ES6

IN PRACTICE

THE COMPLETE DEVELOPER'S GUIDE

Zsolt Nagy

ES6 in Practice

The Complete Developer's Guide

Zsolt Nagy

This book is for sale at http://leanpub.com/es6-in-practice

This version was published on 2016-12-25

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.

On December 30th 2016, you will get access to the following bonuses:

- A chapter on ES2016 and ES2017 to maintain your edge
- A guide on how to learn programming topics effectively

Expect the following bonuses to roll out in January and February 2017:

- Links, where you can play around with the code examples of the book
- The ES6 Coding Interview – collection of interview questions and reference solutions. We will write some code that may get you hired

I will also create a video version of the lessons and exercise solutions. If you purchase the book now, you will get access to the upgraded package for free. I will withdraw this offer once the first videos are published.

The course will be continuously updated and refined based on your feedback. If you have any questions, reach out to me by writing an email. My email address is info@zsoltnagy.eu[1].

If you are interested in JavaScript articles and courses, sign up to my newsletter on http://www.zsoltnagy.eu[2].

[1] info@zsoltnagy.eu
[2] http://www.zsoltnagy.eu

How to read this book

As this is a software development book, you will stumble upon code examples from time to time. Code examples are written in the following format with syntax highlighting:

```
var factorial = function( num ) {
    if ( num > 0 ) return num * factorial( num - 1 );
    return 1;
}
```

Once you execute code in the console, I will use the following notation:

```
factorial( 5 );
> 120
```

The › character denotes the return value of the expression. When the › symbol is absent, I ignore the return value of the expression for didactic reasons.

In each chapter, you will first get a theory section, followed by some exercises. I highly recommend solving the exercises on your own. Once you are ready with the exercises, check out the reference solutions in the *Solutions* chapter of the book.

In order to make navigation easier for you, I created links to easily navigate back and forth between the exercise and its corresponding solution.

The *Solutions* chapter is accessible both in this book and in a separate workbook. Originally, the two books were separated, however, based on your feedback, some of you may prefer keeping all the content in one place.

1. 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.

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

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

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

> Use cases of fat arrows: syntactic sugar, more compact way of writing functions.

Context binding

In ES5, function scope often requires us to bind the context to a function. Context binding is usually performed in one of the following two ways:

1. by defining a `self = this` variable,
2. by using the `bind` function.

In our first example, we will attempt to animate a ball using the `setInterval` method.

```
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 );
    }, this.dt );
}

var ball = new Ball( 0, 0, 1, 1 );
> NaN NaN  // 25ms later
> NaN NaN  // 50ms later
> NaN NaN  // 75ms later
// ...
```

The animation failed, because inside the function argument of `setInterval`, based on the rule of function scoping, the value of `this` is different.

In order to access and modify the variables in the scope of the `ball` object, we have to make the context of the ball accessible inside the function argument. Our first solution looks like this:

```
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
    var self = this;
    setInterval( function() {
        self.x += vx;
        self.y += vy;
        console.log( self.x, self.y );
    }, this.dt );
}

var ball = new Ball( 0, 0, 1, 2 );
> 1 2  // 25ms later
> 2 4  // 50ms later
> 3 6  // 75ms later
// ...
```

This solution is still a bit awkward, as we have to maintain the `self` and `this` references. It is very easy to make a mistake and use `this` instead of `self` somewhere. Therefore, in ES5, best practices suggest using the `bind` method.

```
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 );
}

var ball = new Ball( 0, 0, 1, 2 );
> 1 2  // 25ms later
> 2 4  // 50ms later
> 3 6  // 75ms later
// ...
```

The `bind` method binds the context of the `setInterval` function argument to `this`.

In ES6, 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.

Let's rewrite the above example in ES6:

```
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 );
> 1 2  // 25ms later
> 2 4  // 50ms later
> 3 6  // 75ms later
// ...
```

> 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 and prototype extensions, it 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!'`. (Solution)

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. (Solution)

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();
```

(Solution)

2. 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.

Var vs Let

Variables declared with `var` have function scope. They are accessible 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. Hoisting is a transformation that transforms the code

```
() => {
    JAVASCRIPT_STATEMENTS;
    var guessMe = 5;
};
```

in the following form:

```
() => {
    var guessMe;
    JAVASCRIPT_STATEMENTS;
    guessMe = 5;
};
```

Variables declared with `var` are initialized to `undefined`. This is why the value of `guessMe` was `undefined` in comment `B`.

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`.

```
function logAge() {
    console.log( 'age:', age );
    var age = 25;
}

logAge();
> age: undefined
```

In `logAge`, we log `undefined`, as `age` is hoisted and initialized to `undefined`.

```
function logName() {
    console.log( 'name:', name );
    let name = 'Ben';
}

logName();
> Uncaught ReferenceError: name is not defined
```

In `logName`, we reached the temporal dead zone of `name` by accessing it before the variable was defined.

You may find the temporal dead zone inconvenient at first sight. However, notice that the thrown error grasps your attention a lot better than a silent `undefined` value. Always be grateful for errors pointing out obvious mistakes during development, as the same mistakes tend to be a lot more expensive once they are deployed to production.

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
{
    // Temporal 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[1], 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( _ ) {}
```

[1] http://www.zsoltnagy.eu/eslint-for-better-productivity-and-higher-accuracy/

```
    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. (Solution)

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. (Solution)

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. (Solution)

3. Default Arguments

First we will examine ES5 hacks to provide default arguments. Then we will explore default arguments in ES6.

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' );
```

Three arguments of `addCalendarEntry` are optional.

A popular shorthand for optional parameters in ES5 uses the `||` (logical or) operator. You can make use of the shortcuts of logical operations.

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

    // ...
}


addCalendarEntry( 'meeting' );
```

The value `value || defaultValue` is `value`, whenever `value` is truthy. If the first operand of an `||` expression is truthy, the second operand is not even evaluated. This phenomenon is called a logical shortcut.

When `value` is falsy, `value || defaultValue` becomes `defaultValue`.

While this approach looks nice on paper, shortcuts are sometimes wrong. All falsy values are substituted with the defaults. This includes `0`, `''`, `false`. Sometimes, we may want to keep a `0` or an empty string.

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' );
```

Suppose function `f` is given with two arguments, `a` and `b`.

```
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;
    ...
}
```

Default arguments can have arbitrary types and values.

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`. However, `a0` cannot be a function of `b`, as `b` is declared later.

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 as 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. (Solution)

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 );
}
```

(Solution)

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' );
```

(Solution)

4. 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, even though it is defined for shapes.

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 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.
- The methods are written using the *concise method syntax*. We will learn about this syntax in depth in Chapter 7 - Objects.[1]

You can observe the equivalent ES5 code by pasting the above code into the BabelJs online editor[2].

[1] Whenever I link to another section of the book, expect a backlink in the corresponding section to the original point where you left off reading. This feature makes navigation easier for you. Unfortunately, links don't always work as expected, but worst case you have to scroll down a page to recognize the page where you left off.

[2] http://babeljs.io/repl/

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.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. Recall the `Shape` class in the previous 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 built-in property `new.target` contains a reference to 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 `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 ancestor `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. (Solution)

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`. (Solution)

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

5. Destructuring

One of the most common tasks in JavaScript is to build, mutate, and extract data from objects and arrays. ES2015 makes this process very compact with *destructuring*.

Object property shorthand notation

Suppose the follwing ES5 code is given:

```
var language = 'Markdown';
var extension = 'md';
var fileName = 'Destructuring';

var file = {
    language: language,
    extension: extension,
    fileName: fileName
};
```

It is possible to define the ES6 equivalent of the `file` object in the following way:

```
var file = { language, extension, fileName };
```

The two definitions of `file` are equivalent.

First destructuring examples

```
let user = {
    name        : 'Ashley',
    email       : 'ashley@ilovees2015.net',
    lessonsSeen : [ 2, 5, 6, 7, 9 ],
    nextLesson  : 10
};

let { email, nextLesson } = user;
// email becomes 'ashley@ilovees2015.net'
// nextLesson becomes 10
```

The value of a destructuring assignment of form `L = R` is `R`:

```
console.log({email, nextLesson} = user);
> Object {name: "Ashley", email: "ashley@ilovees2015.net",
>         lessonsSeen: Array[5], nextLesson: 10}
```

As a consequence, don't expect destructuring to be used as an alternative for formatting values.

Destructuring is *right-associative*, i.e. it is evaluated from right to left. `L = M = R` becomes `L = R`, which in turn becomes `R` after evaluation. The side effect is that in `M` and in `L`, variable assignments may take place on any depth.

```
let user2 = {email, nextLesson} = user;

console.log( user2 === user, user2.name );
> true "Ashley"
```

In the above example, `{email, nextLesson} = user` is evaluated. The side effect of the evaluation is that `email` and `nextLesson` are assigned to `"ashley@ilovees2015.net"` and `10` respectively. The value of the expression is `user`. Then `user2 = user` is evaluated, creating another

handle (or call it reference or pointer depending on your taste) for the object accessible via `user`.

Based on the above thought process, the below assignment should not surprise you:

```
let {name} = {email, nextLesson} = user;

console.log( name );
> "Ashley"
```

> Make sure you use the `let` keyword to initialize new variables. You can only destructure an object or an array, if all the variables inside have been declared.

Deeper destructuring

It is possible to destructure objects and arrays in any depth. Default values can also be used. Objects or arrays that don't exist on the right become assigned to `undefined` on the left.

```
let user = {
    name        : 'Ashley',
    email       : 'ashley@ilovees2015.net',
    lessonsSeen : [ 2, 5, 6, 7, 9 ],
    nextLesson  : 10
};

let { lessonsSeen : [ first, second, third, fourth, fifth,
      sixth = null, seventh ], nextLesson : eighth } = user;

console.log(
    first,
    second,
    third,
```

```
    fourth,
    fifth,
    sixth,
    seventh,
    eighth
);
> 2 5 6 7 9 null undefined 10
```

Destructuring function arguments

The arguments in a function signature act as left values of destructuring assignments. The parameters of a function call act as the respective right values of destructuring assignments.

```
function f( L1, L2 );

f( R1, R2 ); // executes L1 = R1, L2 = R2
```

You will use destructuring function arguments in two exercises in the next lesson.

Bug alert with destructuring!

Software developers tend to make mistakes. Don't overuse destructuring, always keep your code readable!

Continuing the above example, suppose you make a typo, and write `neme` instead of name.

```
let { neme } = user;

console.log( neme );
> undefined
```

The typo silently assigns the value `undefined` to `neme`, potentially causing trouble. Always pay attention to fine-tuning your debugging skills!

In an `L = R` destructuring expression, `R` cannot be `null` or `undefined`, otherwise a `TypeError` is thrown:

```
let testUser = null;
let { name, email } = testUser;

// Uncaught TypeError: Cannot match against 'undefined' or 'null'.(…)
```

Another warning for thorough testing. When you write tests, make sure you always take care of all possible input types.

Exercises

Exercise 1. Swap two variables using one destructuring assignment.

(Solution)

Exercise 2. Complete the below function that calculates the nth fibonacci number in the sequence with one destructuring assignment! The definition of Fibonacci numbers is the following:

- `fib( 0 ) = 0`
- `fib( 1 ) = 1`
- `fib( n ) = fib( n-1 ) + fib( n-2 );`

```
function fib( n ) {
    let fibCurrent = 1;
    let fibLast = 0;

    if ( n < 0 ) return NaN;
    if ( n <= 1 ) return n;

    for ( let fibIndex = 1; fibIndex < n; ++fibIndex ) {
        // Insert one destructuring expression here
    }

    return fibCurrent;
}
```

(Solution)

Exercise 3. Determine all the bindings in the following assignment, and describe the execution of the destructuring assignment. Notice that `loft` is not the same variable name as `left`.

```
let node = { left : { left: 3, right: 4 }, right: 5 };

let { loft : {}, right : val } = node;
```

(Solution)

Exercise 4. Create one destructuring expression that declares exactly one variable to retrieve `x.A[2]`.

```
let x = { A: [ 't', 'e', 's', 't' ] };
```

(Solution)

Exercise 5. Suppose the following configuration object of a financial chart is given:

```
let config = {
    chartType : 0,
    bullColor : 'green',
    bearColor : 'red',
    days      : 30
};
```

Complete the function signature below such that the function may be called with any `config` objects (`null` and `undefined` are not allowed as inputs). If any of the four keys are missing, substitute their default values.

```
function drawChart( data, /* insert your solution here */ ) {
    // do not implement the function body
};
```

(Solution)

Exercise 6. Modify your solution in Exercise 5 such that the user may omit the `option` parameter, making its value `undefined`. (Solution)

6. Spread Operator and Rest Parameters

The Spread operator and Rest parameters are two related features in ES2015 that are worth learning. You can do cool things with them, and they often make your code a lot more compact.

Rest parameters

In some cases, you might want to deal with processing a variable number of arguments. In ES5, it was possible to use the `arguments` array inside a function to access them as an array:

```
( function() { console.log( arguments ); } )( 1, 'Second', 3 );
> [1, "Second", 3]
```

In ES2015, the last argument of a function can be preceded by `...`. This argument collects all the remaining arguments of the function in an array. The name for this construct is *rest parameters*, because it contains the rest of the parameters passed to a function.

Let's rewrite the above function in ES2015:

```
( (...args) => { console.log( args ); } )( 1, 'Second', 3 );
> [1, "Second", 3]
```

Note that the argument list containing rest parameter is placed in parentheses. This is mandatory, as `...args` is equivalent to `arg1, arg2, arg3`.

The rest parameter has to be the last argument of a function. As a consequence, there can only be one rest parameter in a function. If the rest parameter is not the last argument of the argument list of a function, an error is thrown.

Spread operator

In ES5, we often used the `apply` method to call a function with variable number of arguments. The spread operator makes it possible to achieve the exact same thing in a compact way.

Suppose you would like to write a method that returns the sum of its arguments. Let's write this function in ES5:

```
function sumArgs() {
    var result = 0;
    for( var i = 0; i < arguments.length; ++i ) {
        result += arguments[i];
    }
    return result;
}

sumArgs( 1, 2, 3, 4, 5 );
> 15
```

When we know the parameters passed to a function, we have an easy job calling `sumArgs`. However, sometimes it makes little to no sense to write out 100 parameters. In other cases, the number of parameters is not known. This is when the `apply` method of JavaScript was used in ES5.

```
var arr = [];
for( var i = 0; i < 100; ++i ) arr[i] = Math.random();
sumArgs.apply( null, arr );
```

In ES2015, our job is a lot easier. We can simply use the *spread operator* to call `sumArgs` in the exact same way as above. The spread operator spreads the elements of an array, transforming them into a parameter list.

```
sumArgs( ...arr );
```

Opposed to rest parameters, there are no restrictions on the location where the Spread operator is used in the parameter list. Therefore, the following call is also valid:

```
sumArgs( ...arr, ...arr, 100 );
```

Strings are spread as arrays of characters

If you would like to process a string character by character, use the spread operator to create an array of one character long strings in the following way:

```
let spreadingStrings = 'Spreading Strings';
let charArray = [ ...spreadingStrings ];
```

Destructuring with the spread operator

Let's create an array that contains the last four characters of another array:

```
let notgood = 'not good'.split( '' );
let [ ,,,, ...good ] = notgood;

console.log( good );
// ["g", "o", "o", "d"]
```

If there are no elements left, the result of a destructuring assignment involving a spread operator is `[]`.

```
let notgood = 'not good'.split( '' );
let [ ,,,,,,,,,,,,,,,, ...empty ] = notgood;

console.log( empty );
// []
```

Similarly to the rest parameter in functions, using ... on the left of a destructuring expression creates a match for all the remaining elements of the array:

```
[,...A] = [1,2,3,4]
// A becomes [2,3,4]
```

Similarly to the rest parameter in functions, on the left side of a destructuring assignment, it is only allowed to use the rest parameter as the last element of an array.

```
[...A,] = [1,2]
// Uncaught SyntaxError: Rest element must be last element in array
```

In order to fully understand the utility of the spread operator and rest parameters, I encourage you to do the exercises. This is a very important section, and we will build on it in the future.

Exercises

Exercise 1. Make a shallow copy of an array of any length in one destructuring assignment! If you don't know what a shallow copy is, make sure you read about it, as you will need these concepts during your programming career. I can highly recommend my article on Cloning Objects in JavaScript[1]. (Solution)

Exercise 2: Determine the value logged to the console on Google Chrome without running it. Write down the mechanism behind it using your own words.

[1] http://www.zsoltnagy.eu/cloning-objects-in-javascript/

```
let f = () => [..."12345"];

let A = f().map( f );

console.table( A );
```

(Solution)

Exercise 3. Create an 10x10 matrix of `null` values. (Solution)

Exercise 4. Rewrite the `sumArgs` function of this tutorial in ES2015, using a rest parameter and arrow functions.

```
function sumArgs() {
    var result = 0;
    for( var i = 0; i < arguments.length; ++i ) {
        result += arguments[i];
    }
    return result;
}
```

(Solution)

Exercise 5. Complete the following ES2015 function that accepts two String arguments, and returns the length of the longest common substring in the two strings. The algorithmic complexity of the solution does not matter.

```
let maxCommon = ([head1,...tail1], [head2,...tail2], len = 0) => {
    if ( typeof head1 === 'undefined' || typeof head2 === 'undefined\
' ) /* Write code here */
    if ( head1 === head2 ) /* Write code here */
    let firstBranch = /* Write code here */
    let secondBranch = /* Write code here */
    return Math.max( ...[len, firstBranch, secondBranch ] );
}

maxCommon( '123', '1' );
// 1

maxCommon( '11111', '11f111g');
// 3

maxCommon( 'abc', '111cab' );
// 2
```

(Solution)

7. Solutions

Arrow Functions

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

Solution:

```
() => 'Hello World!'
```

Back to the Exercises

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.

Solution:

```
arr => arr.reduce( ( a, b ) => a + b );
```

Back to the Exercises

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();
```

Solution:

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

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

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

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

Notes:

- it does not make sense to replace the `Entity` constructor, because we need the context
- it does not make sense to replace the prototype extension `greet`, as we make use of its default context
- it makes perfect sense to replace the function argument of `setTimeout` with an arrow function. Notice that the context binding also disappeared in the solution

Back to the Exercises

Function scope, block scope, constants

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.

Solution:

```
1 3
1 2
1 2
```

Back to the Exercises

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.

Solution:

```
'use strict';

var guessMe1 = 1;
let guessMe2 = 2;

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

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

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

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

In the first block, move the `let` declaration to the top of the block, to avoid the temporary dead zone belonging to `guessMe2` in the first console log. Now that the temporal dead zone is removed, the try-catch block serves no purpose anymore.

In order to execute the fourth and the fifth console logs, we have to invoke the function. Note that self-invoking functions are not necessary anymore in `ES2015`, as we should only be using block scoped variables anyway, and creating blocks to separate scope is sufficient. We still kept the block scope in the solution.

Without moving the declaration of `guessMe1` inside the self-invoking function to the top of the function, the fourth console log would log `undefined`. Even though it does not make a difference, I also moved the `let` declaration of `guessMe2` to the top.

Back to the Exercises

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.

Solution: There is no exact solution to this exercise. Read my article about ESLint[1] if you are interested in learning the basics of linting. Configure the text editor of your choice to run with ESLint.

[1] http://www.zsoltnagy.eu/eslint-for-better-productivity-and-higher-accuracy/

For `var` declarations, I used the vars-on-top[2] rule. There was no such rule in ESLint for `let` and `const` declarations (proof[3]).

If you go down the line of not allowing `var` at all, use the no-var[4] rule.

Regarding the temporal dead zone, the no-use-before-define[5] rule parses and spots variable use in the temporal dead zone.

Back to the Exercises

[2] http://eslint.org/docs/rules/vars-on-top
[3] https://github.com/eslint/eslint/issues/2865
[4] http://eslint.org/docs/rules/no-var
[5] http://eslint.org/docs/rules/no-use-before-define

Default Arguments

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

Solution:

```
function executeCallback( callback, delay = 1000 ) {
    setTimeout( callback, delay );
}
```

Exercise the code by running:

```
executeCallback( () => console.log('done') );
```

Your function prints `'done'` in one second.

Back to the Exercises

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

Solution:

Default values can be any JavaScript expressions, so the `++` post-increment is perfectly fine.

```
let lineNumber = 1;

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

We want to avoid complicated expressions in the function signature for stylistic reasons, based on the principles of writing maintainable, easy to understand code. Therefore, consider moving more complex logic from default argument value expressions to separate functions:

```
let lineNumber = 0;

let getLineNumber = function() {
    lineNumber += 1;
    return lineNumber;
}

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

This way, if the line numbering logic changes, you don't have to edit hard to read default parameter expressions.

Notice that the code is side-effect free. This will be an important concept in React, Redux, and ImmutableJs. We generally want to avoid side-effects, this is why I suggest not using `++` and `--` at all.

Back to the Exercises

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' );
```

Solution:

The output value is:

```
1
true
false
```

To understand the role of `arguments`, let's see the original function:

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

Let's transform this function into its equivalent on condition that the second argument is not supplied:

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

From this form, it is easy to conclude that

- `arguments.length` is 1,
- `arguments[1]` is undefined, and is not equal to `100`.

Back to the Exercises

Classes

Exercise 1. Create a `PlayerCharacter` and a `NonPlayerCharacter` with a common ancestor `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.

Solution:

This exercise has many solutions. We will stick to just one.

For the sake of simplicity, we will choose not to model the game field. We will place `x` and `y` inside the character objects as coordinates.

At this stage, there is no difference between player and non-player characters. We still created them to match the requirements.

```
class Character {
    constructor( id, name, x, y ) {
        this.id = id;
        this.name = name;
        this.x = x;
        this.y = y;
    }
    get position() {
        return { x: this.x, y: this.y };
    }
}

class PlayerCharacter extends Character {
}

class NonPlayerCharacter extends Character {
}

function createPlayer( id, name ) {
```

```
    let x = Math.floor( Math.random() * 10 ),
        y = Math.floor( Math.random() * 10 );
    return new PlayerCharacter( id, name, x, y );
}

function createNonPlayer( id, name ) {
    let x = Math.floor( Math.random() * 10 ),
        y = Math.floor( Math.random() * 10 );
    return new NonPlayerCharacter( id, name, x, y );
}
```

Example execution:

```
createNonPlayer( 1, 'Wumpus' ).position
// Object {x: 6, y: 5} note that the actual values are random
```

Back to the Exercises

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.

Solution:

We will model the direction of each character with the `dx` and `dy` variables, describing the change in coordinates during one step. For instance, if the character faces upwards, `dx` is `0`, and `dy` is `-1`.

The specification allows non-player characters to occupy the same position.

```
class Character {
    constructor( id, name, x, y ) {
        this.id = id;
        this.name = name;
        this.x = x;
        this.y = y;
        // initially the character will face right
        this.dx = 1;
        this.dy = 0;
    }
    get position() {
        return { x: this.x, y: this.y };
    }
    move() {
        this.x += this.dx;
        this.y += this.dy;
        if ( this.x < 0 ) this.x = 0;
        if ( this.x > 9 ) this.x = 9;
        if ( this.y < 0 ) this.y = 0;
        if ( this.y > 9 ) this.y = 9;
    }
    logPosition() {
        console.log( this.name, this.position );
    }
    collidesWith( character ) {
        return character.position.x === this.x &&
               character.position.y === this.y;
    }
}

class PlayerCharacter extends Character {
    constructor( id, name, x, y ) {
        super( id, name, x, y );
        this.score = 0;
    }
```

```
    faceUp() { this.dx = 0; this.dy = -1; }
    faceDown() { this.dx = 0; this.dy = 1; }
    faceLeft() { this.dx = -1; this.dy = 0; }
    faceRight() { this.dx = 1; this.dy = 0; }
    increaseScore( points ) { this.score += points; }

}

class NonPlayerCharacter extends Character {
    faceRandom() {
        let dir = Math.floor( Math.random() * 4 );
        this.dx = [ 0, 0, -1, 1 ][ dir ];
        this.dy = [ -1, 1, 0, 0 ][ dir ];
    }
}

function createPlayer( id, name ) {
    let x = Math.floor( Math.random() * 10 ),
        y = Math.floor( Math.random() * 10 );
    return new PlayerCharacter( id, name, x, y );
}

function createNonPlayer( id, name ) {
    let x = Math.floor( Math.random() * 10 ),
        y = Math.floor( Math.random() * 10 );
    return new NonPlayerCharacter( id, name, x, y );
}

let npcArray = '23456'.split('').map( i => {
        return createNonPlayer( i, 'Wumpus_' + i )
    } );
let player = createPlayer( 1, 'Hero' );

function gameLoop() {
    function changeNpcDirections() {
```

```
        npcArray.forEach( npc => { npc.faceRandom(); } );
    }
    function moveCharacters() {
        player.move();
        npcArray.forEach( npc => { npc.move(); } );
    }
    function logPositions() {
        player.logPosition();
        npcArray.forEach( npc => { npc.logPosition(); } );
    }
    function processCollisions() {
        let len = npcArray.length;
        npcArray = npcArray.filter(
            npc => !npc.collidesWith( player ) );
        player.increaseScore( len - npcArray.length );
    }

    console.log( 'move starts' );
    changeNpcDirections();
    moveCharacters();
    logPositions();
    processCollisions();
}

setInterval( gameLoop, 5000 );

// influence the movement of the player by executing
// player.faceUp()
// player.faceDown()
// player.faceLeft()
// player.faceRight()
```

Feel free to play around with the game in the console. If you want to test that updating the score works, you have a 50% chance of catching a wumpus by executing the following sequence:

```
player.faceLeft();
player.x = 0;
player.y = 0;
npcArray[0].x = 0;
npcArray[0].y = 0;
```

Back to the Exercises

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

Solution: Place the following lines in the constructor of the `Character` class.

```
if ( this.new === Character ) {
    throw new Error(
        'Abstract class Character cannot be instantiated.' );
}
```

Back to the Exercises

Destructuring

Exercise 1. Swap two variables using one destructuring assignment.

Solution:

```
let text1 = 'swap', text2 = 'me';

[ text1, text2 ] = [ text2, text1 ];
```

The `text1 = text2` and the `text2 = text1` assignments take place in parallel from the perspective of the whole expression. The expression on the right is evaluated, and becomes `[ 'me', 'swap' ]`. This evaluation happens before interpreting the expression on the left.

Back to the Exercises

Question 2. Complete the below function that calculates the nth fibonacci number in the sequence with one destructuring assignment! The definition of Fibonacci numbers is the following:

- `fib( 0 ) = 0`
- `fib( 1 ) = 1`
- `fib( n ) = fib( n-1 ) + fib( n-2 );`

```
function fib( n ) {
    let fibCurrent = 1;
    let fibLast = 0;

    if ( n < 0 ) return NaN;
    if ( n <= 1 ) return n;

    for ( let fibIndex = 1; fibIndex < n; ++fibIndex ) {
        // Insert one destructuring expression here
    }

    return fibCurrent;
}
```

Solution:

```
function fib( n ) {
    let fibCurrent = 1;
    let fibLast = 0;

    if ( n < 0 ) return NaN;
    if ( n <= 1 ) return n;

    for ( let fibIndex = 1; fibIndex < n; ++fibIndex ) {
        [fibCurrent, fibLast] = [fibCurrent + fibLast, fibCurrent];
    }

    return fibCurrent;
}
```

Back to the Exercises

Question 3. Determine all the bindings in the following assignment, and describe the execution:

```
let node = { left : { left: 3, right: 4 }, right: 5 };

let { loft : {}, right : val } = node;
```

Solution: Given that `loft` is a typo, there is no corresponding right value to the value of `loft`. Therefore, the recursive destructuring assignment `{} = undefined` will be executed. Given that `undefined` cannot stand on the right of a destructuring assignment, an error will be thrown, and no value bindings will take place.

In case we modified the code, removing the value of `loft`, the following bindings would take place:

```
let node = { left : { left: 3, right: 4 }, right: 5 };

let { loft, right : val } = node;
```

- `loft` becomes `undefined`,
- `val` becomes `5`.

Back to the Exercises

Question 4. Create one destructuring expression that declares exactly one variable to retrieve `x.A[2]`.

```
let x = { A: [ 't', 'e', 's', 't' ] };
```

Solution:

```
let { A : [ , , A_2 ] } = x;
```

You don't have to provide variable names to match `A[0]`, `A[1]`, or `A[3]`. For `A[3]`, you don't even need to create a comma, symbolizing that `A[3]` exists. Similarly, adding two commas after `A_2` does not make a difference either, as in JavaScript, indexing outside the bounds of an array gives us `undefined`.

Note that `A` was not created as a variable in the expression. You cannot assign the name of a variable and destructure its contents at the same time.

Back to the Exercises

Question 5. Suppose the following configuration object of a financial chart is given:

```
let config = {
    chartType : 0,
    bullColor : 'green',
    bearColor : 'red',
    days      : 30
};
```

Complete the function signature below such that the function may be called with any `config` objects (`null` and `undefined` are not allowed as inputs). If any of the four keys are missing, substitute their default values.

```
function drawChart( data, /* insert your solution here */ ) {
    // do not implement the function body
};
```

Solution:

```
let config = {
    chartType : 0,
    bullColor : 'green',
    bearColor : 'red',
    days      : 30
};

function drawChart( data, {
    chartType = 0,
    bullColor = 'green',
    bearColor = 'red',
    days = 30 } ) {
    // do not implement the function body
    console.log( chartType, bullColor, bearColor, days );
};

drawChart( [], {} );
// 0 "green" "red" 30

drawChart( [], { chartType: 1, days: 60 } );
// 1 "green" "red" 60
```

Back to the Exercises

Question 6 Modify your solution in Question 5 such that the user may omit the `option` parameter, making its value `undefined`.

Solution: We will solve this exercise without destructuring first. We will rename the `drawChart` function to `_drawChart`, and call it from `drawChart` after taking care of the `options` value:

```
function drawChart( data, options = {} ) {
    _drawChart( data, options );
}
```

The second solution uses a default value for the second argument, which lets us construct a solution without renaming `drawChart`.

```
function drawChart( data, {
    chartType = 0,
    bullColor = 'green',
    bearColor = 'red',
    days = 30 } = {} ) {
    // do not implement the function body
    console.log( chartType, bullColor, bearColor, days );
};

drawChart( [] );
// 0 "green" "red" 30
```

When `undefined` is passed, the following steps take place:

- `drawChart` is called with `undefined` as its second argument
- as the second argument has a default value of `{}`, `undefined` is replaced by `{}`
- as neither of the four keys occur in `{}`, the `chartType`, `bullColor`, `bearColor`, `days` will be initialized to their default values

Note that the solution won't work with an `options` value of `null`.

Back to the Exercises

Spread and Rest

Exercise 1. Make a shallow copy of an array of any length in one destructuring assignment! If you don't know what a shallow copy is, make sure you read about it, as you will need these concepts during your programming career. I can highly recommend my article on Cloning Objects in JavaScript[6].

Solution:

Use the Spread operator on the left to create assignments between each element in the original array and the cloned array.

```
let originalArray = [ 2, 3, 4 ];
let [...clonedArray] = originalArray;

console.log(
    clonedArray == originalArray,
    clonedArray,
    originalArray );
// false [2, 3, 4] [2, 3, 4]

console.log( originalArray[2] === clonedArray[2] );
// true
```

Back to the Exercises

Exercise 2: Determine the value logged to the console on Google Chrome without running it. Write down the mechanism behind it using your own words.

[6] http://www.zsoltnagy.eu/cloning-objects-in-javascript/

```
let f = () => [..."12345"];

let A = f().map( f );

console.table( A );
```

Solution:

An array of five vectors of `['1', '2', '3', '4', '5']` is printed out as a table.

The mechanism is the exact same as the explanation in the next exercise.

The function `f` creates the array `['1', '2', '3', '4', '5']`.

In `f().map( f )`, only the length of `f()` matters, as the values are thrown away by the `map` function. Each element of the `f()` array is mapped to the array `['1', '2', '3', '4', '5']`, making a 2 dimensional array of vectors `['1', '2', '3', '4', '5']`.

Back to the Exercises

If you came here from Chapter 7, click here to resume reading the capter

---

Exercise 3. Create an 10x10 matrix of `null` values.

Solution:

```
let nullVector = () => new Array( 10 ).fill( null );
let nullArray = nullVector().map( nullVector );
```

Study the `fill` method for more details here[7].

[7] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/fill

We create a `null` vector of size 10 with the `nullVector` function. The values of the first `nullVector()` return value don't matter, as we map each of the ten elements to another value. The `nullVector` mapping function throws each `null` value away, and inserts an array of ten nulls in its place.

Back to the Exercises

Exercise 4. Rewrite the `sumArgs` function of this tutorial in ES2015, using a rest parameter and arrow functions.

```
function sumArgs() {
    var result = 0;
    for( var i = 0; i < arguments.length; ++i ) {
        result += arguments[i];
    }
    return result;
}
```

Solution:

```
function sumArgs( ...numbers ) {
    return numbers.reduce( (sum,num) => sum+num, 0 );
}
```

Reminder of how `reduce` works: the first argument of the `reduce` function is an accumulator, which is initialized to `0`. The upcoming element of the original array is stored in `num`. Let's log each iteration of the reduction:

```
function sumArgs( ...numbers ) {
    return numbers.reduce( (sum,num) => {
        console.log( '(sum, num): ', sum, num );
        return sum+num;
    }, 0 );
}

sumArgs(2,3,4,5,6)

// console output:
> (sum, num):  0 2
> (sum, num):  2 3
> (sum, num):  5 4
> (sum, num):  9 5
> (sum, num):  14 6

// result:
> 20
```

Back to the Exercises

Exercise 5. Complete the following ES2015 function that accepts two String arguments, and returns the length of the longest common substring in the two strings. The algorithmic complexity of the solution does not matter.

```
let maxCommon = ([head1,...tail1], [head2,...tail2], len = 0) => {
    if ( typeof head1 === 'undefined' ||
         typeof head2 === 'undefined' ) {
        /* Write code here */
    }
    if ( head1 === head2 ) /* Write code here */
    let firstBranch = /* Write code here */
    let secondBranch = /* Write code here */
    return Math.max( ...[len, firstBranch, secondBranch ] );
}

maxCommon( '123', '1' );
// 1

maxCommon( '11111', '11f111g');
// 3

maxCommon( 'abc', '111cab' );
// 2
```

Solution:

We will use an optional `len` argument to store the number of character matches before the current iteration of `maxCommon` was called.

We will use recursion to process the strings.

If any of the strings have a length of `0`, either `head1`, or `head2` becomes `undefined`. This is our exit condition for the recursion, and we return `len`, i.e. the number of matching characters right before one of the strings became empty.

If both strings are non-empty, and the heads match, we recursively call `maxCommon` on the tails of the strings, and increase the length of the counter of the preceding common substring sequence by `1`.

If the heads don't match, we remove one character from either the first string or from the second string, and calculate their `maxCommon` score, with `len` initialized to `0` again. The longest string may

either be in one of these branches, or it is equal to `len`, counting the matches preceding the current strings `[head1,...tail1]` and `[head2,...tail2]`.

```
maxCommon = ([head1,...tail1], [head2,...tail2], len = 0) => {
    if ( typeof head1 === 'undefined' ||
         typeof head2 === 'undefined' ) {
        return len;
    }
    if ( head1 === head2 ) return maxCommon( tail1, tail2, len+1 );
    let firstBranch = maxCommon( tail1, [head2, ...tail2], 0 );
    let secondBranch = maxCommon([head1,...tail1], tail2, 0 );
    return Math.max( ...[len, firstBranch, secondBranch ] );
}
```

Note that this solution is very complex, and requires a magnitude of `#(s1)! * #(s2)!` steps, where `s1` and `s2` are the two input strings, and `#(...)` denotes the length of a string. For those practising for a Google interview, note that you can solve the same problem in steps of `O( #(s1) * #(s2) )` magnitude using dynamic programming.

Back to the Exercises