4. Core ES6 features

This chapter describes the core ES6 features. These features are easy to adopt; the remaining features are mainly of interest to library authors. I explain each feature via the corresponding ES5 code.

4.1 From `var` to `const`/`let`

In ES5, you declare variables via `var`. Such variables are *function-scoped*, their scopes are the innermost enclosing functions. The behavior of `var` is occasionally

4. Core ES6 features
Table of contents
Please support this book: buy it (PDF, EPUB, MOBI) or donate

```
function func(randomize) {
    if (randomize) {
        var x = Math.random(); // (A) scope: whole function
        return x;
    }
    return x; // accesses the x from line A
}
func(false); // undefined
```

That `func()` returns `undefined` may be surprising. You can see why if you rewrite the code so that it more closely reflects what is actually going on:

```
var x = 3;
function func(randomize) {
    var x;
    if (randomize) {
        x = Math.random();
        return x;
    }
    return x;
}
func(false); // undefined
```

In ES6, you can additionally declare variables via `let` and `const`. Such variables are *block-scoped*, their scopes are the innermost enclosing blocks. `let` is roughly a block-scoped version of `var`. `const` works like `let`, but creates variables whose values can't be changed.

`let` and `const` behave more strictly and throw more exceptions (e.g. when you access their variables inside their scope before they are declared). Block-scoping helps with keeping the effects of code fragments more local (see the next section for a demonstration). And it's more mainstream than function-scoping, which eases moving between JavaScript and other programming languages.

If you replace `var` with `let` in the initial version, you get different behavior:

```
let x = 3;
function func(randomize) {
    if (randomize) {
        let x = Math.random();
        return x;
    }
    return x;
}
func(false); // 3
```

That means that you can't blindly replace `var` with `let` or `const` in existing code; you have to be careful during refactoring.

My advice is:

- Prefer `const`. You can use it for all variables whose values never change.
- Otherwise, use `let` – for variables whose values do change.
- Avoid `var`.

More information: chapter "Variables and scoping".

Expression) if you wanted to restrict the scope of a variable `tmp` to a block:

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

console.log(tmp); // ReferenceError
```

In ECMAScript 6, you can simply use a block and a `let` declaration (or a `const` declaration):

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

console.log(tmp); // ReferenceError
```

More information: section “Avoid IIFEs in ES6”.

4.3 From concatenating strings to template literals

With ES6, JavaScript finally gets literals for string interpolation and multi-line strings.

4.3.1 String interpolation

In ES5, you put values into strings by concatenating those values and string fragments:

```
function printCoord(x, y) {
    console.log('('+x+', '+y+')');
}
```

In ES6 you can use string interpolation via template literals:

```
function printCoord(x, y) {
    console.log(`(${x}, ${y})`);
}
```

4.3.2 Multi-line strings

Template literals also help with representing multi-line strings.

For example, this is what you have to do to represent one in ES5:

```
var HTML5_SKELETON =
    '<!doctype html>\n' +
    '<html>\n' +
    '<head>\n' +
    '    <meta charset="UTF-8">\n' +
    '    <title></title>\n' +
    '</head>\n' +
    '<body>\n' +
    '</body>\n' +
    '</html>\n';
```

```
var HTML5_SKELETON = `\
    <!doctype html>\n\
    <html>\n\
    <head>\n\
        <meta charset="UTF-8">\n\
        <title></title>\n\
    </head>\n\
    <body>\n\
    </body>\n\
    </html>';
```

ES6 template literals can span multiple lines:

```
const HTML5_SKELETON = `
    <!doctype html>
    <html>
    <head>
        <meta charset="UTF-8">
        <title></title>
    </head>
    <body>
    </body>
    </html>`;
```

(The examples differ in how much whitespace is included, but that doesn't matter in this case.)

More information: chapter "Template literals and tagged templates".

4.4 From function expressions to arrow functions

In current ES5 code, you have to be careful with `this` whenever you are using function expressions. In the following example, I create the helper variable `_this` (line A) so that the `this` of `UiComponent` can be accessed in line B.

```
function UiComponent() {
    var _this = this; // (A)
    var button = document.getElementById('myButton');
    button.addEventListener('click', function () {
        console.log('CLICK');
        _this.handleClick(); // (B)
    });
}
UiComponent.prototype.handleClick = function () {
    ...
};
```

In ES6, you can use arrow functions, which don't shadow `this` (line A):

```
function UiComponent() {
    var button = document.getElementById('myButton');
    button.addEventListener('click', () => {
        console.log('CLICK');
        this.handleClick(); // (A)
    });
}
```

Arrow functions are especially handy for short callbacks that only return results of expressions.

In ES5, such callbacks are relatively verbose:

```
var arr = [1, 2, 3];
var squares = arr.map(function (x) { return x * x });
```

In ES6, arrow functions are much more concise:

```
const arr = [1, 2, 3];
const squares = arr.map(x => x * x);
```

When defining parameters, you can even omit parentheses if the parameters are just a single identifier. Thus: `(x) => x * x` and `x => x * x` are both allowed.

More information: chapter "Arrow functions".

4.5 Handling multiple return values

Some functions or methods return multiple values via arrays or objects. In ES5, you always need to create intermediate variables if you want to access those values. In ES6, you can avoid intermediate variables via destructuring.

4.5.1 Multiple return values via arrays

`exec()` returns captured groups via an Array-like object. In ES5, you need an intermediate variable (`matchObj` in the example below), even if you are only interested in the groups:

```
var matchObj =
    /^(\d\d\d\d)-(\d\d)-(\d\d)$/
    .exec('2999-12-31');
var year = matchObj[1];
var month = matchObj[2];
var day = matchObj[3];
```

In ES6, destructuring makes this code simpler:

```
const [, year, month, day] =
    /^(\d\d\d\d)-(\d\d)-(\d\d)$/
    .exec('2999-12-31');
```

The empty slot at the beginning of the Array pattern skips the Array element at index zero.

4.5.2 Multiple return values via objects

The method `Object.getOwnPropertyDescriptor()` returns a *property descriptor*, an object that holds multiple values in its properties.

In ES5, even if you are only interested in the properties of an object, you still need an intermediate variable (`propDesc` in the example below):

```
var configurable = propDesc.configurable;

console.log(writable, configurable); // true true
```

In ES6, you can use destructuring:

```
const obj = { foo: 123 };

const {writable, configurable} =
    Object.getOwnPropertyDescriptor(obj, 'foo');

console.log(writable, configurable); // true true
```

`{writable, configurable}` is an abbreviation for:

```
{ writable: writable, configurable: configurable }
```

More information: chapter “Destructuring”.

4.6 From `for` to `forEach()` to `for-of`

Prior to ES5, you iterated over Arrays as follows:

```
var arr = ['a', 'b', 'c'];
for (var i=0; i<arr.length; i++) {
    var elem = arr[i];
    console.log(elem);
}
```

In ES5, you have the option of using the Array method `forEach()`:

```
arr.forEach(function (elem) {
    console.log(elem);
});
```

A `for` loop has the advantage that you can break from it, `forEach()` has the advantage of conciseness.

In ES6, the `for-of` loop combines both advantages:

```
const arr = ['a', 'b', 'c'];
for (const elem of arr) {
    console.log(elem);
}
```

If you want both index and value of each array element, `for-of` has got you covered, too, via the new Array method `entries()` and destructuring:

```
for (const [index, elem] of arr.entries()) {
    console.log(index+'. '+elem);
}
```

More information: Chap. “The `for-of` loop”.

4.7 Handling parameter default values

```
    x = x || 0;
    y = y || 0;
    ...
}
```

ES6 has nicer syntax:

```
function foo(x=0, y=0) {
    ...
}
```

An added benefit is that in ES6, a parameter default value is only triggered by `undefined`, while it is triggered by any falsy value in the previous ES5 code.

More information: section “Parameter default values”.

4.8 Handling named parameters

A common way of naming parameters in JavaScript is via object literals (the so-called *options object pattern*):

```
selectEntries({ start: 0, end: -1 });
```

Two advantages of this approach are: Code becomes more self-descriptive and it is easier to omit arbitrary parameters.

In ES5, you can implement `selectEntries()` as follows:

```
function selectEntries(options) {
    var start = options.start || 0;
    var end = options.end || -1;
    var step = options.step || 1;
    ...
}
```

In ES6, you can use destructuring in parameter definitions and the code becomes simpler:

```
function selectEntries({ start=0, end=-1, step=1 }) {
    ...
}
```

4.8.1 Making the parameter optional

To make the parameter `options` optional in ES5, you’d add line A to the code:

```
function selectEntries(options) {
    options = options || {}; // (A)
    var start = options.start || 0;
    var end = options.end || -1;
    var step = options.step || 1;
    ...
}
```

In ES6 you can specify `{}` as a parameter default value:

More information: section “Simulating named parameters”.

4.9 From `arguments` to rest parameters

In ES5, if you want a function (or method) to accept an arbitrary number of arguments, you must use the special variable `arguments`:

```
function logAllArguments() {
    for (var i=0; i < arguments.length; i++) {
        console.log(arguments[i]);
    }
}
```

In ES6, you can declare a rest parameter (`args` in the example below) via the `...` operator:

```
function logAllArguments(...args) {
    for (const arg of args) {
        console.log(arg);
    }
}
```

Rest parameters are even nicer if you are only interested in trailing parameters:

```
function format(pattern, ...args) {
    ...
}
```

Handling this case in ES5 is clumsy:

```
function format(pattern) {
    var args = [].slice.call(arguments, 1);
    ...
}
```

Rest parameters make code easier to read: You can tell that a function has a variable number of parameters just by looking at its parameter definitions.

More information: section “Rest parameters”.

4.10 From `apply()` to the spread operator (`...`)

In ES5, you turn arrays into parameters via `apply()`. ES6 has the spread operator for this purpose.

4.10.1 `Math.max()`

`Math.max()` returns the numerically greatest of its arguments. It works for an arbitrary number of arguments, but not for Arrays.

ES5 – `apply()`:

```
> Math.max.apply(Math, [-1, 5, 11, 3])
11
```

```
11
```

4.10.2 `Array.prototype.push()`

`Array.prototype.push()` appends all of its arguments as elements to its receiver. There is no method that destructively appends an Array to another one.

ES5 – `apply()`:

```
var arr1 = ['a', 'b'];
var arr2 = ['c', 'd'];

arr1.push.apply(arr1, arr2);
    // arr1 is now ['a', 'b', 'c', 'd']
```

ES6 – spread operator:

```
const arr1 = ['a', 'b'];
const arr2 = ['c', 'd'];

arr1.push(...arr2);
    // arr1 is now ['a', 'b', 'c', 'd']
```

More information: section "The spread operator (. . .)".

4.11 From `concat()` to the spread operator (. . .)

The spread operator can also (non-destructively) turn the contents of its operand into Array elements. That means that it becomes an alternative to the Array method `concat()`.

ES5 – `concat()`:

```
var arr1 = ['a', 'b'];
var arr2 = ['c'];
var arr3 = ['d', 'e'];

console.log(arr1.concat(arr2, arr3));
    // [ 'a', 'b', 'c', 'd', 'e' ]
```

ES6 – spread operator:

```
const arr1 = ['a', 'b'];
const arr2 = ['c'];
const arr3 = ['d', 'e'];

console.log([...arr1, ...arr2, ...arr3]);
    // [ 'a', 'b', 'c', 'd', 'e' ]
```

More information: section "The spread operator (. . .)".

4.12 From function expressions in object literals to method definitions

In JavaScript, methods are properties whose values are functions.

```
var obj = {
    foo: function () {
        ...
    },
    bar: function () {
        this.foo();
    }, // trailing comma is legal in ES5
}
```

ES6 has *method definitions*, special syntax for creating methods:

```
const obj = {
    foo() {
        ...
    },
    bar() {
        this.foo();
    },
}
```

More information: section “Method definitions”.

4.13 From constructors to classes

ES6 classes are mostly just more convenient syntax for constructor functions.

4.13.1 Base classes

In ES5, you implement constructor functions directly:

```
function Person(name) {
    this.name = name;
}
Person.prototype.describe = function () {
    return 'Person called '+this.name;
};
```

In ES6, classes provide slightly more convenient syntax for constructor functions:

```
class Person {
    constructor(name) {
        this.name = name;
    }
    describe() {
        return 'Person called '+this.name;
    }
}
```

Note the compact syntax for method definitions – no keyword `function` needed. Also note that there are no commas between the parts of a class.

4.13.2 Derived classes

Subclassing is complicated in ES5, especially referring to super-constructors and super-properties. This is the canonical way of creating a sub-constructor `Employee` of `Person`:

```
Employee.prototype = Object.create(Person.prototype);
Employee.prototype.constructor = Employee;
Employee.prototype.describe = function () {
    return Person.prototype.describe.call(this) // super.describe()
           + ' (' + this.title + ')';
};
```

ES6 has built-in support for subclassing, via the `extends` clause:

```
class Employee extends Person {
    constructor(name, title) {
        super(name);
        this.title = title;
    }
    describe() {
        return super.describe() + ' (' + this.title + ')';
    }
}
```

More information: chapter “Classes”.

4.14 From custom error constructors to subclasses of `Error`

In ES5, it is impossible to subclass the built-in constructor for exceptions, `Error`. The following code shows a work-around that gives the constructor `MyError` important features such as a stack trace:

```
function MyError() {
    // Use Error as a function
    var superInstance = Error.apply(null, arguments);
    copyOwnPropertiesFrom(this, superInstance);
}
MyError.prototype = Object.create(Error.prototype);
MyError.prototype.constructor = MyError;

function copyOwnPropertiesFrom(target, source) {
    Object.getOwnPropertyNames(source)
    .forEach(function(propKey) {
        var desc = Object.getOwnPropertyDescriptor(source, propKey);
        Object.defineProperty(target, propKey, desc);
    });
    return target;
};
```

In ES6, all built-in constructors can be subclassed, which is why the following code achieves what the ES5 code can only simulate:

```
class MyError extends Error {
}
```

More information: section “Subclassing built-in constructors”.

4.15 From objects to Maps

Using the language construct *object* as a map from strings to arbitrary values (a data structure) has always been a makeshift solution in JavaScript. The safest way to do

The following ES5 code contains the function `countWords` that uses the object `dict` as a map:

```
var dict = Object.create(null);
function countWords(word) {
    var escapedWord = escapeKey(word);
    if (escapedWord in dict) {
        dict[escapedWord]++;
    } else {
        dict[escapedWord] = 1;
    }
}
function escapeKey(key) {
    if (key.indexOf('__proto__') === 0) {
        return key+'%';
    } else {
        return key;
    }
}
```

In ES6, you can use the built-in data structure `Map` and don't have to escape keys. As a downside, incrementing values inside Maps is less convenient.

```
const map = new Map();
function countWords(word) {
    const count = map.get(word) || 0;
    map.set(word, count + 1);
}
```

Another benefit of Maps is that you can use arbitrary values as keys, not just strings.

More information:

- Section "The dict Pattern: Objects Without Prototypes Are Better Maps" in "Speaking JavaScript"
- Chapter "Maps and Sets"

4.16 New string methods

The ECMAScript 6 standard library provides several new methods for strings.

From `indexOf` to `startsWith`:

```
if (str.indexOf('x') === 0) {} // ES5
if (str.startsWith('x')) {} // ES6
```

From `indexOf` to `endsWith`:

```
function endsWith(str, suffix) { // ES5
  var index = str.indexOf(suffix);
  return index >= 0
    && index === str.length-suffix.length;
}
str.endsWith(suffix); // ES6
```

From `indexOf` to `includes`:

From `join` to `repeat` (the ES6 way of repeating a string is more of a hack):

```
new Array(3+1).join('#') // ES5
'#'.repeat(3) // ES6
```

More information: Chapter "New string features"

4.17 New Array methods

There are also several new Array methods in ES6.

4.17.1 From `Array.prototype.indexOf` to `Array.prototype.findIndex`

The latter can be used to find `NaN`, which the former can't detect:

```
const arr = ['a', NaN];

arr.indexOf(NaN); // -1
arr.findIndex(x => Number.isNaN(x)); // 1
```

As an aside, the new `Number.isNaN()` provides a safe way to detect `NaN` (because it doesn't coerce non-numbers to numbers):

```
> isNaN('abc')
true
> Number.isNaN('abc')
false
```

4.17.2 From `Array.prototype.slice()` to `Array.from()` or the spread operator

In ES5, `Array.prototype.slice()` was used to convert Array-like objects to Arrays. In ES6, you have `Array.from()`:

```
var arr1 = Array.prototype.slice.call(arguments); // ES5
const arr2 = Array.from(arguments); // ES6
```

If a value is iterable (as all Array-like DOM data structure are by now), you can also use the spread operator (`...`) to convert it to an Array:

```
const arr1 = [...'abc'];
    // ['a', 'b', 'c']
const arr2 = [...new Set().add('a').add('b')];
    // ['a', 'b']
```

4.17.3 From `apply()` to `Array.prototype.fill()`

In ES5, you can use `apply()`, as a hack, to create in Array of arbitrary length that is filled with `undefined`:

```
// Same as Array(undefined, undefined)
var arr1 = Array.apply(null, new Array(2));
    // [undefined, undefined]
```

In ES6, `fill()` is a simpler alternative:

`fill()` is even more convenient if you want to create an Array that is filled with an arbitrary value:

```
// ES5
var arr3 = Array.apply(null, new Array(2))
    .map(function (x) { return 'x' });
    // ['x', 'x']

// ES6
const arr4 = new Array(2).fill('x');
    // ['x', 'x']
```

`fill()` replaces all Array elements with the given value. Holes are treated as if they were elements.

More information: Sect. “Creating Arrays filled with values”

4.18 From CommonJS modules to ES6 modules

Even in ES5, module systems based on either AMD syntax or CommonJS syntax have mostly replaced hand-written solutions such as the revealing module pattern.

ES6 has built-in support for modules. Alas, no JavaScript engine supports them natively, yet. But tools such as browserify, webpack or jspm let you use ES6 syntax to create modules, making the code you write future-proof.

4.18.1 Multiple exports

4.18.1.1 Multiple exports in CommonJS

In CommonJS, you export multiple entities as follows:

```
//------ lib.js ------
var sqrt = Math.sqrt;
function square(x) {
    return x * x;
}
function diag(x, y) {
    return sqrt(square(x) + square(y));
}
module.exports = {
    sqrt: sqrt,
    square: square,
    diag: diag,
};

//------ main1.js ------
var square = require('lib').square;
var diag = require('lib').diag;

console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

Alternatively, you can import the whole module as an object and access `square` and `diag` via it:

4.18.1.2 Multiple exports in ES6

In ES6, multiple exports are called *named exports* and handled like this:

```
//------ lib.js ------
export const sqrt = Math.sqrt;
export function square(x) {
    return x * x;
}
export function diag(x, y) {
    return sqrt(square(x) + square(y));
}

//------ main1.js ------
import { square, diag } from 'lib';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

The syntax for importing modules as objects looks as follows (line A):

```
//------ main2.js ------
import * as lib from 'lib'; // (A)
console.log(lib.square(11)); // 121
console.log(lib.diag(4, 3)); // 5
```

4.18.2 Single exports

4.18.2.1 Single exports in CommonJS

Node.js extends CommonJS and lets you export single values from modules, via `module.exports`:

```
//------ myFunc.js ------
module.exports = function () { ··· };

//------ main1.js ------
var myFunc = require('myFunc');
myFunc();
```

4.18.2.2 Single exports in ES6

In ES6, the same thing is done via a so-called *default export* (declared via `export default`):

```
//------ myFunc.js ------
export default function () { ··· } // no semicolon!

//------ main1.js ------
import myFunc from 'myFunc';
myFunc();
```

More information: chapter “Modules”.

4.19 What to do next

location.

Next: II Data