *pending* in its initial state when it hasn't completed yet and thus hasn't succeeded or failed. For instance, a Promise that's waiting for the result of an AJAX call is pending. A Promise is *fulfilled* if its operations have completed successfully or *rejected* if its operations have failed.

To create a Promise object, you pass a function definition to the Promise constructor. The syntax for this is shown below the first table. The function that's passed to the Promise constructor will execute right away. Usually, though, it will start an asynchronous operation. Then, once the asynchronous operation completes, the function invokes one of the two callback functions it receives as parameters.

The second table in this figure describes the parameters of the function that's passed to the Promise constructor. The first parameter is a callback function that's invoked if the operation is successful, while the second is a callback function that's invoked if the operation fails.

After you create a Promise object, you can use the methods described in the third table. The then() method passes the callback functions that the Promise object will invoke when fulfilled or rejected. You can pass in only a function for fulfilled operations (the first parameter), or only a function for rejected operations (the second parameter), or functions for each. If you pass only a function for rejected operations, though, you need to pass null as a placeholder for the first parameter. So in a case like that it's better to use the catch() method, which only accepts a function for rejected operations.

Both the then() and catch() methods return a new Promise object, so they are chainable. In fact, a common use of a Promise object is to chain several then() methods and end with a catch() method. Then, the function passed to the catch() method will execute if any of the functions in the chained then() methods fail.

The example in this figure shows how this can work. Here, the Promise constructor is passed a function, which runs immediately. Then, the object returned by the Promise constructor is stored in a variable named promise. After that, two ways of working with the instance of the Promise object are presented. In the first, the then() method of the promise is called and passed a function to invoke on success and a function to invoke on failure. In the second, the then() method is only passed a success function, and it's chained with a catch() method that is passed a failure function.

The three states of a Promise object

State	Description
`pending`	The initial state of a promise. Not fulfilled or rejected.
`fulfilled`	The operation has completed successfully.
`rejected`	The operation has failed.

The syntax of the Promise constructor

```
new Promise( function(resolve, reject) { ... } );
```

The parameters of the function passed to the Promise constructor

Parameter	Description
`resolve`	A callback function that's invoked when the promise is fulfilled.
`reject`	A callback function that's invoked when the promise is rejected.

Two methods of the Promise prototype

Method	Description
`then(onFulfilled, onRejected)`	Passes the callback functions to be called on success or failure and returns a new promise.
`catch(onRejected)`	Passes the callback function to be called on failure and returns a new promise.

An example of creating and using a Promise object

```
var promise = new Promise(function(resolve, reject) {
    var isOK = callAsynchronousOperation();
    if( isOK ) {
        resolve("Yay!");
    } else {
        reject("Error");
    }
});
```

Handling success and failure in the then() method

```
promise.then(
    function(val) { alert( val ); },
    function(err) { alert( err ); }
);
```

Handling success in the then() method and failure in the catch() method

```
promise.then(function(val) {
    document.write(val);
}).catch(function(err) {
    document.write(err);
});
```

URLs where you can get more information about Promises

- https://promisesaplus.com/
- developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
- developers.google.com/web/fundamentals/getting-started/primers/promises

Figure 1-7 ES2015: The use of Promise objects

ECMAScript 2015 (ES6) syntax changes

The topics that follow present the new features of ES2015 that are syntax changes. Because they are syntax changes, you can't use polyfills to make them work with browsers that don't support them. However, you can use a transpiler, as shown in figure 1-3.

The syntax for destructuring an array or object

Figure 1-8 presents a new way to use the array literal syntax to load the elements of an array into individual variables. This is called *destructuring*.

The first example in this figure creates an array called names and adds three elements to it. Then, the second example uses the *destructuring assignment* syntax to load the elements in the names array into variables called name1, name2, and name3. Note that this code is functionally the same as the following:

```
var name1 = names[0];
var name2 = names[1];
var name3 = names[2];
```

The third example shows how you can use the destructuring syntax to retrieve only some of the elements of an array. Here, variables are provided in the assignment expression for the first and third elements, but a blank is left for the second. Thus, the first element is loaded in the variable name1, the second element is skipped, and the third element is loaded in name2. For this to work, you need to code a blank as a placeholder for the element you want to skip.

The fourth example shows how you can add default values to the variables in the assignment expression. Then, when the array is destructured, the values of the elements in the array replace the default values. But if there is no array element that corresponds to a variable, its default value remains. Here, there's one more variable than there are elements in the array, so the last variable, name4, doesn't have its default value replaced. Note that if name4 didn't have a default value, its value would be undefined after this line of code ran.

The fifth example shows how to use the assignment syntax to swap values in variables. The sixth example shows how to destructure a string into an array of its component characters. And the seventh example shows a destructuring assignment that retrieves month and year values from an array of date parts that's returned from a function.

So far, the examples show how to destructure arrays, but you can also use the object literal syntax to destructure an object. For instance, the last example in this figure creates an object literal with two properties, first and last. Then, the object literal syntax is used to load the values of the object's properties into variables named first and last. This code is functionally the same as the following:

```
var first = name.first;
var last = name.last;
```

Note that when you destructure an object, you enclose the assignments in braces, but you enclose the assignments in brackets when you destructure an array.

An array literal with three elements

```
var names = ["Grace", "Charles", "Ada"];
```

How to destructure the values of the array into individual variables

```
var [name1, name2, name3] = names;

console.log(name1);    // Grace
console.log(name2);    // Charles
console.log(name3);    // Ada
```

How to skip an array element as you destructure

```
var [name1, , name2] = names;

console.log(name1);    // Grace
console.log(name2);    // Ada
```

How to use default values for the variables

```
var [name1 = "1st", name2 = "2nd", name3 = "3rd", name4 = "4th"] = names;

console.log(name1);    // Grace
console.log(name2);    // Charles
console.log(name3);    // Ada
console.log(name4);    // 4th
```

How to swap variable values

```
var first = "Grace", last = "Hopper";
[first, last] = [last, first];

console.log(first);    // Hopper
console.log(last);     // Grace
```

How to destructure a string

```
var [a,b,c] = "USA";;

console.log(a);        // U
console.log(b);        // S
console.log(c);        // A
```

How to destructure the values of an array returned by a function

```
var getDateParts = function(dt) {
    var parts = [];
    parts.push( dt.getMonth() + 1 );          // month is zero based
    parts.push( dt.getDate() );
    parts.push( dt.getFullYear() );
    return parts;
};
// date is April 15, 2017
var [m, , y] = getDateParts( new Date() );    // m = 4, y = 2017
```

How to destructure an object

```
var name = {
    first: "Grace",
    last: "Hopper"
};
var { first, last } = name;    // first = "Grace", last = "Hopper"
```

Figure 1-8 ES2015: The syntax for destructuring an array or object

The Spread/Rest operator

The *Spread/Rest operator* is a new ES2015 operator, which is coded as three dots (…). This operator either expands an array into individual variables (Spread), or it collects individual variables into an array (Rest). What it does depends on where and how it is coded.

Figure 1-9 summarizes the use of this operator. Here, the first group of examples shows how this operator can be used to take the values in an array and "spread" them into individual variables. The first example in this group shows how you can use the Spread operator in the array literal syntax to create a new array from an existing array. This is functionally the same as using the slice() method of an array with no parameters.

One benefit of using the Spread operator in array literals is that you can easily combine existing arrays with individual elements, as shown in the second example. Here, two arrays and two individual numbers are combined to create a single array with seven numeric elements. As you can see, the Spread operator can be used multiple times and in multiple places within a single expression.

The Spread operator can also be used to pass the values in an array as parameters to a function. The next example illustrates how this works. It spreads the values of an array out to a list of elements and then passes that list to the push method. Although you can get the same effect with the concat() method of an array or with a for loop, this code is simpler.

The last example in this group shows how to use the Spread operator when creating a new Date object. Here, an array named dateparts contains the values for the optional year, month, and day parameters of the Date object constructor. Then, the Spread operator spreads the parameters out to their places in the constructor.

In contrast to the Spread operator, the Rest operator takes values in individual variables and collects them into an array. Because of this, you can use the Rest operator along with the syntax you learned in the last figure to destructure an array. The first example in the next group of examples shows how this works. Unlike the Spread operator, the Rest operator can only be used once per expression, and it must be the last parameter so it gets the "rest" of the values.

The Rest operator can also be used in the parameter list of a function. This is illustrated by the next example. Here, a function named myPetIs() accepts three parameters, the last of which is a rest parameter. Then, when the code that calls the myPetIs() function passes it five strings, the rest parameter gathers the last three in an array, and the function uses the join method of this array to build the return string.

The last example in this figure shows the use of both the Spread and Rest operators. Here, the Spread operator is used with an array of strings when the myPetIs() function is called. This spreads the elements into individual strings. Then, inside the function, the Rest operator collects the individual strings back into an array.

The Spread/Rest operator

Operator	Name	Description
...	Spread	Expand an array into individual variables. Can use the operator multiple times per expression and in any order.
...	Rest	Collapse individual variables into an array. Can use the operator only once per assignment or list, and it must be the last parameter.

How to use the Spread operator

Copy an array

```
var list = [1, 2, 3];
var newlist = [...list];                      // newlist = [1,2,3]
```

Use multiple Spread operators in an array literal

```
var list1 = [2, 3, 4];
var list2 = [6, 7];
var newlist = [1, ...list1, 5, ...list2];     // newlist = [1,2,3,4,5,6,7]
```

Call the push() method to add array elements to the end of an existing array

```
var list = [1, 2, 3];
list.push(...[4, 5, 6]);                      // list = [1,2,3,4,5,6]
```

Create a new Date object from an array of date parts

```
var dateparts = [2017, 3, 15];
var taxday = new Date(...dateparts);
console.log( taxday.toDateString() );         // Sat Apr 15 2017
```

How to use the Rest operator

Destructure an array

```
var list = [1, 2, 3, 4, 5];
var [first, ...rest] = list;                  // first = 1, rest = [2,3,4,5]

var [first, ...middle, last];                 // SyntaxError
```

A function that uses a Rest operator in its parameter list

```
var myPetIs = function(name, species, ...descriptors) {
   return name + " is a " + species + " who is " + descriptors.join(", ");
};
var str = myPetIs("Emma", "cat", "sweet", "furry", "a character");
// Emma is a cat who is sweet, furry, a character
```

How to use a Spread operator to pass an array to a function that uses a Rest operator

```
var myPetIs = function(name, species, ...descriptors) {
   return name + " is a " + species + " who is " + descriptors.join(", ");
};
var descriptors = ["smart", "loyal", "enthusiastic", "poorly behaved"];
var str = myPetIs("Chelsea", "dog", ...descriptors);
// Chelsea is a dog who is smart, loyal, enthusiastic, poorly behaved
```

Description

- The Spread and Rest operations both use the same three dot operator. Whether this operator is a Spread operator or a Rest operator depends on how it's used.

Figure 1-9 ES2015: The Spread/Rest operator

The let keyword

Variables that are declared outside a function have *global scope* so they are available to all functions. Conversely, variables that are declared inside a function have *local scope* so they are only available within the function.

Prior to ES2015, those were the only types of scope that were available. But now, ES2015 provides a *let keyword* that lets you create *block scope*. This means that the variable is only available within the block of code in which it's coded, like a while statement or a for loop.

This is illustrated by the first example in figure 1-10. Here, a for loop uses the let keyword to declare the counter variable i. As a result, the i variable has block scope and isn't available outside the for loop. That's why the last statement in this function throws an error.

The let keyword can also be used to correct a problem that can occur with closures inside loops. This is illustrated by the second example. Here, click event handlers are assigned within a loop that's inside the load event handler for the window.

If you use the *var* keyword to declare the counter in this loop, the click event handler that's created always displays the last element in the topSites array, no matter which link is clicked. Previously, you had to use an immediately invoked function expression (IIFE) to correct this problem. But with ES2015, you can make this code work simply by changing the *var* keyword to *let*, as shown here.

The const keyword

Variables declared with the var or let keyword can have their values re-assigned. But with ES2015, you can use the const keyword to create a *constant*. When you declare a constant, you must also assign its value, and its value can't ever be re-assigned. This is illustrated by the last example in this figure, which throws a TypeError when an attempt to re-assign the constant is made.

Note, however, that this works a little differently if the value that's assigned to a constant is an object. Then, the value of the object's properties and methods *can* be re-assigned, even though the object itself can't be.

How to use the let keyword to create block scope

```
var letLoop = function() {
    var sum = 0;
    for( let i = 0; i < 5; i++) {
        sum += i;
    }
    console.log( i );    // Reference error: i is not defined
};
```

How to use the let keyword to fix the closure loop problem

Assigning click event handlers in a loop

```
window.onload = function() {
    var topSites = ["Google", "Facebook", "Twitter"];
    var links = $("top_sites").getElementsByTagName("a");
    for (let i in topSites) {
        links[i].text = topSites[i];
        links[i].onclick = function() {
            alert("You clicked on " + topSites[i]);
        };
    }
};
```

How to use a constant

```
const TAXRATE = 0.87;
var tax = subtotal * TAXRATE;

TAXRATE = 0.75;          // TypeError: Assignment to constant variable.
```

Description

- The let keyword allows you to create *block scope* and fixes a problem with closures in loops.
- The const keyword allows you to create *constants*, also known as *immutable variables*. The value of a constant can't be changed once it's assigned. If the value of a constant is an object, though, the object's properties and methods can still be changed.

Figure 1-10 ES2015: The let and const keywords

Default values for the parameter list of a function

Figure 1-11 shows a new way to set default values for the parameters of a function. This is illustrated in the first example. Here, the second parameter for the calculateSalesTax() function is optional because it's set to a default value of 0.087. As a result, if the function is called with one argument, the taxRate variable is set to the default. But if the function is called with two arguments, the second argument overrides the default value for the taxRate variable.

Before ES2015, you could provide default values for a function by working with the arguments property of the function, or by testing whether a parameter value is undefined and providing a value if it is. This new syntax makes for code that's more concise, and thus easier to read and understand.

Arrow functions

Arrow functions make it easier to pass a function definition as an argument to a function or method. They do that by removing a lot of the boilerplate code of a function definition.

This is illustrated by the next two examples in figure 1-11. Both examples start with the traditional code for passing a function to a method. This code is followed by code that uses arrow functions to get the same results.

In the first of these examples, a function with one parameter is passed to the forEach() method of an array. When you pass an arrow function, the syntax starts with the parameter list in parentheses. This is followed by the *arrow operator* (`=>`), which is followed by the code in the body of the function. Also, if there's only one parameter, the parentheses around the parameter are optional. Either way, this is more concise than the traditional code so you can focus on the code in the function.

The last example illustrates the same idea but with two parameters. This time, a function is passed to the reduce() method of an array.

When you use arrow functions, you should know that they don't have arguments properties. Also, the body of an arrow function doesn't have its own *this* keyword value. Instead, the value of *this* in the body of an arrow function comes from the scope that contains the arrow function.

How to set default values for the parameters in a function

A function with a default value

```
function calculateSalesTax( amt, taxRate = 0.087 ) {
    return amt * taxRate;
};
```

A function call that uses the default value

```
var tax = calculateSalesTax( 100 );            // tax = 8.7
```

A function call that overrides the default value

```
var tax = calculateSalesTax( 100 , 0.075 );    // tax = 7.5
```

How to pass a function with one parameter to a method

Traditional code that passes a function to the forEach() method of an array

```
var sum = 0;
[1,2,3,4,5].forEach( function(v) {
    sum += v
} );                       // sum = 15
```

Code that passes an arrow function to the forEach() method of an array

```
var sum = 0;
[1,2,3,4,5].forEach( (v) => sum += v );
```

An arrow function that omits the optional parentheses for a single parameter

```
[1,2,3,4,5].forEach( v => sum += v );
```

How to pass a function with two parameters to a method

Traditional code that passes a function to the reduce() method of an array

```
var sum = [1,2,3,4,5].reduce( function(prev, current) {
    return prev + current;
} );                       // sum = 15
```

Code that passes an arrow function to the reduce() method of an array

```
var sum = [1,2,3,4,5].reduce( (prev, current) => prev + current );
```

Description

- To add a default value to a parameter in the parameter list of a function, code the equals sign and a value after the parameter name. Also, it's a good practice to code the parameters with default values last in the list.
- To code an *arrow function*, code the parameter list in parentheses, the *arrow operator* (`=>`), and the statements of the function. This makes it easier to pass function definitions as arguments to a function or method.
- When the parameter list only has one parameter, the parentheses around it can be dropped.
- An arrow function doesn't have an arguments property, and it gets the value of its *this* keyword from the scope that contains it.

Figure 1-11 ES2015: Default parameter values and arrow functions

The shortcut syntax for object literals

The shortcut syntax for object literals provides an easier way to add a method to an object literal. This is illustrated in the first set of examples in figure 1-12. Here, the methods of a tasklist object are first coded in the usual way. Then, they're coded with the new shortcut syntax, which removes the function keyword and the colon.

The shortcut syntax for immediately invoked function expressions (IIFEs)

Figure 1-12 also presents some new syntax for working with immediately invoked function expressions (IIFEs). Here, the first set of examples shows how to call an IIFE to create block scope in a function. To do that, you place the code that you want to have block scope inside two curly braces. This tells the JavaScript engine to execute the code inside the braces. Thus, the second example in this set works the same as the first example.

You should know, however, this curly braces technique only works if the IIFE doesn't need to accept any arguments when it's invoked. If it does, you can use the syntax shown in the second set of examples in this figure. Here, the arrow syntax that you learned about in figure 1-11 is used to simplify the IIFE call. Thus, the second example in this set works the same as the first example.

Remember that you can also use the let keyword to create block scope, as shown in figure 1-10.

The shortcut syntax for coding a method in an object literal

Traditional syntax

```
var tasklist = {
    sort: function() { /* sort code goes here */ },
    display: function(div) { /* display code goes here */ }
};
```

Shortcut syntax

```
var tasklist = {
    sort() { /* sort code goes here */ },
    display(div) { /* display code goes here */ }
};
```

How to use the shortcut syntax to invoke an IIFE

A traditional IIFE

```
var blockScopeLoop = function() {
    var arr = [];
    (function() {
        for (var i = 0; i < 5; i++) {
            arr.push(i);
        }
    })();
};
```

Shortcut syntax for an IIFE

```
var es6blockScopeLoop = function() {
    var arr = [];
    {
        for (var i = 0; i < 5; i++) {
            arr.push(i);
        }
    }
};
```

How to use the shortcut syntax with IIFEs that accept arguments

Traditional code that invokes an IIFE and passes an argument

```
(function(mod) {
    mod.changeSpeed = function(speed) {
        /* code goes here */
    };
})(myapp.slideshow);
```

Code that uses an arrow function to invoke an IIFE and pass an argument

```
(mod => {
    mod.changeSpeed = function(speed) {
        /* code goes here */
    };
})(myapp.slideshow);
```

Description

- The shortcut syntax for object literals makes it easier to add methods to them.
- The shortcut syntax for IIFEs simplifies the code for working with them.

Figure 1-12 ES2015: The shortcut syntax for object literals and IIFEs

The new syntax for creating and using objects

The new syntax for creating and using objects looks more like the syntax of a classical language in which you declare *classes* that contain *constructors* that are used to create *objects*. That makes the code easier to understand. However, the objects being created are still prototypal, and the methods are still being added to the object's prototype.

This new syntax is illustrated by the first example in figure 1-13. Here, a Percent object is coded in two ways: first as a *class declaration*, then as a *class expression*. Both examples get the same results so in both cases, a Percent class is created that contains a constructor that receives a rate parameter. When this constructor is called, as in the third part of this example, a Percent *object* (or *instance* of the class) is created.

In this example, the constructor creates a Percent object that has a getPercent() method that receives a subtotal parameter. After you create a Percent object, you can call this method to get the percent of the subtotal. This is illustrated by the last part of this example.

To *inherit* methods from a *base* (or *parent*) *class*, you can use the new *extends* and *super* keywords to create *derived* (or *child*) *classes*. This is illustrated by the second example in this figure. Here, the *extends* keyword is used to create a derived class called Commission that inherits the methods of the Percent class. Then, in the constructor method for the Commission class, the *super* keyword is used to call the constructor of the base class. This is important, because it initializes the base class. If you don't do this, you won't be able to use the *this* keyword in your derived class.

In a derived class, you can also use the *super* keyword to the access the properties and methods of the base class. You can see this in this second example, where the Commission class overrides the getPercent() method of the Percent class. To do that, it first uses the super keyword to call the getPercent() method of the Percent class. Then, it divides the value returned by the method in the parent class by 2 if the isSplit property is true.

As mentioned earlier, the methods you add in a class declaration or expression are added to the object's prototype. That means these methods can be accessed by a derived class. The properties you add in the constructor, by contrast, aren't added to the object's prototype. However, when the base constructor is called, any properties coded in that constructor are created. Thus, these properties can be accessed by a derived class as well.

You should also know that you don't have to explicitly code a constructor method when you create a class. If you don't, this definition is created by default:

```
constructor() {}
```

Similarly, you don't need to explicitly code a constructor method in a derived class. If you don't, one will be added that calls the base constructor function and passes any arguments that it needs.

How to declare a class

A class declaration

```
class Percent {
    constructor( rate ) {
        this.rate = rate
    }
    getPercent(subtotal) {
        return subtotal * this.rate;
    }
};
```

A class expression

```
var Percent = class {
    constructor( rate ) {
        this.rate = rate
    }
    getPercent(subtotal) {
        return subtotal * this.rate;
    }
};
```

How to create an instance (object) of the Percent class

```
var percent = new Percent( 0.1 );
```

How to call the getPercent() method of the object

```
var salesrepPercent = percent.getPercent( 2500.00 );       // returns 250
```

How to inherit methods from a class

A class declaration

```
class Commission extends Percent {              // inherits the Percent class
    constructor( rate, isSplit ) {
        super(rate);                            // calls the parent constructor
        this.isSplit = isSplit
    }
    getPercent(subtotal) {
        var percent = super.getPercent(subtotal);     // call parent method
        return (this.isSplit) ? percent / 2 : percent;
    }
};
```

How to create an instance (object) of the Commission class

```
var commission = new Commission( 0.1, true );
```

How to call the getPercent() method of the object

```
var salesrepPercent = commission.getPercent( 2500.00 );    // returns 125
```

Description

- The *class*, *constructor*, *extends*, and *super* keywords let you declare *classes* and *inherit* methods.
- This makes JavaScript similar to a classical language. However, the objects are still prototypal, and the methods are still added to the object's prototype.

Figure 1-13 ES2015: The new syntax for creating and using objects

ECMAScript 2016 (ES7)

ECMAScript 2016 (ES2016, also known as ES7) was released in June 2016. Because it was released so soon after ES2015, it contains only the three new features that are summarized in figure 1-14.

The includes() method of the Array object

The includes() method accepts a value and then tests to see whether the array contains, or includes, that value. This is similar to using the indexOf() method of an array to look for the index of a value. However, the includes method returns true or false instead of an index. Since the includes() method is an addition to an existing object, you can use the shim file in figure 1-2 to support it in older browsers.

You can see how the includes() method works in first example in this figure. Here, an array named list is created, and the first two statements call the includes() method to see whether the array contains the values that are passed to it.

The next two statements show how the optional second parameter works. This parameter sets the index at which the search should begin. For example, the third statement says to begin the search for the value 3 at index 3, which is the fourth element in the array. Since this element contains a value of 4, this statement returns false. The fourth statement shows that if you tell the includes() method to start at an index that doesn't exist, it will return false rather than throw an error.

The includes() method also differs from the indexOf() method in the way it handles the value NaN. This is illustrated by the statements in the second example. Here, NaN is added to the array using the push() method. Then, the includes() method is used to see if the array contains NaN, and it returns true. In contrast, the indexOf() method returns -1, which means that value wasn't found.

The exponentiation operator

With ES2016, the *exponentiation operator* (**) can be used to raise a value to a "power of", as shown in this figure. In other words, this operator can be used in place of the Math.pow() method. Since this is a syntax change, you need to use a transpiler as shown in figure 1-3 for browsers that don't support it.

Changes to the "use strict" directive

With ES5, the "use strict" directive can be placed either at the top of a file or as the first line of code in a function. However, if your function uses default parameter values or destructuring, ES2017 says you need to place the "use strict" directive at the top of the file. If you start a function that accepts non-simple parameters with a "use strict" directive, it will throw a syntax error.

The includes() method of the Array object

Method	Description
`includes(`***`search, index`***`)`	Accepts a value and tests whether any elements in the array contain that value. Returns a Boolean value that indicates the result of the test. Accepts an optional second parameter that sets the index in the array from which to start the search.

How to use the includes() method to test an array for a specific value

```
var list = [1, 2, 3, 4];
var value = list.includes(5);           // value = false
var value = list.includes(3);           // value = true

var value = list.includes(3, 3);        // value = false
var value = list.includes(4, 8);        // value = false
```

How to use the includes() method to test for NaN

```
list.push(NaN);
var value = list.includes(NaN);         // value = true
var value = (list.indexOf(NaN) > -1);   // value = false
```

The exponentiation operator (**)

```
var result = 2 ** 3;                    // result = 8
// same as Math.pow(2,3)
```

The strict directive

- When used with functions that have non-simple parameters, such as default values or destructuring, the “use strict” directive must be at the top of the file, not the first line of code for the function.

Description

- The includes() method lets you test whether the elements of an array contain a specific value. Since this is an addition to an existing object, you need to use the polyfill file presented in figure 1-2 for browsers that don’t support it.
- The exponentiation operator is a shortcut for the Math.pow() method. Since this is a syntax change, you need to use a transpiler as shown in figure 1-3 for browsers that don’t support it.

Figure 1-14 ES2016: The three new features

The Internationalization API

The *ECMAScript Internationalization API Specification* is sometimes included as a new feature of ES2015. However, it's actually a separate specification that complements the ECMAScript Language Specification.

In the topics that follow, you'll learn how to use the Internationalization API to format numbers, currency, and dates in a language-sensitive way. First, though, you'll learn about the codes that the Internationalization API uses to do its work, and you'll learn how to include the API in your projects.

The codes used by the API

When you use this API, you'll use codes to identify the locales, countries, and currencies that you want to work with. A few of these codes are summarized in the tables in figure 1-15. To get the right codes for the locales and currencies that you're working with, you can search the websites listed in this figure. In the next figure, you'll see examples of how these codes are used.

How to include the API in your applications

Since the Internationalization API is separate from the ES2015 specification, it isn't part of the ES2015 shim that you learned about in figure 1-2. As a result, you need to use a different polyfill file if you want to use the Internationalization API with non-compliant browsers.

Figure 1-15 presents two examples of URLs for the polyfill files that you need to include with script statements. After the URL, you use language codes to specify the locale or locales that you want to support. For instance, the first URL specifies just English, while the second specifies English and French.

The codes used by the Internationalization API

- The Internationalization API uses the IETF Best Current Practice (BCP) 47 standard for languages and countries/regions, and the ISO 4217 standard for currencies.
- The language and country codes of the BCP 47 standard are also used in HTML, CSS, and other protocols.

Some of the language codes

Code	Language	Code	Language
`en`	English	`es`	Spanish, Castilian
`fr`	French	`it`	Italian
`de`	German	`zh`	Chinese

Some of the country codes

Code	Country/Region	Code	Country/Region
`US`	United States	`AU`	Australia
`GB`	United Kingdom	`MX`	Mexico
`DE`	Germany	`GT`	Guatemala

Some of the currency codes

Code	Currency	Code	Currency
`USD`	U.S dollar	`CAD`	Canadian dollar
`GBP`	British pound	`AUD`	Australian dollar
`EUR`	Euro	`MXN`	Mexican peso

Websites that list all the codes

- Language codes: http://www.loc.gov/standards/iso639-2/php/code_list.php
- Country codes: http://www.nationsonline.org/oneworld/country_code_list.htm
- Currency codes: https://www.iban.com/currency-codes.html

URLs for the Internationalization API polyfill

The URL with one locale

```
https://cdn.polyfill.io/v2/polyfill.min.js?features=Intl.~locale.en
```

With two locales

```
https://cdn.polyfill.io/v2/polyfill.min.js?features=Intl.~locale.en,
Intl.~locale.fr
```

Description

- The *ECMAScript Internationalization API Specification* allows you to format numbers, currency, and dates in language-sensitive ways.
- To do the formatting, you use the *language* and *country* codes provided by the BCP 47 standards and the currency codes provided by the ISO 4217 standards.

Figure 1-15 Internationalization API: Language, country, and currency codes

How to format numbers, currency, and dates

The Internationalization API has a single object named Intl. This object has constructors that are used to create objects that do language-sensitive formatting. Two of these constructors are illustrated in figure 1-16.

To format numbers with this API, you create a NumberFormat object by passing a language and country code to the NumberFormat constructor, as shown in this figure. Although you could pass just a language code, it's usually better to be more precise. After you create the NumberFormat object, you use its format() method to format numbers for that language. This is illustrated by the first group of examples.

To format currency with this API, you use a similar procedure. However, you also pass a second parameter to the constructor. That parameter is an object that specifies currency formatting along with a currency code. This is illustrated by the second group of examples.

To format dates, you use the same general procedure, but you use the DateTimeFormat constructor to create a DateTimeFormat object. Then, you use the format() method of that object to format dates. This is illustrated by the third group of examples.

At the bottom of this figure is a URL for the documentation of the Intl object. This is a good resource for more information on working with this API.

Two of the constructor functions of the Intl object

Constructor	Description
`NumberFormat(`*`langCode`*`, [`*`styleObject`*`])`	Formats numbers and currency.
`DateTimeFormat(`*`langCode`*`)`	Formats dates.

How to use the NumberFormat and DateTimeFormat constructors

Example 1: How to use NumberFormat objects to format numbers

```
var us = new Intl.NumberFormat("en-US");
var de = new Intl.NumberFormat("de-DE");
var result_1a = us.format(1234567.89);      // 1,234,567.89
var result_1b = de.format(1234567.89);      // 1.234.567,89
```

Example 2: How to use NumberFormat objects to format currency

```
var us = new Intl.NumberFormat("en-US", {style:"currency", currency:"USD"});
var gb = new Intl.NumberFormat("en-GB", {style:"currency", currency:"GBP"});
var de = new Intl.NumberFormat("de-DE", {style:"currency", currency:"EUR"});
var result_2a = us.format(100200300.40);    // $100,200,300.40
var result_2b = gb.format(100200300.40);    // £100,200,300.40
var result_2c = de.format(100200300.40);    // 100.200.300,40
```

Example 3: How to use DateTimeFormat objects to format dates

```
var dt = new Date("7/4/2017");
var us = new Intl.DateTimeFormat("en-US");
var de = new Intl.DateTimeFormat("de-DE");
var result_3a = us.format(dt);              // 7/4/2017
var result_3b = de.format(dt);              // 4.7.2017
```

Documentation for the Intl object

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl

Description

- The Internationalization API provides a single Intl object with constructors that let you create objects that do the formatting.

Figure 1-16 Internationalization API: How to format numbers, currency, and dates

Perspective

Now that you've completed this document, you can decide which of the new features you want to add to your programming skillset. Although you may never find the need for some of these features, many of them will help you write code that's shorter and easier to understand. At the least, you can keep this document as a reference so you can refer to it whenever you want to refresh your memory about what's available and how it works.

Terms

ECMAScript specification
ECMAScript 2015 (ES2015)
ECMAScript 2016 (ES2017)
polyfill file
shim file
sham file
transpiler
promise
pending promise
fulfilled promise
rejected promise
destructure
destructuring assignment
Spread/Rest operator
global scope
local scope
block scope
let keyword
constant
immutable variable
arrow function
arrow operator
class
constructor
object
instance of a class
class declaration
class expression
inherit methods
base class
parent class
derived class
child class
exponentiation operator
Internationalization API
language code
country code
currency code

Mike Murach, Publisher