ES6

IN PRACTICE

THE COMPLETE DEVELOPER'S GUIDE

Zsolt Nagy

Table of Contents

Introduction

ES6/ES2015 knowledge is now expected among JavaScript developers. The new language constructs have not only become popular, but they are now also widely supported.

At the same time, ES6 learning resources do not necessarily facilitate learning. Some courses spend a lot of your valuable time on explaining and repeating the obvious. Others tend to compete with the reference documentation, and provide you with all details you will never use.

My vision with this tutorial was to create a comprehensive guide that will not only focus on practical applications of the language, but it will also give you the opportunity to put theory into practice.

At the end of each section, there are exercises. Solve them in order to put theory into practice. You can verify your solutions by reading my reference solutions in the workbook.

The ES6 bundle currently contains two parts:

- ES6 in Practice - The Complete Developer's Guide
- ES6 in Practice - The Complete Developer's Guide Workbook

I will add other bonuses later.

If you have any questions, reach out to me by writing an email. My email address is info@zsoltnagy.eu.

If you are interested in JavaScript articles and courses, sign up to my newsletter on http://www.zsoltnagy.eu.

Arrow Functions

We will first describe the fat arrow syntax. Then we will discover the main advantage of arrow functions: context binding.

Fat Arrow Syntax

Let's write an ES5 function to sum two numbers.

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

Using fat arrows (`=>`), we will rewrite the same function in two steps.

Step 1: replace the `function` keyword with a fat arrow.

```
var sum = ( a, b ) => {
    return a + b;
};
```

Step 2: if the return value of the function can be described by one expression, and the function body has no side-effects, then we can omit the braces and the `return` keyword too.

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

If a function only has one argument, parentheses are not needed on the left of the fat arrow:

```
var square = a => a * a;
```

> Use case: syntactic sugar, more compact way of writing functions.

Context binding

In ES5, function scope often requires the awkward and unmaintainable `self = this` trick or the usage of context binding. Observe a simple example for the latter.

```
var Ball = function( x, y, vx, vy ) {
    this.x = x;
    this.y = y;
    this.vx = vx;
    this.vy = vy;
    this.dt = 25; // 1000/25 = 40 frames per second
    setInterval( function() {
        this.x += vx;
        this.y += vy;
        console.log( this.x, this.y );
    }.bind( this ), this.dt );
}

b = new Ball( 0, 0, 1, 1 );
```

Arrow functions come with automatic context binding. The lexical value of this isn't shadowed by the scope of the arrow function. Therefore, you save yourself thinking about context binding.

```
var Ball = function( x, y, vx, vy ) {
    this.x = x;
    this.y = y;
    this.vx = vx;
    this.vy = vy;
    this.dt = 25; // 1000/25 = 40 frames per second
    setInterval( () => {
        this.x += vx;
        this.y += vy;
        console.log( this.x, this.y );
    }, this.dt );
}

b = new Ball( 0, 0, 1, 1 );
```

> Use case: Whenever you want to use the lexical value of `this` coming from outside the scope of the function, use arrow functions.

Don't forget that the equivalence transformation for fat arrows is the following:

```
// ES2015
ARGUMENTS => VALUE;

// ES5
function ARGUMENTS { return VALUE; }.bind( this );
```

The same holds for blocks:

```
// ES2015
ARGUMENTS => {
    // ...
};

// ES5
function ARGUMENTS {
    // ...
}.bind( this );
```

In constructor functions, prototype extensions, it typically does not make sense to use fat arrows. This is why we kept the `Ball` constructor a regular function.

We will introduce the `class` syntax later to provide an alternative for construction functions and prototype extensions.

Exercises

Exercise 1: Write an arrow function that returns the string `'Hello World!'`.

Exercise 2: Write an arrow function that expects an array of integers, and returns the sum of the elements of the array. Use the built-in method `reduce` on the array argument.

Exercise 3: Rewrite the following code by using arrow functions wherever it makes sense to use them:

```
var Entity = function( name, delay ) {
  this.name = name;
  this.delay = delay;
};

Entity.prototype.greet = function() {
    setTimeout( function() {
        console.log( 'Hi, I am ' + this.name );
    }.bind( this ), this.delay );
};

var java = new Entity( 'Java', 5000 );
var cpp = new Entity( 'C++', 30 );

java.greet();
cpp.greet();
```

Function scope, block scope, constants

In this lesson, we will introduce the `let` keyword for the purpose of declaring block scoped variables. You will also learn about defining block scoped constants, and the dangers of scoping, such as the **temporal dead zone**. We will conclude the lesson with best practices that simplify your task and combat all dangers with ease.

Var vs Let

Variables declared with `var` have function scope. They are valid inside the function they are defined in.

```
var guessMe = 2;
// A: guessMe is 2
( () => {
    // B: guessMe is undefined
    var guessMe = 5;
    // C: guessMe is 5
} )();
// D: guessMe is 2
```

Comment `B` may surprise you if you have not heard of hoisting. JavaScript hoists all declarations:

```
var guessMe; // initialized to undefined
// B: guessMe is undefined
guessMe = 5;
```

Variables declared with `let` have block scope. They are valid inside the block they are defined in.

```
// A: guessMe is undeclared
{
    // B: guessMe is uninitialized. Accessing guessMe throws an error
    let guessMe = 5;
    // C: guessMe is 5
}
// D: guessMe is undeclared
```

Comment `B` may surprise you again. Even though `let guessMe` is hoisted similarly to `var`, its value is not initialized to `undefined`. Retrieving uninitialized values throws a JavaScript error.

The area described by comment `B` is the *temporal dead zone* of variable `guessMe`.

```
console.log( age ); // logs undefined
var age = 25;

console.log( name ); // we are in the temporal dead zone of name, accessing name throw
s an error
var name = 'Ben';
```

The temporal dead zone exists even if a variable with the same name exists outside the scope of the dead zone.

```
let guessMe = 1;
// A: guessMe is 1
{
    // Dead Zone of guessMe
    let guessMe = 2;
    // C: guessMe is 2
}
// D: guessMe is 1
```

For a complete reference, the temporal dead zone exists for `let`, `const`, and `class` declarations. It does not exist for `var`, `function`, and `function*` declarations.

Constants

Declarations with `const` are blocked scope, they have to be initialized, and their value cannot be changed after initialization.

```
const PI = 3.1415;
PI = 3.14;
// Uncaught TypeError: Assignment to constant variable.(…)
```

Not initializing a constant also throws an error:

```
const PI;
// Uncaught SyntaxError: Missing initializer in const declaration
```

Const may also have a *temporal dead zone*.

```
// temporal dead zone of PI
const PI = 3.1415;
// PI is 3.1415 and its value is final
```

Redeclaring another variable with the same name in the same scope will throw an error.

Use cases of let, const, and var

> Rule 1: use `let` for variables, and `const` for constants whenever possible. Use `var` only when you have to maintain legacy code.

The following rule is also worth keeping.

> Rule 2: Always declare and initialize all your variables at the beginning of your scope.

Using rule 2 implies that you never have to face with the temporal dead zone.

If you have a linter such as ESLint, set it up accordingly, so that it warns you when you violate the second rule.

If you stick to these two rules, you will get rid of most of the anomalies developers face.

Exercises

Exercise 1: Check the following riddle:

```
'use strict';

var guessMe1 = 1;
let guessMe2 = 2;

{
    try {
        console.log( guessMe1, guessMe2 );
    } catch( _ ) {}

    let guessMe2 = 3;
    console.log( guessMe1, guessMe2 );
}

console.log( guessMe1, guessMe2 );

() => {

    console.log( guessMe1 );
    var guessMe1 = 5;
    let guessMe2 = 6;
    console.log( guessMe1, guessMe2 );
};

console.log( guessMe1, guessMe2 );
```

Determine the values logged to the console.

Exercise 2: Modify the code such that all six console logs print out their values exactly once, and the printed values are the following:

```
1 3
1 3
1 2
5
5 6
1 2
```

You are not allowed to touch the console logs, just the rest of the code.

Exercise 3: Add the linter of your choice to your text editor or IDE. Configure your linter such that you never have to worry about leaving a temporal dead zone unnoticed.

Default Arguments

First we will examine ES5 hacks to provide default arguments. Then we will explore the ES6 version. This is a short lesson, therefore, it is your chance to learn the most from the exercises.

Hacks in ES5

In some cases, function arguments are optional. For instance, let's check the following code:

```
function addCalendarEntry( event, date, length, timeout ) {
    date = typeof date === 'undefined' ? new Date().getTime() : date;
    length = typeof length === 'undefined' ? 60 : length;
    timeout = typeof timeout === 'undefined' ? 1000 : timeout;

    // ...
}


addCalendarEntry( 'meeting' );
```

In the above example, three arguments are optional. Most people use the `||` for default parameters.

```
function addCalendarEntry( event, date, length, timeout ) {
    date = date || new Date().getTime();
    length = length || 60;
    timeout = timeout || 1000;

    // ...
}


addCalendarEntry( 'meeting' );
```

The reason why this approach is wrong is that all falsy values are substituted with the defaults. This includes `0`, `''`, `false`.

The ES6 way

ES6 supports default values. Whenever an argument is not given, the default value is substituted. The syntax is quite compact:

```
function addCalendarEntry( event, date = new Date().getTime(), length = 60, timeout =
1000 ) {

    // ...
}


addCalendarEntry( 'meeting' );
```

Default argument values can be any values.

```
function f( a = a0, b = b0 ) { ... }
```

When `a` and `b` are not supplied, the above function is equivalent with

```
function f() {
    let a = a0;
    let b = b0;
    ...
}
```

All considerations for let declarations including the temporal dead zone holds. `a0` and `b0` can be any JavaScript expressions, in fact, `b0` may even be a function of `a`.

The `arguments` array is not affected by the default parameter values in any way. See the third exercise for more details.

> Use default arguments at the end of the argument list for optional arguments. Document their default values.

Exercises

Exercise 1. Write a function that executes a callback function after a given delay in milliseconds. The default value of delay is one second.

Exercise 2. Change the below code such that the second argument of `printComment` has a default value that's initially `1`, and is incremented by `1` after each call.

```
function printComment( comment, line ) {
    console.log( line, comment );
}
```

Exercise 3 Determine the values written to the console.

```
function argList( productName, price = 100 ) {
    console.log( arguments.length );
    console.log( productName === arguments[0] );
    console.log( price === arguments[1] );
};

argList( 'Krill Oil Capsules' );
```

Classes

The concept of prototypes and prototypal inheritance in ES5 are hard to understand for many developers transitioning from another programming language to JavaScript development.

ES6 classes introduce *syntactic sugar* to make prototypes look like classical inheritance.

For this reason, some people applaud classes, as it makes JavaScript appear more familiar to them.

Other people seem to have launched a holy war against classes, claiming, that the class syntax is flawed, and it takes away the main advantages of using JavaScript.

On some level, all opinions have merit. My advice to you is that the market is always right. Knowing classes gives you the advantage that you can maintain code written in the class syntax. It does not mean that you have to use it. If your judgement justifies that classes should be used, go for it!

Not knowing the class syntax is a disadvantage.

Judgement on the class syntax, or offering alternatives are beyond the scope of this section.

Prototypal Inheritance in ES5

Let's start with an example, where we implement a classical inheritance scenario in JavaScript.

```
function Shape( color ) {
    this.color = color;
}

Shape.prototype.getColor = function() {
    return this.color;
}

function Rectangle( color, width, height ) {
    Shape.call( this, color );
    this.width = width;
    this.height = height;
};

Rectangle.prototype = Object.create( Shape.prototype );
Rectangle.prototype.constructor = Rectangle;

Rectangle.prototype.getArea = function() {
    return this.width * this.height;
};

let rectangle = new Rectangle( 'red', 5, 8 );
console.log( rectangle.getArea() );
console.log( rectangle.getColor() );
console.log( rectangle.toString() );
```

`Rectangle` is a constructor function. Even though there were no classes in ES5, many people called constructor functions and their prototype extensions classes.

We instantiate a class with the `new` keyword, creating an object out of it. In ES5 terminology, constructor functions return new objects, having defined of properties and operations.

Prototypal inheritance is defined between `Shape` and `Rectangle` , as a rectangle *is a* shape. Therefore, we can call the `getColor` method on a rectangle.

Prototypal inheritance is implicitly defined between `Object` and `Shape` . As the prototype chain is transitive, we can call the `toString` built-in method on a rectangle object, even though it comes from the prototype of `Object` .

The code looks a bit weird, and chunks of code that should belong together are separated.

The ES6 way

Let's see the ES6 version. As we'll see later, the two versions are not equivalent to each other, we just describe the same problem domain with ES5 and ES6 code.

```
class Shape {
    constructor( color ) {
        this.color = color;
    }

    getColor() {
        return this.color;
    }
}

class Rectangle extends Shape {
    constructor( color, width, height ) {
        super( color );
        this.width = width;
        this.height = height;
    }

    getArea() {
        return this.width * this.height;
    }
}

let rectangle = new Rectangle( 'red', 5, 8 );
console.log( rectangle.getArea() );
console.log( rectangle.getColor() );
console.log( rectangle.toString() );
```

Classes may encapsulate

- a constructor function
- additional operations extending the prototype
- reference to the parent prototype.

Notice the following:

- the `extends` keyword defines the is-a relationship between `Shape` and `Rectangle`. All instances of `Rectangle` are also instances of `Shape`.
- the `constructor` method is a method that runs when you instantiate a class. You can call the constructor method of your parent class with `super`. More on `super` later.
- methods can be defined inside classes. All objects are able to call methods of their class and all classes that are higher in the inheritance chain.
- Instantiation works in the exact same way as the instantiation of an ES5 constructor function.

You can observe the equivalent ES5 code by pasting the above code into the BabelJs online editor.

The reason why the generated code is not equivalent with the ES5 code we studied is that the class syntax comes with additional features. You will never need the protection provided by these features during regular use. For instance, if you call the class name as a regular function, or you call a method of the class with the `new` operator as a constructor, you get an error.

> Your code becomes more readable, when you capitalize class names, and start object names and method names with a lower case letter. For instance, `Person` should be a class, and `person` should be an object.

Super

Calling `super` in a constructor should happen before accessing `this`. As a rule of thumb:

> Call `super` as the first thing in a constructor of a class defined with `extends`.

If you fail to call `super`, an error will be thrown. If you don't define a constructor in a class defined with `extends`, one will automatically be created for you, calling `super`.

```
class A { constructor() { console.log( 'A' ); } }
class B extends A { constructor() { console.log( 'B' ); } }

new B()
B
Uncaught ReferenceError: this is not defined(…)

class C extends A {}

new C()
A
> C {}

C.constructor
> Function() { [native code] }
```

Shadowing

Methods of the parent class can be redefined in the child class.

```
class User {
    constructor() {
        this.accessMatrix = {};
    }
    hasAccess( page ) {
        return this.accessMatrix[ page ];
    }
}

class SuperUser extends User {
    hasAccess( page ) {
        return true;
    }
}

var su = new SuperUser();

su.hasAccess( 'ADMIN_DASHBOARD' );
> true
```

Creating abstract classes

Abstract classes are classes that cannot be instantiated. One example is `Shape` in the above example. Until we know what kind of shape we are talking about, we cannot do much with a generic shape.

Often times, you have a couple of business objects on the same level. Assuming that you are not in the WET (We Enjoy Typing) group of developers, it is natural that you abstract the common functionalities into a base class. For instance, in case of stock trading, you may have a `BarChartView`, a `LineChartView`, and a `CandlestickChartView`. The common functionalities related to these three views are abstracted into a `ChartView`. If you want to make `ChartView` abstract, do the following:

```
class ChartView {
    constructor( /* ... */ ) {
        if ( this.new === ChartView ) {
            throw new Error( 'Abstract class ChartView cannot be instantiated.' );
        }
        // ...
    }
    // ...
}
```

The property `new.target` contains the class written next to the `new` keyword during instantiation. This is the name of the class whose constructor was first called in the inheritance chain.

Getters and Setters

Getters and setters are used to create computed properties.

In the below example, I will use `>` to indicate the response of an expression. Feel free to experiment with the below classes using your Chrome console.

```
class Square {
    constructor( width ) { this.width = width; }
    get area() {
        console.log( 'getter' );
        return this.width * this.width;
    }
}

let square = new Square( 5 );

square.area
get area
> 25

square.area = 36
> undefined

square.area
get area
> 25
```

Note that in the above example, `area` only has a getter. Setting `area` does not change anything, as `area` is a computed property that depends on the width of the square.

For the sake of demonstrating setters, let's define a `height` computed property.

```
class Square {
    constructor( width ) { this.width = width; }
    get height() {
        console.log( 'get height' );
        return this.width;
    }
    set height( h ) {
        console.log( 'set height', h );
        this.width = h;
    }
    get area() {
        console.log( 'get area' );
        return this.width * this.height;
    }
}

let square = new Square( 5 );

square.width
> 5

square.height
get height
> 5

square.height = 6
set height 6
> 6

square.width
> 6

square.area
get area
get height
> 36

square.width = 4
> 4

square.height
get height
> 4
```

Width and height can be used as regular properties of a `Square` object, and the two values are kept in sync using the height getter and setter.

Advantages of getters and setters:

- *Elimination of redundancy*: computed fields can be derived using an algorithm depending on other properties.

- *Information hiding*: do not expose properties that are retrievable or settable through getters or setters.
- *Encapsulation*: couple other functionality with getting/setting a value.
- *Defining a public interface*: keep these definitions constant and reliable, while you are free to change the internal representation used for computing these fields. This comes handy e.g. when dealing with a DOM structure, where the template may change
- *Easier debugging*: just add debugging commands or breakpoints to a setter, and you will know what caused a value to change.

Static methods

Static methods are operations defined on a class. These methods can only be referenced from the class itself, not from objects.

```
class C {
    static create() { return new C(); }
    constructor()   { console.log( 'constructor'); }
}

var c = C.create();
constructor

c.create();
> Uncaught TypeError: e.create is not a function(…)
```

Exercises

Exercise 1. Create a `PlayerCharacter` and a `NonPlayerCharacter` with a common anchestor `Character` . The characters are located in a 10x10 game field. All characters appear at a random location. Create the three classes, and make sure you can query where each character is.

Exercise 2. Each character has a direction (up, down, left, right). Player characters initially go right, and their direction can be changed using the `faceUp` , `faceDown` , `faceLeft` , `faceRight` methods. Non-player characters move randomly. A move is automatically taken every 5 seconds in real time. Right after the synchronized moves, each character console logs its position. The player character can only influence the direction he is facing. When a player meets a non-player character, the non-player character is eliminated from the game, and the player's score is increased by 1.

Exercise 3. Make sure the `Character` class cannot be instantiated.