

Introduction

A basic overview of what to expect in this course

ES6 is the sixth release of ECMA Script, more commonly known as JavaScript. ES6 revamped the way we use JavaScript, bringing in several new features which greatly simplify writing and reading the language.

We'll explore all the fundamentals of ES6 and look at how they practically improve our JavaScript experience.

To start things off, we will discuss Functions, which play an essential role in JavaScript. As you'll see, ES6 has refined function syntax, making them easier and more flexible.

So let's get right into it. Welcome aboard!

var vs let

introduction to the `let` keyword for declaring block-scoped variables; and the dangers of scoping, such as the temporal dead zone

Variables declared with `var` have function scope. This means that they are accessible inside the function/block they are defined in. Take a look at the following code:

```
var guessMe = 2;
console.log("guessMe: "+guessMe); // A: guessMe is 2
( function() {
    console.log("guessMe: "+guessMe); // B: guessMe is undefined
    var guessMe = 5;
    console.log("guessMe: "+guessMe); // C: guessMe is 5
} )();
console.log("guessMe: "+guessMe); // D: guessMe is 2
```



Comment B may surprise you if you have not heard of hoisting. If a variable is declared using `var` inside a function, the Javascript engine treats them as if they are declared at the top of a functional scope. However, if that variable has been declared outside the function, it has a global scope regardless of where the actual declaration occurs. This is called **hoisting**.

```
() => {
    JAVASCRIPT_STATEMENTS;
    var guessMe = 5;
};
//accessing guessMe will give an error here
```



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
    //console.log(guessMe); <-This gives an error
    let guessMe = 5;
    console.log("guessMe: "+guessMe); // 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();
```



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

```
function logName() {
```



```
console.log( 'name:', name );
let name = 'Ben';

}
logName();
```



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 the 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;
console.log( 'guessMe: ', guessMe );// A: guessMe is 1
{
    // Temporal Dead Zone of guessMe
    //console.log( 'guessMe: ', guessMe ); <- This would give an error
    let guessMe = 2;
    console.log( 'guessMe: ', guessMe );// C: guessMe is 2
}
console.log( 'guessMe: ', guessMe );// 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.

Now, let's talk about the `const` keyword.

Constants

introduction to Javascript constants; potential errors, such as the temporal dead zone

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

```
const PI = 3.1415;  
PI = 3.14;
```



This code will give uncaught TypeError: Assignment to constant variable.

Not initializing a constant also throws an error:

```
const PI; //const PI=8; won't give an error
```



Uncaught SyntaxError: Missing initializer in const declaration

Const may also have a *temporal dead zone*.

```
// temporal dead zone of PI  
//PI cannot be accessed here  
const PI = 3.1415;  
// PI is 3.1415 and its value is final  
//PI can be accessed here
```



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

In the next lesson, let's discuss the use cases of `var`, `let` and `const` formally.

let, const, and var

the use case of let, const and var; avoiding errors, such as the temporal dead zone, and using a linter

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.

Now, let's do some exercises before learning new concepts.

Exercise on Function Scope, Block Scope, Constants

This exercise will test your knowledge on the scope and sequence of function execution. You will have to keep track of the function's path using `console.log()`.

Exercise 1:

Check the following code snippet riddle:

Determine the values logged to the console before you execute it.

```
'use strict';

var guessMe1 = 1;
let guessMe2 = 2;

{
    try {
        console.log( guessMe1, guessMe2 ); // (A)
    } catch(err) {
        console.log("Error");
    }

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

console.log( guessMe1, guessMe2 ); // (C)

const print_func = () => {

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

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



Explanation:

The output in the console will be as follows:

The output in the console will be as follows:

```
guessMe1 and/or guessMe2 not defined  
1 3  
1 2  
1 2
```

Let's examine the six console logs one by one.

(A): Here, `guessMe1` holds the value original value of `1` as it has not been defined again in the block. The error is thrown because of `guessMe2`. Why? This is because we have defined it with the `let` keyword in line 14 of the block:

```
let guessMe2 = 3;
```

If we remove the `try` method and simply print line 9:

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

an error will still tell you that `guessMe2` is uninitialized. Hence `Error` was the first output.

(B): This `console.log()` will work because now

`guessMe1 == 1` and `guessMe2 == 3`.

However, keep in mind that these values are only true for this particular block.

(C): This will print the original values of both variables as we have moved outside the scope of the block above.

`guessMe1 == 1` and `guessMe2 == 3`.

(D): The `print_func` function is never called in the program, hence its contents never run.

(E): Just as for (D), the `print_func` function is never called in the program, hence its contents never run.

(F): The `console.log` in this line is the same as that in (C). The values of both

variables have not been altered anywhere in the global scope. Hence, the output is the same.

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.

```
//Edit this code  
'use strict';  
  
var guessMe1 = 1;  
let guessMe2 = 2;  
  
{  
    try {  
        console.log( guessMe1, guessMe2 ); // (A)  
    } catch(err){console.log("Error");}  
  
    let guessMe2 = 3;  
    console.log( guessMe1, guessMe2 ); // (B)  
}  
  
console.log( guessMe1, guessMe2 ); // (C)  
  
const print_func = () => {  
  
    console.log( guessMe1 ); // (D)  
    var guessMe1 = 5;  
    let guessMe2 = 6;  
    console.log( guessMe1, guessMe2 ); // (E)  
};  
  
console.log( guessMe1, guessMe2 ); // (F)
```



Explanation:

The code in the solution is pretty self-explanatory. However, this is not the

only solution.

The basic idea is to clearly define `guessMe1` and `guessMe2` before each `console.log()` call.

Fat Arrow Syntax

introduction to fat arrow syntax and its advantages

An arrow function expression has a shorter syntax than a regular function expression and does not have its own `this`, arguments, `super`, or `new.target`. Let's write an ES5 function to sum two numbers.

```
var sum = function( a, b ) {
    return a + b;
};
console.log(sum(2, 3));
```



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;
};
console.log(sum(2, 3));
```



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;
console.log(sum(2, 3));
```



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

```
var square = a => a * a;  
console.log(square(2));
```



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

In the next lesson, let's discuss context binding in ES6.

Context Binding

introduction to context binding, animation of a ball using the `setInterval` method

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, 10000, 10000 );
```



The code times out because the `setInterval` method runs an infinite number of times.

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

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 );
```



The code times out because the `setInterval` method runs an infinite number of times.

This solution is still a bit awkward, as we have to maintain the `self` and `this` references. It is easy to make a mistake and use `this` instead of `self` somewhere in your code. 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 );
}
```

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 ) {
```

```
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 );
```



The code times out because the `setInterval` method runs an infinite number of times.

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.

Exercise on Arrow Functions

You will be writing and modifying function declarations using the arrow syntax.

Exercise 1:

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

```
// Write your code here!
```



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.

```
//Write your code here!
```



```
//the name of your function should be sum. Program will throw an error if the name is not sum
```



Explanation:

Reduce works with an accumulator to store the value associated with reducing the array, and takes two arguments:

- the initial value of the accumulator,
- a function to define the operation between the accumulator and the upcoming element of the array. Reduce performs the following operations:
 - `(0,1)=>0+1` becomes 1,
 - `(1,2)=>1+2` becomes 3,
 - `(3,3)=>3+3` becomes 6,

- $(6, 4) \Rightarrow 6+4$ becomes 10,

- $(10, 5) \Rightarrow 10+5$ becomes 15.

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();
```



Explanation:

- 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.

Hacks in ES5

optional function arguments and their replacement, limitations of ES5 shortcuts

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

```
function addCalendarEntry( event, date, len, timeout ) {  
    date = typeof date === 'undefined' ? new Date().getTime() : date;  
    len = typeof len === 'undefined' ? 60 : len;  
    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 for logical operations.

```
function addCalendarEntry( event, date, len, timeout ) {  
    date = date || new Date().getTime();  
    len = len || 60;  
    timeout = timeout || 1000;  
    // ...  
}  
addCalendarEntry( 'meeting' );
```



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

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

While this approach looks nice on paper, shortcuts are sometimes not flexible enough. All false values are treated in the same way, including `0`, `''`, `false`. Sometimes, we may want to treat a `0` differently than an `undefined` indicating the absence of a value.

Now, let's see how we handle this using ES6.

The ES6 way

default values and using a variable number of arguments in functions

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(),  
    len = 60,  
    timeout = 1000 ) {  
  
    return len;  
}  
var add=addCalendarEntry( 'meeting' );  
console.log(add); //outputs the default value set earlier
```



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 to

```
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 hold. `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.

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

The `arguments` array is not affected

In earlier versions of JavaScript, we often used the `arguments` array to handle a variable number of arguments:

```
function printArgs() {  
    console.log( arguments );  
}  
  
printArgs( 'first', 'second' );
```



Bear in mind that the `arguments` array is not affected by the default parameter values in any way.

```
function printArgs( first = 'No arguments' ) {  
    console.log( arguments );  
}  
  
printArgs();
```



To get a better understanding, see Exercise 3 in the next lesson.

Exercise on Default Arguments

It's time to play with function arguments. These exercises build further upon what we learned in the previous lesson.

Exercise 1:

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

The `setTimeout()` method can be used to specify the time delay before a function is executed.

| | |
|---|---|
|  Exercise 1 |  Solution |
|---|---|

```
function executeCallback( callback, delay ) {
  console.log('Delay: ' + delay);
}

//Edit above this line
executeCallback( () => console.log('Done'));
```

Explanation:

The main objective of this exercise was to define a default argument for `delay`.

Using ES6 conventions, we can simply state the default value in the function arguments:

```
delay = 1000
```

The statement above sets the delay at 1000. You can define your own delay by passing it into the function. Otherwise, the `executeCallback()` function will execute after 1000 milliseconds.

If we want to use the ES5 syntax, we'd have to write something like this:

```
delay = delay || 1000;
```

We use the built in `setTimeout` method to start the timer and execute our function.

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.

Select the show console button in the widget below to see your output.

```
function printComment( comment, line ) {  
    console.log( line, comment );  
}  
  
//Edit above this line  
  
for (var i = 1; i <= 5; i++)  
    printComment('I should be lineNumber ' + i);
```



Explanation:

We create a new variable `lineNumber` which is initialized to `1`. This way, we are updating a variable rather than changing the default argument of the `printComment` function each time.

Exercise 3:

Determine the values written to the console before executing this script.

```
function argList( productName, price = 100 ) {  
    console.log( arguments.length ); // (A)  
    console.log( productName === arguments[0] ); // (B)  
    console.log( price === arguments[1] ); // (C)  
};  
  
argList( 'Krill Oil Capsules' );
```





Explanation:

The answers are fairly simple. Let's look at them one by one.

(A): Even though we have specified that our function can have 2 arguments, we've already learned that it isn't necessary to provide both. In `argList('Krill Oil Capsules');`, we specify the first argument. Hence, the length of our arguments array is `1`.

(B): Here, we are simply checking whether or not the `productName` argument exists. Since, we passed `'Krill Oil Capsules'` into the function, the console will display `true` for this statement.

(C): This is the interesting part. The `price` argument has a default value of 100. However, it is not considered to be an argument which we have passed into the function. As a result, the comparative statement `price === arguments[1]` will return `false` because `arguments[1]` is `undefined`.

Prototypal Inheritance

prototype-based inheritance and the introduction to classes in ES6

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

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. Others seem to have launched a holy war against classes, claiming that the class syntax is flawed.

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 judgment justifies that classes should be used, go for it.

I use classes on a regular basis, and my [React-Redux tutorials](#) also make use of the class syntax.

Not knowing the class syntax is a disadvantage.

Judgment on the class syntax, or offering alternatives are beyond the scope of this section. In the next lesson, I will talk about prototypal inheritance in more detail.

Prototypal Inheritance in ES5

introduction to prototypal inheritance using an example (rectangle "is a" shape)

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`.

If this example is your first encounter with classes, you may conclude that it is not too intuitive to read this example. Especially the two lines building the prototype chain seem far too complex to write. The definition of classes are also separated: we define the constructor and other methods in different places.

Now, let's talk about classes in ES6.

Inheritance - The ES6 way

inheritance in ES6, accessing constructors and other class methods at various places in the code, introduction to concise method syntax and Javascript code conventions

Let's see the ES6 version of inheritance. As we'll see later, the two versions are not equivalent, 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( "Area:\t\t" + rectangle.getArea() );  
console.log( "Color:\t\t" + rectangle.getColor() );  
console.log( "toString:\t" + 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 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 can call methods of their class and all classes that are higher in the inheritance chain.
- Instantiation works in the 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](#).

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 ES6 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.

Convention: 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.

Let's talk about the `super` keyword in the next lesson.

Super

introduction to the super keyword and its importance in classes that inherit from a parent class

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` with the argument list of the constructor.

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

new B()
//outputs B
//but also gives an uncaught ReferenceError: this is not defined(...)

class C extends A {}

new C()
//> A

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



In the next lesson, we will talk about another concept: shadowing.

Shadowing

Redefining methods of the parent class in a child class.

Methods of the parent class can be redefined in the child class. This redefinition of the function using the same name is called **shadowing**. Consider the following code:

```
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' );
```



In the code above, the child class *SuperUser* has redefined the function *hasAccess*.

Now let's talk about abstract classes in Javascript.

Creating Abstract Classes

introduction to an abstract class and its implementation using a stock trading example

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, 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 the 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 ( new.target === 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 class whose constructor was first called in the inheritance chain.

In the next lesson, let's talk about getters and setters.

Getters and Setters

introduction to getters and setters, and their advantages

Getters and setters are used to create computed properties.

```
class Square {  
    constructor( width ) { this.width = width; }  
    get area() {  
        console.log('get area');  
        return this.width * this.width;  
    }  
}  
  
let square = new Square( 5 );  
  
console.log(square.area)
```



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 );  
let print:
```



```

let print,
print=square.width;
console.log(print+'\n')
//> 5

print=square.height
console.log(print+'\n')
//> get height
//> 5

print=square.height = 6
console.log(print+'\n')
//> set height 6
//> 6

print=square.width
console.log(print+'\n')
//> 6

print=square.area
console.log(print+'\n')
//> get area
//> get height
//> 36

print=square.width = 4
console.log(print+'\n')
//> 4

print=square.height
console.log(print+'\n')
//> 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:** It allows you to hide properties that are retrievable or settable through getters or setters.
- **Encapsulation:** You can couple other functionality with getting/setting a value.
- **Defining a public interface:** It is possible to keep these definitions constant and reliable, while you are free to change the internal

representation used for computing these fields. This comes in 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.

Now, let's move on to static methods.

Static Methods

introduction to static methods in Javascript

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

```
class C {  
    static create() { return new C(); }  
    constructor() { console.log( 'Accessing constructor from the class'); }  
}  
  
var c = C.create();  
//> constructor  
c.create(); //this will give an error
```



Now, let's do some exercises in the next lesson.

Exercise on Classes

We'll apply our knowledge of classes to a video game model in which we there are two types of characters.

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.

```
class Character {  
    constructor( id, name, x, y ) {  
        //Write your code here  
    }  
  
    get position() {  
        //Write your code here  
    }  
}  
  
//Define Player Character and NonPlayerCharacter classes here  
  
function createPlayer( id, name ) {  
    //Write your code here  
}  
  
function createNonPlayer( id, name ) {  
    //Write your code here  
}
```



Explanation:

This exercise has many solutions. We just used one. For the sake of simplicity, we chose not to model the game field. We placed x and y inside the character objects as coordinates.

At this stage, there was no difference between player and non-player characters.

We still created them to match the requirements.

Exercise 2: Judge For Yourself

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.

 WriteHere Solution

```
// Write your code here!
```



Explanation:

We modeled 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.

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;
```

Object Property - Shorthand Notation

introduction to destructuring in ES5 and its equivalent notation in ES6

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*.

Suppose we have the following ES5 code:

```
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.

We will talk about this concept in more detail in the following lesson.

Destructuring Examples

destructuring, associativity and ES6 notation

Consider the following examples:

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



In a destructuring expression $L = R$, we take the right value R , and break it down so that the new variables in L can be assigned a value. In the above code, we used the object property shorthand notation.

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



Without this shorthand notation, our code will look like this:

```
let {  
    email: email,  
    nextLesson: nextLesson  
} = user;  
  
console.log(user)
```



In this case, the above code is equivalent with the following ES5 assignments:

```
let email = user.email;  
let nextLesson = user.nextLesson;
```

Note that the above two lines are executed in parallel. First, the `R` value is fully evaluated before assigning it to left values. For instance, let's increment the variables `a` and `b` using destructuring:

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



If we wanted to transform this destructuring assignment to ES5, we would write the following code:

```
let [a, b] = [5, 3];  
[a, b] = [a + 1, b + 1];  
var temp_a = a + 1;  
var temp_b = b + 1;  
a = temp_a;  
b = temp_b;  
console.log(a, b)
```



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

```
console.log({email, nextLesson} = user);
```



As a consequence, don't expect destructuring to be used as an alternative for filtering 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 `L`, variable assignments may take place on any depth.

```
let user2 = {email, nextLesson} = user;
console.log( user2 === user, user2.name );
```



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 );
```



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

In the next lesson, we will discuss deeper destructuring.

Deeper Destructuring, destructuring functions, and pitfalls

using default values and assigning undefined in destructuring

Destructuring objects and arrays in any depth is possible. We can also use default values. 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:\t\t"+first+"\nsecond:\t\t"+second+"\nthird:\t\t"+third+  
            "\nfourth:\t\t"+fourth+"\nfifth:\t\t"+fifth+"\nsixth:\t\t"+sixth+  
            "\nseventh:\t\t"+seventh+"\neighth:\t\t"+eighth);
```



Notice that the `null` value of the `sixth` field behaves in the same way as a default argument value of function arguments.

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

assignments. The parameters of a function call act as the respective right values of destructuring assignments.

You will use destructuring function arguments in exercises 5 and 6.

```
function f( l1, l2 )  
{  
  console.log(l1, l2);  
}  
let R1 = "R1";  
let R2 = "R2";  
f( R1, R2 ); // executes l1 = R1, l2 = R2
```



Destructuring pitfalls

Now let's talk about the possible bugs that can occur as a result of 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 );
```



Error: 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' ()

Now, let's do some exercises before learning new concepts.

Exercise on Destructuring

Let's try writing a few destructuring assignments and see how they simplify our code. Good luck!

Exercise 1:

Swap two variables using one destructuring assignment.

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

//Write Code here
```



Explanation

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.

Exercise 2:

Complete the function below 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;
}

```



Exercise 3:

Create one destructuring expression that declares exactly one variable to retrieve `x.A[2]`. Return the value in a new variable called `A_2`.

```

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

//Write your Code here
let A_2 = "";

```



Explanation

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.

Exercise 4:

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. The default values are the same as in the example configuration object.

```
function drawChart( data, /* Write your code here */ ) {  
    // do not implement chart drawing functionality or anything  
    // return {chartType, bullColor, bearColor, days};  
};
```



Rest Parameters

introduction to rest parameters, calling functions with a variable number of arguments and its potential errors

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 more compact than the equivalent ES5 code.

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 );
```



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) => // using rest parameters  
  {  
    console.log( args );  
  }  
) ( 1, 'Second', 3 );
```



Note that the argument list containing the 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.

In the next lesson, we'll discuss the spread operator.

Spread Operator

an introduction to spread operator, limitations of rest parameter and their solution using the spread parameter

In ES5, we often used the `apply` method to call a function with a variable number of arguments. The spread operator makes it possible to achieve the exact same thing in a more 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;  
}  
  
console.log(sumArgs( 1, 2, 3, 4, 5 ));
```



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 down 100 parameters. In other cases, the number of parameters is not known. This was when the `apply` method of JavaScript was used in ES5.

```
var arr = [];  
for( var i = 0; i < 100; ++i ) arr[i] = Math.random();  
console.log("Sum:\t"+sumArgs.apply( null, arr ));
```



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

of an array, transforming them into a parameter list.

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



As opposed to rest parameters, there are no restrictions on the location of the *Spread operator* 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 ];
```



In the next lesson, let's move on to destructuring using the spread operator.

Destructuring with the Spread Operator

destructuring in Javascript using spread operator, and it's comparison with destructuring using the rest parameter

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 );
```



Just like 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]
```



Like the rest parameter in functions, on the left side of a destructuring assignment, we are only allowed to use the rest parameter as the last element of an array.

```
[...A,] = [1,2]
```



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

Exercise on Spread Operator and Rest Parameters

Get a hang of the spread operator and rest parameters by trying out these exercises. Remember, the point is to think differently and move away from ES5 conventions.

Exercise 1:

Make a shallow copy of an array of any length in one destructuring assignment! The given array contains random integers, can be of any length and is called `originalArray`. Call the new array you make `clonedArray`. Remember to use this exact spelling or your code won't compile.

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](#).

```
// Original array randomly generated
console.log(originalArray);
let clonedArray = []
```



Exercise 2:

Determine the value logged to the console without running it.

index.js

explanation.txt

```
let f = () => [..."12345"];
let A = f().map( f );
console.log( A );
```



Explanation

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

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']`.

Exercise 3:

Create an 10x10 matrix of `null` values.

```
// fill this array
let nullArray = [];
```



Explanation

Study the fill method for more details [here](#). 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.

Exercise 4:

Rewrite the `sumArgs` function given 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;
}
```



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. Remember to keep the name as `maxCommon()` or your code won't compile.

```
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 = 0 /* Write code here */
    let secondBranch = 0 /* Write code here */
    return Math.max( ...[len, firstBranch, secondBranch] );
}
```



Explanation

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;
    }
    /* Write code here */
}
```



```
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 practicing for a Google interview, note that you can solve the same problem in steps of $O(\#(s1) * \#(s2))$ magnitude using dynamic programming.

Equality

equality in JavaScript ES5 using '==' and '==='; and introduction to the `Object.is()` method in ES6, which is very similar to '==='

In this lesson, we will cover object and function improvements in ES6. As with most updates in ES6, we will be able to solve the same problems as before, with less code, and more clarity.

We will start with a nitpicky subject: comparisons. Most developers prefer `==` to `==`, as the first one considers the type of its operands.

In ES6, `Object.is(a, b)` provides *same value equality*, which is almost the same as `==` except the following differences:

- `Object.is(+0, -0)` is `false`, while `-0 === +0` is `true`
- `Object.is(NaN, NaN)` is `true`, while `NaN === NaN` is `false`

```
console.log(Object.is(+0, -0)) // is `false`, while
console.log(-0 === +0)          // is `true`  
  
Object.is(NaN, NaN)           // is `true`, while
console.log(NaN === NaN)        // is false
```



I will continue using `==` for now, and pay attention to `NaN` values, as they should normally be caught and handled prior to a comparison using the more semantic `isNaN` built-in function. For more details on `Object.is`, visit [this thorough article](#).

In the next lesson, we'll talk about another important concept - class hierarchy using mixins.

Mixins

introduction to inheritance and the importance of mixins in JS; and cloning objects in ES6

Heated debates of composition over inheritance made mixins appear to be the winner construct for composing objects. Therefore, libraries such as UnderscoreJs and LoDash created support for this construct with their methods `_.extend` or `_.mixin`.

In ES6, `Object.assign` does the same thing as `_.extend` or `_.mixin`.

Why are mixins important?

They are important because the alternative of establishing a class hierarchy using inheritance is inefficient and rigid.

Suppose you have a view object, which can be defined with or without the following extensions:

- validation
- tooltips
- abstractions for two-way data binding
- toolbar
- preloader animation

Assuming the order of the extensions does not matter, 32 different view types can be defined using the five enhancements above. To fight the combinatoric explosion, we take these extensions as mixins and extend our object prototypes with the extensions that we need.

For instance, a validating view with a preloader animation can be defined in the following way:

```
let View = { ... };
let ValidationMixin = { ... };
let PreloaderAnimationMixin = { ... };
```



```
let PreloaderAnimationMixin = ...;
let ValidatingMixinWithPreloader = Object.assign(
  {},
  View,
  ValidationMixin,
  PreloaderAnimationMixin
);
```

Why do we extend the empty object?

Because `Object.assign` works in a way that it extends its first argument with the remaining list of arguments. This implies that the first argument of `Object.assign` may get new keys, or its values will be overwritten by a value originating from a mixed in object.

Syntax for `Object.assign`

The syntax for calling `Object.assign` is as follows:

```
Object.assign( targetObject, ...sourceObjects )
```

The return value of `Object.assign` is `targetObject`. The side-effect of calling `Object.assign` is that `targetObject` is mutated.

`Object.assign` makes a *shallow copy* of the properties and operations of `...sourceObjects` into `targetObject`.

For more information on *shallow copies* or cloning, check my article on [Cloning Objects in JavaScript](#), and check the first exercise of the lesson on the [Spread operator and Rest parameters](#).

Consider the following code:

```
let horse = {
  horseName: 'QuickBucks',
  toString: function() {
    return this.horseName;
  }
};

let rider = {
  riderName: 'Frank',
  toString: function() {
```

```
        return this.riderName;
    }

};

let horseRiderStringUtility = {
    toString: function() {
        return this.riderName + ' on ' + this.horseName;
    }
}

let racer = Object.assign(
    {},
    horse,
    rider,
    horseRiderStringUtility
);

console.log( racer.toString() );
```



Had we omitted the `{}` from the assembly of the `racer` object, seemingly, nothing would have changed, as `racer.toString()` would still have been `"Frank on QuickBucks"`. However, notice that `horse` would have been `==` equivalent to `racer`, meaning, that the side-effect of executing `Object.assign` would have been the mutation of the `horse` object.

Shorthand for Creating and Destructuring Objects

objects using pre-defined variables during initialization

In the scope where an object is created, it is possible to use other variables for initialization.

```
let shapeName = 'Rectangle', a = 5, b = 3;  
  
let shape = { shapeName, a, b, id: 0 };  
  
console.log( shape );  
// { shapeName: "Rectangle", a: 5, b: 3, id: 0 }
```



It is possible to use this shorthand in destructuring assignments for the purpose of creating new fields:

```
let { x, y } = { x: 3, y: 4, z: 2 };  
  
console.log( y, typeof y );  
// 4 "number"
```



Now, let's move on to object keys in the next lesson.

Computed Object Keys

using string keys to access their corresponding objects, and behavior of the `toString` method in Javascript

In JavaScript, objects are associative arrays (hashmaps) with String keys. We will refine this statement later with ES6 symbols, but so far, our knowledge is limited to string keys.

It is now possible to create an object property inside the object literal using the bracket notation:

```
let arr = [1,2,3,4,5];

let experimentObject = {
  [ arr.length ]: 2,
  [ arr ]: 1,
  [ {} ]: 3,
  arr
}

console.log(experimentObject)
```



The object will be evaluated as follows:

```
{
  "5": 2,
  "1,2,3,4,5": 1,
  "[object Object)": 3,
  "arr": [1,2,3,4,5]
}
```

We can use any of the above keys to retrieve the above values from `experimentObject`. Consider the following snippet to print these values:

```
console.log(experimentObject.arr)          // [1,2,3,4,5]
console.log(experimentObject[ 'arr' ])      // [1,2,3,4,5]
console.log(experimentObject[ arr ])        // 1
// [1,2,3,4,5]
```



```
console.log(experimentObject[ arr.length ]) // 2  
console.log(experimentObject[ '[object Object]' ]) // 3  
console.log(experimentObject[ experimentObject ]) // 3
```



Conclusion:

- Arrays and objects are converted to their `toString` values.
- `arr.toString()` equals the concatenation of the `toString` value of each of its elements, joined by commas.
- The `toString` value of an object is `[object Object]` regardless of its contents.
- When creating or accessing a property of an object, the respective `toString` values are compared.

Now, let's talk about the various operations of objects in the next lesson.

Shorthand for Defining Operations in Objects

'Concise method syntax' - a new feature of ES6

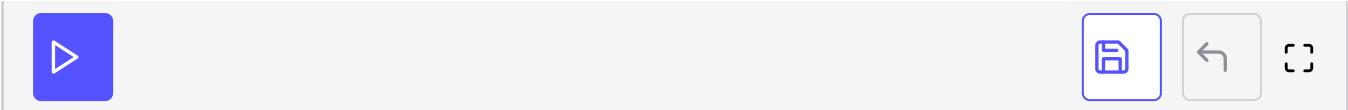
As the name suggests, function declaration using the concise method syntax requires less typing, but that's not the only advantage.

Let's declare a `logArea` method in our `shape` object:

```
let shapeName = 'Rectangle', a = 5, b = 3;

let shape = {
  shapeName,
  a,
  b,
  logArea() { console.log( 'Area: ' + (a*b) ); },
  id: 0
};

shape.logArea();
```



Concise Method Syntax

Notice that in ES5, we would have to write `: function` between `logArea` and `()` to make the same declaration work. This syntax is called the *concise method syntax*. We first used the concise method syntax in [Chapter 4 - Classes](#).

Concise methods have not made it to the specification just to shave off 10 to 11 characters from the code. They also make it possible to access prototypes more easily.

This leads us to the next section on object prototype extensions and super calls.

Object Prototype Extensions and Super Calls

get and set the prototype of an object, using the 'super' keyword to make code more concise

The `Object` API has been extended with two methods to get and set the prototype of an object:

- `Object.getPrototypeOf(o)` returns the prototype of `o`
- `Object.setPrototypeOf(o, proto)` sets the prototype of `o` to `proto`

```
let proto = {
  whoami() { console.log('I am proto'); }
};

let obj = {
  whoami() {
    super.whoami();
    console.log('I am obj');
  }
};

console.log( Object.getPrototypeOf( obj ) );
// {}

Object.setPrototypeOf( obj, proto );

obj.whoami();
// I am proto
// I am obj
```



The following points are worth noting:

- The prototype of an object is `Object` by default.
- `Object.setPrototypeOf` can change this prototype to any object
- The `super` call saves some typing. Without `super`, we would have to write the following:



Note that the concise method notation is mandatory for us to use the `super` keyword. Otherwise, a JavaScript error is thrown.



```
let proto = {
    whoami() { console.log('I am proto'); }
};

let obj = {
    whoami: function() {
        super.whoami();
        console.log('I am obj');
    }
};

Object.setPrototypeOf( obj, proto );

obj.whoami();
```



Uncaught SyntaxError: 'super' keyword unexpected here

Now, let's solve some exercises before learning new concepts.

Exercise on Objects in ES6

Now, we'll create a "Basket" object using all the concepts we have learned in this section.

Exercise 1:

Suppose an array of `firstName`, `email`, `basketValue` triples are given. Create ONE JavaScript expression that puts a default value of `'-'` and `0` to the `firstName` or `basketValue` fields respectively, whenever the `firstName` or the `basketValue` keys are missing. Remember to name your new object, `newBaskets` or your code won't run.

```
// the baskets object is randomly generated
console.log("baskets: ");
console.log(baskets);

let newBaskets = {}
```



Explanation

The hard part of the solution is that we need one JavaScript expression. When it comes to transforming the value of an array, we usually use the map method. As we learned in the first lesson, the shortest way of writing a map function is through using the arrow syntax. Inside the map, we have to fill in the default values in item. If a key is already given in item, it takes precedence. We will use `Object.assign`. For each element of the basket array, we will create an object of default values, and we extend it with the element of the array, mixing in all the properties. If a key exists in both objects, the value in `item` is kept.

Exercise 2:

Create a prototype object with the following methods:

- `addToBasket(value)` adds `value` to the basket value

- `addToBasket(value)` adds `value` to the basket value,
- `clearBasket()` sets the basket value to 0
- `getBasketValue()` returns the basket value
- `pay()` logs the message `{getBasketValue()} has been paid`, where `{getBasketValue()}` is the return value of the method with the same name. We can pay for the same basket as many times as we'd like. Name your object, `basketProto` your code won't compile otherwise.

```
let basketProto = {
  //write-your-code-here
}
```



Exercise 3:

Create an object `myBasket`, and set its prototype to the object created in Exercise 2 (it has already been prepended to the given code). Create an array field in `myBasket`, containing all the items that you purchase in the following format:

```
{ itemName: 'string', itemPrice: 9.99 }
```

Redefine the `addToBasket` method such that it accepts an `itemName` and an `itemPrice`. Call the `addToBasket` method in the prototype for the price administration, and store the `itemName - itemPrice` data locally in your array. Make sure you modify the `clearBasket` method accordingly.

```
myBasket.addToBasket( 'Cream', 5 );
myBasket.addToBasket( 'Cake', 8 );
myBasket.getBasketValue();           // 13
myBasket.items
// [{ itemName: 'Cream', itemPrice: 5},
// { itemName: 'Cake', itemPrice: 8 }]
myBasket.clearBasket();
myBasket.getBasketValue();          // 0
myBasket.items                     // []
```

```
//basketProto object given
```

```
let basketProto = {
```



```
  value: 0
```

```

value: 0,
addToBasket( itemValue ){
  this.value += itemValue;
},
clearBasket() {
  this.value = 0;
},
getBasketValue(){
  return this.value;
},
pay() {
  console.log( this.getBasketValue() + ' has been paid' );
}

};

//write-your-code-here

```



Explanation

Notice,

- the shorthand of constructing `{ itemName, itemPrice }`,
- the short method syntax enabling the super calls,
- the easy way of setting prototypes.

ES6 is powerful and clean.

Exercise 4:

Extend your solution in Exercise 3 by adding a `removeFromBasket(index)` method. The parameter `index` should be the index of the element in the array that you would like to remove.

```

myBasket.addToBasket( 'Cream', 5 );
myBasket.addToBasket( 'Cake', 8 );
myBasket.addToBasket( 'Cookie', 2 );
myBasket.removeFromBasket( 1 );

myBasket.getBasketValue();           // 7
myBasket.items
// [{ itemName: 'Cream', itemPrice: 5 },
// { itemName: 'Cookie', itemPrice: 2 }]
```

```

//basketProto object given

```

let basketProto = {
 value: 0,
 addToBasket(itemValue){
 this.value += itemValue;
 },
 clearBasket() {
 this.value = 0;
 },
 getBasketValue(){
 return this.value;
 },
 pay() {
 console.log(this.getBasketValue() + ' has been paid');
 }
};

let myBasket = {
 items: [],
 addToBasket(itemName, itemPrice)
 {
 this.items.push({ itemName, itemPrice });
 super.addToBasket(itemPrice);
 },
 clearBasket() {
 this.items = [];
 super.clearBasket();
 }
};
Object.setPrototypeOf(myBasket, basketProto);

```



The hard part of this task is synchronizing the value stored in the prototype. By the time you reach Exercise 4, you might have noticed that this redundancy is not optimal from the perspective of modeling the baskets.

The index needs to be valid. For simplicity, I have omitted to check for floating point indices. Notice that it is not possible to use arrow functions here, because of the properties of fat arrows you learned in Lesson 1. Using arrow functions preserves the external context, which is not `myBasket`. Therefore, accessing this would not give us the desired results. The call `A.splice( index, 1 )` method removes `A[index]` from `A` mutating the original array, and returns an array of the removed elements. As we only removed one element, we find our element at index 0.

We already got used to the handy super call in Exercise 3. I added this method as an extension of `myBasket` on purpose to show you that the super call cannot be used with this function syntax. In fact, this form is discouraged. Use the concise method syntax, and define all methods of an object at once. In the

unlikely case you still needed this format, you would have to get the prototype

of the current object, and call its method by setting the context. With the concise method syntax, our solution would look like this:

```
let myBasket = { items: [],
 addToBasket(itemName, itemPrice){ this.items.push({ itemName, itemPrice }); super.addToBasket(itemName, itemPrice) },
 clearBasket() {
 this.items = [];
 super.clearBasket();
 },
 removeFromBasket(index) {
 if (typeof index !== 'number' ||
 index < 0 ||
 index >= this.items.length) return;
 let removedElement = this.items.splice(index, 1)[0]; super.removeFromBasket(removedElement);
 }
};
```



# Stacks

introduction to stacks and stack size limits in Javascript

We will need a basic understanding of how stacks work to understand the next section. If you have never heard of a stack data structure, I will summarize it in this section. I still encourage you to do more thorough research before continuing, as you will need it in your programming career.

Imagine a stack like a JavaScript array whose elements are not accessible. Suppose stack S is given. You can only execute the following operations on it:

- `S.length` : checks the length of the stack
- `S.push(element)` : pushes an element to the stack
- `S.pop()` : removes the top element from the stack and returns it

You can neither access nor modify any elements of the stack other than the element at position `S.length - 1`. In Exercise 1, you will have a chance to implement a stack in ES6.

There are two types of memory available to you: the **stack** and the **heap**. The heap is used for **dynamic memory allocation**, while the stack is used for **static memory allocation**. Accessing the stack is very fast, but the size of the stack is fixed.

A *stack frame* is created for the global scope. Then, for each function call, another stack frame is added to the top. These frames stack on top of each other.

When executing JavaScript code, you get a stack with limited size to work with. To get an idea of typical stack size limits in practice, look at [this page](#).

Regardless of the browser, iterating from zero to a million in a for loop is an easy task.

```
console.time('for loop');
let sum = 0;
for (let i = 0; i < 1000000; ++i) {
 sum += i;
}
console.timeEnd('for loop');
// for loop: 119.11ms
```



Doing the same iteration as a recursive call is an arduous task that most browsers are not able to solve because of the stack limit. Execution is fast, but the stack limit will not make it possible to execute a million calls.

```
function sumToN(n) {
 if (n <= 1) return n;
 return n + sumToN(n - 1);
};

console.time('recursion');
console.log(sumToN(1000000));
console.timeEnd('recursion');
```



> Uncaught RangeError: Maximum call stack size exceeded...

Let's examine how the following program is executed on the stack:

```
function sumToN(n) {
 if (n <= 1) return n;
 return n + sumToN(n - 1);
};

console.log(sumToN(2));
```



1. Initially, a reference to the `sumToN` function is created on the global stack frame.
2. Once `sumToN(2)` is called, another stack frame is created, containing the value of `n`, the expected return value, and a reference to `sumToN`.
3. Once `sumToN(1)` is called, another stack frame is created with another

value of `n`, the expected return value, and a reference to `sumToN`.

4. Once `sumToN` returns `1`, the last stack frame is destroyed.
5. Once `sumToN(2)` is computed, the stack frame created in step (2) is destroyed.

Beyond a five digit stack limit, execution stops, and a JavaScript error is thrown. We will combat this behavior with tail call optimization.

# Tail call optimization

tail call optimization for writing elegant recursive solutions without the performance tax of ES5

A tail call is a subroutine call performed as the final action of a procedure.  
That is,

```
return myFunction()
```

It is important to understand that ES6 does not introduce new syntax for tail call optimization. It is just a different structure of code to make sure that it is efficient.

Let's calculate the Fibonacci using recursion:

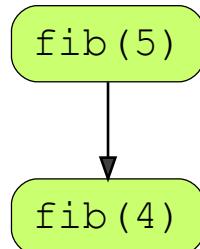
```
function fib(n) {
 if (n <= 1){
 return n;
 } else {
 return fib(n-1) + fib(n - 2);
 }
}
```

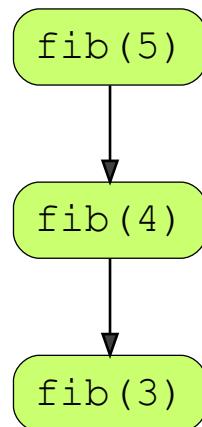


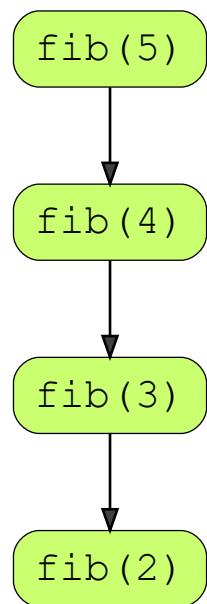
Let's view the function calls in the form of a tree:

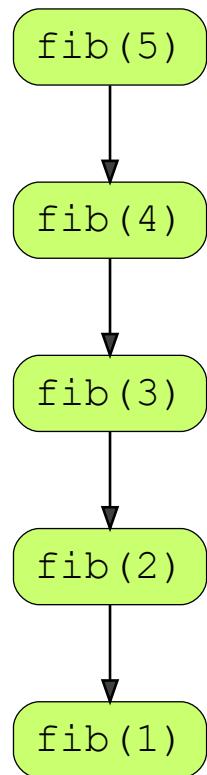
fib(5)

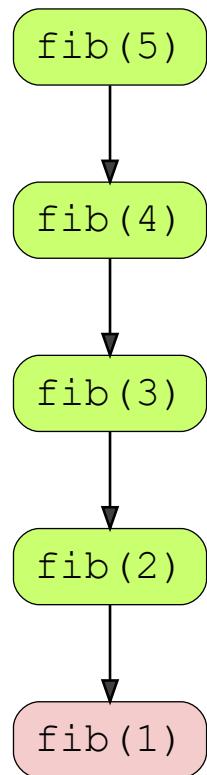
1 of 46

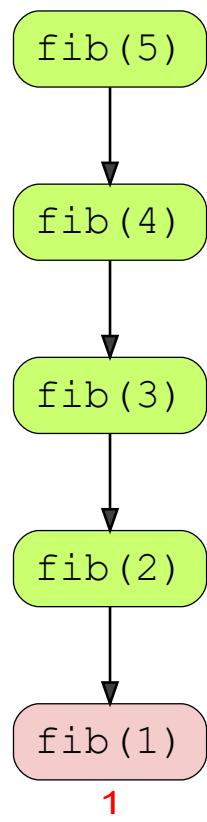


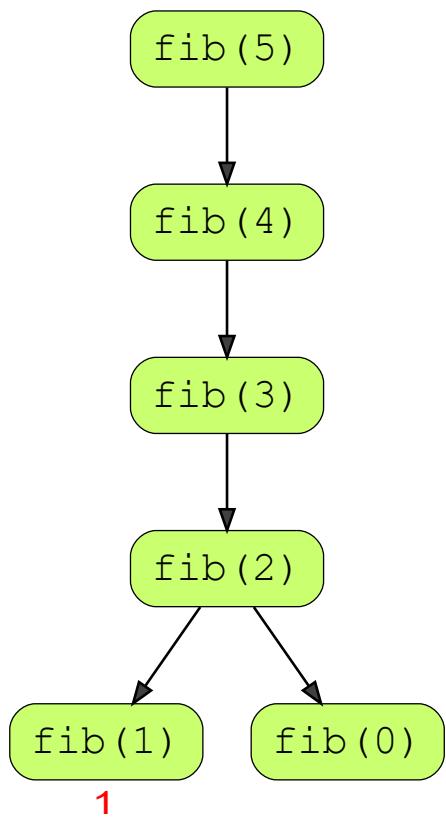


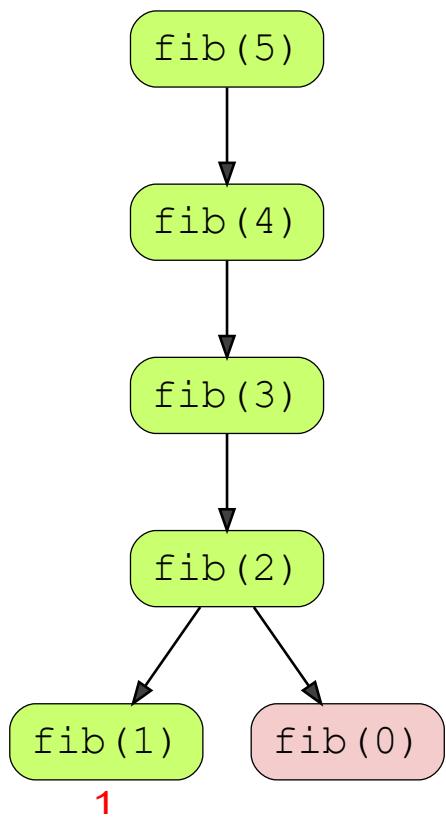


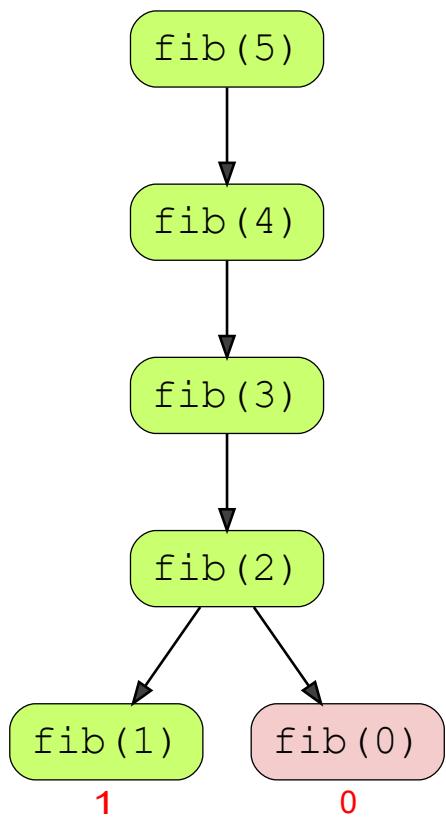


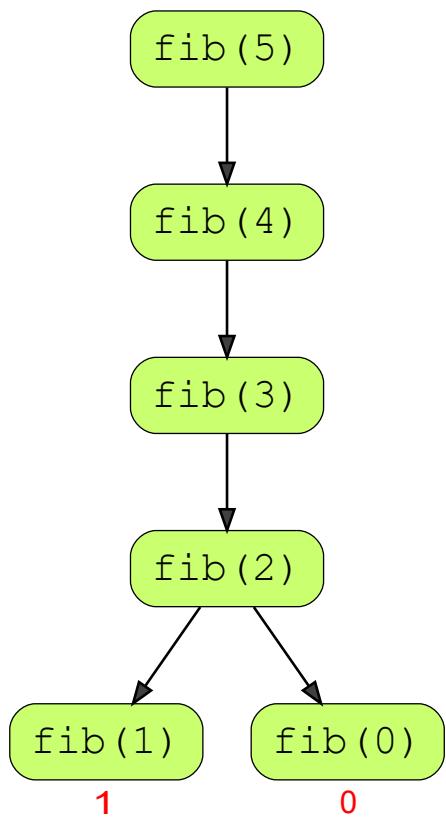


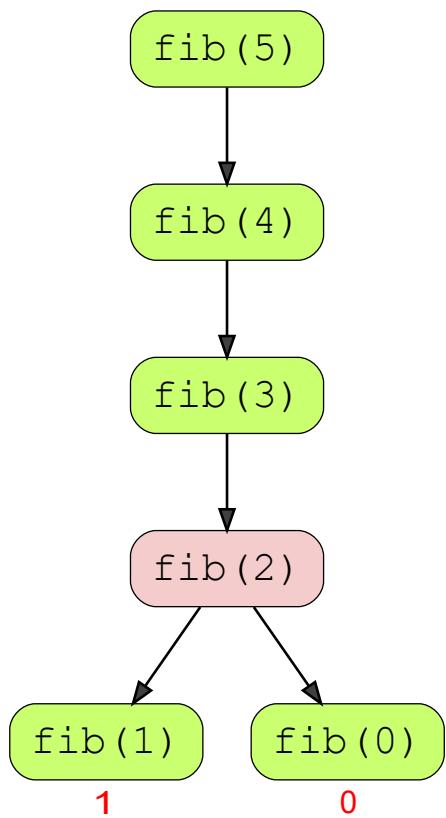


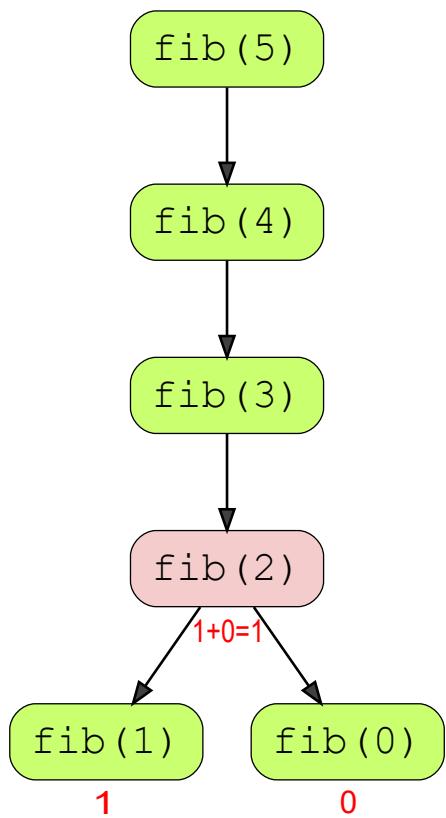


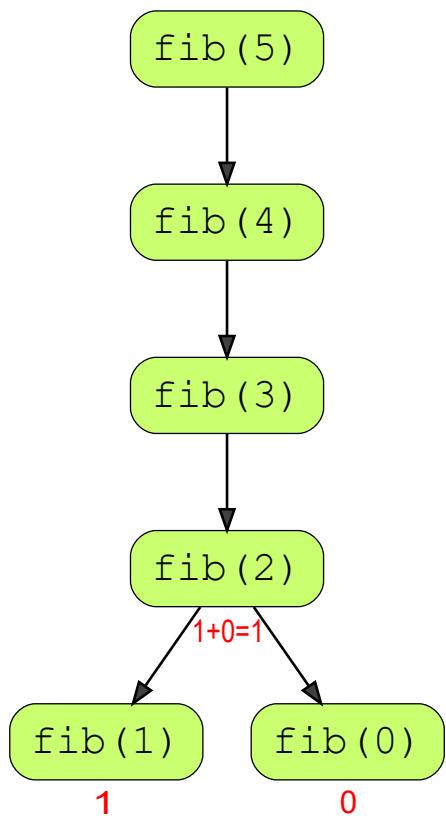


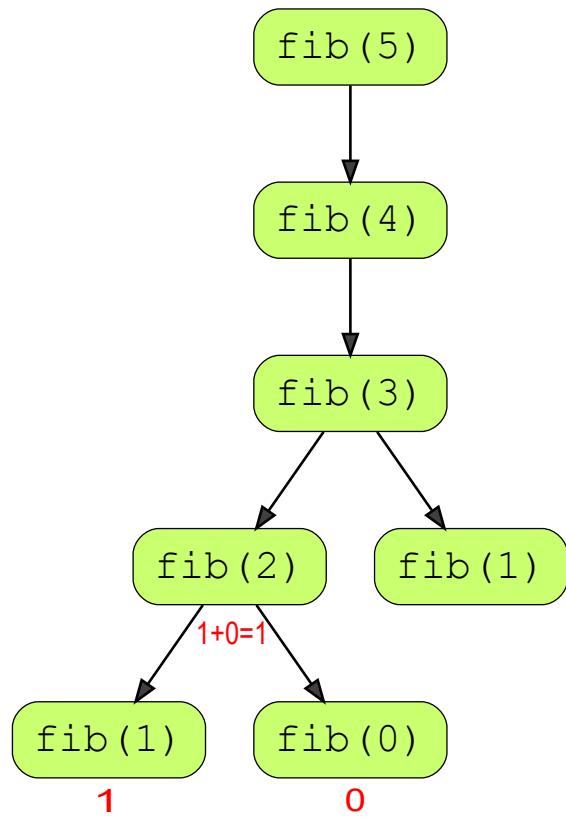


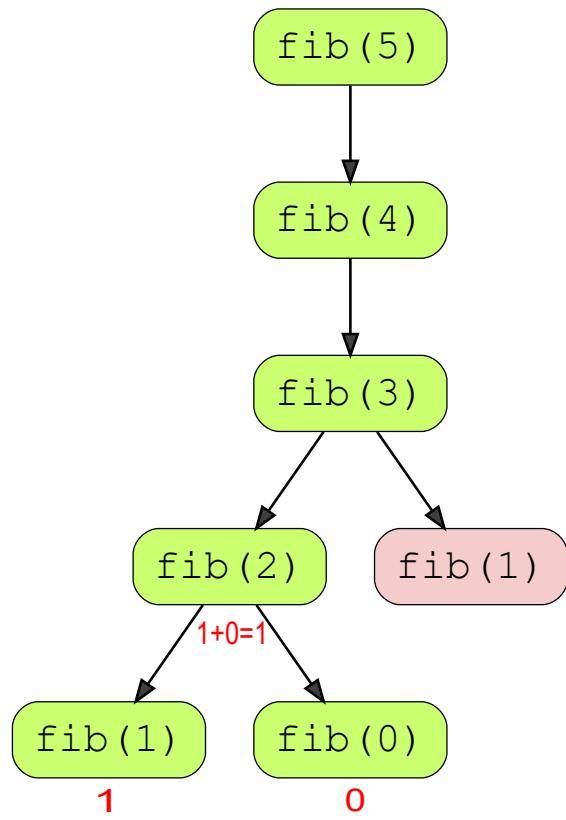


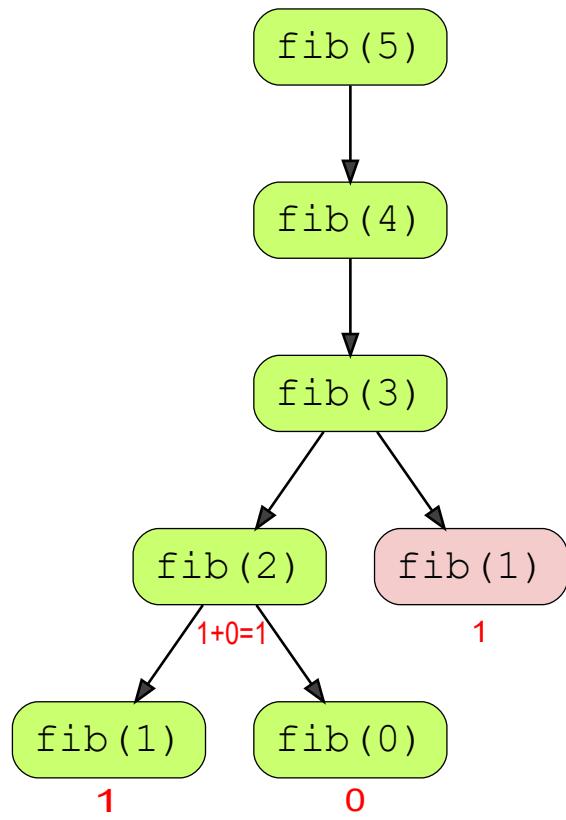


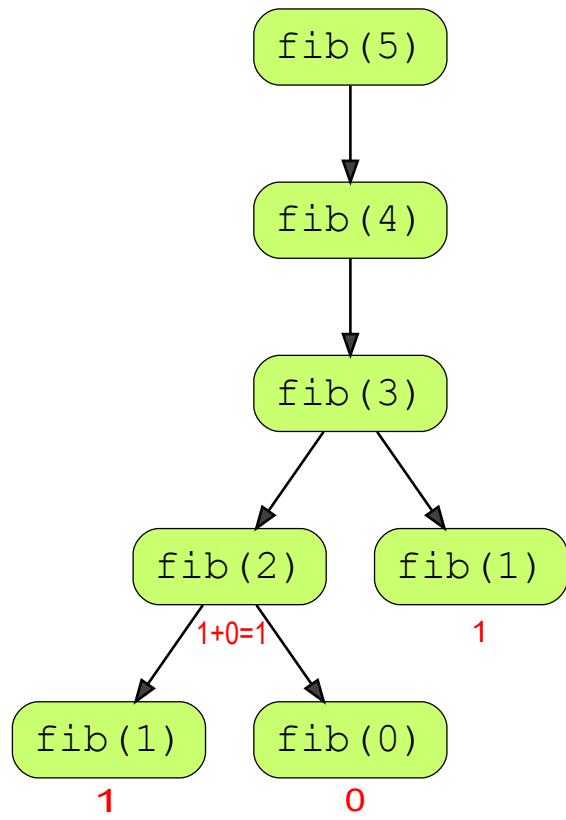


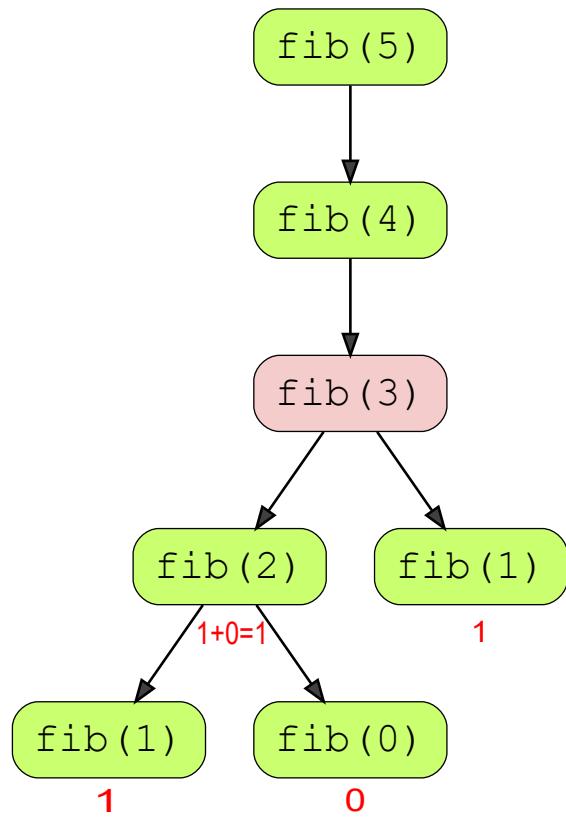


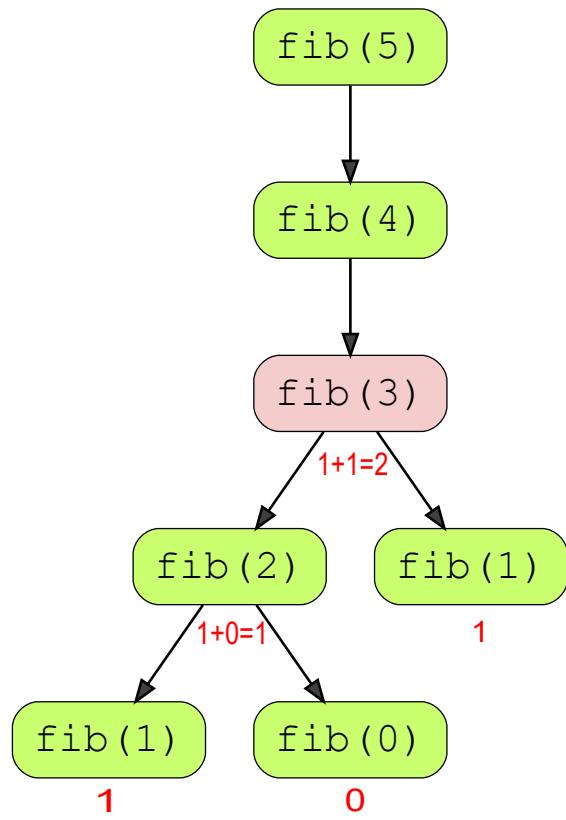


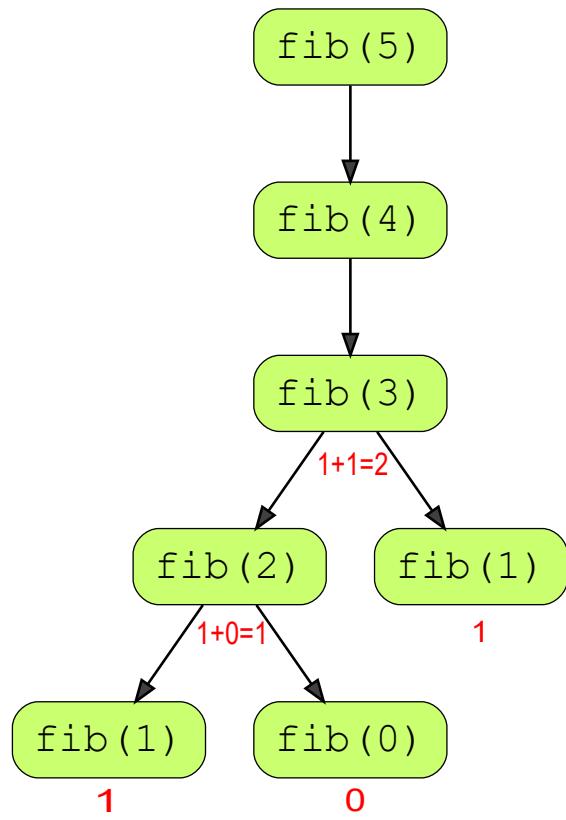


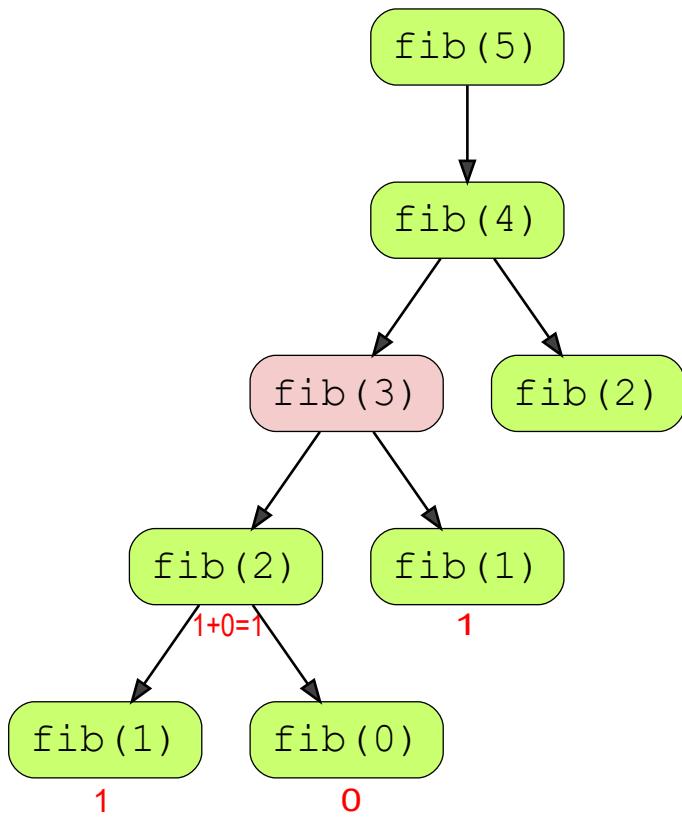


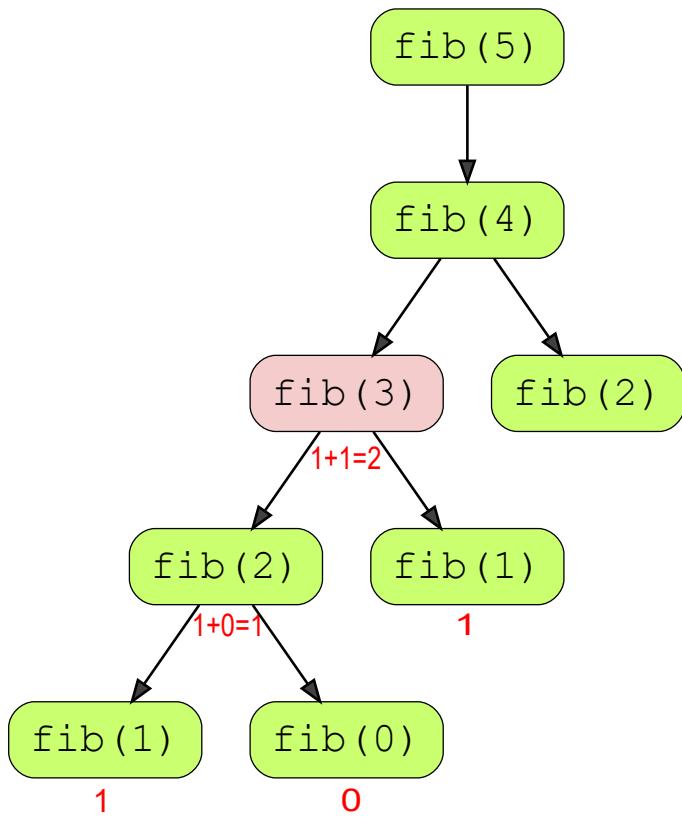


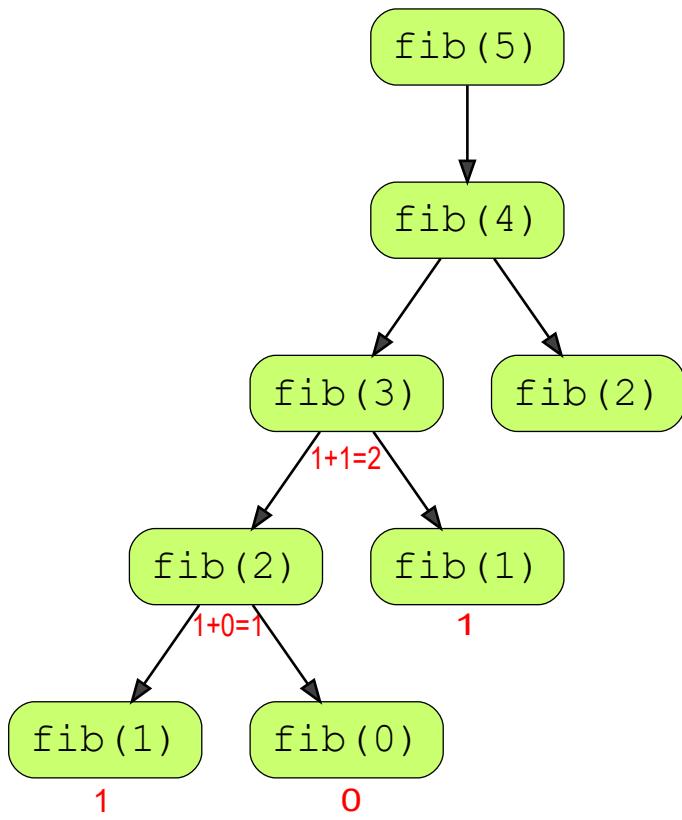


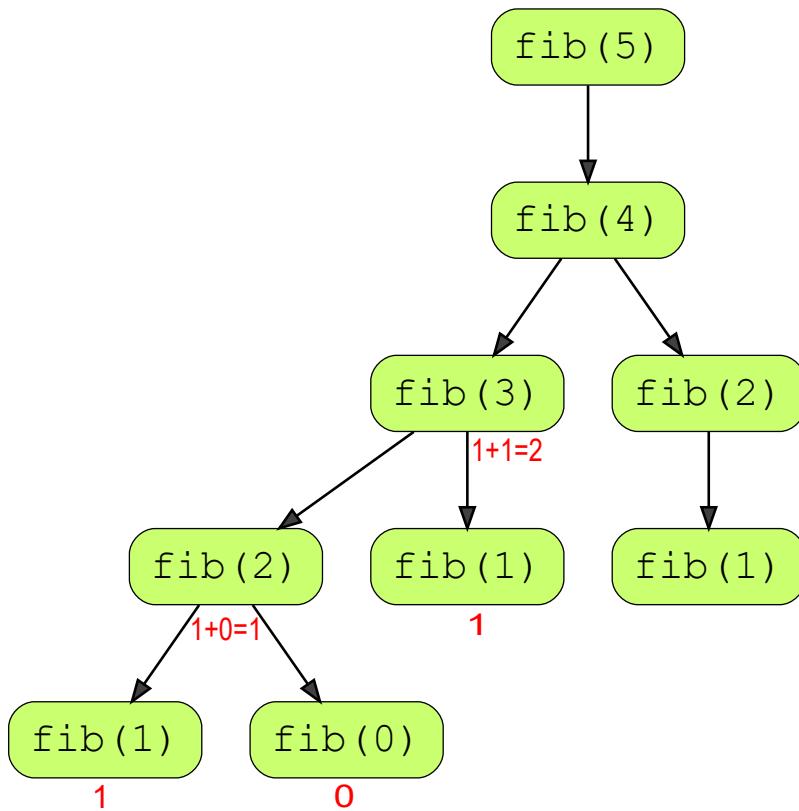


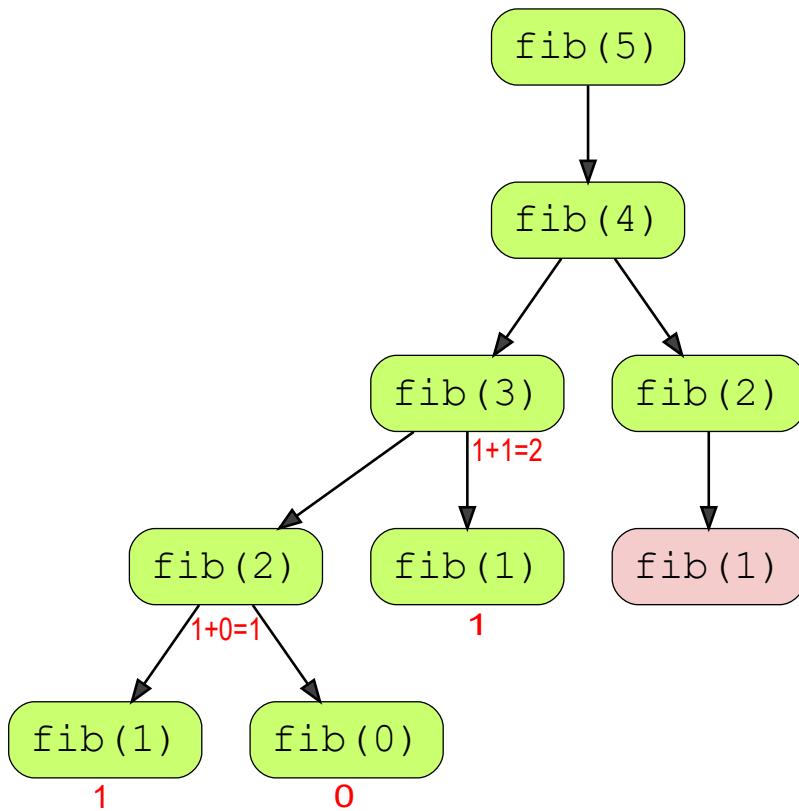


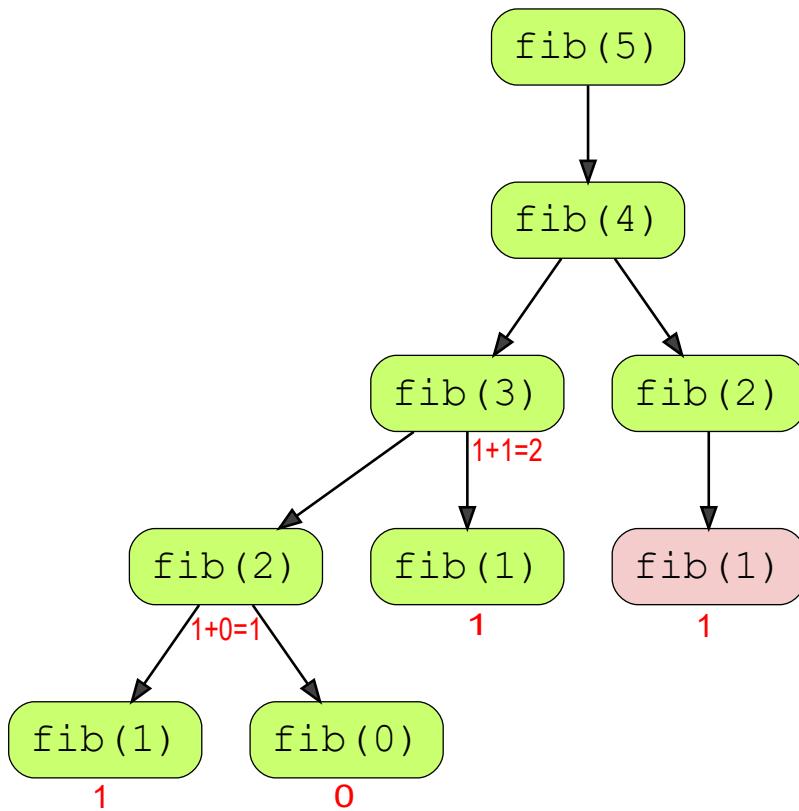


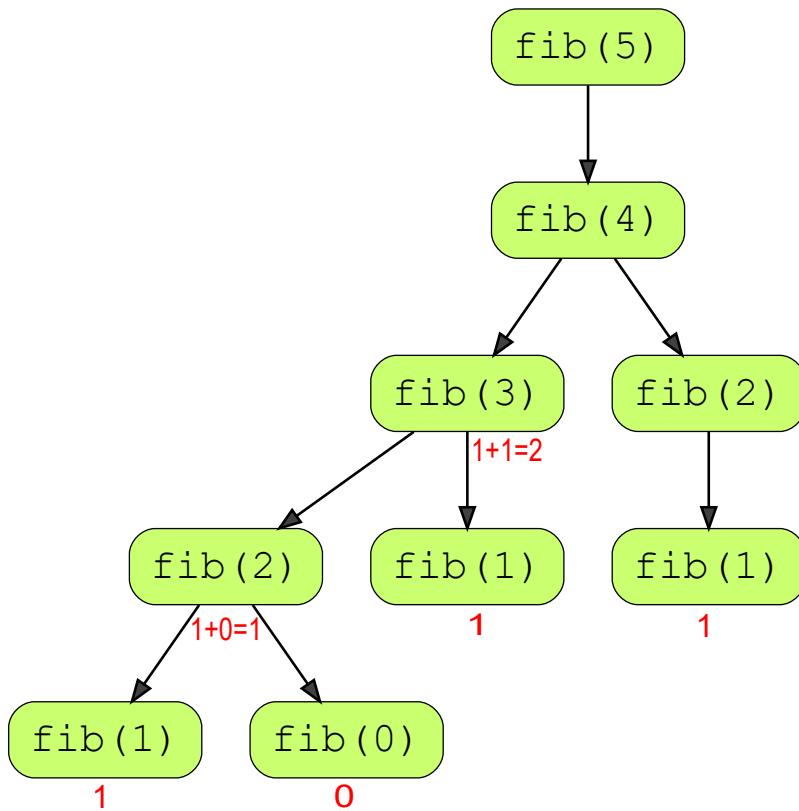


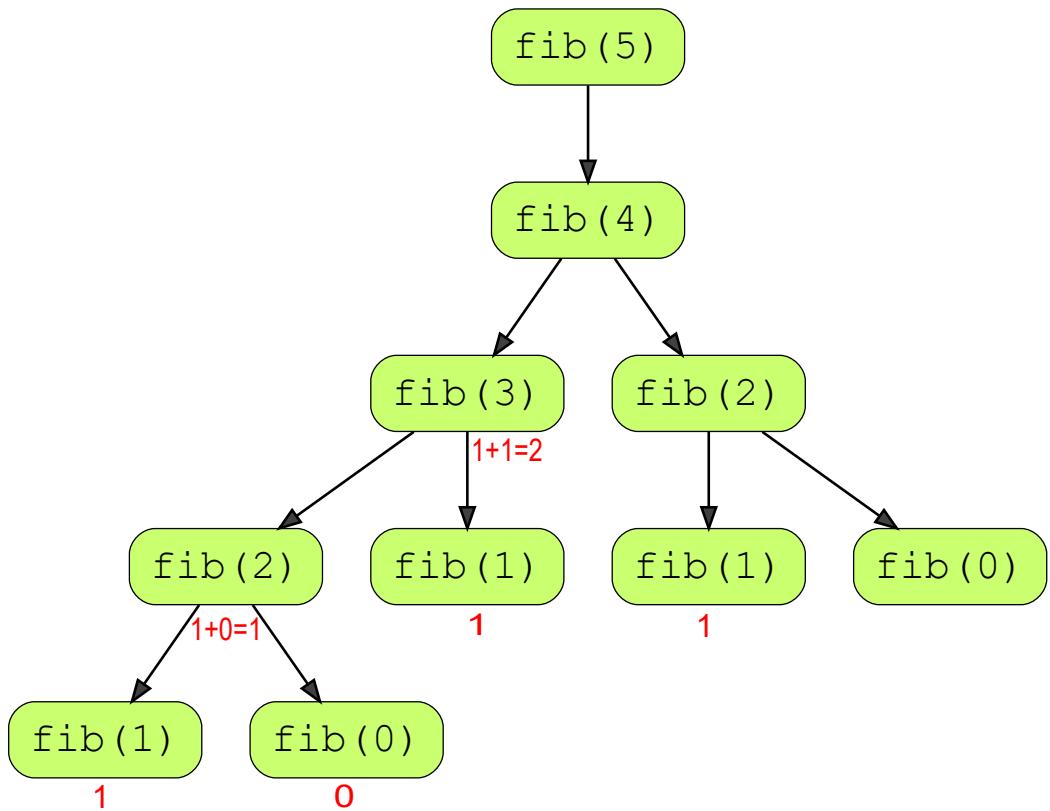


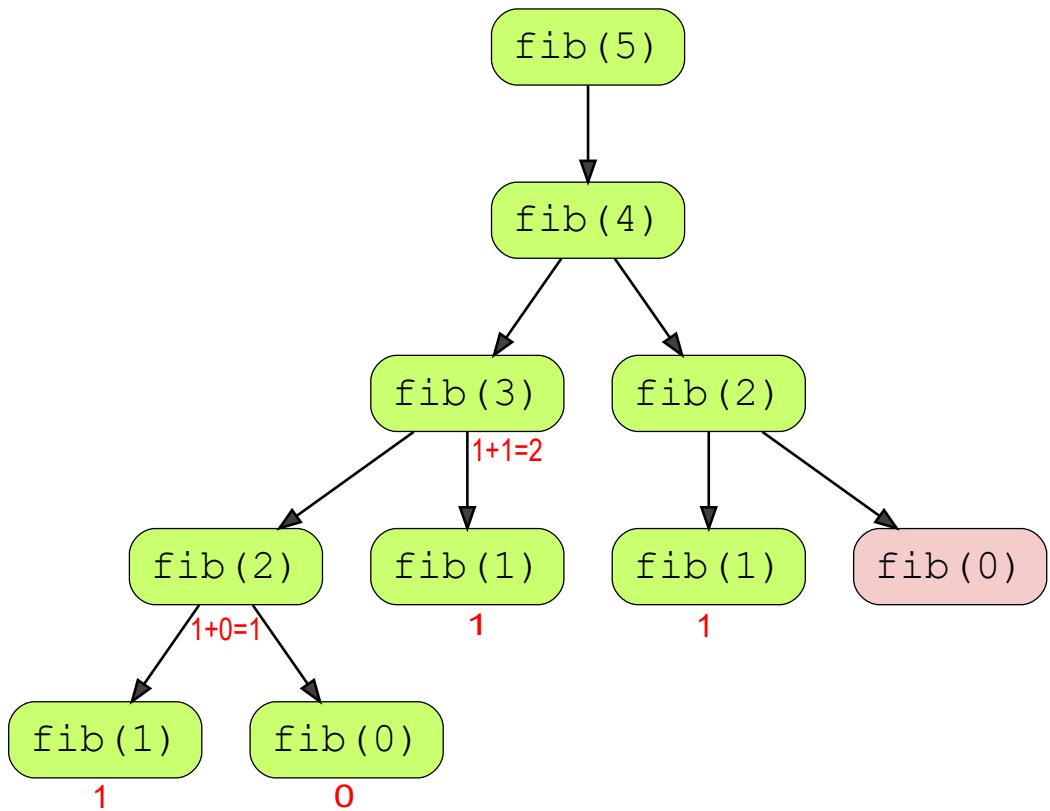


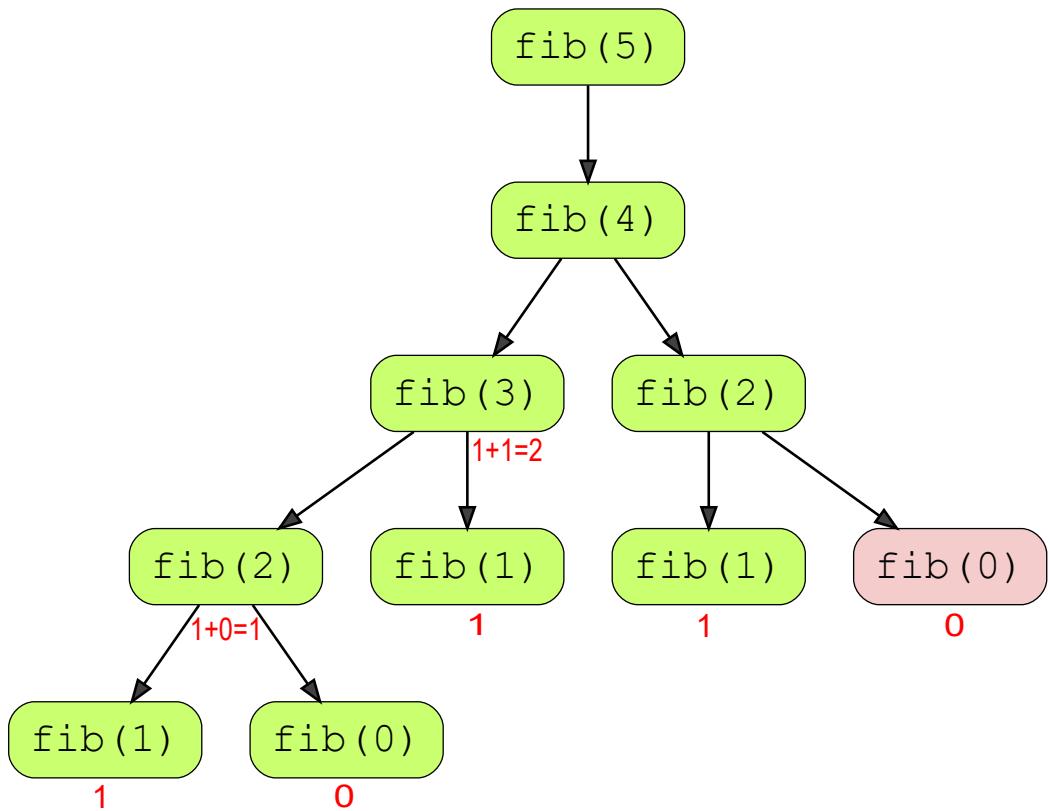


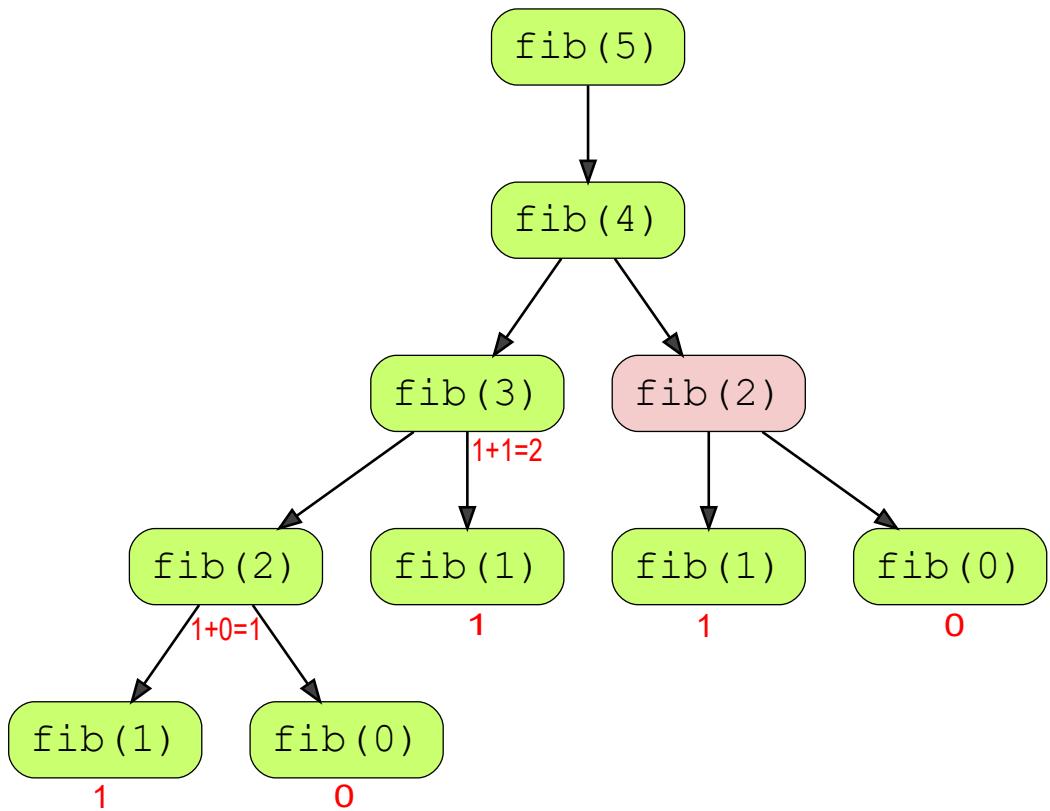


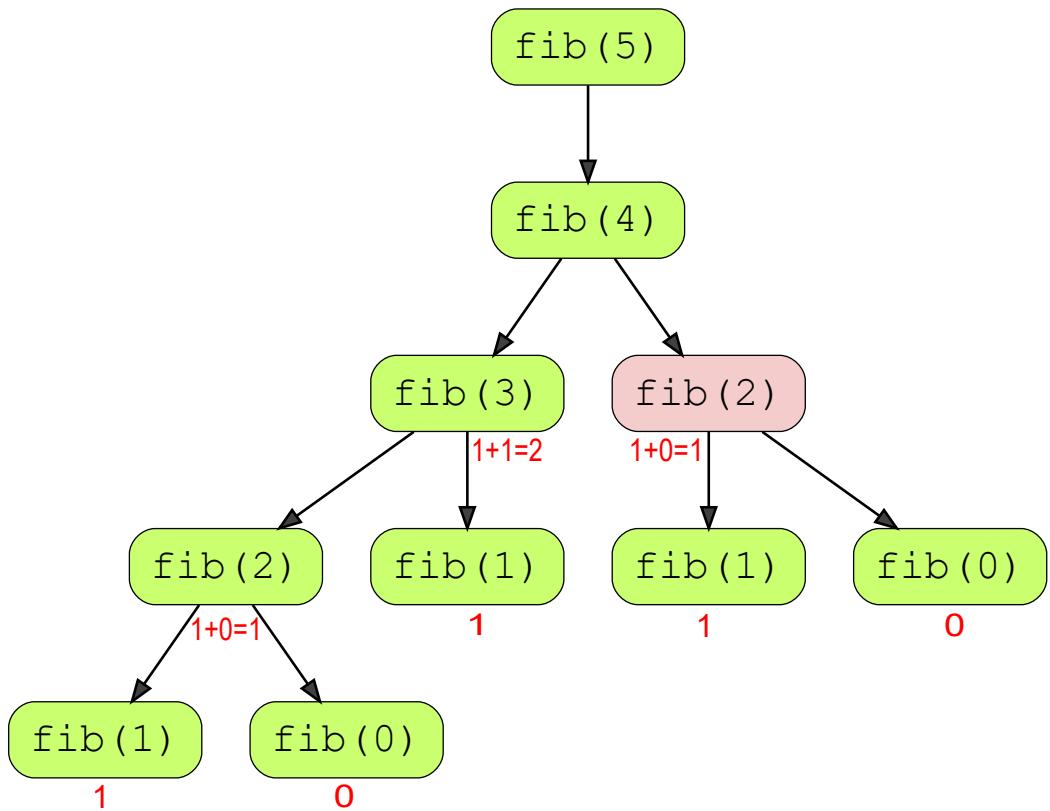


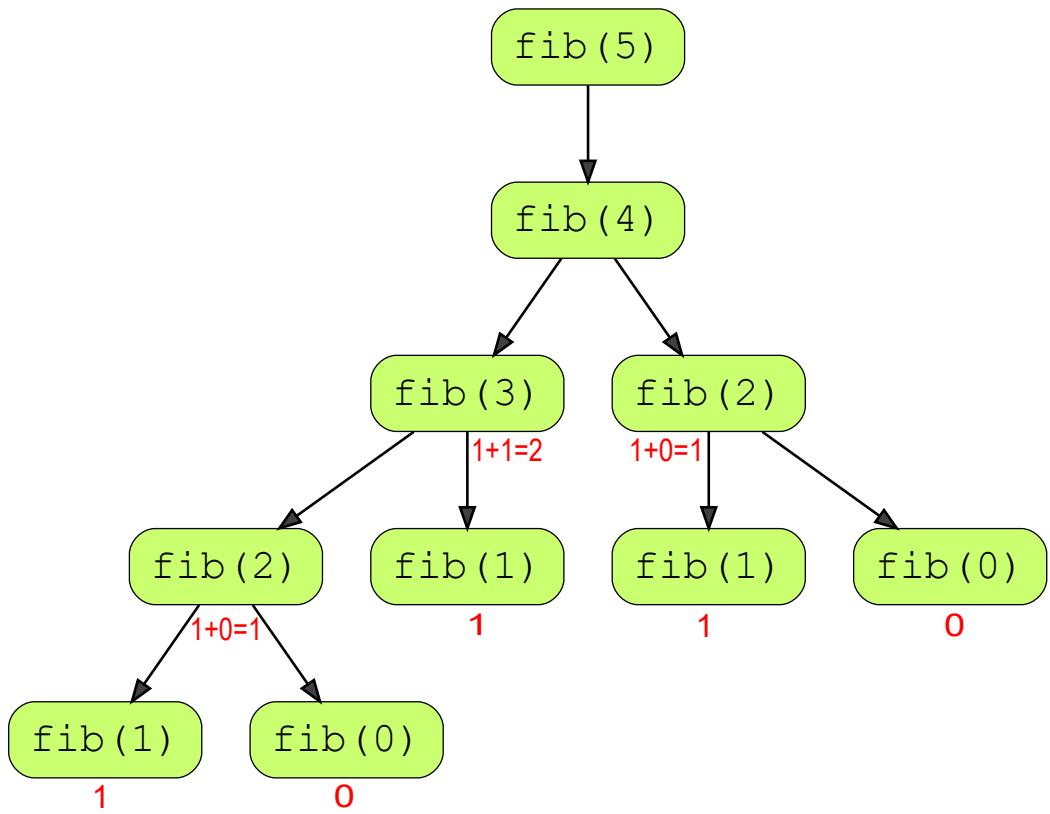


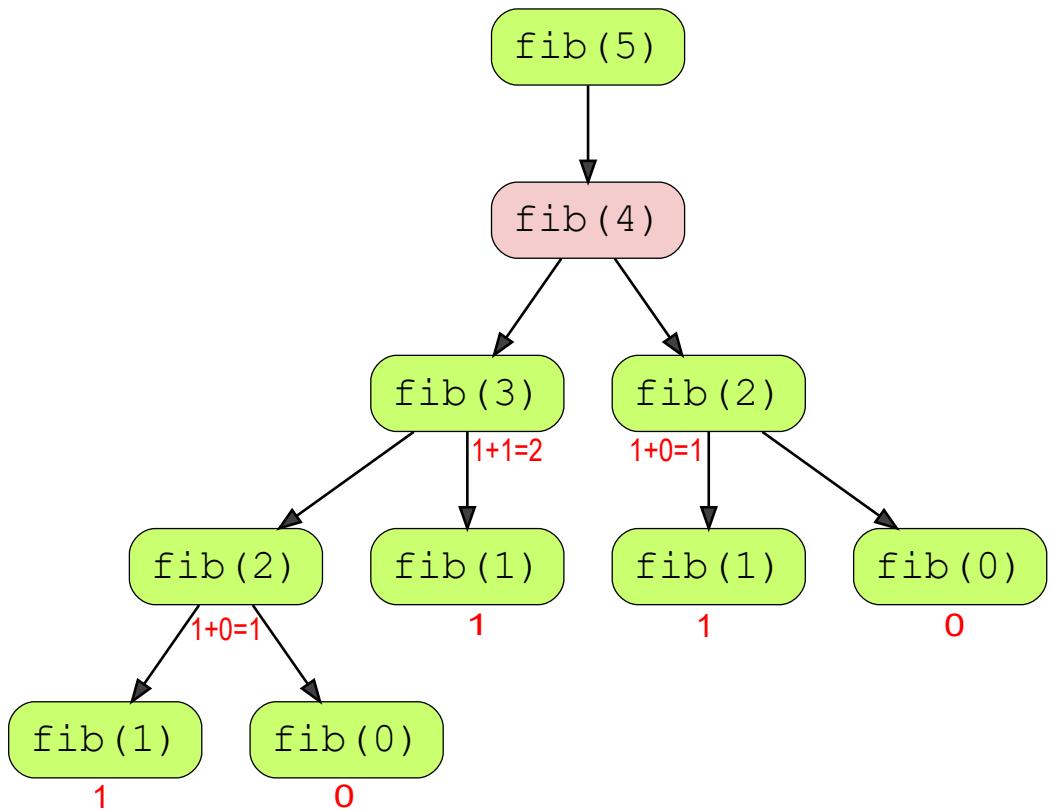


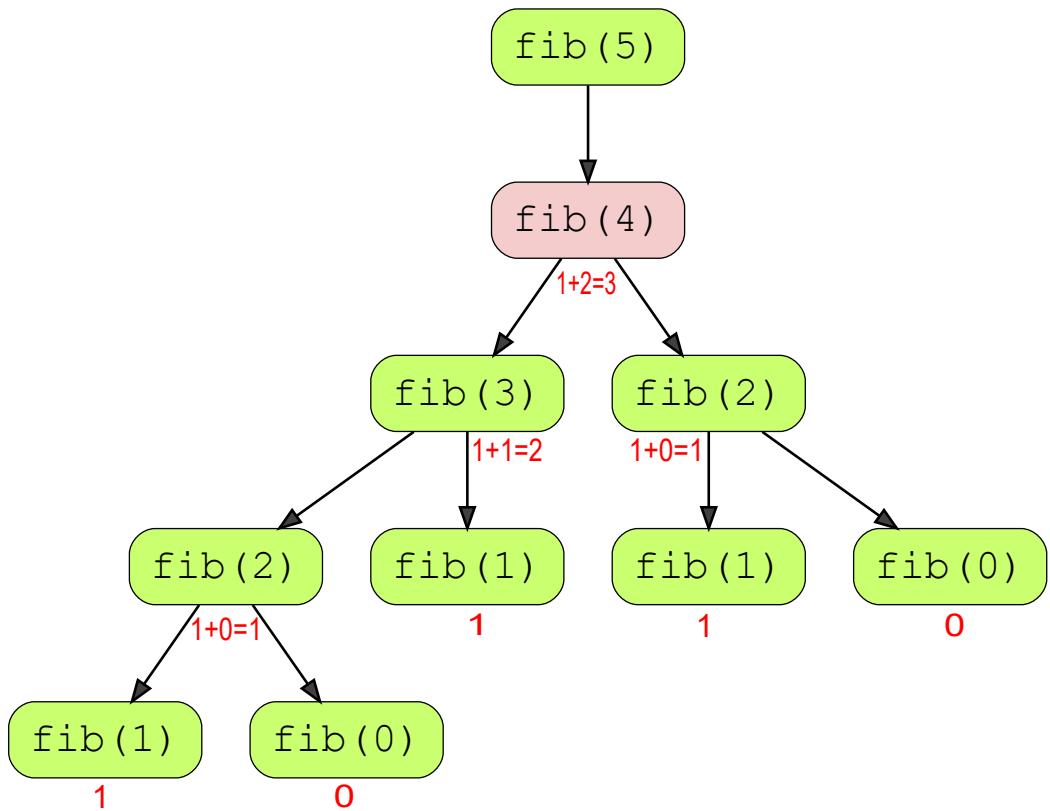


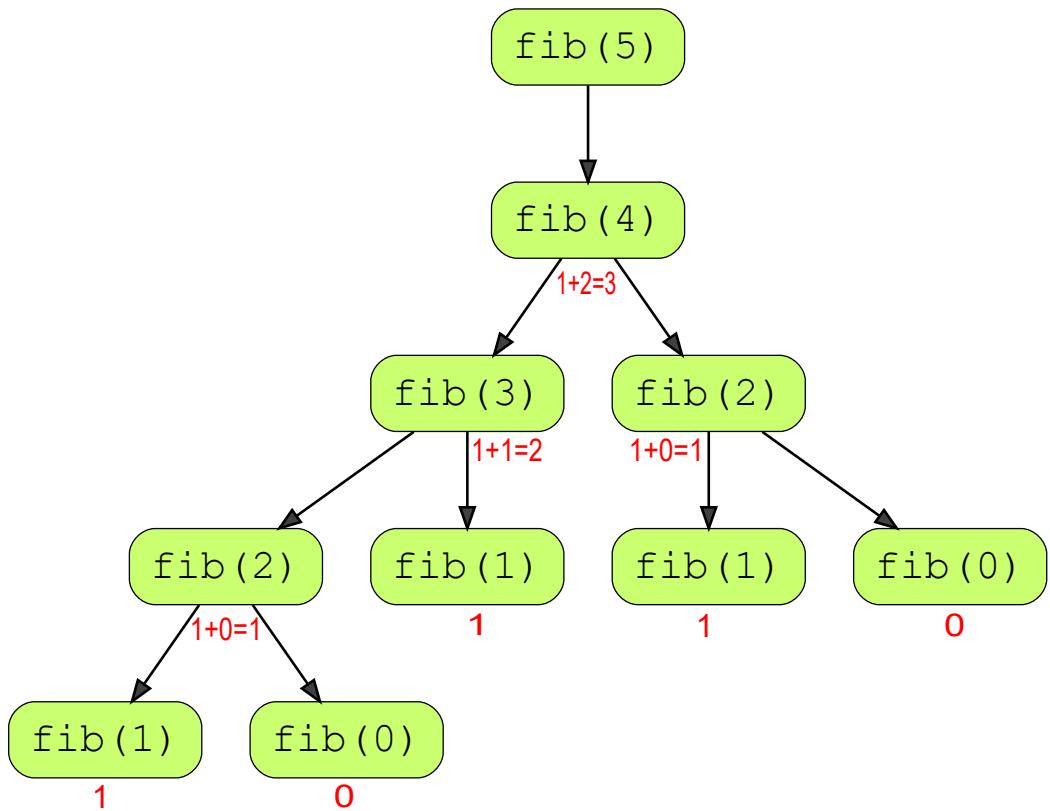


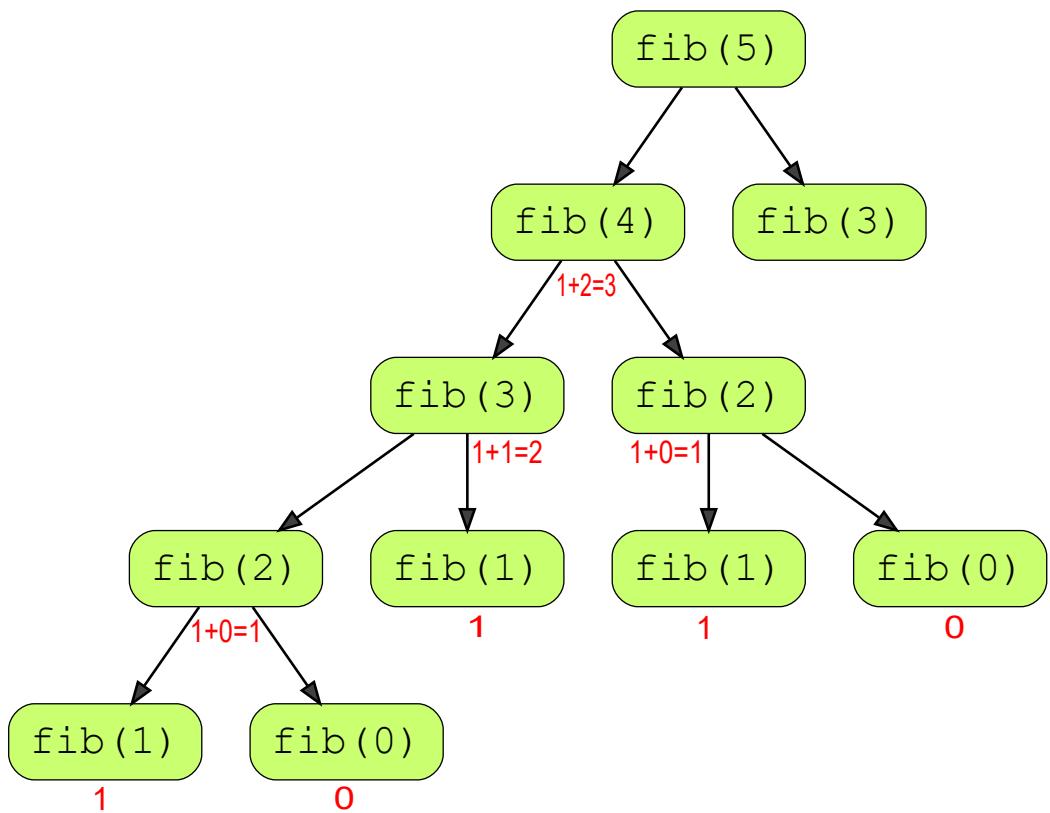


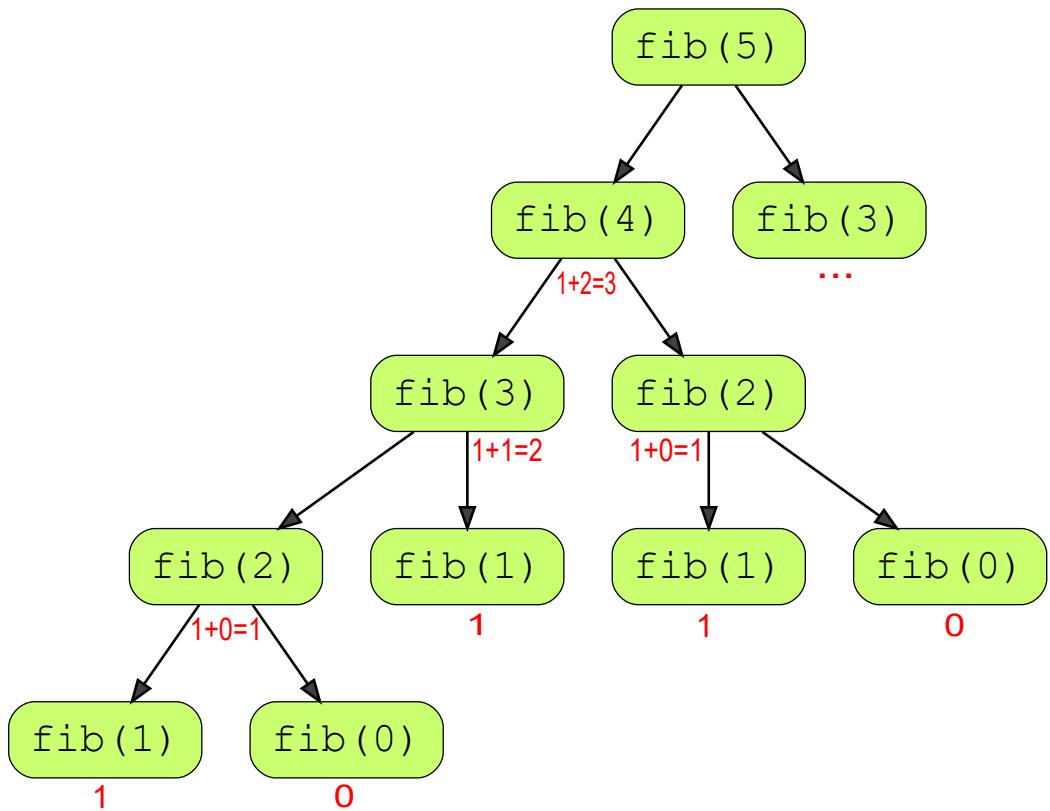


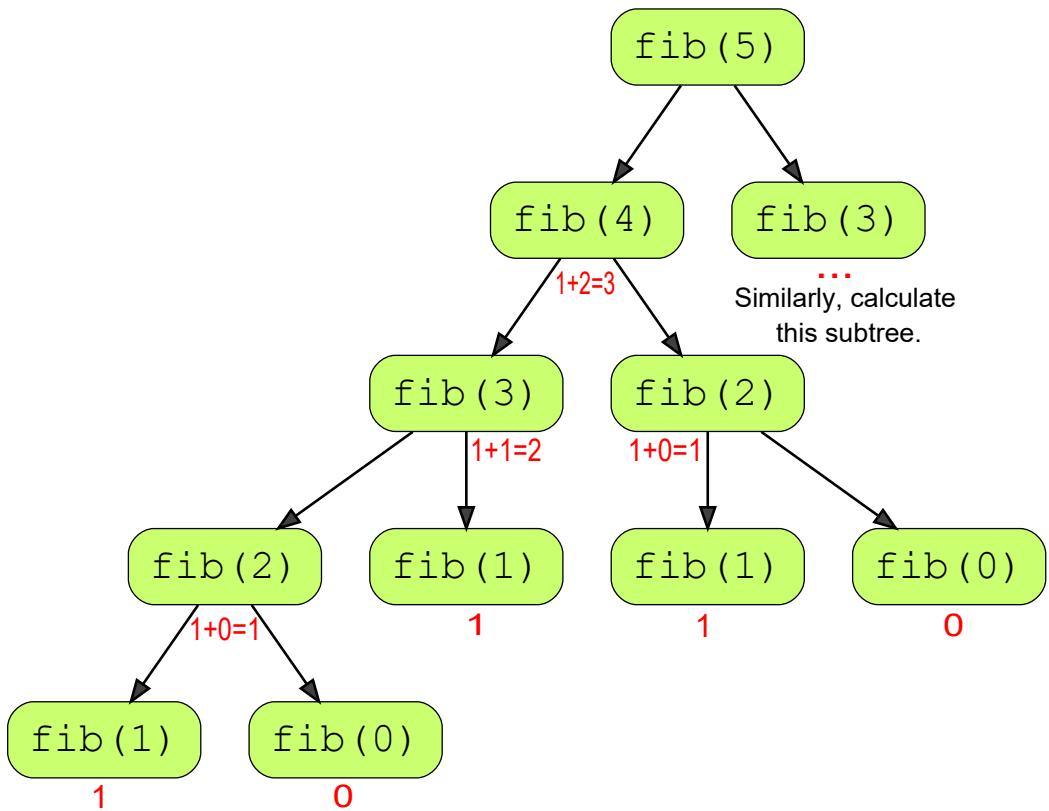


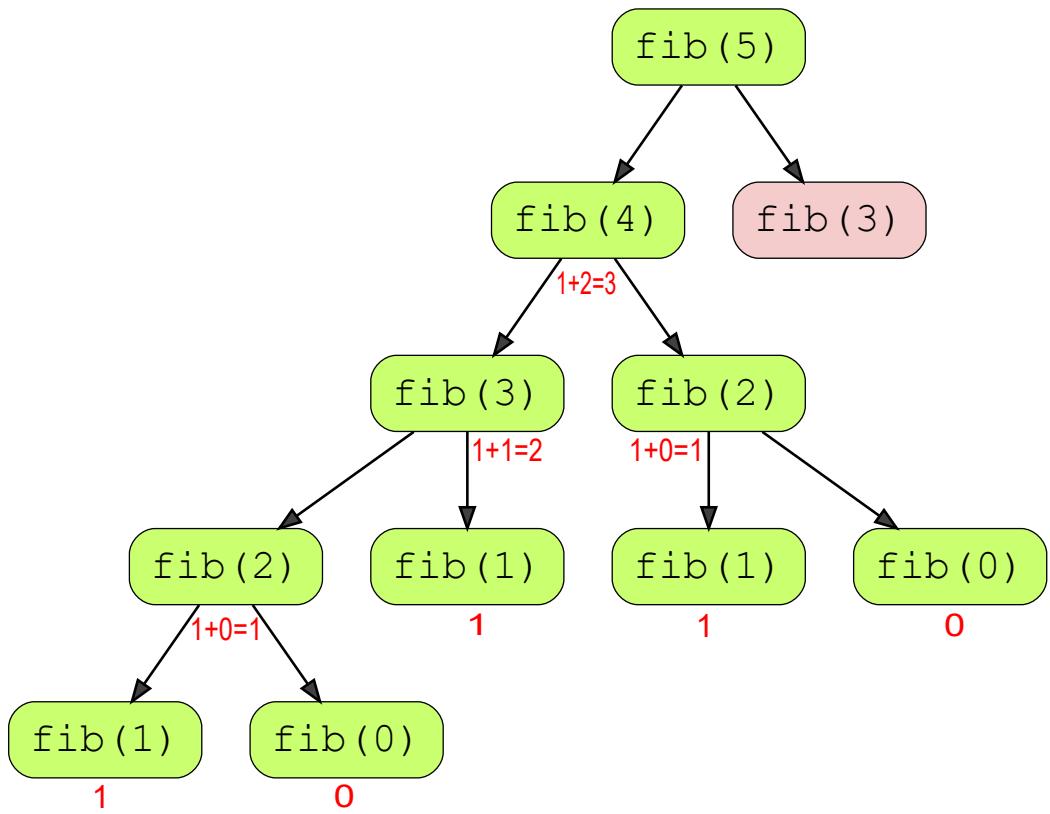


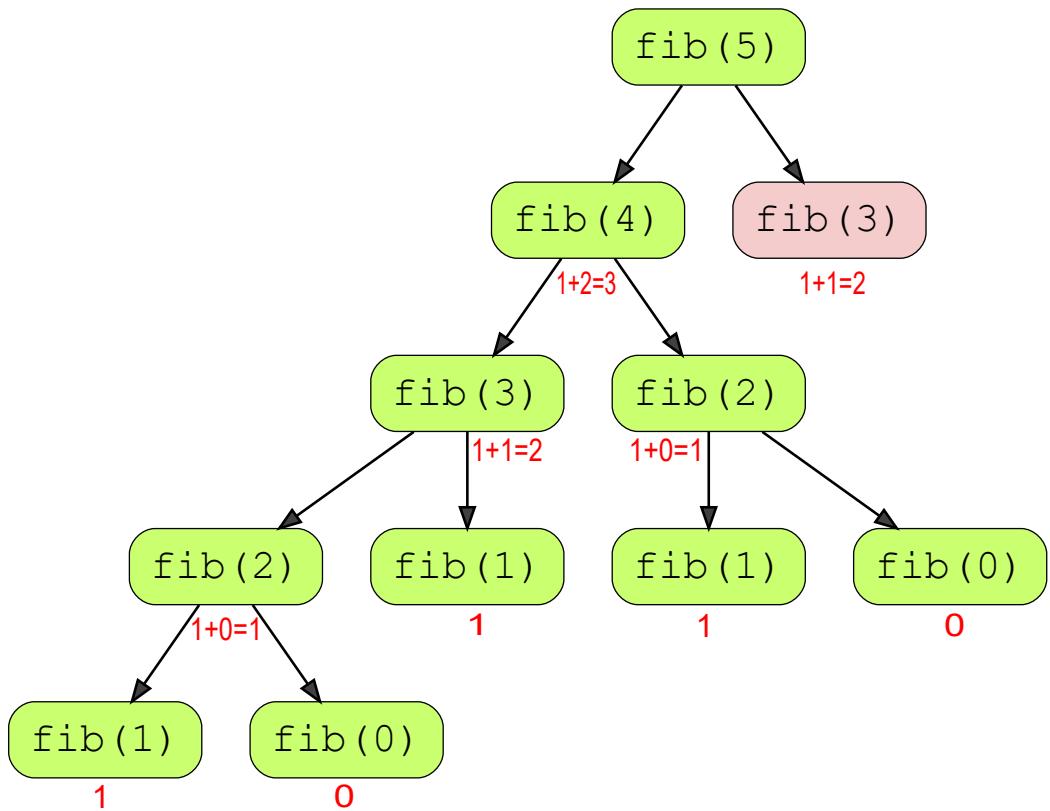


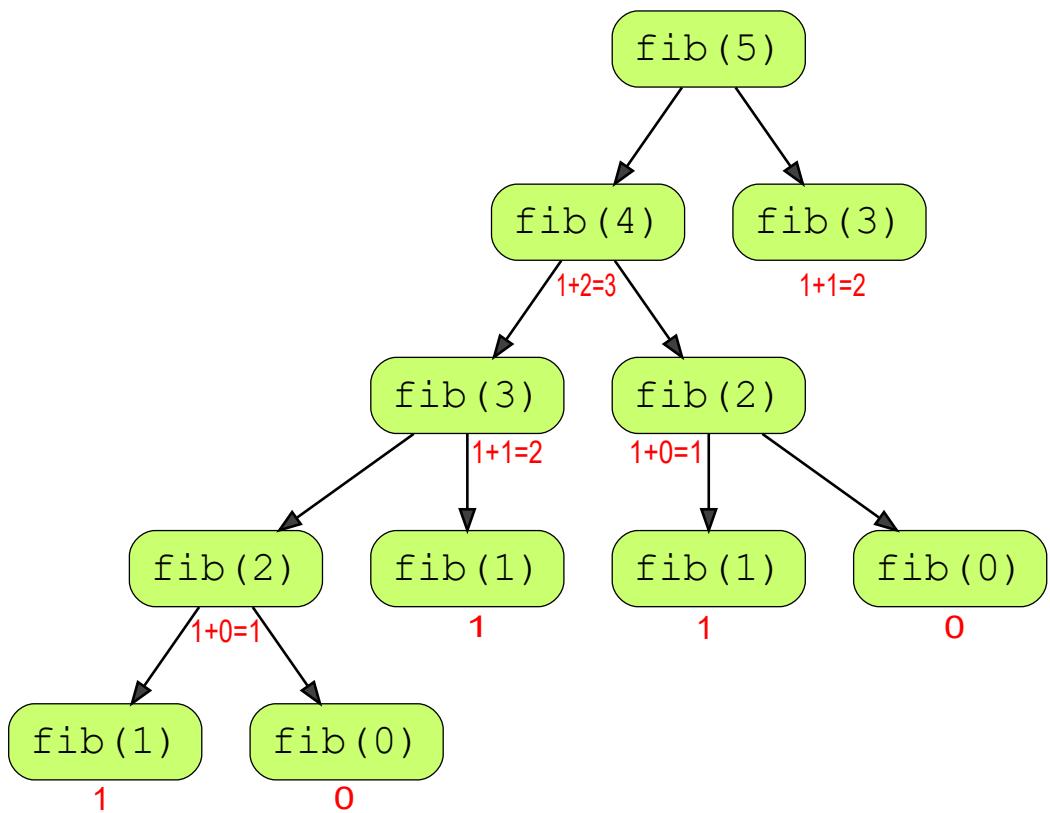


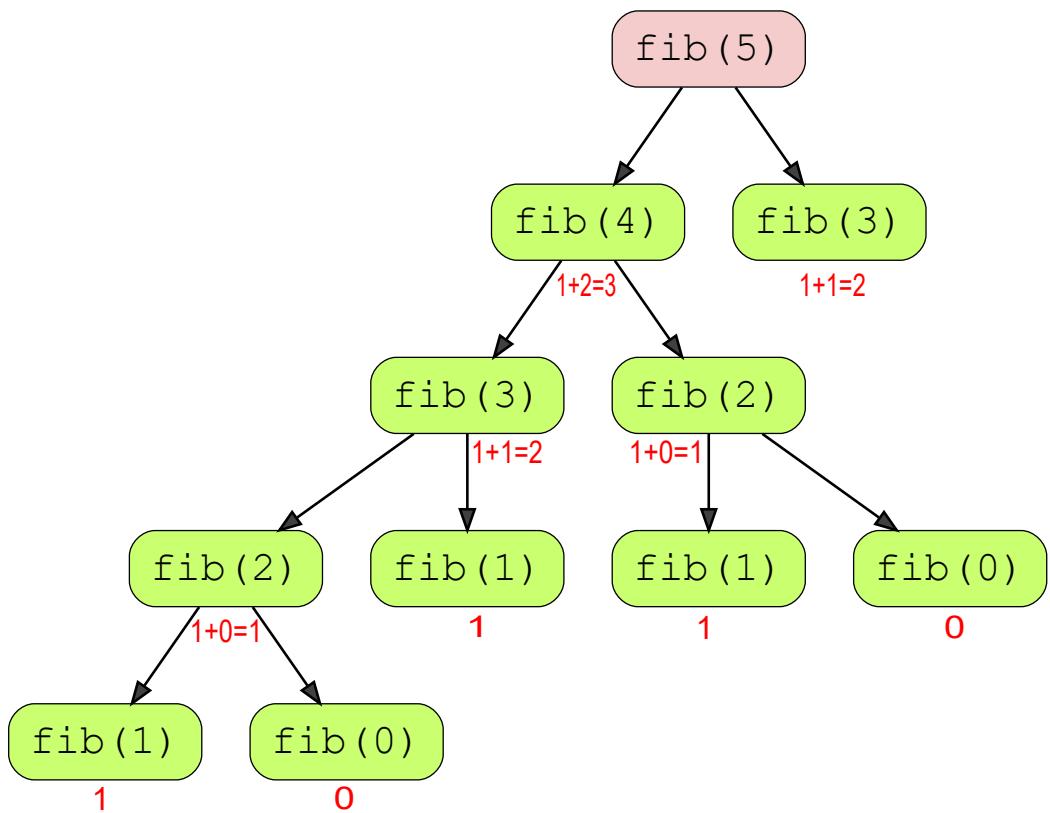


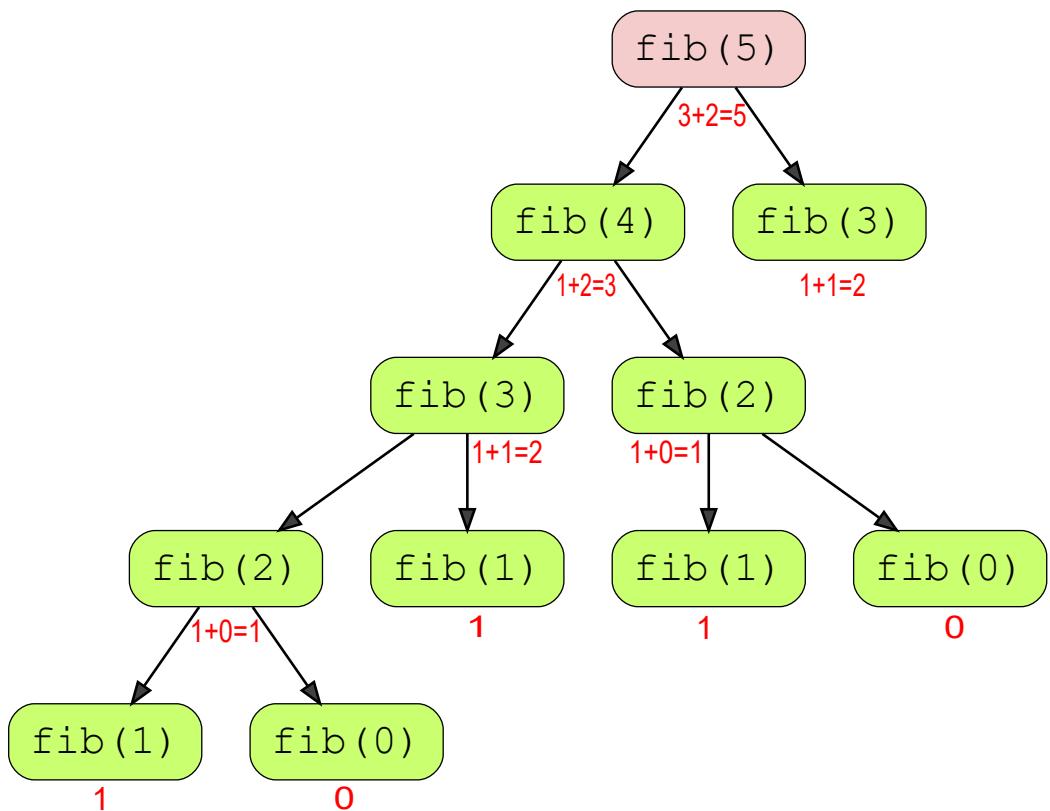




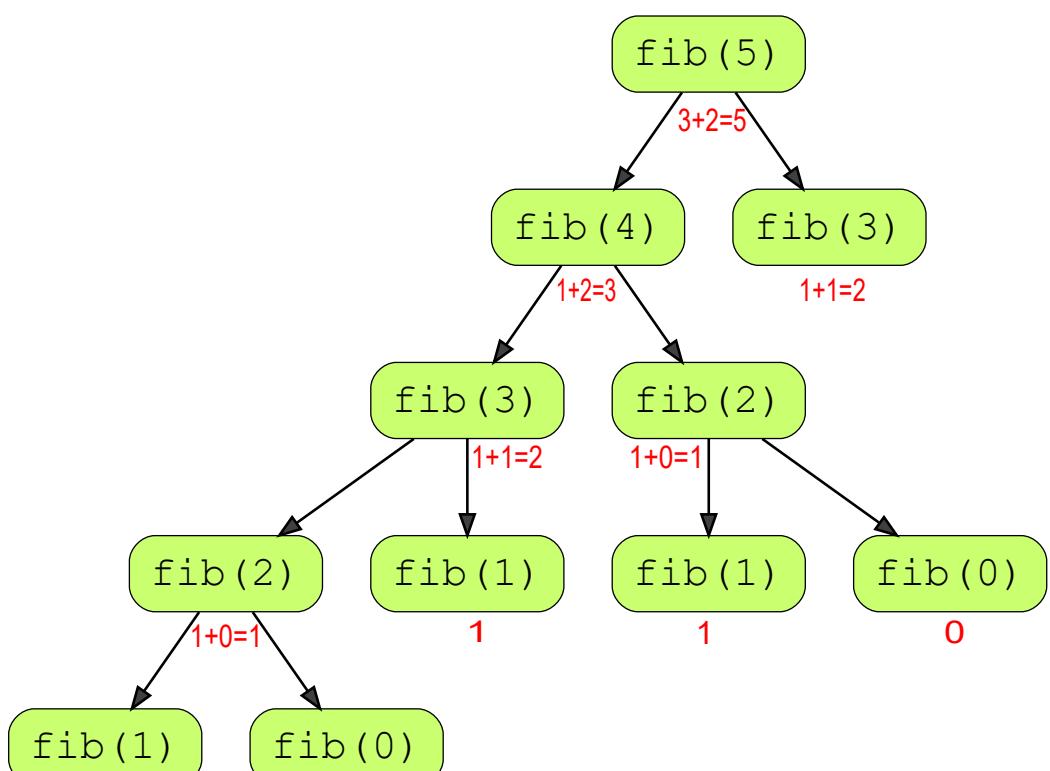








45 of 46





Notice that none of these have any clue that they are actually a part of a bigger process to calculate `fib(5)`. For instance, no `fib(1)` knows that it will be added to `fib(0)` and be used to calculate `fib(5)` at the end.

How do we know that `fib(3)` has to be added to `fib(2)` to make `fib(4)`, or `fib(2)` has to be added to `fib(1)` to make `fib(3)`? Where do we keep this information? All this information is in the call stack.

If at any point during the process, we were to pause and continue the process in a new environment with a clean stack, we would be unable to calculate `fib(5)` because the call stack has been wiped clean.

Think of a recursive process as a series of deferred operations, where there is information hidden to each recursive call - that hidden information is in the call stack.

Notice, that in the above code some functions, `fib(0)` and `fib(1)` for instance, are called multiple times. There is no memory of them ever being called before. As a result, the memory complexity is **O(n)**. If we run this program for large numbers, it throws an error.

```
function sumToN(n, sum = 0) {
 if (n <= 1) return sum;
 let result = sum + n;
 return sumToN(n - 1, result);
};

console.time('recursion');
console.log(sumToN(1000000)); //the code works if you replace 1000000 with a smaller number
console.timeEnd('recursion');
```



RangeError: Maximum call stack size exceeded

Now we will write the same function using tail call optimization:

Now, we will write the same function using tail call optimization.

```
function fib(n, a, b){
 if (n === 0) {
 return b;
 } else {
 return fib(n-1, a + b, a);
 }
};
```



When we run this function, here is what the function calls will look like:

**fib(5, 1, 0)**

1 of 7

**fib(5, 1, 0)  
fib(4, 1, 1)**

2 of 7

**fib(5, 1, 0)  
fib(4, 1, 1)  
fib(3, 2, 1)**

3 of 7

`fib(5, 1, 0)`  
`fib(4, 1, 1)`  
`fib(3, 2, 1)`  
**`fib(2, 3, 2)`**

4 of 7

`fib(5, 1, 0)`  
`fib(4, 1, 1)`  
`fib(3, 2, 1)`  
`fib(2, 3, 2)`  
**`fib(1, 5, 3)`**

5 of 7

`fib(5, 1, 0)`  
`fib(4, 1, 1)`  
`fib(3, 2, 1)`  
`fib(2, 3, 2)`  
`fib(1, 5, 3)`  
**`fib(0, 8, 5)`**

6 of 7

`fib(5, 1, 0)`  
`fib(4, 1, 1)`  
`fib(3, 2, 1)`  
`fib(2, 3, 2)`  
`fib(1, 5, 3)`  
`fib(0, 8, 5)`

7 of 7

In this implementation, the entire state of the process is encapsulated in each function call. If we were to pause the process midway and start with a clean stack, we would still get the correct output. Moreover, the memory complexity is **O(1)**.

The current support for tail call optimization is currently very low. It will take time until you will be able to benefit from properly optimized tail calls. See the current [compatibility table](#).

On Google Chrome, the feature status can be seen [here](#). Currently, Safari is the only main browser that supports tail calls.

Once browser support is better, I will rewrite this section. Until then, know that tail call optimization exists.

We will now move on to the name property in ES6.

# Name Property

name property in ES5 and its limitations

The name of a function can be retrieved using the name property of a function.

```
let guessMyName = function fName() {};
let fName2 = function() {};
let guessMyProperty = {
 prop: 1,
 methodName() {},
 get myProperty() {
 return this.prop;
 },
 set myProperty(prop) {
 this.prop = prop;
 }
};

console.log(guessMyName.name);
//> "fName"
console.log(fName2.name);
//> "fName2"
console.log(guessMyProperty.methodName.name);
//> "methodName"
console.log(guessMyProperty.methodName.bind(this).name);
//> "bound methodName"
```



When it comes to getters and setters, retrieving method names is a bit more complicated:

```
let propertyDescriptor = Object.getOwnPropertyDescriptor(
 guessMyProperty, 'myProperty');

console.log(propertyDescriptor.get.name);
//> "get myProperty"
console.log(propertyDescriptor.set.name);
//> "set myProperty"
```





Function names provide you with limited debugging capabilities. For instance, you can create an automated test that checks if a function name is bound, making sure that no-one will delete the function binding in the code.

In theory, another use case is to retrieve the class name of an object by querying `obj.constructor.name`. Before using this construct, read the [warnings here](#).

If you would still like to use the `name` property to retrieve the class of an object, note that it is bad practice. You can use `new.target` inside the `constructor` function just as well as the `name` property. Set the internal state of your object such that you will never need to retrieve the actual constructor of your object. Encapsulate all the roles and responsibilities of your object inside it. Checking for the constructor types outside the object definition results in worse maintainability and tighter coupling.

I will talk about `new.target` in more detail in the next lesson.

# new.target

using new.target inside constructors

After the introduction of `new.target` for classes in [Chapter 4](#), you will now understand it more deeply by using the same language construct for functions.

You can retrieve the constructor function of an object inside the constructor by using `new.target`.

```
function MyConstructor() {
 console.log(new.target === MyConstructor, typeof new.target);
 if (typeof new.target === 'function') {
 console.log(new.target.name);
 }
}

new MyConstructor();
//> true "function"
//> MyConstructor

MyConstructor();
//> false "undefined"
```



Whenever you call a constructor **with** the `new` operator, `new.target` contains a reference to the constructor. Whenever you call a constructor **without** the `new` operator, `new.target` will be `undefined`.

In order to understand `new.target` on a deeper level, check out the last exercise in the next lesson.

# Exercise on Tail Call Optimization and other Function Features in ES6

In the following exercises, you will use tail call optimization, create a stack, and examine how `new.target` behaves in ES5.

## Exercise 1:

Implement a stack in ES6. In addition to a constructor, it should have a `push()`, `pop()`, and a `len()` function that would return the number of elements in the stack.

```
class Stack {
 //Write your code here
}
```



## Exercise 2:

Write a tail call optimized solution for the following Fibonacci function. See if all of the tests work after you tail-call optimize the given function. Remember to name your function `fib2()` or the tests won't work.

```
function fib(n) {
 if (n <= 1) return n;
 return fib(n - 1) + fib(n - 2);
}

function fib2()//add parameters
{
 //write code here
}
```



Explanation:

To avoid mixing the two Fibonacci functions, we will refer to the tail recursive fib function as fib2. We have to create accumulator variables to create proper tail calls. As we need to memorize the last two values of the sequence, we have to create two accumulators.

The accumulators will keep track of the last two elements that are needed for constructing the current Fibonacci number. Notice that the two accumulator values require us to create two separate exit conditions:

- When calling fib2 with 0, the return value should be 0.
- When calling fib2 with a positive integer, the final return value in the last fib2 call is acc1.

### Example:

- `fib2(5)` becomes `fib2(5, 1, 0)` once the default values are assigned to the second and the third arguments
- `fib2(5, 1, 0)` invokes `fib2(4, 1, 1)`
- `fib2(4, 1, 1)` invokes `fib2(3, 2, 1)`
- `fib2(3, 2, 1)` invokes `fib2(2, 3, 2)`
- `fib2(2, 3, 2)` invokes `fib2(1, 5, 3)`
- as `n` is 1 for `fib2(1, 5, 3)`, 5 is returned as a result. 5 is the value of `acc1`.
- 5 is returned by all recursive calls. Therefore, `fib(5)` also returns 5.

Obviously, this implementation only works with non-negative integer inputs. We could also add another exit condition to handle invalid inputs:

```
if (n < 0 || !Number.isInteger(n)) return NaN;
```

### Exercise 3:

Create a solution for the Fibonacci exercise that does not use recursion. Remember to name it `fib()`

```
function fib(n)
{
 count++;
```



```
counter,
/*write-your-code-here*/
}
```



## Exercise 4:

Rewrite the following code without using the class syntax of ES6. Observe how `new.target` behaves.

Uncaught Error: Abstract class cannot be instantiated.(...)

```
class AbstractUser {
 constructor() {
 if (new.target === AbstractUser) {
 throw new Error('Abstract class.');
 }
 this.accessMatrix = {};
 }
 hasAccess(page) {
 return this.accessMatrix[page];
 }
}

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

let su = new SuperUser();

//let au = new AbstractUser();
// ^ Throws the new error
```



# Symbol

symbol - a new ES6 primitive type and its use cases

ES6 introduces a new primitive type for JavaScript: Symbols. The global `Symbol()` function creates a JavaScript symbol. Each time the `Symbol()` function is called, a new unique symbol is returned.

```
let symbol1 = Symbol();
let symbol2 = Symbol();

console.log(symbol1 === symbol2);
//> false
```



Symbols don't have a literal value. All you should know about the value of a symbol is that each symbol is treated as a unique value. In other words, no two symbols are equal.

A symbol is a new *type* in JavaScript.

```
console.log(typeof symbol1);
//> "symbol"
```



Symbols are useful, because they act as unique object keys.

```
let myObject = {
 publicProperty: 'Value of myObject ["publicProperty"]'
};

myObject[symbol1] = 'Value of myObject [symbol1]';
myObject[symbol2] = 'value of myObject [symbol2]';

console.log(myObject);
//> Object
```

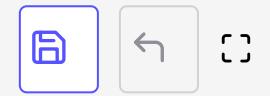


```

//> Object
//> publicProperty: "Value of myObject[\"publicProperty\"]"
//> Symbol(): "Value of myObject[symbol1]"
//> Symbol(): "value of myObject[symbol2]"
//> __proto__: Object

console.log(myObject[symbol1]);
//> Value of myObject[symbol1]

```



When console logging `myObject` inside the *Chrome Developer Tools*, you can see that both symbol properties are stored in the object. The literal `"Symbol()"` is the return value of the `toString()` method called on the symbol. This value denotes the presence of a symbol key in the console. We can retrieve the corresponding values if we have access to the right symbol.

In some environments such as node and Microsoft Edge console, Symbol keys are not logged. In case you don't see the two Symbol properties, the problem is not with your copy-pasting or typing skills.

Symbol properties do not appear in the console of NodeJs (bottom part of the code editor), but they appear in the Google Chrome devtools(below the code editor).

```

~/projects/es6inpractice/test.js (es6inpractice) - Sublime Text
FOLDERS
 es6inpractice
 01 - Arrow Functions
 01 - Fat Arrow Syntax.js
 02 - Context Binding.js
 test.js

test.js
1 let symbol1 = Symbol();
2 let symbol2 = Symbol();
3
4 let myObject = {
5 publicProperty: 'Value of myObject["publicProperty"]'
6 };
7
8 myObject[symbol1] = 'Value of myObject[symbol1]';
9 myObject[symbol2] = 'value of myObject[symbol2]';
10
11 console.log(myObject);

{ publicProperty: 'Value of myObject["publicProperty"]' }
[Finished in 0.1s]

Line 5, Column 41
Elements Console Sources Network Timeline Profiles Application Security Audits
top Preserve log
let myObject = {
 publicProperty: 'Value of myObject["publicProperty"]'
};

myObject[symbol1] = 'Value of myObject[symbol1]';
myObject[symbol2] = 'value of myObject[symbol2]';

console.log(myObject);
Object {
 publicProperty: "Value of myObject[\"publicProperty\"]"
 Symbol(): "Value of myObject[symbol1]"
 Symbol(): "value of myObject[symbol2]"
}
 __proto__: Object
<- undefined
>
VM371:11

```

Following [this node issue](#), using `console.dir`, with `showHidden` flag set to `true`, reveals the Symbol keys.

```
console.dir(myObject, {showHidden: true})
```



Properties with a symbol key don't appear in the JSON representation of your object. Not even the for-in loop or `Object.keys` can enumerate them:

```
JSON.stringify(myObject)
//> {"publicProperty":"Value of myObject[\"publicProperty\"] "}

for(var prop in myObject) {
 console.log(prop, myObject[prop]);
}
//> publicProperty Value of myObject["publicProperty"]

console.log(Object.keys(myObject));
//> ["publicProperty"]
```



Even though properties with Symbol keys don't appear in the above cases, these properties are not fully private in a strict sense.

`Object.getOwnPropertySymbols` provides a way to retrieve the symbol keys of your objects:

```
console.log(Object.getOwnPropertySymbols(myObject));
//> [Symbol(), Symbol()]
```

```
console.log(myObject[Object.getOwnPropertySymbols(myObject)[0]]);
//> "Value of myObject[symbol1]"
```



If you choose to represent private variables with Symbol keys, make sure you don't use `Object.getOwnPropertySymbols` to retrieve properties that are intended to be private. In this case, the only use cases for

`Object.getOwnPropertySymbols` are testing and debugging.

As long as you respect the above rule, your object keys will be private from the perspective of developing your code. In practice, however, be aware that others will be able to access your private values.

Even though symbol keys are not enumerated by `for...of`, the spread operator, or `Object.keys`, they still make it to shallow copies of our objects:

```
clonedObject = Object.assign({}, myObject);

console.log(clonedObject);
//> Object
//> publicProperty: "Value of myObject[\"publicProperty\"]"
//> Symbol(): "Value of myObject[symbol1]"
//> Symbol(): "value of myObject[symbol2]"
//> __proto__: Object
```



Naming your symbols properly is essential in indicating what your symbol is used for. If you need additional semantic guidance, it is also possible to attach a description to your symbol. The description of the symbol appears in the string value of the symbol.

```
let leftNode = Symbol('Binary tree node');
let rightNode = Symbol('Binary tree node');

console.log(leftNode)
//> Symbol(Binary tree node)
```



Always provide a description for your symbols, and make your descriptions unique. If you use symbols for accessing private properties, treat their descriptions as if they were variable names.

Even if you pass the same description to two symbols, their value will still differ. Knowing the description does not make it possible for you to create the same symbol.



```
let leftNode = Symbol('Binary tree node');
let rightNode = Symbol('Binary tree node');
console.log(leftNode === rightNode);
//> false
```



In the next lesson, we will talk about the global resource for creating symbols.

# Global Symbol Registry

relationship between symbol values and their string keys

ES6 has a global resource for creating symbols: the symbol registry. The symbol registry provides us with a one-to-one relationship between strings and symbols. The registry returns symbols using `Symbol.for( key )`.

`Symbol.for( key1 ) === Symbol.for( key2 )` whenever `key1 === key2`. This correspondance works even across service workers and iframes.

```
let privateProperty1 = Symbol.for('firstName');
let privateProperty2 = Symbol.for('firstName');

myObject[privateProperty1] = 'Dave';
myObject[privateProperty2] = 'Zsolt';

console.log(myObject[privateProperty1]);
// Zsolt
```



As there is a one-to-one correspondence between symbol values and their string keys in the symbol registry, it is also possible to retrieve the string key. Use the `Symbol.keyFor` method.

```
console.log(Symbol.keyFor(privateProperty1));
//> "firstName"

console.log(Symbol.keyFor(Symbol()));
//> undefined
```



Now, let's talk about making symbols private.



# Symbols as Semi-Private Property Keys

advantages of making object properties private, its drawbacks and use cases

Creating truly private properties and operations is feasible, but it's not an obvious task in JavaScript. If it was as obvious as in Java, blog posts like [this](#), [this](#), [this](#), and many more wouldn't have emerged.

Check out [Exercise 2](#) to find out more about how to simulate private variables in JavaScript to decide whether it's worth for you.

Even though Symbols do not make attributes private, they can be used as a notation for private properties. You can use symbols to separate the enumeration of public and private properties, and the notation also makes it clear.

```
const _width = Symbol('width');
class Square {
 constructor(width0) {
 this[_width] = width0;
 }
 getWidth() {
 return this[_width];
 }
}
```



As long as you can hide the `_width` constant, you should be fine. One option to hide `_width` is to create a closure:

```
let Square = (function() {
 const _width = Symbol('width');

 return class {
 constructor(width0) {
 this[_width] = width0;
 }
 getWidth() {

```



```
getwidth() {
 return this[_width];
}
}

})();
```



The advantage of this approach is that it becomes intentionally harder to access the private `_width` value of our objects. It is also evident which of our properties are intended to be public, and which private. The solution is not bulletproof, but some developers do use this approach in favor of indicating privacy by starting a variable with an underscore.

The drawbacks are also obvious:

- By calling `Object.getOwnPropertySymbols`, we can get access to the symbol keys. Therefore, private fields are not truly private.
- Developer experience is also worse, as they have to write more code. Accessing private properties is not as convenient as in Java or TypeScript for example.

Some developers will express their opinion on using symbols for indicating privacy. In practice, your team has the freedom of deciding which practices to stick to, and which rules to follow. If you agree on using symbols as private keys, it is a working solution, as long as you don't start writing workarounds to access private field values publicly.

If you use symbols to denote private fields, you have done your best to indicate that a property is not to be accessed publicly. When someone writes code violating this common sense intention, they should bear the consequences.

There are various methods for structuring your code such that you indicate that some of your variables are private in JavaScript. None of them looks as elegant as a `private` access modifier.

If you want real privacy, you can achieve it even without using ES6. Exercise 2 deals with this topic. Try to solve it, or read the reference solution.

The question is not whether it is possible to simulate private fields in

JavaScript. The real question is whether you want to simulate them or not.

Once you figure out that you don't need truly private fields for development, you can agree whether you use symbols, weak maps (covered later), closures, or a simple underscore prefix in front of your variables.

In the next lesson, let's talk about enum types.

# Creating enum Types

a brief introduction to enum types and their syntax

Enums allow you to define constants with semantic names and unique values. Given that the values of symbols are different, they make excellent values for enumerated types.

```
const directions = {
 UP : Symbol('UP'),
 DOWN : Symbol('DOWN'),
 LEFT : Symbol('LEFT'),
 RIGHT: Symbol('RIGHT')
};
console.log(directions);
```



In the next lesson, let's talk about another problem - name clashes.

# Well-known Symbols

name clashes occurring in objects and how to avoid them using Symbols

When using symbols as identifiers for objects, we don't have to set up a global registry of available identifiers. We also save the creation of a new identifier, as all we need to do is create a `Symbol()`.

The same holds for external libraries.

In the next lesson, we'll talk about some well-known symbols and how they change the behavior of your code.

## Well-known Symbols

There are some [well-known symbols](#) defined to access and modify internal JavaScript behavior. For example, you can redefine built-in methods, operators, and loops. Redefining standard language behavior usually confuses other developers. Ask yourself, will this skill move you forward in your career?

We will not focus on well-known symbols in this section. If there is a valid use case for it, I will signal it in the corresponding lesson. Otherwise, I suggest staying away from manipulating the expected behavior of your code.

Before learning newer concepts, let's solve some exercises.

# Exercise on Symbols

The exercises below will give you a deeper understanding of symbol along with the pros and cons of using them.

## Exercise 1:

What are the pros and cons of using an underscore prefix for expressing our intention that a field is private? Compare this approach with symbols!

```
let mySquare = {
 _width: 5,
 getWidth() { return this._width; }
}
```



## Solution 1:

Pros:

- Notation and developer experience is simple, provided that your team spreads this practice
- It does not result in a hard-to-read code structure, all you need is one more character

Cons:

- The properties are not private in practice, they are just denoted as private, which opens up a possibility of hacking quick and dirty solutions
- Unlike symbols, there is no clear separation between public and private properties. Private properties appear in the public interface of an object and they are enumerated in for...of loops, using the spread operator, and Object.keys

## Exercise 2:

This exercise is based on the previous challenge. Continue reading to learn how to solve it.

Find a way to simulate truly private fields in JavaScript!

## Solution 2:

When it comes to constructor functions, private members can be declared inside a constructor function using var, let, or const. As in the following,

```
function F() {
 let privateProperty = 'b';
 this.publicProperty = 'a';
}

let f = new F();
console.log(f.publicProperty); // returns 'a'
console.log(f.privateProperty); // returns undefined
```



In order to use the same idea for classes, we have to place the method definitions that use private properties in the constructor method in a scope where the private properties are accessible. We will use Object.assign to accomplish this goal. This solution was inspired by an article I read on this topic by Dr. Axel Rauschmayer on Managing private data of ES6 classes<sup>11</sup>.

```
class C
{
 constructor()
 {
 let privateProperty = 'a';
 Object.assign(this,
 {
 logPrivateProperty()
 {
 console.log(privateProperty);
 }
 });
 }
}

let c = new C();
c.logPrivateProperty();
```



The field `privateProperty` is not accessible in the `c` object. The solution also

The field `privateProperty` is not accessible in the `C` object. The solution also works when we extend the `C` class.

```
// Class C from the above is prepended to keep the code short
class D extends C { constructor() {
 super();
 console.log('Constructor of D');
}
}

let d = new D()
d.logPrivateProperty()
```



For the sake of completeness, there are two other ways for creating private variables:

- Weak maps: we will introduce it in a later section. We can achieve true privacy with it, at the expense of writing less elegant code,
- TypeScript: introduces compile time checks whether our code treats private variables as private.

# Introducing the for-of loop

Javascript syntax for the for-of loop

The for-of loop is a new powerful construct in ES6 that simplifies your code.

The for-of loop works for any JavaScript iterable object. Don't worry about iterables, we will learn about them later. For now, we will use the for-of loops on arrays, strings, and NodeLists.

In our first example, let's log the characters of a string:

```
let message = 'hello';

for(let i in message) {
 console.log(message[i]);
}
```



Often, you need `message[i]` inside the loop block, and not `i`. Therefore, it makes perfect sense to use a loop that gives us the value `message[i]` instead of `i`. This is what the for-of loop provides us with:

```
let message = 'hello';

for(let ch of message) {
 console.log(ch);
}
```



In the next lesson, let's talk about how the for-of loop handles Unicode characters.



# UTF-32 support

Unicode character support of ES6 and comparison between the for-of and for-in loops

Let me give you some foreshadowing about the next section, where we will deal with template literals. You will learn about the Unicode character support of ES6. You will also learn that in ES5, three and four byte long characters are sometimes broken into 2-byte chunks.

```
let text = '\u{1F601}\u{1F43C}';
console.log('text: ', text);

for(let i in text) {
 console.log(text[i]);
}

console.log('-----');

for (let c of text) {
 console.log(c);
};
```



Execute the above code.

The for-of loop handles UTF-32 characters properly. The for-in loop interprets them as two byte long characters.

Unlike the for-in loop, the for-of loop parses all Unicode characters properly, regardless of their size.

Now, let's move on to destructuring using the for-of loop.



# Destructuring and the DOM in the for-of loop

the syntax of destructuring in the for-of loop using examples

If your elements are objects, it is possible to use destructuring inside the for-of loop:

```
let flights = [
 { source: 'Dublin', destination: 'Warsaw' },
 { source: 'New York', destination: 'Phoenix' }
];

for (let { source, destination } of flights) {
 console.log(source, destination);
}
```



When you use destructuring, you can omit the fields you are not interested in.

```
for (let { source } of flights) {
 console.log(source);
}
```



## The for-loop and the DOM

It is possible to use the for-of loop on a `NodeList`. The below snippet changes the default font color of each div on a website randomly. Try it out on your favorite website!

```
let divs = document.querySelectorAll('div');
for (let div of divs)
{
 let rand = Math.floor (Math.random() * 3);
 div.style.color = ['#990000', '#009900', '#000099'][rand];
}
```



Now, let's solve some exercises before learning new concepts.

# Exercise on the for-of loop

We will use the for-of loop to search through data and obtain our desired values.

## Exercise 1:

Open the developer tools on any website. Locate the first character of all headings, and log the concatenation of the first characters.

|                                                                                              |                                                                                            |
|----------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
|  Exercise 1 |  Solution |
|----------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|

```
let text = '';
let nodes = // Write code here

for (let node of nodes) {
 // Write code here
}

console.log(text);
```



## Explanation:

The data from the headings can be obtained in several ways. The solution uses

```
document.querySelectorAll('h1', 'h2', 'h3', 'h4', 'h5', 'h6')
```

This query returns those 6 headings in the form of nodes. The data is in the child of the node. Hence, we use `node.childNodes[0].textContent[0]` where `textContent[0]` refers to the first character of the text.

The rest is plain concatenation in a for-of loop.

## Exercise 2:

Assemble a string containing all emojis between the hexadecimal codes `1F601` and `1F64F` in the respective order. Use the for-of loop.

Reference: <http://apps.timwhitlock.info/emoji/tables/unicode>

`String.fromCodePoint` converts a decimal number into a character, even if it is a 4-byte long number.

```
//Write your code here
```



## Explanation:

In the first 4 lines of our code, we define our prefixes and ranges because we know all of them and we know how they progress:

```
let prefix = '1F6';
let digits4 = '01234';
let digits5 = '01234567890ABCDEF';
let emojis = '';
```

Then, using a nested loop, we traverse each element of `digits4`, appending it to the prefix, and then further concatenating each element from `digits5` to this hexadecimal value, `hex`.

This value is converted to string using `String.fromCodePoint`.

# Introduction

the importance of learning strings and template literals

It is time to have a look at string manipulation and template literals. There has been a shift in the trends of web development in the last ten years that moved the responsibility of rendering from the server side to the client side.

Therefore, the HTML code that you can see on screen is often assembled using JavaScript. Furthermore, other trends such as **isomorphic Javascript** have also emerged, where full stack developers can use Javascript to render templates both on the **server side** and on the **client side**.

This is why it is important for JavaScript developers to learn about strings and template literals in depth.

As a side benefit, template literals are more convenient to use than strings, especially when it comes to *string concatenation*.

Let's talk about this in more detail in the following lessons.

# New String Methods

built-in Javascript methods to parse strings and analyze them, along with examples

Some popular checks that can be done with regular expressions are now possible in a more semantic way. Here are some functions that we can use:

| Function Name           | Syntax                         | Output                                                                |
|-------------------------|--------------------------------|-----------------------------------------------------------------------|
| <code>startsWith</code> | <code>s1.startsWith(s2)</code> | true if and only if <code>s1</code> starts with <code>s2</code>       |
| <code>endsWith</code>   | <code>s1.endsWith(s2)</code>   | true if and only if <code>s1</code> ends with <code>s2</code>         |
| <code>includes :</code> | <code>s1.includes(s2)</code>   | true if and only if <code>s2</code> is a substring of <code>s1</code> |
| <code>repeat</code>     | <code>s.repeat(n)</code>       | replicates <code>s</code> <code>n</code> times, and joins them        |

## Examples:

```
console.log('Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz'.startsWith('Ri
//> true
console.log('not good'.endsWith('good'));
//> true

console.log('good or bad'.includes(' or '));
//> true

console.log('ha'.repeat(4));
//> 'hahahaha'
```



Side note: if you are wondering where I got the long word from, it was a [valid German word](#) not too long ago.

In the next lesson, let's talk about handling unicode strings.

# Better Unicode Support

handling Unicode strings in ES5 and its alternative in ES6: using the for-of loop and spread operator to process characters

We are going to a lower level in this section.

ES5 handles string operations in two-byte chunks. As a result, handling Unicode strings with a mixture of two and four byte long codes often gets confusing. Characters of Unicode strings were often not printable on their own, and in general, any string operation required extra care.

For instance, if you recall the first exercise of the last section, the reference solution would have failed if a title had started with a three or four-byte long character.

We still calculate the length of a string by dividing the number of bytes allocated by a string by two. However, there are some updates in ES6 that make String handling more user-friendly.

- `codePointAt(n)` returns the code of the character at the `n`th position regardless of whether it is a two byte or a four-byte long character. If `n` points at the second half of a four-byte Unicode character, only the code of the second half is returned

In ES6, it is possible to define four byte long Unicode characters with their code:

```
console.log('\u{1f600}') // becomes an emoji
console.log('\u{1f600}'.length) // is 2
console.log('\u{1f600}'.charCodeAt(0)) // is 55357
console.log('\u{1f600}'.codePointAt(0)) // is 128512
console.log('\u{1f600}'.charCodeAt(1)) // is 56832
console.log('\u{1f600}'.codePointAt(1)) // is 56832
console.log('\u{1f600}'.startsWith('\u{1f600}'[0])) // is true
```



Note that the `startsWith`, `endsWith`, `includes` methods interpret the result in two-byte chunks.

The for-of loop interprets three and four byte long characters as one unit, scoring another convenience point for ES6. Let's execute the following:

```
let str = '\u{1f600}\u{00fa}é';

for (const ch of str) {
 console.log(ch);
}
```



What is printed to the console? Notice that

- The `for-of` loop prints three characters, an emoji, `ú`, and `é`.
- `[...str]` spreads `str` character by character
- Even though `[...str]` has three elements, the length of the `str` string is `4`. This is because the length of `[...str][0]` is `2`.

Use the `for-of` loop or the spread operator to process characters of a string regardless of their length in bytes.

This section contains a minimalistic summary of Unicode features of ES6 from a practical point of view. If you have to handle four byte-long characters on a regular basis, you should do your research. Make sure you know the `normalize` method for Unicode normalization. Make sure you know that there is a new `u` flag to influence whether to consider Unicode characters when testing regular expressions.

Normalization and Unicode support for regular expressions are outside the scope of this course.

In the next lesson, we'll talk about using template literals to work with string templates.



# Template Literals

introduction to template literals and its usage

The purpose of *template literals* is to evaluate and insert values of JavaScript expressions in a template string.

```
let x = 555; //<-notice 555
let evaluatedTemplate = `${x} === 555 is ${x === 555}`;
console.log(evaluatedTemplate);
// evaluatedTemplate becomes "555 === 555 is true"

let y = '555'; //<-notice '555'
evaluatedTemplate = `${y} === 555 is ${y === 555}`;
console.log(evaluatedTemplate);
// evaluatedTemplate becomes "555 === 555 is false"
```



One practical application of template literals is the creation of DOM markup with static structure, and dynamic values.

For more information, check out [my article on Underscore templating](#). If you compare the ES6 syntax, you will find that it is a lot more compact than the Underscore syntax.

Another application of template literals is to simplify writing strings that span multiple lines:

```
`first
second
third
fourth`
```

In the next lesson, let's talk about how to perform a transformation on a template literal.

# Tagged Templates

the ins and outs of tagged templates using a `literalFragments` array, and the `tag` & `format` functions

A template tag is a function performing a transformation on a template literal, returning a string. The signature of a tag function is:

```
tagFunction(literalFragments, ...substitutionValues)
```

- `literalFragments` is an array of Strings that store fragments of the template literal. We use substitutions to split the original template literal.
- the rest parameter `...substitutionValues` contains the values of  `${...}` substitutions.

For the sake of simplicity, suppose that we only use alphanumeric characters in template substitutions. To understand how `literalFragments` are constructed, study the behavior of the JavaScript `split` method:

```
let emulatedSubs = '${sub1}abc ${sub2} def${sub3}'
 .split(/\${\w*}/);
console.log(emulatedSubs);
```



```
> ["", "abc ", " def", ""]
```

The `emulatedSubs` array contains all text fragments in order. The `i`th element of `emulatedSubs` is before the `(i+1)`th argument of the tag function, representing the substitution  `${sub_i}`.

Let's now observe how the real `literalFragments` are constructed. We will create a tag function that prints all of its arguments. Let's execute this tag function on the template  `${sub1}abc ${sub2} def${sub3}`.

```
let sub1=1, sub2=2, sub3 = 3;
((x, ...subs) => {

 console.log(x, ...subs);
})`$ {sub1}abc ${sub2} def${sub3}`
```



```
> ["", "abc ", " def", "", raw: Array[4]] 1 2 3
```

There is one small difference in the construction of the `literalFragments` array: the array has an associative property `raw`, containing the same four literal fragments as raw values.

As a first real example, let's create a salutation tag.

```
let salutation = literalFragments =>
 'Hello, ' + literalFragments[0];

console.log(salutation`Ashley`);
```



```
> "Hello, Ashley"
```

If variable substitutions occur inside the template literal, their values can also be manipulated using tag functions.

```
let price = 5999.9;
let currencySymbol = '€';
let productName = 'Titanium Toothbrush';

let formatCurrency = function(currency, amount) {
 return amount.toFixed(2) + currency;
}

let format = (textArray, ...substitutions) => {
 let template = textArray[0];
 template += substitutions[0];
 template += textArray[1];
 template += formatCurrency(substitutions[1], substitutions[2]);
 template += textArray[3];

 return template;
};
```



In the `format` function, we can access all variable substitutions and template fragments, and we can concatenate them in any order. Substitutions come from evaluating the values of `productName`, `currencySymbol`, and `price` in the scope of the template evaluation.

The result of the tagged template above looks like this:

```
<div class="js-product">
 Product: Titanium Toothbrush
</div>
<div class="js-price">
 Price: 5999.90€
</div>
```



Even though in this specific case, the `format` function relies on knowledge of the structure of the template, in some cases, templates have a variable number of substitutions. One of the exercises will require you to create tag functions handling a variable number of substitutions.

Now, let's solve some exercises on this before moving on to new concepts.

# Exercise on String and Template Literals

Practice your string and template literals by printing an array in a tabular format, manipulating strings, printing emojis, and playing with if-else clauses.

## Exercise 1:

Use template literals to create a table that prints the following array in a tabular format.

```
let departures = [
 {
 id : 'KL 1255',
 destination : 'Amsterdam',
 departureTime : '21:55',
 gate : 'A13',
 },
 {
 id : 'OK 001',
 destination : 'Prague',
 departureTime : '20:40',
 gate : 'A13',
 status : 'Check-in'
 },
 {
 id : '4U 2011',
 destination : 'Stuttgart',
 departureTime : '20:35',
 gate : 'A11',
 status : 'Check-in'
 },
 {
 id : 'LX 911',
 destination : 'Zurich',
 departureTime : '20:15',
 expectedDepartureTime : '21:15',
 status : 'check-in'
 },
 {
 id : 'OS 133',
 destination : 'Vienna',
 departureTime : '19:25',
 gate : 'A06',
 status : 'Departed'
 }
];

let headers = {
 id : 'Id',
 destination : 'Destination',
 departureTime : 'Departure Time',
 gate : 'Gate',
 status : 'Status'
};
```

```

 id : 'Id',
 destination : 'Destination',
 departureTime : 'DepartureTime',
 expectedDepartureTime : 'Expected Departure Time',
 gate : 'Gate',
 status : 'Status'
 }

//Write your code here...

```



- We will build our solution brick by brick and start with the table header.
- We will then create a template literal for one table row. For the sake of simplicity, we will use departures[0] as sample data.
- We already know how to insert one row into a template. We will then insert all rows.

In real life, you might want to research escaping the result, and automatizing some parts of the template building process. By substituting the variables recursively in the template literal, we get the literal as given in the solution.

## Exercise 2:

Create a tag function that transforms all of its String substitutions into upper case.

```

let upper = (textArray, ...substitutions) => {
 //Write your code here...
};

let a = 1, b = 'ab', c = 'DeF';
console.log(upper`x ${a} x ${b} x ${c} x`);

```



Follow the tag function syntax from the theory part. We have a `textArray` of length `substitutions.length - 1`. All we need to do is connect the text and substitution segments, and transform each substitution.

## Exercise 3:

Do you remember Exercise 2 of the section on “for-of loops”? It is now time to

create a template literal that prints out all the emojis in a tabular format. The rows of the table should be the last character in the hexadecimal code of the emoji (0 - F), while the columns should be the fourth character of the emoji (0 - 4). Make sure to print out the table to the console.

Reference: <http://apps.timwhitlock.info/emoji/tables/unicode>

```
let prefix = '1F6';
let digits4 = '01234';
let digits5 = '01234567890ABCDEF';

//Write your code here...
```



Define the header first. Notice the spread operator to handle the digits4 string as an array of characters.

Building the rows should be straightforward based on the first exercise.

- We used `String.fromCodePoint( hexString )` to convert a string of a hexadecimal number to a Unicode character. “0x” denotes that the integer parser should parse a hexadecimal number
- Read the code starting in the rightmost indentation level, and proceed leftwards
- The map function of [...digits4] prepares the cells of each row
- The map function of [...digits5] prepares the five rows of the table

Changing the granularity level of the template for debugging purposes is advised.

If you paste all the code in the Chrome developer tools, and then execute

```
document.body.innerHTML = header
```

you can see the table of emojis appear.

## Exercise 4:

Find a way to create the following `if-else` clause in a template literal.

```
if (10 % 2 == 0) {
 console.log("Even");
} else {
 console.log("Odd");
}
```

```
// Write your code here...
```



First of all, note that you don't have to include conditions, there is always a way to describe the template in more steps.

If-else clauses are used mostly when we want to conditionally hide a template segment, or we want to do type checks, substitute default values, and avoid reference errors. In both cases, the if-else clause should determine if we print out a value or not:

```
if (condition) {
 // value1
} else {
 // value2
}
```

We can simply use the ternary operator to return the proper value.

# Data Structures 101

introduction to sets and maps in Javascript

This lesson introduces some data structures that we often use during coding: sets and maps. Using these data structures prevents us from reinventing the wheel.

A **set** is an unordered collection of distinct elements. An element is said to be a member of the set, if the set contains the element. Adding an element to the set that's already a member does not change that set.

While a mathematical set is unordered, the ES6 `Set` data structure is an ordered list of distinct elements. Don't confuse yourself with the mathematical definition. We can enumerate the elements of an ES6 `Set` in a given order.

There are some operations defined on sets. These are

- union: `e` becomes a member of `union( A, B )`, if `e` is a member of `A`, or `e` is a member of `B`.
- intersection: `e` becomes a member of `intersection( A, B )`, if `e` is a member of `A`, and `e` is a member of `B`.
- difference: `e` becomes a member of `difference( A, B )`, if `e` is a member of `A`, and `e` is **not** a member of `B`.

A **map** is a collection of key-value pairs. If you provide a key, you can retrieve the corresponding value. This sounds very much like a simple object. However, there are significant differences between objects and maps. Objects can only work as maps under special circumstances.

In the next few lessons, we'll cover both these data structures in more detail.

# ES6 Sets

set functions and their usage

A `Set` data structure in ES6 is an *ordered* list of unique elements. Here are some set functions and their usage:

| Keyword             | Type        | Usage                                      |
|---------------------|-------------|--------------------------------------------|
| <code>Set</code>    | constructor | creates a set                              |
| <code>add</code>    | method      | adds elements to the set                   |
| <code>size</code>   | property    | check size of the set                      |
| <code>has</code>    | method      | check if an element is a member of the set |
| <code>delete</code> | method      | remove a value from a set                  |

Consider the following code:

```
let colors = new Set();

colors.add('red');
colors.add('green');
colors.add('red'); // duplicate elements are added only once
console.log(colors);
//> Set {"red", "green"}

console.log('Size: ' + colors.size);
//> 2

console.log('has green: ' + colors.has('green') + '\nhas blue: ' + colors.has('blue'));
```

```
//> true false
```



You can remove a value from a set by calling its `delete` method. The return value of the removal is a boolean, indicating whether the removed element was initially a member of the set or not.

```
console.log('Before deleting: ')
console.log(colors);
colors.delete('green')
//> true
colors.delete('green')
//> false

console.log('\nAfter deleting: ')
console.log(colors);
```



The `Set` constructor accepts an optional array argument with initial values. It eliminates all duplicates.

```
let moreColors = new Set(['red', 'blue', 'red', 'orange']);
console.log(moreColors);
```



We will revisit the set construction when learning about iterators and generators.

Now, let's move on to set iteration.

# Iterating Sets

set traversal using the `forEach` method and `for...of` loop

Sometimes we have to traverse all the elements of a set. This can be done in multiple ways:

- `forEach` method,
- `for...of` loop,
- transforming a set into an array by using the spread operator

```
console.log('forEach function:')
moreColors.forEach(value => { console.log(value) });
//> red
//> blue

console.log('\nfor...of loop:')
for (let value of moreColors) {
 console.log(value);
}
//> red
//> blue

console.log('\nspread operator:')
console.log([...moreColors]);
//> ["red", "blue"]
```



Technically, the function argument of the `forEach` method accepts up to three parameters:

```
moreColors.forEach(console.log);
```



As `console.log` accepts a variable number of arguments, it prints all the arguments the iteration provides. These are:

- the upcoming value stored in the set,
- the belonging key,
- a reference to the set itself.

The second and the third arguments are not too useful. Keys and values are pairwise equal in sets. Therefore, the first two arguments are always equivalent in sets. The reason why a key was provided in `forEach` is compatibility with maps. We may need the third argument in case we don't have a reference to the set. However, use cases for this are rare.

As `Set` is an ordered list, iteration is performed in the order of adding the elements.

Now, let's move on to Maps.

# ES6 Maps

introduction to maps and their methods

Maps represent key-value pairs, similar to objects.

You may be wondering why we need a map data structure if objects can also be used as maps. The answer is simple: maps can have keys of any type, and the keys are not converted to strings. Therefore, `0` and `'0'` are two different, and valid keys.

Objects are also valid keys for maps. They are not converted to `'[object Object]'`. Note though, that objects are reference types. Therefore, using the `{}` object literal twice as keys will result in two different entries in the map.

We can build maps in multiple ways. We can either use the `set` method to add values to the map or pass an array of key-value pairs to the map constructor.

```
let horses = new Map();
horses.set(8, 'Chocolate');
horses.set(3, 'Filippone');
console.log(horses);
```



As the `set` method is chainable, the above code is equivalent to

```
let horses = new Map().set(8, 'Chocolate').set(3, 'Filippone');
console.log(horses);
```



Using arrays of key-value pairs works as follows:



```
let horses = new Map([[8, 'Chocolate'], [3, 'Filippone']]);
console.log(horses);
```



The property `size`, and the methods `has` and `delete` also work for maps. The `has` and `delete` methods expect a key in the map. Also, we can get a value from the map using the `get` method and providing a key.



```
console.log('Size:\t'+horses.size);
//> 2

console.log('has id=3:\t'+horses.has(3));
//> true

console.log('value at key=3:\t'+horses.get(3));
//> "Filippone"

horses.delete(3);
console.log('\nAfter deleting:\t');
console.log(horses);
//> true
```



In the next lesson, we will talk about traversing maps.

# Iterating Maps

the various ways to traverse a Map

We have the same options as in the case of sets. The only exception is that we have to take care of a key and a value.

```
console.log('forEach function:');
horses.forEach((value, key) => { console.log(value, key) });
//> Chocolate 8
//> Filippone 3

console.log('\nfor...of loop:');
for (let [key, value] of horses) {
 console.log(key, value);
}
//> 8 "Chocolate"
//> 3 "Filippone"

console.log('\nspread operators:');
console.log([...horses]);
//> [[8,"Chocolate"],[3,"Filippone"]]
```



Notice the following behavior:

- The destructuring operation in the `for...of` loop. Iterating a map results in a key-value pair.
- The spread operator creates an enumeration of key-value pairs. These key-value pairs can be used to build an array.
- The order of elements in the `forEach` method is value, key. The order in the `for...of` loop is key, value.

In the next lesson, you'll learn weak sets.



# Weak Sets

introduction to weak sets, their usage and comparison with regular sets

Sets and maps hold a reference of their values. This means that the garbage collector won't be able to collect the values in sets, and key-value pairs in maps to free some memory.

This is where weak sets and maps come into play. They only hold *weak references* to their values, allowing garbage collection of the values while they are members of a set or map.

Weak sets are similar to regular sets. The main difference is that their elements may disappear once they are garbage collected. Consider the following code:

```
let firstElement = { order: 1 }, secondElement = { order: 2 };
let ws = new WeakSet([firstElement, secondElement]);

console.log('has firstElement: '+ws.has(firstElement));
//> true

delete firstElement;
// firstElement is removed from the weak set
```



Other characteristics of weak sets include:

- You don't have access to the size of the set.
- Weak sets may only store objects.
- Weak sets are not iterable.

The use cases for weak sets are limited. We can use them whenever we deal with objects that can be garbage collected from other sources, and we only want to check the existence of objects in the weak set.

For instance, whenever we traverse a graph, and we would like to detect cycles, we can store the nodes in weak sets during traversal.

Now, let's talk about weak maps.

# Weak Maps

weak maps and their behavior in Javascript

Weak maps have object keys, and arbitrary values. When all strong references to a key are removed, the key is garbage collected, and the key-value pair is removed from the weak map.

Only the keys of weak maps are weak. Values placed in a weak map have strong references in the map.

Consider the following code:

```
let firstElement = { order: 1 }, secondElement = { order: 2 };
let wm = new WeakMap();

wm.set(firstElement, 1);
wm.set(secondElement, {});

console.log(wm.get(secondElement));
//> {}

delete secondElement;
// secondElement is removed from the weak map
```



As soon as a key of the weak map is not referenced anymore, the key-value pair is removed from the weak map.

Check out Exercise 2 in the following lesson to experience a major use case of weak maps: storing private data.

# Exercise on Sets, Maps, and their Weak Versions

We will use Sets and Maps to perform various operations on arrays, shapes and sets.

## Exercise 1:

Write a function that removes all the duplicates from an array.

Name your function: `removeDuplicatesFromArray`.

```
// Write your function here...
```



## Exercise 2:

Make the property `_width` private for the below class using weak maps. Make sure the property is unique for each created object, and the property cannot be accessed from outside instances of the class.

```
class Square {
 constructor(width) {
 this._width = width;
 }
 get area() {
 return this._width * this._width;
 }
}
```

```
//Write your Code here
```



## Exercise 3:

Implement the `union`, `intersection`, and `difference` operations for sets.

```
// Write your code here...
```



# Iterables and Iterators

introduction to Iterables and Iterators and their behavior in Javascript

Now that we have introduced the `for...of` loop and symbols, we can comfortably talk about iterators.

It is worth for you to learn about iterators, especially if you like lazy evaluation, or you want to be able to describe infinite sequences. Understanding of iterators also helps you understand generators, promises, sets, and maps better.

Once we cover the fundamentals of iterators, we will use our knowledge to understand how generators work.

ES6 comes with the *iterable* protocol. The protocol defines the iterating behavior of JavaScript objects.

An *iterable object* has an iterator method with the key `Symbol.iterator`. This method returns an *iterator object*.

```
let iterableObject = {
 [Symbol.iterator]() { return iteratorObject; }
};
```



`Symbol.iterator` is a *well-known symbol*. We will now use `Symbol.iterator` to describe an iterable object. Note that we are using this construct for the sake of understanding how iterators work. Technically, you will hardly ever need `Symbol.iterator` in your code. You will soon learn another way to define iterables.

An *iterator object* is a data structure that has a `next` method. When calling this method on the iterator, it returns the next element, and a boolean

signaling whether we reached the end of the iteration.

```
// Place this before iterableObject
let iteratorObject = {
 next() {
 return {
 done: true,
 value: null
 };
 }
};
```



The return value of the `next` function has two keys:

- `done` is treated as a boolean. When `done` is true, the iteration ends, and `value` is not considered in the iteration.
- `value` is the upcoming value of the iteration. It is considered in the iteration if and only if `done` is false. When `done` is true, `value` becomes the return value of the iterator.

Let's create a countdown object as an example:

```
let countdownIterator = {
 countdown: 10,
 next() {
 this.countdown -= 1;
 return {
 done: this.countdown === 0,
 value: this.countdown
 };
 }
};

let countdownIterable = {
 [Symbol.iterator]() {
 return Object.assign({}, countdownIterator)
 }
};

let iterator = countdownIterable[Symbol.iterator]();

console.log(iterator.next());
//> Object {done: false, value: 9}

console.log(iterator.next());
//> Object {done: false, value: 8}
```



Note that the state of the iteration is preserved.

The role of `Object.assign` is that we create a shallow copy of the iterator object each time the iterable returns an iterator. This allows us to have multiple iterators on the same iterable object, storing their own internal state. Without `Object.assign`, we would just have multiple references to the same iterator object. Consider the following code:

```
let secondIterator = countdownIterable[Symbol.iterator]();
let thirdIterator = countdownIterable[Symbol.iterator]();

console.log(secondIterator.next());
//> Object {done: false, value: 9}

console.log(thirdIterator.next());
//> Object {done: false, value: 9}

console.log(secondIterator.next());
//> Object {done: false, value: 8}
```



We will now learn how to make use of iterators and iterable objects.

# Consuming Iterables

iterating an iterable object and an introduction to data consumers

Both the `for-of` loop and the spread operator can be used to perform iteration on an iterable object.

```
for (let element of iterableObject) {
 console.log(element);
}

console.log([...iterableObject]);
```



Using the countdown example, we can print out the result of the countdown in an array:

```
let countdownIterator = {
 countdown: 10,
 next() {
 this.countdown -= 1;
 return {
 done: this.countdown === 0,
 value: this.countdown
 };
 }
};

let countdownIterable = {
 [Symbol.iterator]() {
 return Object.assign({}, countdownIterator)
 }
};

console.log([...countdownIterable]);
//> [9, 8, 7, 6, 5, 4, 3, 2, 1]
```



Language constructs that consume iterable data are called **data consumers**.

We will learn about other data consumers soon. But first, let's study Iterables in a bit more detail.

# Built-in Iterables

built-in iterables explained using examples

You have seen four examples for built-in iterables among the previous lessons:

- Arrays are iterables, and work well with the `for-of` loop.
- Strings are iterables as arrays of 2 to 4-byte characters.
- DOM data structures are also iterables. If you want proof, just open a random website, and execute `[...document.querySelectorAll('p')]` in the console
- Maps and Sets are iterables.

Let's experiment with built-in iterables a bit:

```
let message = 'ok';

let stringIterator = message[Symbol.iterator]();
let secondStringIterator = message[Symbol.iterator]();

stringIterator.next();
//> Object {value: "o", done: false}

secondStringIterator.next();
//> Object {value: "o", done: false}

stringIterator.next();
//> Object {value: "k", done: false}

stringIterator.next();
//> Object {value: undefined, done: true}

secondStringIterator.next();
//> Object {value: "k", done: false}
```

Before you think how cool it is to use `Symbol.iterator` to get the iterator of built-in datatypes, I would like to emphasize that using `Symbol.iterator` is generally not cool. There is an easier way to get the iterator of built-in data structures using the public interface of built-in iterables.

You can create an `ArrayIterator` by calling the `entries` method of an array.

`ArrayIterator` objects yield an array of `[key, value]` in each iteration.

Strings can be handled as arrays using the spread operator:

```
let message = [...'ok'];

let pairs = message.entries();

for(let pair of pairs) {
 console.log(pair);
}
```



Now, let's talk about iterables with sets and maps.

# Iterables with Sets and Maps

using the entries, keys and values methods to play with iterables of keys or values

The `entries` method is defined on sets and maps. You can also use the `keys` and `values` method on a set or map to create an iterator/iterable of the keys or values. For example:

```
let colors = new Set(['red', 'yellow', 'green']);
let horses = new Map(
 [5, 'QuickBucks'],
 [8, 'Chocolate'],
 [3, 'Filippone']
]);

console.log(colors.entries());
//> SetIterator {["red", "red"], ["yellow", "yellow"], ["green", "green"]}

console.log('\n')
console.log(colors.keys());
//> SetIterator {"red", "yellow", "green"}

console.log(colors.values());
//> SetIterator {"red", "yellow", "green"}

console.log(horses.entries());
//> MapIterator {[5, "QuickBucks"], [8, "Chocolate"], [3, "Filippone"]}

console.log(horses.keys());
//> MapIterator {5, 8, 3}

console.log(horses.values());
//> MapIterator {"QuickBucks", "Chocolate", "Filippone"}
```



You don't need to create these iterators with the `keys`, `values`, or `entries` method though to perform an iteration on a set or a map. Sets and maps are iterable themselves. Therefore, they can be used in `for-of` loops.

A common destructuring pattern is to iterate the keys and values of a map using destructuring in a `for-of` loop:

Map using destructuring in a for...of loop.

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



When creating a set or a map, you can pass any iterable as an argument, provided that the results of the iteration can form a set or a map:

```
let nineToOne = new Set(countdownIterable);
console.log(nineToOne);

let horses = new Map([
 [5, 'QuickBucks'],
 [8, 'Chocolate'],
 [3, 'Filippone']
]);
console.log(horses);
```



In the first example, we used a custom iterable, while in the second example, we used an array or key-value pairs.

Now, let's talk about the role of the iterable interface.

# The Role of the Iterable Interface

understand iterators and their data flow, create independent iterators and connect the dots using ES6

We can understand iterables a bit better by concentrating on data flow:

- The `for-of` loop, the Spread operator, and some other language constructs are *data consumers*. They consume iterable data.
- Iterable data structures such as arrays, strings, dom data structures, maps, and sets are *data sources*
- The *iterable interface* specifies how to connect data consumers with data sources
- *Iterable objects* are created according to the iterable interface specification. Iterable objects can create iterator objects that facilitate the iteration on their data source, and prepare the result for a data consumer.

We can create independent iterator objects on the same iterable. Each iterator acts like a pointer to the upcoming element the linked data source can consume.

In the lesson on sets and maps, you have learned that you can convert sets to arrays using the spread operator:

```
let arr = [...set];
```

You now know that a set is an iterable object, and the spread operator is a data consumer. The formation of the array is based on the iterable interface. ES6 makes a lot of sense once you start connecting the dots. Now, let's talk about generators in the next lesson.

# Generators

comparison between generators and regular functions, use case of a generator

A **generator** is a special function that returns an iterator. There are some differences between generator functions and regular functions:

- There is an `*` after the `function` keyword.
- Generator functions create iterators.
- We use the `yield` keyword in the created iterator function. By writing `yield v`, the iterator returns `{ value: v, done: false }` as a value.
- We can also use the `return` keyword to end the iteration. Similar to iterators, the returned value won't be enumerated by a data consumer.
- The yielded result is the next value of the iteration process. Execution of the generator function is stopped at the point of yielding. Once a data consumer asks for another value, execution of the generator function is resumed by executing the statement after the last `yield`.

Consider the following example:

```
function *getLampIterator() {
 yield 'red';
 yield 'green';
 return 'lastValue';
 // implicit: return undefined;
}

let lampIterator = getLampIterator();

console.log(lampIterator.next());
//> {value: "red", done: false}

console.log(lampIterator.next());
//> {value: "green", done: false}

console.log(lampIterator.next());
//> {value: "lastValue", done: true}
```



If the return value were missing, the function would return `{value: undefined, done: true}`.

Use generators to define custom iterables to avoid using the well-known symbol `Symbol.iterator`.

In the next lesson, we'll discuss how generators can return iterables using various methods and operators.

# Generators and Iterators

generators returning Iterators that are also Iterables

Recall our string iterator example to refresh what iterable objects and iterators are:

```
let message = 'ok';
let stringIterator = message[Symbol.iterator]();
```



We call the `next` method of `stringIterator` to get the next element:

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



However, before we learned about iterators, we used the *iterable object* in a `for-of` loop, not the iterator:

```
let message = 'ok';
for (let ch of message) {
 console.log(ch);
}
```



Let's summarize what *iterables* and *iterators* are once more:

- Iterable objects have a `[Symbol.iterator]` method that returns an iterator.
- Iterator objects have a `next` method that returns an object with keys `value` and `done`.

`value` and `done`.

Generator functions return an object that is both an iterable and an iterator.  
Generator functions have:

- a `[Symbol.iterator]` method to return their iterator,
- a `next` method to perform the iteration.

As a consequence, the return value of generator functions can be used in `for-of` loops, after the spread operator, and in all places where iterables are consumed.

```
function *getLampIterator() {
 yield 'red';
 yield 'green';
 return 'lastValue';
 // implicit: return undefined;
}

let lampIterator = getLampIterator();

console.log(lampIterator.next());
//> {value: 'red', done: false}

console.log([...lampIterator]);
//> ['green']
```



In the above example, `[...lampIterator]` contains the remaining values of the iteration in an array.

Now, let's move on to Iterators and destructuring.

# Iterators and Destructuring

destructuring using iterators and their behavior

When equating an array to an iterable, iteration takes place.

```
let lampIterator = getLampIterator();
let [head,] = lampIterator;
console.log(head, [...lampIterator]);
//> red []
```



The destructuring assignment is executed as follows:

- First, `lampIterator` is substituted by an array of form `[...lampIterator]`.
- Then the array is destructured, and `head` is assigned to the first element of the array.
- The rest of the values are consumed, but they are not assigned to any value on the left.
- As `lampIterator` was used to build an array with all elements on the right-hand side, `[...lampIterator]` is empty in the console log.

Now, let's learn how to combine generators in the next lesson.

# Combining Generators

using the `yield` method to combine multiple sequences in one iterable

It is possible to combine two sequences in one iterable. All you need to do is use `yield*` to include an iterable, which will enumerate all of its values one by one.

`Yield*` delegates iteration to another generator or iterable object.

```
let countdownGenerator = function *() {
 let i = 10;
 while (i > 0) yield --i;
}

let lampGenerator = function *() {
 yield 'red';
 yield 'green';
}

let countdownThenLampGenerator = function *() {
 yield *countdownGenerator();
 yield *lampGenerator();
}

console.log([...countdownThenLampGenerator()]);
```



In the next lesson, let's learn how to pass parameters to an iterable using the `next` function.

# Passing Parameters to Iterables

using `yield*` expression to delegate to another iterable object

The `next` method of iterators can be used to pass a value that becomes the value of the previous `yield` statement.

```
let greetings = function *() {
 let name = yield 'Hi!';
 yield `Hello, ${name}!`;
}

let greetingIterator = greetings();

console.log(greetingIterator.next());
//> Object {value: "Hi!", done: false}

console.log(greetingIterator.next('Lewis'));
//> Object {value: "Hello, Lewis!", done: false}
```



The return value of a generator becomes the return value of a `yield *` expression.

```
let sumSequence = function *(num) {
 let sum = 0;
 for (let i = 1; i <= num; ++i) {
 sum += i;
 yield i;
 }
 return sum;
}

let wrapSumSequence = function *(num) {
 let sum = yield *sumSequence(num);
 yield `The sum is: ${sum}.`;
}

for (let elem of wrapSumSequence(3)) {
 console.log(elem);
}
```





Now, let's discuss some practical applications of generators.

# Practical Applications

infinite sequences, code that is evaluated lazily and asynchronous programming

You now know everything required to be able to write generator functions. This is one of the hardest topics in ES6, so you will get a chance to solve more exercises than usual.

After practicing the foundations, you will find out how to use generators in practice to:

- define infinite sequences (exercise 5),
- create code that is evaluated lazily (exercise 6).

For the sake of completeness, it is worth mentioning that generators can be used for asynchronous programming. Running asynchronous code is outside the scope of this lesson. We will use promises for handling asynchronous code.

# Exercise on Iterators and Generators

We will play around with Iterators and Generators to get a deeper understanding of how they work.

These exercises help you explore how iterators and generators work. You will get a chance to play around with iterators and generators, which will result in a higher depth of learning experience for you than reading about the edge cases.

You can also find out if you already know enough to command these edge cases without learning more about iterators and generators.

## Exercise 1:

What happens if we use a string iterator instead of an iterable object in a `for-of` loop?

```
let message = 'ok';
let messageIterator = message[Symbol.iterator]();

messageIterator.next();

for (let item of messageIterator) {
 console.log(item);
}
```

## Solution

```
let message = 'ok';
let messageIterator = message[Symbol.iterator]();

messageIterator.next();

for (let item of messageIterator) {
 console.log(item);
}
```



Similarly to generators, in case of strings, arrays, DOM elements, sets, and maps, an iterator object is also an iterable. Therefore, in the `for-of` loop, the remaining k letter is printed out.

## Exercise 2:

Create a countdown iterator that counts from 9 to 1. Use generator functions!

```
let getCountdownIterator = // Your code comes here

console.log([...getCountdownIterator()]);
> [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

```
let getCountdownIterator = function *() {
 //Write your Code here
}
```



## Exercise 3:

Make the following object iterable:

```
let todoList = {
 todoItems: [],
 addItem(description) {
 this.todoItems.push({ description, done: false });
 return this;
 },
 crossOutItem(index) {
 if (index < this.todoItems.length) {
 this.todoItems[index].done = true;
 }
 return this;
 }
};

todoList.addItem('task 1').addItem('task 2').crossOutItem(0);

let iterableTodoList = /// ???;

for (let item of iterableTodoList) {
 console.log(item);
}

// Without your code, you get the following error:
// TypeError: todoList is not iterable
```

```
// TypeError: iterableTodoList is not iterable
```



We could use well known symbols to make `todoList` iterable. We can add a `*` [\[Symbol.iterator\]](#) generator function that yields the elements of the array. This will make the `todoList` object iterable, yielding the elements of `todoItems` one by one.

```
let todoList = {
 todoItems: [],
 *[Symbol.iterator]() {
 yield* this.todoItems;
 }
 addItem(description) {
 this.todoItems.push({ description, done: false });
 return this;
 },
 crossOutItem(index) {
 if (index < this.todoItems.length) {
 this.todoItems[index].done = true;
 }
 return this;
 }
};
let iterableTodoList = todoList;
```

This solution is not compatible with the code of the exercise. Furthermore, we prefer staying away from well-known symbols. This solution reads a bit like a hack.

So make the code more semantic by following the patterns of the exercise. The solution given for this exercise looks cleaner even if you have to type more characters.

## Exercise 4:

Determine the values logged to the console without running the code. Instead of just writing down the values, formulate your thought process and explain to yourself how the code runs line by line. You should execute each statement, loop, and declaration one by one.

```
let errorDemo = function *() {
 yield 1;
 yeild 'Error yielding the next result';
 yield 2;

}

let it = errorDemo();

// Execute one statement at a time to avoid
// skipping lines after the first thrown error.

console.log(it.next());

console.log(it.next());

console.log([...errorDemo()]);

for (let element of errorDemo()) {
 console.log(element);
}
```

## Solution

```
let errorDemo = function *() {
 yield 1;
 yield 'Error yielding the next result';
 yield 2;
}

let it = errorDemo();

// Execute one statement at a time to avoid
// skipping lines after the first thrown error.

console.log(it.next());

console.log(it.next());

console.log([...errorDemo()]);

for (let element of errorDemo()) {
 console.log(element);
}
```



We created three iterables in total: it, one in the statement in the spread operator, and one in the for-of loop.

In the example with the next calls, the second call results in a thrown error. In the spread operator example, the expression cannot be evaluated, because an error is thrown.

In the for-of example, the first element is printed out, then the error stopped the execution of the loop.

## Exercise 5:

Create an infinite sequence that generates the next value of the Fibonacci sequence.

The Fibonacci sequence is defined as follows:

- `fib( 0 ) = 0`
- `fib( 1 ) = 1`
- `for n > 1, fib( n ) = fib( n - 1 ) + fib( n - 2 )`

```
function *fibonacci() {
 //Write your code here
}
```



Note that you only want to get the `next()` element of an infinite sequence.

Executing `[...fibonacci()]` will skyrocket your CPU usage, speed up your CPU fan, and then crash your browser.

## Exercise 6:

Create a lazy `filter` generator function. Filter the elements of the Fibonacci sequence by keeping the even values only.

```
function *filter(iterable, filterFunction) {
 // insert code here
}
```

```
//Assume Fibonacci function already defined

function *filter(iterable, filterFunction) {
 // Write your code here
}
```



# Promise States

pending, fulfilled and rejected states of Promises

Promises represent the eventual result of an asynchronous operation. They give us a way to handle asynchronous processing in a more synchronous fashion. A promise represents a value we can handle in the future, with the following guarantees:

- Promises are immutable.
- Promises are either kept or broken.
- When a promise is kept, we are guaranteed to receive a value.
- When a promise is broken, we are guaranteed to receive the reason why the promise cannot be fulfilled.

## Promise States

- `pending`: may transition to `fulfilled` or `rejected`
- `fulfilled` (kept promise): must have a value
- `rejected` (broken promise): must have a reason for rejecting the promise

Now, let's learn how to create promises in ES6.

# Creating Promises in ES6

creating promises and resolving/rejecting them using the 'resolve' and 'reject' keywords

Consider the following code:

```
let promise1 = new Promise(function(resolve, reject) {
 // call resolve(value) to resolve a promise
 // call reject(reason) to reject a promise
});

// Create a resolved promise
let promise2 = Promise.resolve(5);
console.log(promise1)
console.log(promise2)
```



When instantiating a promise, the handler function decides whether to resolve or reject the promise. When you call `resolve`, the promise moves to Fulfilled state. When you call `reject`, the promise moves to the Rejected state.

`Promise.resolve(value)` creates a promise that's already resolved.

Now, let's talk about promises that have either been fulfilled or rejected.

# Handling the Fulfilled or Rejected States

'then', 'onFulfilled' and 'onRejected' methods

Promises can be passed around as values, as function arguments, and as return values. Values and reasons for rejection can be handled by handlers inside the `then` method of the promise.

```
promise.then(onFulfilled, onRejected);
```



- `then` may be called more than once to register multiple callbacks. The callbacks have a fixed order of execution.
- `then` is chainable, it returns a promise.
- if any of the arguments are not functions, they have to be ignored.
- `onFulfilled` is called once with the argument of the fulfilled value if it exists.
- `onRejected` is called once with the argument of the reason why the promise was rejected.

## Examples:

```
let promisePaymentAmount = Promise.resolve(50);

promisePaymentAmount
 .then(amount => {
 amount *= 1.25;
 console.log('amount: '+amount)
 console.log('amount * 1.25: ', amount);
 return amount;
}).then(amount => {
 console.log('amount: ', amount);
 return amount;
});
```



Notice the return value of the callback function of the first `then` call. This

Notice the return value of the callback function of the first `then` call. This value is passed as `amount` in the second `then` clause.

```
let promiseIntro = new Promise(function(resolve, reject) {
 setTimeout(() => reject('Error demo'), 2000);
});

promiseIntro.then(null, error => console.log(error));
```



Instead of

```
promise.then(null, errorHandler);
```

You can also write:

```
promise.catch(errorHandler);
```

to make error handling more semantic.

The best practice is to always use `catch` for handling errors, and place it at the end of the promise handler chain. Reason: `catch` also catches errors thrown inside the resolved handlers. Example:

```
var p = Promise.resolve(5);

p.then((value) => console.log('Value:', value))
 .then(() => { throw new Error('Error in second handler') })
 .catch((error) => console.log(error.toString()));
```



As `p` is resolved, the first handler logs its value, and the second handler throws an error. The error is caught by the `catch` method, displaying the error message.

It is also possible to handle multiple promises at the same time. We will study this in detail in the next lesson.



# Handling Multiple Promises

using the `promise.all()` function to handle multiple promises

`Promise.all()` takes an iterable object of promises. In this section, we will use arrays. Once all of them are fulfilled, it returns an array of fulfilled values. If any of the promises in the array fails, `Promise.all()` also fails. Study the following code:

```
var loan1 = new Promise((resolve, reject) => {
 setTimeout(() => resolve(110) , 1000);
});
var loan2 = new Promise((resolve, reject) => {
 setTimeout(() => resolve(120) , 2000);
});
var loan3 = new Promise((resolve, reject) => {
 reject('Bankrupt');
});

Promise.all([loan1, loan2, loan3]).then(value => {
 console.log(value);
}, reason => {
 console.log(reason);
});
```



Now, let's solve some exercises.

# Exercise on ES6 Promises

In the following exercises, we will practice promises by handling success and failure in data retrieval.

## Exercise 1:

List the names of the users retrieved via [this endpoint using ajax](#).

Make sure you don't use jQuery, and you use promises to handle success and failure. Place the names in the `div` below, having a class `.js-names`, and separate them with a simple comma. Make sure you do not use jQuery to access or modify the content of the below div.

```
<div style="border:1px black solid;"
 class="js-names"></div>
```

### Solution:

If you want to test the solution, you can simply prepend the test div to the beginning of the body:

```
document.body.innerHTML = `
<div style="border:1px black solid;"
 class="js-names"></div>` +
document.body.innerHTML;
```

Notice the template literal that enables us to add new lines without closing the literal.

I assume that at least half of my readers came up with a solution that uses jQuery.

jQuery lowered the entry barriers to frontend development, and we can be grateful for it. However, jQuery has done its job, and we don't necessarily need it now that ES6 is expressive enough.

This solution will work without jQuery.

Feel free to study the documentation if you would like to learn more about the XMLHttpRequest or ready states:

- XMLHttpRequest.send<sup>14</sup>
- XMLHttpRequest.onreadystatechange<sup>15</sup>
- XMLHttpRequest.readyState<sup>16</sup>

Let's get the response first, and log it to the screen with an empty parse function.

```
let parse = function(response) {
 console.log(response);
}

let errorHandler = function() {
 console.log('error');
}

new Promise(function(resolve, reject) {
 let request = new XMLHttpRequest();
 request.onreadystatechange = function() {
 if (this.status === 200 && this.readyState === 4) {
 resolve(this.response);
 }
 }
 request.open('GET',
 'http://jsonplaceholder.typicode.com/users'
);
 request.send();
}).then(parse).catch(errorHandler);
```

Getting the response works as follows:

- we create a new promise
- inside the promise, we create an XMLHttpRequest
- we can parse the response once the XMLHttpRequest transitions to readyState 4, and the status becomes 200
- once the state change handler is registered, we open and send the request
- once all conditions are fulfilled, we resolve the promise

Note that there is no error handling in the code. In real life systems, you should create a catch after the then and handle errors

should create a `catch` after the `then`, and handle errors.

Also note that in real life code, it makes sense to abstract your API handlers. Let's continue with the proper `parse` method:

```
let parse = function(response) {
 let element = document.querySelector('.js-names');
 element.innerHTML =
 JSON.parse(response)
 .map(element => element.name)
 .join(',');
}
```

Our last task is to handle errors in the promise. We will test this scenario with a wrong API URL:

```
new Promise(function(resolve, reject) {
 let request = new XMLHttpRequest();
 request.onreadystatechange = function() {
 if (this.status === 200 && this.readyState === 4) {
 resolve(this.response);
 }
 }
 request.onerror = function() {
 reject(new Error(this.statusText));
 }
 request.open(
 'GET',
 'http://erroneousurl.com/users'
);
 request.send();
}).then(parse).catch(errorHandler);
```

Defining an `onerror` handler does the task. Reject the promise to transition to the `catch` clause.

Output

JavaScript

HTML

CSS (SCSS)

Hello, Hello



## Exercise 2:

Extend the exercise such that you disable the below text field and button while fetching takes place. Regardless of whether fetching is done or fetching fails, enable the text field and the button. The input field and the text field can be referenced using the `js-textfield` and `js-button` classes respectively.

```
<input type="text" class="js-textfield" />
<button class="js-button">
 Enable the Textfield
</button>
```

## Solution:

Let's add the elements to the DOM first:

```
document.body.innerHTML = `
<input type="text" class="js-textfield" />
<button class="js-button">
 Enable the Textfield
</button>` +
document.body.innerHTML;
```

We will now define two functions: one to enable, and one to disable the fields.

```
let enableFields = function() {
 document.querySelector('.js-textfield')
 .removeAttribute('disabled');
 document.querySelector('.js-button')
 .removeAttribute('disabled');
}

let disableFields = function() {
```

```

document.querySelector('.js-textfield')
 .setAttribute('disabled', true);

document.querySelector('.js-button')
 .setAttribute('disabled', true);
}

```

Our first plan is to call the `disableFields` function in the `then` clause, and call the `enableFields` function at the end of the `then` callback and the `catch` callback.

```

let parse = function(response) {
 let element = document.querySelector('.js-names');

 element.innerHTML =
 JSON.parse(response)
 .map(element => element.name)
 .join(',');
 enableFields();
}

let errorHandler = function() {
 console.log('error');
 enableFields();
}

new Promise(function(resolve, reject) {
 disableFields();
 let request = new XMLHttpRequest();
 request.onreadystatechange = function() {
 if (this.status === 200 && this.readyState === 4) {
 resolve(this.response);
 }
 }
 request.onerror = function() {
 reject(new Error(this.statusText));
 }
 request.open(
 'GET',
 'http://erroneousurl.com/users'
);
 request.send();
}).then(parse)
 .catch(errorHandler);

```

Technically, it is possible to add a second `then` callback after the `catch` to call `enableFields` instead of repeating it in the `parse` and `errorHandler` methods.

If you chose to implement your code this way, make sure you catch all errors inside your cleanup method.

# The Task

Write a simple ES6 module, and run the code using Webpack.

In order to save you time, I uploaded each step of the tutorial on my GitHub account. [Fork it from here](#).

## Code Modularity

Code modularity is essential for writing maintainable code. ES6 modules offer solutions for encapsulation, packaging and information hiding. Implementing the [module pattern](#) enriches your application.

## Module for financial account

We will create an ES6 module representing a financial account. We will store transactions belonging to the account in an array. Each transaction has the following attributes:

- amount : Integer in cents. Floating points are unreliable. Depositing 0.1 then 0.2 won't give you 0.3 as a result.
- date : String in `"yyyy-MM-dd"` format. We will simplify things. If you feel like experimenting with the Date object or you just want to laugh a bit, [read my post on Javascript dates!](#)

We will import this ES6 module in another file, fill it with some data for demo purposes, and populate an ordered list with the three largest transactions.

Now, let's start working on this task.

## Step 1: index.html

Create a folder for this experiment and place an `index.html` file in it with the following contents:

```
<!doctype html>
<html>
 <head>
 <title>Deposits and Withdrawals</title>
 </head>
 <body>
 <h1>Top 3 Transactions</h1>
 <ul class="js-top-transactions">

 <script src="myaccount.dist.js"></script>
 </body>
</html>
```

The file `myaccount.dist.js` does not exist yet. We will use [Webpack](#) to create it.

## Step 2: Initialize your Application and Configure Webpack

To continue with the setup, you have to have [nodejs](#) installed on your machine. If you haven't installed node, visit [nodejs.org](#), and follow the installation instructions there.

Execute

```
npm init
```



inside the folder of your application. Provide the necessary details about your application by answering the questions in the terminal. Once you are done answering or skipping these questions, a [package.json](#) file is created in your folder.

Webpack will be our default tool for bundling our packages for distribution, and managing dependencies. Install Webpack as a local dependency:

```
npm install webpack --save-dev
```



Npm package installations can be global or local. In case of global installations, you may have to run the command as admin, using [sudo](#) in Linux and Mac, or you may have to run your command line as an administrator in Windows.

Note that it is rarely a good idea to install global packages, as you have no control over the runtime environment of other people running your code. If you work in a team, I suggest installing every npm dependency locally. If you prefer a global installation for experimenting, feel free to do so.

Webpack does not support the ES6 syntax by default. We will use a module loader called [babel-loader](#). You can install it using [npm](#):

loader called `babel-loader`. You can install it using `npm`.

```
npm install babel-loader babel-core --save-dev
npm install babel-preset-es2015 babel-preset-stage-2 --save-dev
```

We have installed the following dev dependencies:

- `babel-loader`: babel module loader,
- `babel-core`: the core transpiler used for transpiling JavaScript,
- `babel-preset-es2015`: ES6/ES2015 support,
- `babel-preset-stage-2`: babel preset for some extra features such as destructuring.

Now that Webpack is available, let's create a configuration file

`webpack.config.js`.

```
module.exports = {
 entry : './src/main.js',
 output : {
 path : __dirname,
 filename : 'myaccount.dist.js'
 },
 module : {
 loaders: [{
 test : /\.js$/,
 loader : 'babel-loader',
 exclude: /node_modules/,
 query: {
 presets: [
 'es2015',
 'stage-2'
]
 }
 }]
 }
};
```

Note that in Webpack 4, you have to write `rules` instead of `loaders` in the `module` property:

```
module.exports = {
 entry : './src/main.js',
 output : {
 path : __dirname,
 filename : 'myaccount.dist.js'
 },
 module : {
```

```
rules: [{
 test : /\.js$/,
 loader : 'babel-loader',
 exclude: /node_modules/,
 query: {
 presets: [
 'es2015',
 'stage-2'
]
 }
},
{
 test: /\.js$/,
 exclude: /node_modules/,
 loader: 'babel-loader',
 query: {
 presets: [
 'es2015',
 'stage-2'
]
 }
}
];
```

Our entry point `src/main.js` will be created shortly. All dependencies specified in this file will be copied to `myaccount.dist.js` recursively. Babel-loader is also invoked, transpiling all Javascript files it finds.

## Step 3: Account Module

To process transactions, we need an `Account` class. We will use the class syntax of ES6 to write the code. A short explanation will follow the example. Before reading the explanation, try to figure out what the code does.

```
import { sortBy, first } from 'underscore';

class Account {
 constructor() {
 this.transactions = [];
 }
 getTopTransactions() {
 var getSortKey = transaction =>
 -Math.abs(transaction.amount);
 var sortedTransactions = sortBy(
 this.transactions,
 getSortKey
);
 return first(sortedTransactions, 3);
 }
 deposit(amount, date) {
 this.transactions.push({
 amount : amount,
 date : date
 });
 }
 withdraw(amount, date) {
 this.transactions.push({
 amount : -amount,
 date : date
 });
 }
};

export default Account;
```



Save the above code in `src/Account.js`. We will also need UnderscoreJs as it's defined as a dependency. Grab it using

```
npm install underscore --save
```

Underscore is a regular `dependency`, hence the `--save` flag. All other packages have been `devDependencies`, and they were added to the `package.json` using `--save-dev`. Development dependencies, such as transpilers or automated testing frameworks, are only used during development.

The code works in the following way:

- We import the `sortBy` and `first` functions from the Underscore functional programming utility belt
- When creating an account with the `new` operator, we initialize the transactions to `[]`
- To demonstrate how to import Underscore functions, we derive the top 3 transactions. After sorting the transactions based on descending absolute transaction amount, we take the first three elements from the list.
- Depositing and withdrawing are both straightforward: we push a new transaction object to the `transactions` array, setting its amount and date

## Step 4: Create the Entry Point

The last missing file is `src/main.js`. We will import the `Account` class, create an account, add a couple of transactions and finish this step by displaying the top 3 transactions. We [don't need jQuery](#).

```
import Account from './Account';

let myAccount = new Account();
let format = transaction =>
`${transaction.amount} (${transaction.date})`;
let list = document.querySelector('.js-top-transactions');

myAccount.deposit(200000, '2017-01-01');
myAccount.deposit(500000, '2017-02-01');
myAccount.deposit(100000, '2017-03-01');
myAccount.withdraw(300000, '2017-04-01');

// Top 3: 500000, -300000, 200000
list.innerHTML = myAccount.getTopTransactions().map(format).join('');
```

## Step 5: Compile and Run the Application

As we installed `webpack` locally, we have to configure a script to execute it. In your `package.json` file, the following key was generated by default:

```
"scripts": {
 "test": "echo \\\"Error: no test specified\\\" && exit 1"
},
```

Replace the `test` script with a `webpack` script:

```
"scripts": {
 "webpack": "webpack"
},
```

Enter `npm run webpack` in your terminal to run all the tasks specified in Step 2. The script should run without errors. Alternatively, if you would like to continue developing this demo and you don't want to bother executing Webpack after every change, configure a script in your `package.json` that will run `webpack` with a `--watch` flag:

```
"scripts": {
 "webpack": "webpack",
 "webpack:watch": "webpack --watch"
},
```

Execute the `webpack:watch` script:

```
npm run webpack:watch
```

If you make a change in one of your JavaScript files, the distribution file is automatically recompiled. Press `Ctrl + C` to exit the `webpack:watch` task whenever you finish development.

Once `myaccount.dist.js` is created, view it in a browser window.

If you are interested in a slightly more complex setup of a React boilerplate with a development server, read [this article](#).

If you would like to add ESLint support to your application, [this article](#) will help you.

# Reflection

introduction to reflection and using the method `Reflect.apply`

**Reflection** in a programming language is the act of inspecting, dynamically calling, and modifying classes, objects, properties, and methods. In other words, reflection is the ability of the programming language to *reflect* on the structure of the code.

The ES6 Reflect API gives you a Reflect object that lets you call methods, construct objects, getting and setting prototypes, manipulating and extending properties.

Reflection is important, because it lets you write programs and frameworks that are able to handle a non-static code structure.

One use case is **automated testing**. You don't have to do expensive and complicated setup operations in order to test some methods of a class in isolation.

We can also use the Reflect API in proxies. We will learn about proxies in the next chapter.

**Reflect.apply** calls functions or methods.

```
let target = function getArea(width, height) {
 return `${ width * height }${this.units}`;
}
let thisValue = { units: 'cm' };
let args = [5, 3];

console.log('Area: '+Reflect.apply(target, thisValue, args));
```



Now let's learn how to create objects in reflection.



# Creating Objects

Reflect.construct and its behavior

We can instantiate classes with `Reflect.construct`. Consider the following code:

```
let target = class Account {
 constructor(name, email) {
 this.name = name;
 this.email = email;
 }
 get contact() {
 return `${this.name} <${this.email}>`;
 }
};
let args = [
 'Zsolt',
 'info@zsoltnagy.eu'
];
let myAccount = Reflect.construct(
 target,
 args);
console.log(myAccount.contact);
//> "Zsolt <info@zsoltnagy.eu>"
```



Arguments of `Reflect.construct`:

- `target`: the function we will construct
- `args`: array of arguments
- `newTarget`: `new.target` value (optional)

With the following example, we have full control over the value of `new.target` in the constructor:

```
let target = class Account {
 constructor(name, email) {
```



```
this.name = name;
this.email = email;
}

get contact() {
 return `${this.name} <${this.email}>`;
}
};

let args = [
 'Zsolt',
 'info@zsoltnagy.eu'
];
let newTarget = class PrivateAccount {
 get contact() {
 return 'Private';
 }
}

let myAccount = Reflect.construct(
 target,
 args,
 newTarget);

console.log(myAccount.contact);
//> "Private"

console.log(myAccount);
//> PrivateAccount {name: "Zsolt", email: "info@zsoltnagy.eu"}
```



Now, let's talk about how we can manipulate prototypes of objects using Reflect API.

# Manipulating Prototypes

getting and setting prototypes of objects using the Reflect API

We can get the prototype of an object using the Reflect API.

```
let classOfMyAccount = Reflect.getPrototypeOf(myAccount);
console.log(classOfMyAccount.prototype === myAccount.prototype);
```



As you can see from the example, this prototype is the same as the prototype of our `PrivateAccount` class.

We can also set prototypes using `Reflect.setPrototypeOf`:

```
let newProto = {
 get contact() {
 return `${this.name} - 555-1269`;
 }
}

Reflect.setPrototypeOf(myAccount, newProto);

console.log(myAccount.contact);
//> "Zsolt - 555-1269"
```



In the above example, we set the prototype of `myAccount` to `newProto`, and define a `contact` getter function. After changing the prototype, the new getter method is executed.

In the next lesson, we will discuss how we can access properties of objects and modify them using Reflection.



# Property Access and Modification

access, modify (update and delete) properties of Objects

`Reflect.has` determines if a property exists for a given target object. The call enumerates all properties, not only own properties.

```
let target = class Account {
 constructor(name, email) {
 this.name = name;
 this.email = email;
 }
 get contact() {
 return `${this.name} <${this.email}>`;
 }
};
let args = [
 'Zsolt',
 'info@zsoltnagy.eu'
];
let myAccount = Reflect.construct(
 target,
 args);
console.log(Reflect.has(myAccount, 'name'));
console.log(Reflect.has(myAccount, 'contact'));
```



`Reflect.ownKeys` returns all own properties of a target in an array.

```
console.log(Reflect.ownKeys(myAccount));
```



`Reflect.get` gets a property based on a key. As an optional parameter, the `this` context can be specified.

```
console.log(Reflect.get(myAccount, 'name'));
//> "Zsolt"

console.log(Reflect.get(myAccount, 'contact'));
//> "Zsolt - 555-1269"
```



Getting the `name` property of `myAccount` is straightforward. In the second example, getting the `contact` property requires the execution of a getter method. As we redefined this getter method using `Reflect.setPrototypeOf` not too long ago, the result becomes `"Zsolt - 555-1269"`.

If we specify the context as the third argument of `Reflect.get`, we will get another result for the `contact` property:

```
console.log(Reflect.get(
 myAccount,
 'contact',
 { name: 'Bob' }
));
//> "Bob - 555-1269"
```



We can also set a property of our target using `Reflect.set`.

```
let target = myAccount;
let property = 'age';
let newValue = 32;

Reflect.set(
 myAccount,
 property,
 newValue
);

console.log(myAccount.age);
//> 32
```



The fourth argument of `Reflect.set` is an optional context. Solve **exercise 4** to figure out how the context works in `Reflect.set`.

`Reflect.defineProperty` defines a new property. It is similar to calling `Object.defineProperty`. The difference between the two calls is that `Reflect.defineProperty` returns a boolean, while `Object.defineProperty` returns the object itself.

In the following example, we will define a writable property:

```
let target = {};
let key = 'response';
let attributes = {
 value: 200,
 writable: true,
 enumerable: true
};

Reflect.defineProperty(
 target,
 key,
 attributes
);
```



For a complete list of flags and their default values, check out [the documentation of Object.defineProperty](#). Notice that all configuration flags inside the `attributes` object have a default value of `false`. This means that without specifying the `writable` flag, `target.response` would have been a read-only property.

We have accessed, created, and updated properties. The last operation missing is the deletion of properties.

```
let response = {
 status: 200
};

console.log(Reflect.deleteProperty(response, 'status'));
//> true

console.log(response);
//> {}
```



**Exercise 5** is about `Reflect.deleteProperty` and `Reflect.defineProperty`. You will find out that in some circumstances, deleting a property is not possible.

It is possible to prevent property extensions by calling the `preventExtensions` method of the `Reflect` API:

```
let test = {
 title: 'Petri Nets',
 maxScore: 100
};

console.log(Reflect.preventExtensions(test));
//> true

test.score = 55;

console.log(test);
//> Object {title: "Petri Nets", maxScore: 100}
```



As you can see, the `score` field is not added to the `test` object, as `test` was locked down by `Reflect.preventExtensions`.

`Reflect.isExtensible` tests whether an object is extensible:

```
console.log(Reflect.isExtensible(test));
//> false

console.log(Reflect.isExtensible({}));
//> true
```



Now, let's solve some exercises before learning new concepts.

# Exercise on the Reflect API

In this exercise, we will use various methods of the Reflect API to create and alter objects.

## Exercise 1:

Create a `Movie` class, and initialize the `title` (`String`) and `movieLength` (`Number`) properties in the constructor. Create a `toString` method that prints the movie out using the following format:

```
 ${this.title} (${this.movieLength} minutes)
```

```
class Movie {
 constructor(title, movieLength) {
 //Write your code here
 }

 toString() {
 return "Your Answer";
 }
};
```



## Exercise 2:

Use `Reflect.apply` to call the `toString` method of the `Movie` class with the following redefined properties: `"Rush"`, `123`.

```
//Assume class Movie already defined as in the preceding solution

const rush123 = Reflect.apply(
 //Write your code here

);

//Without your code, this is going to throw
//TypeError: Function.prototype.apply was called on undefined
```



## Exercise 3:

Use the `Reflect` API to access a reference to the `Building` class assuming that initially you only have access to the `myBuilding` object. Then extend the prototype of the `Building` class by adding a `toString` method.

```
let myBuilding = (function() {
 class Building {
 constructor(address) {
 this.address = address;
 }
 }

 class ResidentialBuilding extends Building {
 constructor(address, capacity) {
 super(address);
 this.capacity = capacity;
 }
 }

 let myBuilding = new ResidentialBuilding(
 'Java Street 3',
 16
);

 return myBuilding;
})();

let toString = function() {
 return `Address: ${this.address}`;
};

//Add your code here
```



## Exercise 4:

Suppose a `Person` class is given.

```
class Person {
 constructor(name) {
 this.name = name;
 }
 set name(name) {
 let [first, last] = name.split(' ');
 this.first = first;
 this.last = last;
 }
}
```



Let's create a `person` object and a `newContext` variable.

```
let person = new Person('Julius Caesar');
let newContext = { name: 'Marcus Aurelius' };
```



If we query the contents of a `person`, we can see how the setter transformed the name into a `first` and a `last` field.

```
person
//> Person {first: "Julius", last: "Caesar"}
```



Let's call a `Reflect.set` operation, setting the `name` field of our `person` object, and let's add the new context in the fourth variable.

```
Reflect.set(
 person,
 'name',
 'Alexander Severus',
 newContext
);
```



Determine the following values without executing the code:

- the return value of the above `Reflect.set` call
- `person.first` and `person.last`
- `person.name`
- `newContext.first` and `newContext.last`
- `newContext.name`

```
class Person {
 constructor(name) {
 this.name = name;
 }
 set name(name) {
 let [first, last] = name.split(' ');
 this.first = first;
 this.last = last;
 }
}

let person = new Person('Julius Caesar');
let newContext = { name: 'Marcus Aurelius' };
```



```
const returnVal = Reflect.set(
 person,
 'name',
 'Alexander Severus',
 newContext
);

console.log(returnVal);
console.log(person.first + " " + person.last);
console.log(person.name);
console.log(newContext.first + " " + newContext.last);
console.log(newContext.name);
```



### 💡 Pro Tip

`propertyDescriptor.configurable` is the value you should look for when judging whether an object property is configurable

## Exercise 5:

```
let target = {};
let key = 'response';
let attributes = {
 value: 200,
 writable: true,
 enumerable: true
};

Reflect.defineProperty(
 target,
 key,
 attributes
);
```



Let's try to delete `target.response`.

```
console.log(Reflect.deleteProperty(target, key));
//> false

console.log(target)
//> Object {response: 200}
```





The return value of the `deleteProperty` call indicates that the deletion is unsuccessful. Why? How can we modify the code such that the same `Reflect.deleteProperty` call returns `true`, and `target.response` is deleted?

```
let target = {};
let key = 'response';
let attributes = {
 value: 200,
 writable: true,
 enumerable: true
};

Reflect.defineProperty(
 target,
 key,
 attributes
);
```



# Introduction

introduction to proxy and some traps while handling the access of the target

A **proxy** is an object that wraps an object or a function and monitors access to the wrapped item, a.k.a. the **target**. We use proxies for the intention of blocking direct access to the target function or object.

The proxy object has some *traps* which handle access to the target. The traps are the same as the methods used in the Reflect API. Therefore, this section assumes that you are familiar with the Reflect API, as you will use the same calls. The following traps are available:

- `apply`
- `construct`
- `defineProperty`
- `deleteProperty`
- `get`
- `getOwnPropertyDescriptor`
- `getPrototypeOf`
- `has`
- `isExtensible`
- `ownKeys`
- `preventExtensions`
- `set`
- `setPrototypeOf`

Once a trap of a proxy object is executed, we can run any code, even without accessing the target. The proxy decides if it wants to provide you access to the target, or handle the request on its own.

The available traps allow you to monitor almost any use case. Some behavior cannot be trapped though

cannot be trapped thought.

We cannot tell if our object is compared to another value, or wrapped by another object. We cannot tell if our object is an operand of an operator. In reality, the absence of these use cases is hardly problematic.

Now, let's learn how to define these proxies.

# Defining Proxies

solve fundamental operations (like property lookup, assignment, enumeration, function invocation, etc.) using proxies

We can create a proxy in the following way:

```
let proxy = new Proxy(target, trapObject);
```



The first argument is `target` representing the proxied constructor function, class, or object.

The second argument is an object containing traps that are executed once the proxy is used in a specific way.

Let's put our knowledge into practice by proxying the following target:

```
class Student {
 constructor(first, last, scores) {
 this.firstName = first;
 this.lastName = last;
 this.testScores = scores;
 }
 get average() {
 let average = this.testScores.reduce(
 (a,b) => a + b,
 0
) / this.testScores.length;
 return average;
 }
}

let john = new Student('John', 'Dwan', [60, 80, 80]);
console.log(john);
```



We will now define a proxy on the target object `john`:

```
let johnProxy = new Proxy(john, {
 get: function(target, key, context) {
 console.log(`john[${key}] was accessed.`);
 }
});
```



```
// return undefined;
}
});
```



The `get` method of the proxy is executed whenever we try to access a property.

```
console.log(johnProxy.getGrade);
//> john[getGrade] was accessed.
//> undefined

console.log(johnProxy.testScores);
//> john[testScores] was accessed.
//> undefined
```



The above defined `get` trap chose to ignore all the field values of the target object to return `undefined` instead.

Let's define a slightly more useful proxy that allows access to the `average` getter function, but returns `undefined` for anything else:

```
let johnMethodProxy = new Proxy(john, {
 get: function(target, key, context) {
 if (key === 'average') {
 return target.average;
 }
 }
});

console.log(johnMethodProxy.firstName);
//undefined
console.log(johnMethodProxy.average);
//73.33333333333333
```



We can conclude that proxies can be used to

- define access modifiers,
- provide validation through the public interface of an object.

The target of proxies can also be functions.

```
let factorial = n =>
 n <= 1 ? n : n * factorial(n - 1);
console.log(factorial(6));
```



For instance, you might wonder how many times the `factorial` function was called in the expression `factorial(5)`. This is a natural question to ask to formulate an automated test.

Let's define a proxy to find the answer out.

```
factorial = new Proxy(factorial, {
 apply: function(target, thisValue, args) {
 console.log('I am called with', args);
 return target(...args);
 }
});

console.log('Factorial: '+factorial(5));
```



We added a proxy that traps all the calls of the `factorial` method, including the recursive calls. We equate the proxy reference to the `factorial` reference so that all recursive function calls are proxied. The reference to the original factorial function is accessible via the `target` argument inside the proxy.

We will now make two small modifications to the code.

First of all, we will replace

```
return target(...args);
```

with

```
return Reflect.apply(
 target,
 thisValue,
 args);
```

We can use the Reflect API inside proxies. There is no real reason for the replacement other than demonstrating the usage of the Reflect API.

Second, instead of logging, we will now count the number of function calls in the `numOfCalls` variable.

If you are executing this code in your developer tools, make sure you reload your page, because the existing code may interact with this

```
let factorial = n =>
 n <= 1 ? n : n * factorial(n - 1);

let numOfCalls = 0;
factorial = new Proxy(factorial, {
 apply: function(target, thisValue, args) {
 numOfCalls += 1;
 return Reflect.apply(
 target,
 thisValue,
 args
);
 }
});

console.log(factorial(5) && numOfCalls);
//> 5
```



In the next lesson, we will learn about controlling proxies and revoking them.

# Revocable Proxies

Proxy.revocable and its behavior

We can also create revocable proxies using `Proxy.revocable`. This is useful when we pass proxies to other objects, but we want to keep a centralized control of when we want to shut down our proxy.

```
let payload = {
 website: 'zsoltznagy.eu',
 article: 'Proxies in Practice',
 viewCount: 15496
}

let revocable = Proxy.revocable(payload, {
 get: function(...args) {
 console.log('Proxy');
 return Reflect.get(...args);
 }
});

let proxy = revocable.proxy;

console.log(proxy.website);
//> Proxy
//> "zsoltznagy.eu"

revocable.revoke();

proxy.website;
//> Uncaught TypeError: Cannot perform 'get' on a proxy that
//> has been revoked
//> at <anonymous>:3:6
```



Once we revoke the proxy, it throws an error when we try using it.

As both the `revoke` method and `proxy` are accessible inside `revocable`, we can use the ES6 shorthand notation for objects to shorten our code:

```
// ...
```



```
// Create a revocable proxy
let {proxy, revoke} = Proxy.revocable(payload, {
 get: function(...args) {
 console.log('Proxy');
 return Reflect.get(...args);
 }
});

// Revoke the proxy
revoke();
```



Now, let's talk about the various use cases of proxies in the next lesson.

# Use Cases

automated testing, building fake servers for development, making memorization proxies, establishing a logging layer, controlling client-side validation and handling access rights

When studying `Proxy.revocable`, we concluded that we could centralize control of data access via revocable proxies. All we need to do is pass the revocable proxy object to other consumer objects, and revoke their access once we want to make data inaccessible.

In **Exercise 1**, we will build a proxy that counts the number of times a method was called. Proxies can be used in automated testing. If you read the [SinonJs documentation](#), you can see multiple use cases for proxies.

SinonJs spies, stubs, etc. were initially implemented and used in ES5. The reference is for the purpose that proxies are used in automated testing.

You can also build a fake server for development, which intercepts some API calls and answers them using static JSON files. For the API calls that are not being developed, the proxy simply passes the request through, and lets the client interact with the real server.

In **Exercise 2**, we will build a memorization proxy on the famous Fibonacci problem, and we will check out how many function calls we save with the lookup. If you extend this idea, you can also use a proxy to memorize the response of expensive API calls, and serve these calls without recalculating the results.

**Exercise 3** highlights three use cases.

First, we may need to deal with JSON data that has a non-restricted structure. If we cannot make assumptions on the structure of a document, proxies come in handy.

Second, we can establish a logging layer including logs, warning messages, and errors using proxies.

Third, if we have control over when we log errors, we can also use proxies to control client-side validation.

If we combine the ideas of **Exercise 2** and **Exercise 3**, we can use proxies to restrict access to endpoints based on user credentials. This method does not save the server side implementation, but proxies give you a convenient way to handle access rights.

Now, let's solve some exercises.

# Exercise on Proxies

Using your knowledge of proxies object, you must use them to:

- i) log the number of times a function is accessed
- ii) create a revocable discount object

## Exercise 1:

Suppose the following `fibonacci` implementation is given:

```
fibonacci = n =>
 n <= 1 ? n :
 fibonacci(n - 1) + fibonacci(n - 2);
```

Determine how many times the `fibonacci` function is called when evaluating `fibonacci( 12 )`.

Determine how many times `fibonacci` is called with the argument `2` when evaluating `fibonacci( 12 )`.

```
let fibonacci = n =>
 n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2); //the fibonacci function

let fibCalls = 0; // total calls to the fibonacci function
let fibCallsWith2 = 0; // calls to the fibonacci function with 2 as an argument

//Write your code here

// Do not edit code below this line
console.log('fibCalls:', fibCalls);
console.log('fibCallsWith2:', fibCallsWith2);
```



## Explanation:

The solution is not that hard. As we learned earlier, `fibonacci = new Proxy( fibonacci, {` uses the same reference name as our fibonacci function so that

the proxy can be applied on all recursive calls as well.

Within the proxy, we increment `fibCalls` and `fibCallsWith2` whenever needed.

```
return Reflect.apply(target, thisValue, args); could also be written as
return target(...args);
```

## Exercise 2:

Create a proxy that builds a lookup table of `fibonacci` calls, memorizing the previously computed values. For any `n`, if the value of `fibonacci( n )` had been computed before, use the lookup table, and return the corresponding value instead of performing recursive calls.

Use this proxy, and determine how many times the `fibonacci` function was called

- altogether
- with the argument `2`

while evaluating `fibonacci( 12 )`.

```
let fibonacci = n =>
 n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2); // the fibonacci function

let fibCalls = 0; // total calls to the fibonacci function
let fibCallsWith2 = 0; // total calls to the fibonacci function with 2 as an argument

//Write your code here

//Do not edit the code below this line

console.log('Result:', fibonacci(12));
console.log('fibCalls', fibCalls);
console.log('fibCallsWith2', fibCallsWith2);
```



## Explanation:

This is very similar to what we did in exercise 1. The only difference is that we

now use a lookup table to see if we already have the Fibonacci value for a certain number. If it doesn't exist, we add it to the table.

There are several ways to implement your table I have used `Map()` because of its simplicity. The rest is simple code where we increment our `fibCalls` and `fibCallsWith2` variables.

Then comes the lookup functionality, where we check if the key from `args[0]` already exists in the table.

## Exercise 3:

Given the object

```
let course = {
 name: 'ES6 in Practice',
 _price: 99,
 currency: '€',
 get price() {
 return `${this._price}${this.currency}`;
 }
};
```

Define a revocable proxy that gives you a 90% discount on the price for the duration of 5 minutes (or 300,000 milliseconds). Revoke the discount after 5 minutes.

The original `course` object should always provide access to the original price.

 Exercise 3

 Solution

```
let course = {
 name: 'ES6 in Practice',
 _price: 99,
 currency: '€',
 get price() {
 return `${this._price}${this.currency}`;
 }
};

//Write your Code here
```



## Explanation:

We create a revocable proxy `revocableDiscount`. In the `get` function, we check if our key to the target is ‘price’:

```
if (key == 'price')
```

We then set the new discounted price and return it.

`return target[ key ]` makes sure that other properties of the `course` object are returned as they are.

Now all that’s left is to set the delay after which the proxy is to be revoked.

This is done using `setTimeout`.

# Math Extensions

in-built mathematical functions in ES6

The `Number` and the `Math` objects have been extended with many useful methods. There are many new mathematical functions in ES6.

In my opinion, the most useful and semantic addition is `Math.trunc`, which gives you the truncated integer value of a number.

```
console.log(Math.trunc(1.99));
console.log(Math.trunc(-1.99));
```



Another extension is `Math.sign` which gives you the sign of a number. Even though the `>` and `<` operators can still form proper boolean expressions, when using calculations, sometimes the `sign` function is still useful.

```
console.log(Math.sign(5));
//> 1

console.log(Math.sign('5'));
//> 1

console.log(Math.sign(0));
//> 0

console.log(Math.sign(-0));
//> -0

console.log(Math.sign(''));
//> 0

console.log(Math.sign('twenty'));
//> NaN

console.log(Math.sign(NaN));
//> NaN
```



Other new `Math` functions include:

| Function                                                                                                                                                                               | Usage                                            |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------|
| <code>Math.cbrt</code>                                                                                                                                                                 | cube root                                        |
| <code>Math.clz32</code>                                                                                                                                                                | count leading zeros of 32bit integers            |
| <code>Math.exp(num)</code>                                                                                                                                                             | Returns $e^{\text{num}}$ ( $e=2.718\dots$ )      |
| <code>Math.expm1(num)</code>                                                                                                                                                           | <code>Math.exp(num) - 1</code>                   |
| <code>Math.fround</code>                                                                                                                                                               | round to the nearest 32 bit floating point value |
| <code>Math.sinh</code> , <code>Math.cosh</code> , <code>Math.tanh</code> ,<br><code>Math.asinh</code> , <code>Math.acosh</code> ,<br><code>Math.atanh</code> , <code>Math.hypot</code> | Hyperbolic functions                             |
| <code>Math.imul</code>                                                                                                                                                                 | 32 bit integer multiplication                    |
| <code>Math.log2</code>                                                                                                                                                                 | base 2 logarithm                                 |
| <code>Math.log10</code>                                                                                                                                                                | base 10 logarithm                                |
| <code>Math.log(num)</code>                                                                                                                                                             | natural logarithm (ln) of num                    |
| <code>Math.log1p(num)</code>                                                                                                                                                           | <code>Math.log(num + 1)</code>                   |

You can read about these in more detail [here](#).

In the next lesson, we will talk about the Number extensions in ES6.



# Number Extensions

in-built methods in ES6 to play with numbers

Finally, ES6 gives you a built-in `isInteger` check. Earlier, we had to implement different tricks to determine if a number is an integer. These tricks were not too semantic. In ES6, all we need to do is call

```
console.log(Number.isInteger(5));
```



**Safe integers** are integers that can be represented in JavaScript without loss of precision.

```
let maxVal = Number.MAX_SAFE_INTEGER;
let minVal = Number.MIN_SAFE_INTEGER;

console.log(Number.isSafeInteger(maxVal));
//> true

console.log(Number.isSafeInteger(maxVal + 1));
//> false
```



`Number.EPSILON` represents the difference between one and the smallest value greater than one. We can use this constant to compare values against each other considering that arithmetic operations may be distorted. Example:

```
console.log(0.1 + 0.2 <= 0.3);
//> false

console.log(0.1 + 0.2 <= 0.3 + Number.EPSILON);
//> true
```



The `parseInt` and `parseFloat` methods are now available from the `Number` object. In order to provide more context, I recommend using `Number.parseInt` and `Number.parseFloat` from now on.

```
// base 10 = decimal input (default):
console.log(Number.parseInt('1234', 10));
//> 1234

// base 16 = hexadecimal input:
console.log(Number.parseInt('ff', 16));
//> 255

console.log(Number.parseFloat('1.2'));
//> 1.2
```



The global `isFinite` and `isNaN` functions are also available in the `Number` object.

```
console.log(Number.isFinite(5));
//> true

console.log(Number isNaN(0/0));
//> true
```



There is one difference between the old global functions and the new `Number` methods:

```
console.log(isFinite('0'), Number.isFinite('0'));
//> true false

console.log(
 isNaN('IamNotANumber'),
 Number.isNaN('IamNotANumber'));
//> true false
```



The old global methods convert strings into numbers. The corresponding

The original methods convert strings into numbers. The corresponding **Number** methods keep the types.

In the next chapter, we will study some features that Es2016 provides us and how they make writing javascript code easier.

# The Exponential Operator

introduction to ES2016 features and using the exponential operator - the syntactic sugar of ES2016

There is not too much to consider for ES2016, as only two new features surfaced: the exponential operator, and the `includes` method of arrays.

As we will see, these two updates are nothing more than syntactic sugar.

Check out the [ES2016 plus compatibility table](#) for more information on the current browser support. Most likely, you will need a transpiler.

Use the [ES2017 Babel Preset](#) for code transpilation if you want to support older browsers.

Before ES2016, we could use the `Math.pow` function to calculate powers of numbers.

```
let base = 10;
let exponent = 3;

console.log(Math.pow(base, exponent));
```



In ES2016, the `**` operator makes the same operation more compact:

```
let base = 10;
let exponent = 3;

console.log(base ** exponent);
```



Now, let's discuss the `includes` method in the next lesson.



# Array `includes`

using the `includes` method to process elements in an array

Even in ES5, it was easy to figure out when an element is a member of an array.

```
console.log([1, 2, 3].indexOf(2) >= 0);
```



In ES2016, the `includes` method makes this check more semantic.

```
console.log([1, 2, 3].includes(2));
```



Similar to `indexOf`, `includes` also accepts a second argument, which specifies the first element of the array that should be checked.

```
// check starts from the 2nd element
console.log([1, 2, 3].includes(2, 1));
```

```
// check starts from the 3rd element
console.log([1, 2, 3].includes(2, 2));
```



Notice that `includes` was already defined for strings. For strings `s` and `t`, `s.includes(t)` is true, whenever `t` is a substring of `s`.

```
console.log('JavaScript'.includes('Java'));
```





Now, let's talk about the features in ES2017.

# New Object Extensions

`Object.entries`, `Object.values`, `Object.getOwnPropertyDescriptors`

Check out the [ES2016 plus compatibility table](#) for more information on the current browser support. Most likely, you will need a transpiler.

Use the [ES2017 Babel Preset](#) for code transpilation if you want to support older browsers.

In this section, we will introduce three `Object` methods:

- `Object.entries`
- `Object.values`
- `Object.getOwnPropertyDescriptors`

Let's discuss each in detail in the following lessons.

# Object.entries and Object.values

check the entries and values of a Map using ES2017

We already know about `Object.keys`:

```
let account = {
 first: 'Zsolt',
 last: 'Nagy',
 email: 'info@zsoltnagy.eu'
};

console.log(Object.keys(account));
```



In ES2017, `Object.values` and `Object.entries` are also available:

```
let account = {
 first: 'Zsolt',
 last: 'Nagy',
 email: 'info@zsoltnagy.eu'
};

console.log(Object.values(account));
console.log();
console.log(Object.entries(account));
```



`Object.entries` can also be used for creating maps.

```
let account = {
 first: 'Zsolt',
 last: 'Nagy',
 email: 'info@zsoltnagy.eu'
};
let accountMap = new Map(Object.entries(account));
console.log(accountMap);
```



Symbol keys are ignored from the keys, values, and entries arrays.

The `entries` method was already defined for arrays in ES6, and it returned an `ArrayIterator`.

```
let iterator = Object.values(account).entries();
console.log(iterator);
//> ArrayIterator {}

console.log(iterator.next());
//> { value: [0, "Zsolt"], done: false }

for (let [val, key] of iterator) {
 console.log(val, key);
}
//> 1 "Nagy"
//> 2 "info@zsoltnagy.eu"
```



Now, let's learn how to manipulate the properties of Objects in the next lesson.

# Object.getOwnPropertyDescriptors

the behavior of value, writable, get, set, configurable and enumerable

`Object.getOwnPropertyDescriptors` returns all property descriptors of its first argument:

```
let player = {
 cards: ['Ah', 'Qc'],
 chips: 1000
};

let descriptors =
 Object.getOwnPropertyDescriptors(player);

console.log(descriptors);
console.log();
console.log(descriptors.cards);
```



`Object.getOwnPropertyDescriptors` returns all property descriptors in an object with the same keys as the keys of the original object. The following four property descriptors are returned (source: [developer.mozilla.com](https://developer.mozilla.com)):

- `value`: the value of the property
- `writable`: `true` if and only if the value associated with the property may be changed (data descriptors only)
- `get`: A function which serves as a getter for the property, or `undefined` if there is no getter (accessor descriptors only)
- `set`: A function which serves as a setter for the property, or `undefined` if there is no setter (accessor descriptors only)
- `configurable`: `true` if and only if the type of this property descriptor may be changed and if the property may be deleted from the corresponding object
- `enumerable`: `true` if and only if this property shows up during

enumeration of the properties on the corresponding object

Let's construct an example for getters and setters:

```
let player = {
 cards: ['Ah', 'Qc'],
 chips: 1000,
 flop: ['7d', '7c', '2c'],
 get hand() {
 return [...this.cards, ...this.flop];
 },
 set hand(newHand) {
 if (newHand.length && newHand.length === 5) {
 [this.cards[0],
 this.cards[1],
 ...this.flop
] = newHand;
 }
 }
};

let descriptors =
 Object.getOwnPropertyDescriptors(player);

console.log(descriptors);
//> Object {
// cards: Object,
// chips: Object,
// flop: Object,
// hand: Object
//}

console.log(Object.keys(descriptors.hand));
//> ["get", "set", "enumerable", "configurable"]

console.log(descriptors.hand.get);
//> function get hand() {
// return [...this.cards, ...this.flop];
// }
```



`Object.getOwnPropertyDescriptors` handles String keys as well as Symbol keys.

To show Symbol keys in node and some browsers, instead of `console.log`, use `console.dir` with the flag `showHidden: true`. Check out [this node issue](#) for more information, or re-read the [relevant section](#) of Chapter 9.

```
let s = Symbol('test');
let test = {
 [s]: 'test'
};
```



```
console.log(Object.getOwnPropertyDescriptors(test));
```



As a result, `Object.getOwnPropertyDescriptors` can be used to make shallow copies of objects using `Object.create`.

`Object.create` takes two arguments:

- the prototype of the object we wish to clone,
- the property descriptors of the object.

To illustrate the difference between a *shallow copy* and a *deep copy*, let's create a shallow copy of the `player` object defined above.

```
let player = {
 cards: ['Ah', 'Qc'],
 chips: 1000,
 flop: ['7d', '7c', '2c'],
 get hand() {
 return [...this.cards, ...this.flop];
 },
 set hand(newHand) {
 if (newHand.length && newHand.length === 5) {
 [this.cards[0],
 this.cards[1],
 ...this.flop
] = newHand;
 }
 }
};

let proto = Object.getPrototypeOf(player);
let descriptors =
 Object.getOwnPropertyDescriptors(player);

let newPlayer = Object.create(proto, descriptors);

newPlayer.chips = 1500;

console.log(player.chips, newPlayer.chips);
```



We have created two seemingly independent entities. However, when trying to change a card of the new player, the change will be made in the context of the old player as well.

```
newPlayer.cards[1] = 'Ad';
console.log(newPlayer.cards[1], player.cards[1]);
```



This is because shallow copying only copied the reference of the cards array to the new player object. The original and the copied reference point to the same array.

Now, let's study some new array extensions.

# New String Extensions

padding in strings for alignment

This section is about two String prototype extensions:

- [padStart](#)
- [padEnd](#)

These two methods are not yet implemented in all browsers. You need to open the Firefox developer tools or Chrome 57 to experiment with them.

Padding is used to add additional characters to the start or the end of a string so that it reaches a given size.

Padding is useful in character mode for alignment.

In the following example, let's format the amounts such that the integer part contains six characters, and the fractional part contains two characters. Let's pad the characters in front of the integer part with spaces, and the decimal part with zeros.

Let's console log the result.

```
let amounts = [
 '1234.0',
 '1',
 '2.56'
];

console.log(`|dddddd.ff|`);
for (let amount of amounts) {
 let [front, back = ''] = amount.split('.');
 front = front.padStart(6);
 back = back.padEnd(2, '0');
 console.log(`|${front}.${back}|`);
}
```



If the second argument of `padStart` or `padEnd` is not given, `' '` characters will be used by default.

Let's talk about asynchronous functions in the next lesson.

# Async-Await

asynchronous functions and controlling the execution of functions using the await operator

The ability to write asynchronous functions is a major update in ES2017.

To understand this chapter, I suggest that you review the [chapter on promises](#).

## What are the asynchronous functions?

Asynchronous functions are functions that return a promise. We denote them by using the `async` keyword.

```
const loadData = async function(value) {
 if (value > 0) {
 return { data: value };
 } else {
 throw new Error('Value must be greater than 0');
 }
}

loadData(1).then(response => console.log(response));
loadData(0).catch(error => console.log(error));
```



When `loadData` returns an object, the return value is wrapped into a promise. As this promise is resolved, the `then` callback is executed, console logging the response.

When `loadData` is called with the argument `0`, an error is thrown. This error is wrapped into a rejected promise, which is handled by the `catch` callback.

In general, return values of an `async` function are wrapped into a resolved promise, except if the return value is a promise itself. In the latter case, the promise is returned. Errors thrown in an `async` function are caught and wrapped into a rejected promise.

## The `await` operator

## The `await` operator

Await is a prefix operator standing in front of a promise.

As long as the promise behind the `await` operator is pending, `await` blocks execution. As soon as the promise is resolved, `await` returns the fulfillment value of the promise. As soon as the promise is rejected, `await` throws the value of rejection.

Let's see an example:

```
const delayedPromise = async () => {
 let p = new Promise((resolve, reject) => {
 setTimeout(() => resolve('done'), 1000);
 });
 const promiseValue = await p;
 console.log('Promise value: ', promiseValue);
}

delayedPromise();

// ... after 1 second
//> Promise value: done
```



The `await` operator can only be used in asynchronous functions. If the `async` keyword is removed from the previous example, an error is thrown:

```
const delayedPromise2 = () => {
 let p = new Promise((resolve, reject) => {
 setTimeout(() => resolve('done'), 1000);
 });
 const promiseValue = await p;
 console.log('Promise value: ', promiseValue);
}
```



## Combining `async` and `await`

We already know that asynchronous functions return a promise.

We also know that the `await` keyword

- expects a promise as its operand,
- should be placed in asynchronous functions

As a result, we can wait for the response of asynchronous functions inside asynchronous functions.

```
const loadData = async () => {
 disableSave();
 const resultSet1 = await asyncQuery1();
 displayResultSet1(resultSet1);
 const resultSet2 = await asyncQuery2();
 displayResultSet2(resultSet2);
 enableSave();
}
```



This hypothetical `loadData` function loads two tables by accessing a server via an API.

First query 1 is executed. The execution of `loadData` is blocked until the promise returned by `asyncQuery1` is resolved.

Once `resultSet1` is available, the `displayResultSet1` function is executed.

Afterward, `asyncQuery2` is executed. Notice that this function is only called after the return value of `asyncQuery1` is resolved. In other words, `asyncQuery1` and `asyncQuery2` are *executed synchronously*.

Once `resultSet2` becomes available, the results are displayed.

There is only one problem with this example. Imagine a web application accessing ten API endpoints. Assume that each server call takes one second in average. If our page can only be rendered after all ten asynchronous calls are executed, we will have to wait ten seconds until the user can browse our page. This is unacceptable.

This is why it makes sense to execute asynchronous queries in parallel. We can use `Promise.all` to create a promise that combines and executes its arguments in parallel.

```
const loadData = async () => {
 disableSave();
 const [resultSet1, resultSet2] = await Promise.all([
```



```
 asyncQuery1(),
 asyncQuery2()
]);
displayResultSet1(resultSet1);
displayResultSet2(resultSet2);
enableSave();
}
```



In this example, all queries are executed asynchronously. If the array inside `Promise.all` contained ten queries, and each query took one second to execute, the execution time of the whole `Promise.all` expression would still be one second.

The two solutions are not equivalent though. Suppose that the average time taken to retrieve `resultSet1` is 0.1 seconds, while `resultSet2` can only be retrieved in one second.

In this case,

- the asynchronous version saves 0.1 seconds compared to the synchronous one,
- However, `displayResultSet1` is only executed after all queries are retrieved in the asynchronous version. This means that we can expect a 0.9 seconds delay compared to the synchronous version.

We can combine the advantages of the two versions by making use of the chainability of the `then` callback of promises.

```
const loadData = async () => {
 disableSave();
 const [resultSet1, resultSet2] = await Promise.all([
 asyncQuery1().then(displayResultSet1),
 asyncQuery2().then(displayResultSet2)
]);
 enableSave();
}
```



In this version of the code, the queries are retrieved asynchronously, and the corresponding `displayResultSet` handler function is executed as soon as the

corresponding promise is resolved. This means that the first query is rendered in 0.1 seconds, while the second query is rendered in one second.

## Parallel execution without await

Let's remove the `disableSave` and `enableSave` functions from the previous example:

```
const loadData = async () => {
 const [resultSet1, resultSet2] = await Promise.all([
 asyncQuery1().then(displayResultSet1),
 asyncQuery2().then(displayResultSet2)
]);
}
```



The function is still working as expected. However, the implementation is made complex for no reason.

We could simply execute the two asynchronous queries and their corresponding handlers one after the other without wrapping them in `Promise.all`:

```
const loadData = () => {
 asyncQuery1().then(displayResultSet1);
 asyncQuery2().then(displayResultSet2);
}
```



By not using `await`, we are not blocking execution of `asyncQuery2` before the promise of `asyncQuery1` is resolved. Therefore, the two queries are still executed in parallel.

Notice that this implementation of `loadData` is not even declared as `async`, as we don't need to return a promise in a vacuum, and we are not using the `await` keyword inside the function anymore.

## Awaiting a rejected promise

There are cases when the operand of `await` becomes a rejected promise. For instance,

- when reading a file that does not exist,
- encountering an I/O error, and
- encountering a session timeout in case of an API call,

our promise becomes rejected.

When promise `p` is rejected, `await p` throws an error. As a result, we have to handle all sources of errors by placing error-prone `await` expressions in `try-catch` blocks.

```
try {
 await p;
} catch(e) {
 /* handle error */
}
```



## Position of the `async` keyword

First, we can create named function expressions of asynchronous regular functions or arrow functions.

```
const name1 = async function() { ... }
const name2 = (...args) => returnValue;
```

When creating function expressions, `async` is written in front of the function keyword.

```
async function name3() { ... }
```



## Summary

Asynchronous functions are functions that return a promise. These functions can handle I/O operations, API calls, and other forms of delayed execution.

Awaiting for the resolution of a promise returns the resolved value of the promise or throws an error upon rejection. The `await` operator makes it possible to execute asynchronous functions sequentially or in parallel.

Async-await gives you an elegant way of handling asynchronous functions, and therefore, it is one of the most useful updates of ES2017.

# Introduction

This bonus section is a collection of helpful job interview questions related to ES6. Be sure to take a look.

If you need to apply your coding skills to get hired, there is a chance that your coding interview questions are in this resource. Imagine that you are in an interview, and the interviewer formulates the question you have already solved. Would you get excited?

If you are not planning to attend a coding interview, you can use this bonus as a final exam to give you pointers to what skills you have not mastered. You will get a chance to solve ES6 exercises covering the whole book. If you get stuck, the reference solutions will give you pointers to the sections you should review.

If you would like to submit a challenging ES6 question, I may include it in this book, or I may create a Youtube video out of it. My email address is [info@zsoltnagy.eu](mailto:info@zsoltnagy.eu).

# Writing an Array Extension

Write a function that creates a palindrome of any array.

Suppose an array of numbers is given. Create a method `toPalindrome` that creates a palindrome out of your array in the following way:

```
const arr = [1,2,3];
//[1, 2, 3]
const arr2 = arr.toPalindrome()
//[1, 2, 3, 2, 1]
const arr3 = arr2.toPalindrome()
//[1, 2, 3, 2, 1, 2, 3, 2, 1]
console.log(arr, arr2, arr3);
//[1, 2, 3] [1, 2, 3, 2, 1] [1, 2, 3, 2, 1, 2, 3, 2, 1]
//undefined
```

```
Array.prototype.toPalindrome = function(){
//write your code here
}
```



gives an error

`toPalindrome()` returns a new array. It keeps the element `arr[arr.length - 1]` the same, and concatenates all the other elements of the array after the end in reverse order.

## Solution:

This exercise is straightforward, it only requires basic JavaScript knowledge, including JavaScript prototypes. We can go on the safe route, and only use basic ES5 constructs.

```
Array.prototype.toPalindrome = function() {
 const result = this.slice();
 for (var i = this.length - 2; i >= 0; --i) {

 result.push(this[i]);
 }
 return result;
}

const arr = [1,2,3];
console.log(arr.toPalindrome())
//[1, 2, 3, 2, 1]
console.log(arr.toPalindrome().toPalindrome())
//[1, 2, 3, 2, 1, 2, 3, 2, 1]
```



To solve this task, you need to know the following about JavaScript:

- To create an array method, we need to extend the prototype of the [Array](#) object
- The [slice](#) method without arguments *clones* an array of integers. In reality, this is a *shallow copy*, which is fine in case of atomic values like integers. To read more about cloning, check out my blog posts [Cloning Objects in JavaScript](#) and [Understanding Value and Reference Types in JavaScript](#).
- We use a simple [for](#) loop to iterate on the elements of the original array that we want to [push](#) to the end of the array. Push modifies the original [result](#) array.

This is a safe and straightforward solution. You can use some more native array methods to make the solution more compact:

```
Array.prototype.toPalindrome = function() {
 return this.slice().concat(this.slice(0, this.length - 1).reverse())
}
const arr = [1,2,3];
console.log(arr.toPalindrome())
//[1, 2, 3, 2, 1]

console.log(arr.toPalindrome().toPalindrome())
//[1, 2, 3, 2, 1, 2, 3, 2, 1]
```



The solution can be explained as follows:

- `slice` still makes a shallow copy of the original array
- `this.slice( 0, this.length - 1 )` makes a shallow copy of the original array, excluding the last element
- `reverse` reverses the elements of the array. Although `reverse` mutates the original array, we are mutating the shallow copy `this.slice( 0, this.length - 1 )`.
- `concat` concatenates two arrays

```
const arr = [1,2,3];

// slice: shallow copy
const arr2 = arr.slice();
//[1, 2, 3]
arr2[1] = 5;
console.log(arr, arr2);
//[1, 2, 3] [1, 5, 3]
const arr3 = arr.slice(0, arr.length - 1);
//[1, 2]

// reverse
arr2.reverse();
console.log(arr, arr2);
//[1, 2, 3] [3, 5, 1]

// concat
console.log(arr.concat(arr3));
//[1, 2, 3, 1, 2]

console.log(arr, arr3)
//[1, 2, 3] [1, 2]

console.log(arr.concat(arr3.reverse()))
//[1, 2, 3, 2, 1]

console.log(arr3)
//[2, 1]
```



If we use ES6, we can replace the `slice` and the `concat` methods with the spread operator:

```
Array.prototype.toPalindrome = function() {
 return [...this, ...this.slice(0, this.length - 1).reverse()];
}

const arr = [1,2,3];
console.log(arr.toPalindrome())
```





If you would like to read more about the spread operator, sign up for my ES6 minicourse or check out my article on the [Spread Operator and Rest Parameters](#).

As you can see, this simple exercise is linked to a lot of JavaScript knowledge that you need to be aware of.

# Binary Gap Exercise in Codility

Find a specific binary sequence.

Suppose a positive integer  $N$  is given. Determine the binary representation of  $N$ , and find the longest subsequence of the form  $10^*1$  in this representation, where  $0^*$  stands for any number of zeros in the sequence. Examples:  $11$ ,  $101$ ,  $1001$ ,  $10001$  etc. Return the number of zeros in the longest sequence you found. If you didn't find such a sequence, return zero.

You can read the original task description on [Codility](#).

## Solution:

Whenever you deal with a riddle, bear in mind, it doesn't matter what techniques you use as long as your solution is correct. Don't try to impress your interviewers with fancy techniques, don't even think about announcing that you are going to use "functional programming" or "recursion" or anything else. Just get the job done.

Do explain your thought process! If you are on the right track, your interviewers will appreciate relating to how you think. If you are on the wrong track, your interviewers will often help you out, because they relate to you, and they want you to succeed.

You can read more interviewing tips in [The Developer's Edge](#).

Before coding, always plan your solution, and explain how you want to solve your task. Your interviewers may correct you, and in the best case, say that you can start coding. In our case, the plan looks as follows:

1. Convert  $N$  into a binary string
2. Set a sequence counter to zero. Set a maximum sequence counter to zero.
3. Iterate over each digit of the binary representation
  - If you find a zero, increase the sequence counter by one.

- If you find a one, compare the sequence counter to the maximum sequence counter, and save the higher value in the maximum sequence counter. Then set the sequence counter to zero to read the upcoming sequence lengths.

4. Once you finish, return the maximum sequence counter value.

### Obtaining the binary representation:

You may or may not know that integers have a `toString` method, and the first argument of `toString` is the base in which the number should be interpreted. Base `2` is binary, so all you need to do to convert an integer into its binary representation is:

```
const n1 = 256, n2 = 257;
var print = "";

print=n1.toString(2)
console.log(print)
// "100000000"

print=n2.toString(2)
console.log(print)
// "100000001"
```



Chances are, you don't know this trick. No problem. In most tech interviews, you can use google. If you formulate the right search expression such as “javascript binary representation of a number”, most of the time, you get a nice and short StackOverflow page explaining the solution. Be careful with copy-pasting tens of lines of code. Look for a deep understanding of the problem, and implement a compact solution.

Never google for the exact solution of the task, because your interviewers may not know how to handle such an attempt.

In the unlikely case you are not allowed to use Google, nothing is lost. You can still solve the same problem in vanilla JavaScript. How do we convert a decimal number to binary on paper?

Suppose your number is 18.

- $18 / 2 = 9$ , and the remainder is `0`.

- $9 / 2 = 4$ , and the remainder is  $1$ .
- $4 / 2 = 2$ , and the remainder is  $0$ .
- $2 / 2 = 1$ , and the remainder is  $0$ .
- $1 / 2 = 0$ , and the remainder is  $1$ .

Read the digits from bottom-up to get the result:  $10010$ .

Let's write some code to get the same result:

```
const IntToBinary = N => {
 let result = '';
 while (N > 0) {
 result = (N % 2) + result;
 N = Math.trunc(N / 2);
 }
 return result;
}

console.log(IntToBinary(256)) //print the output
```



The `%` (modulus) operator gives you the remainder of the division. The `trunc` function truncates the results. For instance, `Math.trunc(9.5)` becomes  $9$ .

If you can't come up with this algorithm on your own, think in another way:

```
// 18 is
(1 * 16) + (0 * 8) + (0 * 4) + (1 * 2) + (0 * 1)
// yielding 10010
```

First, we have to get the largest digit value, which is  $16$ :

```
// Constraint: N > 0.
const getLargestBinaryDigit = N => {
 let digit = 2;
 while (N >= digit) digit *= 2;
 return digit / 2;
}
console.log(getLargestBinaryDigit(20))
```



Then we divide this digit value by  $2$  until we get  $1$  to retrieve the digits of the

binary number one by one. Whenever `N` is greater than or equal to the digit

value, our upcoming digit is `1`, and we have to subtract `digit` from `N`. Otherwise, our upcoming digit value is `0`:

```
const IntToBinary = N => {
 let result = '';
 for (let digit = getLargestBinaryDigit(N); digit >= 1; digit /= 2) {
 if (N >= digit) {
 N -= digit;
 result += '1';
 } else {
 result += '0';
 }
 }
 return result;
}
console.log(IntToBinary(10))
```



Enough said about the integer to binary conversion. Let's continue with the state space of the solution.

Determining the state space:

```
function solution(N) {
 let str = N.toString(2),
 zeroCount = 0,
 result = 0;

 // ...

 return result;
}
```



We will use `N.toString( 2 )` here to get the binary representation of `N`.

To identify a sequence of zeros bounded by ones, we have to know if the sequence has a left border. As every single positive binary number starts with `1`, this condition is automatically true.

Side note: if `N` was allowed to be `0`, even then, our function would return the correct result, because the string `'0'` does not have a trailing `1` in the

sequence.

Therefore, the state space is quite simple: we need to know the binary string of the input, the number of zeros currently read in the sequence, and the longest string found so far.

**Iteration:**

We have to read each digit of the solution one by one. The traditional way in most programming languages is a `for` loop.

```
function solution(N) {
 let str = N.toString(2),
 zeroCount = 0,
 result = 0;

 for (let i = 0; i < str.length; ++i) {
 // ...
 }

 return result;
}
```



We can also use the `for..of` loop of ES6 that enumerates each character of the string. Strings work as *iterators* and *iterable objects* in ES6. For more information, read my article titled [ES6 Iterators and Generators in Practice](#). You can also find six more exercises belonging to this topic in [this blog post](#).

```
function solution(N) {
 let str = N.toString(2),
 zeroCount = 0,
 result = 0;

 for (let digit of str) {
 // ...
 }

 return result;
}
```



**Reading the digits:**

Each digit can either be a zero or a one. We will branch off with an `if` else

Each digit can either be a zero or a one. We will branch off with an if-else statement:

```
function solution(N) {
 let str = N.toString(2),
 zeroCount = 0,
 result = 0;

 for (let digit of str) {
 if (digit === '0') {
 // ...
 } else /* if (digit === '1') */ {
 // ...
 }
 }

 return result;
}
```



### Process the digits:

If we read a zero, we have to increment the zero counter by one. If we read a one, we have to determine if we have just read the longest sequence of zeros by taking the maximum of `result` and `zeroCount`, and saving this maximum in `result`. After determining the new `result` value, we have to make sure to reset `zeroCount` to `0`.

```
function solution(N) {
 let str = N.toString(2),
 zeroCount = 0,
 result = 0;

 for (let digit of str) {
 if (digit === '0') {
 zeroCount += 1;
 } else /* if (digit === '1') */ {
 result = Math.max(result, zeroCount);
 zeroCount = 0;
 }
 }

 return result;
}
console.log(solution(456))
```



If you execute this algorithm in Codility, you can see that all your tests pass. I

encourage you to solve other Codility tasks, as Codility is a great platform to practice coding challenges.

# Ten JavaScript Theory Questions

These are popular conceptual questions asked in JavaScript interviews.

There are countless questions your interviewers may ask when it comes to how JavaScript works. The idea behind asking these questions is to assess whether you have recent experience in writing JavaScript code.

Some more clueless interviewers tend to ask you for lexical knowledge and edge cases that you could simply look up online. I, personally, think that this signals a lack of competence from the end of my interviewers, and I tend to start getting concerned whether I am in the right place.

Usually, you are not allowed to use Google to find the answer, and you have to answer on the spot.

## Question 1

Is JavaScript a “pass by value” or a “pass by reference” type of language when it comes to passing function arguments?

**Answer:** JavaScript passes function arguments by value. In case we pass an array or an object, the passed value is a reference. This means you can change the contents of the array or the object through that reference.

Read [this article on value and reference types](#) for more details.

## Question 2

Study the following code snippet:

```
let user1 = { name: 'FrontendTroll', email: 'ihatepopups@hatemail.com' };
let user2 = { name: 'ElectroModulator', email: 't2@coolmail.com' };

let users = [user1, user2];

let swapUsers = function(users) {
 let temp = users[0];
 users[0] = users[1];
 users[1] = temp;
}
```



```

 users[1] = temp;
 return users;
 }

let setCredit = function(users, index, credit) {
 users[index].credit = credit;
 return users;
}

console.table(swapUsers([...users]));
console.table(setCredit([...users], 0, 10));
console.table(users);

```



What does `[...users]` do? What is printed to the console?

**Answer:** `[...users]` makes a *shallow copy* of the `users` array. This means we assemble a brand new array from scratch. The elements of the new array are the same as the elements of the original array.

However, each element is an object in each array. These objects are *reference types*, which means that their content is reachable from both arrays. For instance, modifying `[...users][0].name` results in a modification in `users[0].name`.

Let's see the printed results one by one.

In the first console table, we expect the two elements to be swapped. This change left the `users` array intact, because none of its elements were modified.

```
console.table(swapUsers([...users]));
```



This is printed on the screen:

| (index) | name               | email                      |
|---------|--------------------|----------------------------|
| 0       | “ElectroModulator” | "t2@coolmail.com"          |
| 1       | “FrontendTroll”    | "ihatepopups@hatemail.com" |

Let's see the second result. We shallow copied the elements of the `users` array again, and added a credit of `10` to the first user. The order of the users is still `FrontendTroll` before `ElectroModulator`, as the order of the elements of the `users` array were not changed by `swapUsers` due to shallow copying. `FrontendTroll` receives ten credits in the cloned array. As we only shallow copied the `users` array, this credit will make it to the original array as well.

```
console.table(setCredit([...users], 0, 10));
```

The above line of code results in:

| (index) | name               | email                      | credit |
|---------|--------------------|----------------------------|--------|
| 0       | “FrontendTroll”    | "ihatepopups@hatemail.com" | 10     |
| 1       | “ElectroModulator” | "t2@coolmail.com"          |        |

Based on the explanation, the third `console.table` will be identical with the second, including the credit of `10`:

```
console.table(users);
```

| 0 | “FrontendTroll”    | "ihatepopups@hatemail.com" | 10 |
|---|--------------------|----------------------------|----|
| 1 | “ElectroModulator” | "t2@coolmail.com"          |    |

Read more on shallow and deep cloning in my article [Cloning Objects in JavaScript](#)

You can execute and visualize this code on [pythontutor.com](http://pythontutor.com).

## Question 3

Does the code in Question 2 conform to the principles of pure functional programming?

**Answer:** No. A function is pure if and only if it is free of side-effects.

`setCredit` modifies a field in the global object `users[0]` as a side-effect of the function execution.

Read the first few paragraphs of the article [Functional and Object Oriented Programming with Higher Order Functions](#) for more details.

## Question 4

How can we prevent `setCredit` from modifying the original array?

**Answer:** Instead of `[...users]`, use deep cloning. As we only work with data that can be represented using a finite JSON string, we can stringify, then we can parse our original object to get a deep copy.

For example:

```
const user1 = { name: 'FrontendTroll', email: 'ihatepopups@hatemail.com' };
const user2 = { name: 'ElectroModulator', email: 't2@coolmail.com' };

const users = [user1, user2];

const deepClone = function(o) {
 return JSON.parse(JSON.stringify(o));
}

let swapUsers = function(users) {
 let temp = users[0];
 users[0] = users[1];
 users[1] = temp;
 return users;
}

let setCredit = function(users, index, credit) {
 users[index].credit = credit;
 return users;
}

console.table (swapUsers(deepClone(users)));
console.table (setCredit(deepClone(users), 0, 10));
console.table(users);
```



## Question 5

What is wrong with the following code?

```
let sum = (...args) => args.reduce((a,b) => a+b, 0);
let oneTwoThree = [1, 2, 3];

let moreNumbers = [...oneTwoThree, 4];
console.log(sum(...moreNumbers, 5));

let [...lessNumbers,] = oneTwoThree;
console.log(sum(...lessNumbers));
```



**Answer:** In ES6, `...` denotes both the Spread operator and rest parameters.

In the first line, `...args` is a rest parameter. The rest parameter has to be the last parameter of the argument list, symbolizing all the remaining arguments of the function. Given there are no more arguments left after `...args`, the rest parameter is in its correct place.

The Spread operator spreads its elements into comma separated values.

Therefore:

```
moreNumbers = [...[1, 2, 3], 4] = [1, 2, 3, 4]
```

because `...[1, 2, 3]` becomes `1, 2, 3`.

In a function call, the spread operator can also be used:

```
sum(...[1, 2, 3, 4], 5) = sum(1, 2, 3, 4, 5)
```

because `...[1, 2, 3, 4]` becomes `1, 2, 3, 4`.

Inside the *destructuring* assignment,

```
let [...lessNumbers,] = oneTwoThree;
```

`...lessNumbers` is a rest parameter. It has to stand at the very end of the array. Given there is a comma after the rest parameter, we expect the code to throw a `SyntaxError`, because the rest parameter has to be the last element of the array. Due to the syntax error, the last line of the code cannot be executed.

## Question 6

Consider the following function:

```
let printArity = function() {
 console.log(typeof arguments, arguments.length);
 console.log(arguments.pop());
}
printArity(1, 2, 3, 4, 5);
```

Determine the output without running the code!



### Header

`[1,2,3].pop()`

3

## Solution:

`arguments` is an object. It is not an array! See the [Mozilla documentation](#) for more details.

For some reason, `arguments` has a `length` property, and it equals the number of arguments of the function, which is `5`.

Given that `arguments` is not an array, the `Array` prototype method `pop` is not available. Therefore, after printing `object 5`, the code throws a `TypeError`, because `arguments.pop` is not a function.

Whenever you can, use rest parameters instead of the `arguments` array. You can read more details on the relationship between the arguments array and rest parameters in [ES6 in Practice](#).

## Question 7

Why isn't `0.1 + 0.2` equal to `0.3`?

**Answer:** This is strictly speaking a computer science question and not a JavaScript question. All you need to know is that JavaScript uses floating point arithmetics, where a number is represented using a finite number of bits.

In this specific example, `0.1 + 0.2` adds up to `0.3000000000000004` due to floating point arithmetics.

## Question 8

How can we retrieve a DOM node collection of all `div` elements on a website?  
How can we retrieve a DOM node collection of all `div` elements having the class `row-fluid` on a website?

**Answer:** It is important that we do not need jQuery for this purpose. If your answer is based on jQuery, please think again, because reliance on jQuery in 2017 is not always optimal.

Regarding the first question, you can either use

`document.getElementsByTagName` or `document.querySelectorAll`. The latter solution uses the same selectors as jQuery does.

```
document.getElementsByTagName('div')
//HTMLCollection(274) [...]

document.querySelectorAll('div')
//HTMLCollection(274) [...]
```



Regarding the second question, you *could* filter the DOM node collection obtained using `getElementsByTagName`:

```
Array.from(
 document.getElementsByTagName('div')
).filter(
 x => x.className.split(' ').indexOf('row-fluid') >= 0
);
```



We have to know how to convert a DOM node collection to an array. We have to understand how to access the class list of a DOM node, and how `indexOf` works in case of arrays. High risk, low reward solution. Let's figure out

works in case of arrays. High risk, low reward solution. Let's figure out something simpler.

As you have just read, `document.querySelectorAll` can process complex selectors. `div.row-fluid` will do all the filtering for you:

```
document.querySelectorAll('div.row-fluid')
```



Strictly speaking, the first solution is wrong, because the result is not a `NodeList`, but an array of two nodes.

## Question 9

Swap the contents of two variables without introducing a third variable!

**Solution:** We can use destructuring to accomplish the desired result. Example:

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



See [this blog post](#) for more exercises on destructuring.

## Question 10

Write a JavaScript function that determines if a string consists of hexadecimal digits only. The digits `A - F` can either be in lower case or upper case.

**Solution:** The easiest way is to formulate a *regular expression*.

```
hexString => /^[0-9a-fA-F]+$/ .test(hexString);
```



## Explanation:

- `^` at the beginning specifies that the string has to start with the specified sequence `[0-9a-fA-F]+`

- `$` at the end specifies that the string has to end with the specified sequence `[0-9a-fA-F]+`
- `[0-9a-fA-F]` is one arbitrary character, which is either a digit or a letter between `a` and `f`, or a letter between `A` and `F`. Note that the solution `[0123456789abcdefABCDEF]` is equally acceptable, just longer.
- `+` specifies that you can repeat the character `[0-9a-fA-F]` as many times as you want, given that you have provided at least one character.

Common sense dictates that a problem with the above solution is that it allows the first digit to be `0`, which is not possible. Notice the task description didn't ask us to take care of this case, so we can simply omit it. However, if we want to go the extra mile, we could write:

```
let checkHexNum = hexString =>
 /^[1-9a-fA-F][0-9a-fA-F]*$/.test(hexString) ||
 hexString == '0';
```



For more details, check out my article on [Regular Expressions in JavaScript](#).

If you want to avoid using regular expressions, you can write a simple loop. Pay attention to the boolean condition though.

```
let checkHexNum = hexString => {
 for (let ch of hexString) {
 if ('0123456789abcdefABCDEF'.indexOf(ch) === -1)
 return false;
 }
 return true;
}
```



# JavaScript Theory Quiz

Take the quiz and find out how much you know about JavaScript!

A JavaScript interview often consists of fundamental JavaScript theory questions. Although it is not possible to cover everything about JavaScript, this post will give you an opportunity to test your knowledge.

You have probably seen that most tests contain questions that are a bit outdated. This test reflects 2017-18 standards, which means that you can, and sometimes you have to make use of your ES6 knowledge.

I suggest answering the questions one by one, so that you can grade yourself. Once you figure out your shortcomings, you can construct a learning plan for yourself.

## Questions

Without looking at the solutions, you can access all ten questions at once here. I encourage you to solve the exercises on your own. Take out your favorite text editor or a piece of paper, and write down your answers. At first, do not use Google.

1. Explain scoping in JavaScript! Enumerate the different types of scopes you know. (6 points)
2. Explain hoisting with one or more examples including `var` and `let` variables. What is the temporal dead zone? (6 points)
3. Explain the role of the `prototype` property via an example! (5 points)
4. Extend your example from question 3 to demonstrate prototypal inheritance! (5 points)
5. Use the ES6 class syntax to rewrite the code you wrote in questions 3 and 4. (7 points)
6. Explain the `this` value in JavaScript! Illustrate your explanation with an example! (6 points)

7. Explain context binding using an example. (3 points)

8. Explain the difference between `==` and `===` in general. Determine the result of a comparison, when two values are:

- of the same type (for all types)
- `null` and `undefined`,
- `Nan` to itself
- 5 to `'5'` (6 points)

9. How can you check if a variable is an array? (2 points)

10. Suppose we would like to detect if a variable is an object using the following code. What can go wrong? How would you fix this error? (4 points)

```
if (typeof x === 'object') {
 x.visited = true;
}
```



The maximum score is 50 points. Multiply your score by 2 to get your percentage. As you are not reading a meaningless horoscope test, I will not describe what each score range means to you. You can quickly figure it out yourself. Note though that this test is in strict mode, in most interviews, your interviewers are a lot more forgiving.

Once you are done with all the answers, start searching for relevant information that can improve your solutions. Make sure you write these enhancements with a different color, or in a separate document.

Verify your original solutions, the ones without using Google. Determine your score.

Verify your enhanced solutions. Have you missed anything?

## Question 1: Scoping

Explain scoping in JavaScript! Enumerate the different types of scopes you know. (6 points)

## **Answer:**

The scope of variables determines where you can access them in your code. (1 point)

Without considering ES6 modules, the scope can be global or local. (1 point)

In modules, a third scope is the module scope.

The global scope consists of global variables and constants that can be accessed from anywhere in your code. (1 point)

The local scope can be local to a function or a block. (1 point)

Variables defined using `var` inside a function have function scope. These variables are accessible/visible inside the function they are defined in. (1 point)

Variables and constants defined inside a block using `let` and `const` have block scope. They are accessible/visible inside the block they are defined in. (1 point)

When `var`, `let`, or `const` are used in global or module scope, their scope is going to be global or module.

## **Question 2: Hoisting**

Explain hoisting with one or more examples including `var` and `let` variables. What is the temporal dead zone? (6 points)

**Answer:** In JavaScript, variable declarations are hoisted to the top of their scope. (1 point)

Both function and block-scoped variables are hoisted to the top of their scope. (1 point)

In the case of function-scoped variables, the value of a variable before its first assignment takes place is `undefined`. (1 point)

In case of a block scoped variable, its value is inaccessible before the intended location of its declaration. This is called the *temporal dead zone*. Accessing a variable in its temporal dead zone throws an error. (1 point)

## Example for block scope: (1 point)

```
{
 console.log(x);
 let x = 3;
}
```



Uncaught ReferenceError: x is not defined

### Reason:

```
{
 // let x; is hoisted to the top of the block
 console.log(x); // temporal dead zone
 let x = 3;
}
```



### Tricky example:

```
{
 try { console.log(x); } catch(_) { console.log('error'); }
 let x;
 console.log(x);
 x = 3;
}
//> error
//> undefined
```



### Reason:

```
{
 // let x; // hoisting
 // start of temporal dead zone for x
 try { console.log(x); } catch(_) { console.log('error'); }
 // end of temporal dead zone for x
 // x = undefined; // at the point of its intended declaration
 console.log(x);
 x = 3;
}
```



## Example for function scope: (1 point)

```
const f = function() {
 // var x; is hoisted to the top
 console.log(x);
 var x = 3;
}

f();
//> undefined
```



If you need some more information on these concepts, check out the first few chapters of this book.

## Question 3: Prototypes

Explain the role of the `prototype` property via an example! (5 points)

**Answer:** JavaScript functions defined using the ES5 syntax have prototypes. (1 point)

Remark: ES6 arrow functions don't have prototypes. Methods defined using the concise method syntax don't have prototypes.

These prototypes become important once a function is used for instantiation. In JavaScript terminology, these functions are *constructor functions*. (1 point)

A prototype may contain functions that are available in every instance created by the constructor functions. (1 point)

### Example: (2 points)

```
function Wallet() {
 this.amount = 0;
}

Wallet.prototype.deposit = function(amount) {
 this.amount += amount;
}
Wallet.prototype.withdraw = function(amount) {
```



```

 if (this.amount >= amount) {
 this.amount -= amount;
 } else {
 throw 'Insufficient funds.';
 }
 }

let myWallet = new Wallet();
myWallet.deposit(100);
console.log(myWallet.amount)
//> 100

```



## Question 4: Prototypal inheritance

Extend your example from question 3 to demonstrate prototypal inheritance!  
(5 points)

### Answer:

```

function BoundedWallet(maxAmount) {
 Wallet.call(this); // 1 point
 this.maxAmount = maxAmount; // 1 point
}

BoundedWallet.prototype = Object.create(Wallet.prototype); // 1 point
BoundedWallet.prototype.constructor = BoundedWallet; // 1 point

BoundedWallet.prototype.deposit = function(amount) {
 if (this.amount + amount > this.maxAmount) {
 throw 'Insufficient wallet capacity';
 }
 Wallet.prototype.deposit.call(this, amount); // this.amount += amount;
} // 1 point

```



One construct worth mentioning is `Wallet.prototype.deposit.call( this, amount );`. Here, instead of writing `this.amount += amount;`, it is semantically more correct to reuse the `deposit` functionality of the base class. In ES5, this is the way to go.

Although the rest of the code is not self-explanatory, we have already covered a similar exercise in Chapter 4 on Classes.

## Question 5: ES6 classes

Use the ES6 class syntax to rewrite the code you wrote in questions 3 and 4. (7 points)

**Answer:**

```
class Wallet { // 1 point
 constructor() { this.amount = 0; } // 1 point
 deposit(amount) { this.amount += amount; }
 withdraw(amount) {
 if (this.amount >= amount) {
 this.amount -= amount;
 } else {
 throw 'Insufficient funds.';
 }
 } // 1 point
}

class BoundedWallet extends Wallet { // 1 point
 constructor(maxAmount) {
 super(); // 1 point
 this.maxAmount = maxAmount;
 }
 deposit(amount) {
 if (this.amount + amount > this.maxAmount) {
 throw 'Insufficient wallet capacity';
 }
 super.deposit(amount);
 } // 1 point
}

let myWallet = new Wallet();
myWallet.deposit(100);
console.log(myWallet.amount) // 1 point
//> 100
```



You need to demonstrate the following six items for a complete solution:

- `class` keyword, class name, and braces to define a constructor function
- a properly working constructor using the *concise method syntax*
- at least one method using the *concise method syntax*
- proper usage of the `extends` keyword
- proper usage of `super` in the constructor of the child class
- at least one redefined method. It is optional to access the shadowed base class method using `super`. In this example, `super.deposit(amount);`

accessed the deposit method of the `Wallet` class.

- some code demonstrating instantiation, which should be unchanged compared to the ES5 prototypal inheritance syntax

## Question 6: this

Explain the `this` value in JavaScript! Illustrate your explanation with an example! (6 points)

**Answer:** In JavaScript, `this` is a global or function scoped variable. (1 point)

When used in global scope, `this` equals the global object, which is `window` in the browser. (1 point)

When used inside a function, the value of `this` is dynamically determined when the function is called and its value equals the context of the function. (2 points)

### Example:

```
class C { f() { return this; } }
class D extends C {}

const d = new D();
console.log(d.f() === d);
//> true
```



Here, `f()` was inherited with prototypal inheritance from class `C`. When creating the `d` object using the class (constructor function) `D`, any method of `d` gets `this` assigned to `d`. (2 points)

The value of `this` can be changed using *context binding*. (1 point)

## Question 7: context binding

Explain context binding using an example. (3 points)

### Answer:

```
let f = function() {
 console.log(this);
}

f.bind({a: 5})();
//> {a: 5}
//(2 points)
```



`f.bind({a: 5})` creates a function, where the value of `this` becomes `{a: 5}`.  
(1 point)

## Question 8: Truthiness

Explain the difference between `==` and `===` in general. Determine the result of a comparison, when two values are:

- of the same type (for all types)
- `null` and `undefined`,
- `NaN` to itself
- 5 to `'5'`
- `Symbol.for('a')` to `Symbol.for('a')` (6 points)

**Answer:** `==` converts its operands to the same type, while `===` is type safe, and is only equal when the types and values are equal. (1 point)

There are six primitive datatypes in JavaScript: boolean, null, undefined, number, string, and symbol.

For null values, undefined values, booleans, strings, and numbers except `NaN`, `a == b` and `a === b` yield the same result, which is `true` whenever the values are the same. (1 point, you may miss the `NaN` exception)

For `NaN`, `NaN == NaN` and `NaN === NaN` is `false`. (1 point)

When it comes to arrays and objects, both `==` and `===` compare the references. So, for instance,

```
let a = [], b = [], c = a;
```

```
console.log(a == b)// or a === b
//false

console.log(a == c) // or a === c
//true
//(1 point)
```



The `null == undefined` comparison is `true` by definition. `null === undefined` is false. You may argue why this is worth a point. In practice, you must have encountered this case during writing code. If you are unsure about this comparison, this may indicate a lack of hands-on experience. (1 point)

`5 === '5'` is false, because the types don't match. In case of `5 == '5'`, the string operand is converted to a number. As the number values match, the result is true. (1 point)

Whenever a `Symbol` is created, they are always unique. Therefore,

```
console.log(Symbol() == Symbol())
//false
```



When using the global symbol registry, we get the same symbol belonging to the string `'a'` whenever we call `Symbol.for('a')`. Therefore,

```
console.log(Symbol.for('a') === Symbol.for('a')) // or ==
//true
//(1 point)
```



Don't confuse `Symbol.for('a')` with `Symbol('a')`. Only the former accesses the global symbol registry. The latter is a mere label associated with the symbol, and different symbols may have the same label:

```
console.log(Symbol('a') == Symbol('a'))
//false

console.log(Symbol.for('a') == Symbol('a'))
```



```
//false
```



## Question 9: Arrays

How can you check if a variable is an array? (2 points)

**Answer:**

`Array.isArray( x )` gives you a `true` result if and only if `x` is an array. (2 points)

If you don't know the above check, you can still invent your own array type checker function. This function may not be as correct as `Array.isArray`, because it may not work in exceptional cases.

The check `x.constructor === Array` is incorrect and is worth zero points, because we could extend the Array class:

```
class MyArray extends Array {}
let x = new MyArray();

console.log(Array.isArray(x))
//> true

console.log(x.constructor === Array)
//> false
```



You might have used this check in your code before, so if you can recall the `'[object Array]'` value of the `Object.prototype.toString` function applied on an array; then you definitely deserve the points.

```
console.log(Object.prototype.toString.call([]))
//"[object Array]"

console.log(Object.prototype.toString.call(x) === "[object Array]")
```



An almost correct check is the usage of the `instanceof` operator:

```
console.log(x instanceof Array)
//true
```



In an interview setting, I would accept the `instanceof` solution, because it is generally not expected to look up edge cases described by e.g. [this StackOverflow article](#), where `instanceof` gives a different result than `Array.isArray`.

In case the solution mentions that it is not perfect, a thorough definition using the `typeof` operator is worth 1 point:

```
typeof x === 'object' &&
typeof x.length === 'number' &&
typeof x.push === 'function'
```



## Question 10: Objects

Suppose we would like to detect if a variable is an object using the following code:

```
if (typeof x === 'object') {
 x.visited = true;
}
```



What can go wrong? How would you fix this error? (4 points)

**Answer:**

`typeof null` is also an object. (1 point)

The value `null` cannot have properties; so the code will crash with an error. (1 point)

We have to add a null check in the condition. (1 point)

```
if (x !== null && typeof x === 'object') {
 x.visited = true;
}
```



If you prefer not including arrays, you can use your array checker from question 8 to exclude arrays. (1 point)

# Connect-4 Solver

Implement the famous game in JavaScript

Suppose an 8\*6 Connect-Four table is given. Each cell of the table is either empty (`null`), or contains the player's number from the possible values `1` and `2`. Determine if any player has won the game by connecting four of their symbols horizontally or vertically. For simplicity, ignore diagonal matches.

## Solution:

As there is no example data, we have to model the table ourselves.

```
const createEmptyTable = () =>
 new Array(8).fill(null).map(
 () => new Array(6).fill(null)
);
```



This simple arrow function returns an empty 6\*8 array:

```
let table = createEmptyTable()
console.log(table)
```



It is evident that we will need a function that checks all elements of the array for four consecutive matches. I encourage you to implement this function yourself. Reading my solution will be more beneficial to you in case you put in the effort to understand what is going on.

```
const checkElements = (
 [head, ...tail],
 matchCount = 0,
 lastElement = null
```



```
) => {
 if (matchCount === 3 && head === lastElement) return true;
 if (tail.length === 0) return false;
 if (head === null) return checkElements(tail);
 if (head === lastElement) {
 return checkElements(tail, matchCount + 1, head);
 }
 return checkElements(tail, 1, head);
}
```



The solution is based on simple recursion. If we find four matches, the function returns `true`.

If there are no more elements left, and there is no match possible anymore, the function returns `false`. Note that the second `if` is only reachable if the first condition is evaluated to `false`. In general, due to the `return` statements, we know that in each line, all `if` conditions of the lines above are `false`.

In the last two conditions, we check if the head is `null`, or matches the sequence we are looking for. In both cases, our task is to recursively call our function with the correct argument list. Eventually, in the last line, we know that `head` contains a non-null element that is different than the last element. In this case, we have to restart the matching process.

Note that if you know how regular expressions work, you could simply write a regex to perform the same work:

```
const checkElements = arr =>
 /([12]),\1,\1,\1/.test(arr.toString());
```



`[12]` is an arbitrary character that is either a `1` or a `2`. We capture it using parentheses, then repeat the captured character using the `\1` capture group reference. We insert the commas in-between. If you want to brush up your regex skills, check out [my articles on regular expressions](#).

We can use the `reduce` array method to match columns:

```
const checkColumns = table =>
```

```
table.reduce(
 (hasMatch, column) => hasMatch || checkElements(column),

 false
)
```



If you still have trouble interpreting what is going on, insert a console log inside the arrow function, and study the logged output:

```
const checkColumns = table =>
 table.reduce(
 (hasMatch, column) => {
 console.log(hasMatch, column);
 return hasMatch || checkElements(column);
 },
 false
)
```



We now need to check the rows. We could google how transposing an array works in JavaScript. However, there is no need to make the solution more complicated than it is. A simple `for` loop will do:

```
const checkRows = table => {
 for (let i = 0; i < table[0].length; ++i) {
 let rowArray = table.map(column => column[i]);
 if (checkElements(rowArray)) return true;
 }
 return false;
}
```



The function works as follows: the `for` loop goes through each element of the first column. Note that in a table, each column has the same number of elements. For this reason, we can form `rowArray` by taking the `i`th element from each column using the `map` function. The map function takes each column of the table, and substitutes it with the `i`th element in the column.

Now that we have an array of consecutive elements, we can use our

`checkElements` function to determine whether any consecutive elements in the array are equal.

`checkElements` function to derive the matches. As soon as we find a match, we

can return `true`. If execution reaches the end of the for loop, we know that none of the rows matched. Therefore, we can safely return `false`.

Let's create a function that checks the whole table for matches:

```
const checkTable = table =>
 checkRows(table) || checkColumns(table);
```



# Binary Trees, Recursion and Tail Call Optimization in Javascript

Use recursion to determine the height of a binary tree. You must figure out how to implement the binary tree in JavaScript.

We are covering each aspect of the job interviewing process. You have already seen some theoretical questions that demonstrate how well you can use JavaScript. You have seen some coding challenges that not only let you showcase your problem-solving abilities, but also demonstrate your theoretical knowledge and your algorithmic skills.

You are yet to experience some longer homework assignment type of tasks that challenge your abilities to write maintainable software. Some of these challenges are timed, some require you to use some frameworks or libraries, while others need you to structure your code.

The challenge I have chosen for this session is an online coding challenge on a site called [HackerRank](#).

I have already recommended that you go through the challenges of a similar site called [Codility](#). HackerRank ups the ante a bit more by giving you problems of continuously increasing difficulty.

Once you sign up, HackerRank recommends a thirty-day challenge for you. You get one exercise a day, which often takes just a couple of minutes. The thirty-day challenge is very healthy, because it builds a habit of coding just a bit every single day.

Consider the option of using challenges like the ones HackerRank provides to improve your problem-solving skills.

As an illustration, I will now solve a coding challenge that you can find in the Data Structures section of HackerRank. The problem is called [Height of a Binary Tree](#).

For advanced positions, you will be expected to know some data structures and algorithms, as well as some programming techniques like pure functional programming and recursion. We will build on some of this knowledge.

You can either read the task by signing up on HackerRank and visiting the link, or by reading my summary here:

*Suppose a binary tree is given with root  $R$ . Each node may be connected to zero, one, or two child nodes. The edges of the tree are directed from the parent nodes towards child nodes. Determine the height of the tree, defined as the maximal number of edges from  $R$  to any node in the tree.*

The JavaScript data structure of a node is as follows:

```
type Node = {
 data: number,
 left: Node | null,
 right: Node | null
}
```



## Solution:

Let's sketch a plan:

- The height of a tree with one node with no children is  $0$
- The height of a tree with a left and a right subtree is  $1 + \max(\text{height of left subtree}, \text{height of right subtree})$

These are all the ideas you need to demonstrate to be able to solve this exercise. Let's create a solution function.

```
const treeHeight = tree =>
 Math.max(
 tree.left === null ? 0 : 1 + treeHeight(tree.left),
 tree.right === null ? 0 : 1 + treeHeight(tree.right)
);
```



That's it. We have solved this exercise with recursion.

Notice the solution is purely functional, as it is not relying on any side-effects. Also, notice the elegance of the solution in a sense that we just described the

input (`tree`) and the return value.

Now, your interviewer may ask you to solve this exercise without recursion. Remember, for every recursive solution there exists an equivalent iterative solution. To find the iterative solution, we need to save the upcoming recursive calls in a queue-like data structure (a simple array will do), and introduce some *accumulator variables* that store the current state of the computation.

Let's start writing the frame of substituting recursion:

```
const treeHeight = root => {
 let nodes = [{ root, distance: 0 }];
 let maxHeight = 0;

 while (nodes.length > 0) {
 let node = nodes.pop();
 // ...
 }
}
```



We put the tree in a data structure where we save the distance from the root. We also initialize the maximum height of the tree to zero.

Instead of recursion, we have a while loop. As long as there are nodes in the `nodes` array, we pop one, and process its consequences. During the processing, we may push more nodes to the array:

```
const treeHeight = root => {
 let nodes = [{ node: root, distance: 0 }];
 let maxHeight = 0;

 while (nodes.length > 0) {
 let currentTree = nodes.pop();
 maxHeight = Math.max(maxHeight, currentTree.distance);
 if (currentTree.node.left !== null) {
 nodes.push({
 node: currentTree.node.left,
 distance: currentTree.distance + 1
 });
 }
 if (currentTree.node.right !== null) {
 nodes.push({
 node: currentTree.node.right,
 distance: currentTree.distance + 1
 });
 }
 }
}
```



```
 j);
 }

 return maxHeight;
}
```



Now that you have completed the iterative solution, a common question is whether you can write a recursive solution that is tail call optimized. Let's see the original recursive solution:

```
const treeHeight = tree =>
 Math.max(
 tree.left === null ? 0 : 1 + treeHeight(tree.left),
 tree.right === null ? 0 : 1 + treeHeight(tree.right)
);
```



The recursive calls are inside `Math.max`, so they are not in tail position. We have to extract them out from the `Math.max`. The question is how.

The iterative solution always gives you an idea for tail recursion. Even if you are unsure about the exact definition of *tail position* for recursive function calls, you can take the state space of the iterative function and implement the while loop using recursion:

```
const treeHeight = root =>
 treeHeightRecursive([{ node: root, distance: 0 }], 0);

const treeHeightRecursive = (nodes, maxHeight) => {
 let currentTree = nodes.pop();
 maxHeight = Math.max(maxHeight, currentTree.distance);
 if (currentTree.node.left !== null) {
 nodes.push({
 node: currentTree.node.left,
 distance: currentTree.distance + 1
 });
 }
 if (currentTree.node.right !== null) {
 nodes.push({
 node: currentTree.node.right,
 distance: currentTree.distance + 1
 });
 }
 if (nodes.length === 0) return maxHeight;
 return treeHeightRecursive(nodes, maxHeight);
}
```



```
}
```



One minor difference concerning the iterative solution is that we have to manually create an exit condition from recursion with the condition

```
nodes.length === 0.
```

# Painting on an HTML5 Canvas

The title explains it all. Let's create a paint canvas with a color palette and line thickness options!

This exercise is designed as an on-site interview question, which means you don't have hours to complete it. Therefore, we will not use heavy templates, Babel, or any tooling.

## Exercise:

Create a webpage, where you can paint on a canvas. The user should be able to select the color and the thickness (pixel) of the drawn line. You may use any HTML5 elements.

## Solution:

Different browsers render the same elements differently. To combat this problem, the demonstration of using CSS resets or normalizers is beneficial. Resets remove all element styles, while normalizers make the default styles consistent for as many browsers as possible. I chose to use `normalize.css`. You can get it using

```
npm install normalize.css
```

We can reference this normalizer in our HTML file. Let's create our `index.html` file:

```
<!doctype html>
<html>
 <head>
 <title>Paint - zsolt nagy.eu</title>
 <link rel="stylesheet"
 href="node_modules/normalize.css/normalize.css">
 <link rel="stylesheet" href="styles/styles.css">
 </head>
 <body>
```

```
<input type="color" class="js-color-picker" color-picker">
<canvas class="js-paint paint-canvas"
 width="600"
 height="300"></canvas>

</body>
</html>
```

Save the file as `index.html`.

The markup contains a color picker, and the canvas element.

Notice that we created a reference to `styles/styles.css`. Let's create it, and put some border settings around the canvas for clarity:

```
.paint-canvas {
 border: 1px black solid;
 display: block;
 margin: 1rem;
}

.color-picker {
 margin: 1rem 1rem 0 1rem;
}
```



The canvas is now clearly visible, and we can also select the color.

In the JavaScript file, we will first reference our canvas element by selecting the `.js-canvas` element. Notice I used the `.js-` prefix in the class in order to make it clear, this class is used for functionality, not for styling. This is *separation of concerns* in action.

Using a `.js-` prefixed class in the CSS is discouraged. Imagine a web styler using your `.js-` class to hack some styles in your application. Then one day, the implementer of a feature decides the `.js-` class is not needed anymore. Relying on the naming, he deletes the `.js-` class, breaking the styles. We don't want these things to happen. We also want to give both parties the flexibility of owning their own class names. Styling and functionality are two independent aspects. They should be separated properly.

In the canvas, we can choose between a 2D and a 3D graphical context. We will retrieve the two-dimensional context for drawing on the canvas.



```
const paintCanvas = document.querySelector('.js-paint');
const context = paintCanvas.getContext('2d');
```

The color picker reference is also needed. Let's use this reference to add a *change event listener* that console logs the chosen color:



```
const colorPicker = document.querySelector('.js-color-picker');

colorPicker.addEventListener('change', event => {
 console.log(event.target.value);
});
```

As we know the chosen color, we can set the *stroke style* of the graphical context to this color:



```
colorPicker.addEventListener('change', event => {
 context.strokeStyle = event.target.value;
});
```

Let's get to business and start drawing. We need to listen to three events of the canvas:

- **mousedown** indicates that we have to start drawing
- **mousemove** indicates that we have to draw a line from the *last position* where the cursor was to the current position.
- **mouseup** indicates that we have to stop drawing

Why do we need to draw a line from the last position to the current position? Because by just drawing a dot at the current position, our image would depend on the frame rate of the browser. This frame rate is not constant. The garbage collector may start running; your browser may start slowing down; a notification may appear, and so on. We don't want our image to depend on external conditions.

To make the drawing happen, we need to determine the state space of the application. We need to keep track of the **x** and **y** coordinates, and the state of the mouse.

```
let x = 0, y = 0;
let isMouseDown = false;
```



Thinking about the state space is always an important step when it comes to animation. In more complex examples, you might want to store the position, velocity, and acceleration of objects that may collide. This is where your high school physics studies come in handy.

In a canvas, the top-left point has the coordinates `(x, y) = (0, 0)`, and the bottom-right point has the coordinates `(x, y) = (canvas.width, canvas.height)`. You can get the current mouse coordinates from the `mousemove` event.

Let's implement the three canvas event listeners:

```
paintCanvas.addEventListener('mousedown', () => {
 isMouseDown = true;
});
paintCanvas.addEventListener('mousemove', event => {
 if (isMouseDown) {
 console.log(event);
 }
});
paintCanvas.addEventListener('mouseup', () => {
 isMouseDown = false;
});
```



For now, the `mousemove` event only contains a conditional console log. When we execute the code and start logging some values, we may get confused seeing all the different values. Let me give you an example:

```
event
.clientX: 425
.clientY: 109
.layerX: 405
.layerY: 35
.offsetX: 409
.offsetY: 109
.pageX: 425
.pageY: 109
.screenX: 426
.screenY: 560
.x: 425
.y: 109
```



If you randomly select a pair of values that make sense to you, you increase

your chances of failing an interview. Not knowing which value stands for what is entirely acceptable. Just start searching for the answer and move on.

Don't guess. Don't experiment either, because these values are tricky. The fact that `clientX`, `pageX`, and `x` have the same value in this one object, *does not imply* that they are always equal. They are only equal if their definitions say so. Therefore, it's time to look up these definitions.

We can conclude that `x` is indeed equal to `clientX`, because in the [documentation of MouseEvent.x](#), we can read that “*The `MouseEvent.x` property is an alias for the `MouseEvent.clientX` property.*” However, after reading a bit more about these properties, it turns out that they are not the ones we need.

Once we read the [documentation of MouseEvent.offsetX](#), depending on temperament, we could imitate the Backstreet Boys singing “*You're the one I need.*” Oh well, they are right about the mouse event, just don't listen to their songs for dating advice.

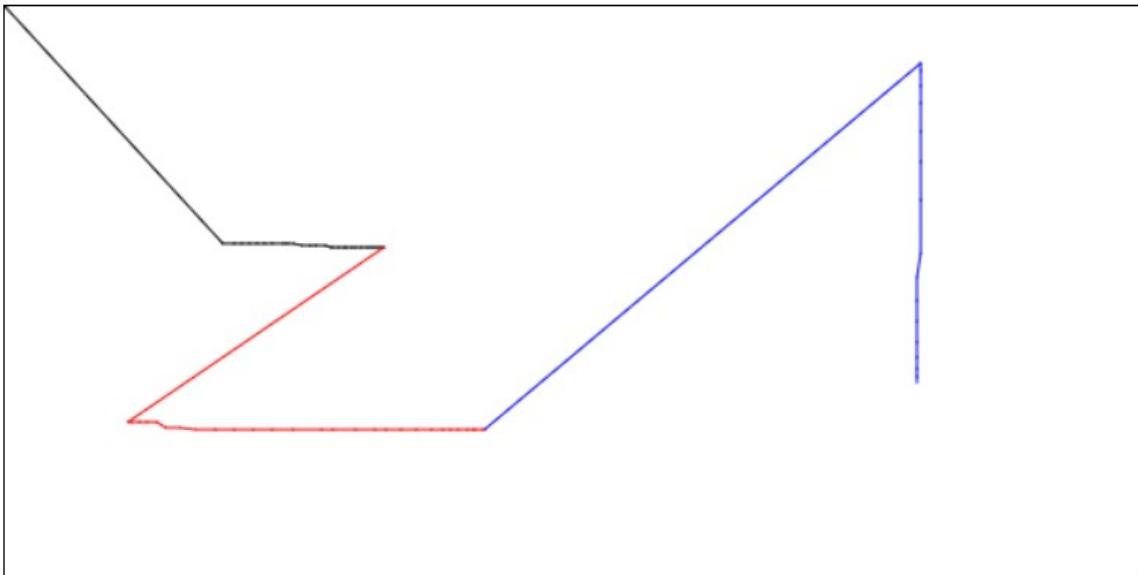
Now that we know that the current coordinates are `event.offsetX` and `event.offsetY`, and the initial coordinates are saved in our state space as `x` and `y`, we know everything to draw our line:

```
paintCanvas.addEventListener('mousemove', event => {
 if (isMouseDown) {
 const newX = event.offsetX;
 const newY = event.offsetY;
 context.beginPath();
 context.moveTo(x, y);
 context.lineTo(newX, newY);
 context.stroke();
 [x, y] = [newX, newY];
 }
});
```



As we start drawing, we can notice a couple of problems.

First of all, when we first start drawing, a line connects the point we first moved to with `(0,0)`. Second, after releasing the mouse, once we start drawing again, a line connects our last drawing with the current one.



Erroneous drawing: the last dot of the previous sequence is connected to the first dot of the new sequence.

Both problems are due to not setting `x` and `y` in the state space to an initial value once we pressed the mouse.

We can solve this problem in multiple ways. One fix involves setting `x` and `y` to `null` whenever we stop drawing. Their initial values should also be `null`. Then, once we start drawing, in the condition, we have to check if `x` and `y` are numbers. If `x` and `y` are null, we ignore drawing.

This solution looks all right on screen, but it is not pixel perfect, because it ignores the very first line of our path. Even if no-one pointed it out, notice that you would have to change the code in at least three different places. Let alone any future modifications in case we wanted to define more events.

We will therefore look for an easier fix. `event.offsetX` and `event.offsetY` are also available inside the `mousedown` event. Therefore, we can initialize the value of `x` and `y` there.

```
paintCanvas.addEventListener('mousedown', event => {
 isMouseDown = true;
 [x, y] = [event.offsetX, event.offsetY]
});
```



Wow! Drawing is now working like a charm. There is just one task left: the ability to draw on the lines this

ability to change the line thickness.

For this purpose, we will use a HTML5 `<slider>` element. We will initialize its value to `1`, and allow our range of thickness between `1Px` and `72Px`.

To read the value of the slider, we will also add a label, displaying the thickness of the line:

```
<!doctype html>
<html>
 <head>
 <title>Paint - zsolt nagy.eu</title>
 <link rel="stylesheet"
 href="node_modules/normalize.css/normalize.css">
 <link rel="stylesheet" href="styles/styles.css">
 </head>
 <body>
 <input type="color" class="js-color-picker">
 <input type="range"
 class="js-line-range"
 min="1"
 max="72"
 value="1">
 <label class="js-range-value">1</label>Px
 <canvas class="js-paint paint-canvas"
 width="600"
 height="300"></canvas>
 </body>
</html>
```

In the JavaScript code, after getting the reference of both objects, we will listen to an event of the slider. If we used the `change` event, we would get an unwanted surprise: the `change` event only fires once we release the slider. After looking up the documentation, you can find the `input` event, which fires upon changing the value of the slider.

```
const lineWidthRange = document.querySelector('.js-line-range');
const lineWidthLabel = document.querySelector('.js-range-value');

lineWidthRange.addEventListener('input', event => {
 const width = event.target.value;
 lineWidthLabel.innerHTML = width;
 context.lineWidth = width;
});
```

As we start drawing a thicker line, another unwanted phenomenon occurs: whenever we make curves, our lines do not form a curvy path. We can see some irregularities instead.

This is because we have not set up the `lineCap` property of the graphical context to `round`:

```
context.lineCap = 'round';
```



This wraps up the canvas painter exercise. We can draw a line of any color and any thickness ranging from `1px` to `72px`.

Play around with it.

If you are thorough, you might have done the following scenario: press the mouse button on the canvas and start drawing. Without releasing the mouse button, exit the canvas. Release the mouse button outside the canvas. Scroll back to the canvas with your mouse button in the released state. Surprise: drawing continues.

This is because we modeled the mouse button in our internal state, but we never took care of the mouse release if it happened outside the scope of the canvas.

How can we fix this?

We set the `isMouseDown` method to `false` in the `mouseup` event of `paintCanvas`.

```
paintCanvas.addEventListener('mouseup', () => {
 isMouseDown = false;
});
```



A dirty trick might inspire you to change the `paintCanvas` to `document`. If you do this and test your code shallowly, you might even succeed to a certain extent.

However, dirty tricks often get caught. In a Windows machine, for instance, you can start drawing, then press Alt+tab to change the currently active task, and click your current window. After the click, your mouse button is in the released state, and we keep drawing.

A proper fix entails listening to the `mouseout` event of `paintCanvas`. For the sake of maintainability, we can also refactor the handler function to indicate

sake of maintainability, we can also refactor the handler function to indicate that `mouseup` and `mouseout` handlers take care of the same piece of functionality. This way, other people maintaining your code will not forget adding their fix to one of the event handlers:

```
const stopDrawing = () => { isMouseDown = false; }
paintCanvas.addEventListener('mouseup', stopDrawing);
paintCanvas.addEventListener('mouseout', stopDrawing);
```

To make our code more semantic, we can do this refactoring for all our drawing event handlers.

```
const startDrawing = event => {
 isMouseDown = true;
 [x, y] = [event.offsetX, event.offsetY];
}
const stopDrawing = () => { isMouseDown = false; }
const drawLine = event => {
 if (isMouseDown) {
 const newX = event.offsetX;
 const newY = event.offsetY;
 context.beginPath();
 context.moveTo(x, y);
 context.lineTo(newX, newY);
 context.stroke();
 [x, y] = [newX, newY];
 }
}

paintCanvas.addEventListener('mousedown', startDrawing);
paintCanvas.addEventListener('mousemove', drawLine);
paintCanvas.addEventListener('mouseup', stopDrawing);
paintCanvas.addEventListener('mouseout', stopDrawing);
```

Refactoring is always great. For instance, suppose an angry customer comes to you with a complaint. He says, our canvas is a piece of crap, because if we click without moving the mouse, nothing is drawn on the screen. As you examine the code, you may already be grateful for the refactoring step above, because the request can be handled by adding just one line to `startDrawing`:

```
const startDrawing = event => {
 isMouseDown = true;
 [x, y] = [event.offsetX, event.offsetY];
 drawLine(event);
}
```

Problem solved!

Check out the final result in [this CodePen](#).

# Video Player

Create a web video player in which you can alter playback speed and skip forward/backward.

## Task:

Implement a video player that can play an mp4 video. Add five buttons below the video player:

- `1x`, `1.5x`, `2x`: when clicked, it sets the playback speed to the displayed value on the button
- `-30s`, `+30s`: when clicked, it offsets the current time of the video by the displayed value

You can use HTML5 tags in the exercise, and you don't have to worry about cross-browser compatibility.

As an example, you can use [this video](#).

Make your solution extensible so that it will be easy to add more fully functional playback speed and offset buttons without changing anything in your JavaScript code.

## Solution:

If you have never used the HTML5 video API, it's time to google it. The video markup looks as follows:

```
<video class="js-video"
 width="540"
 height="360"
 controls
 src="http://clips.vorwaerts-gmbh.de/big_buck_bunny.mp4"></video>
```

We will also need some buttons:

```
<button class="js-speed-button" data-speed="1">1x</button>
<button class="js-speed-button" data-speed="1.5">1.5x</button>
<button class="js-speed-button" data-speed="2">2x</button>

<button class="js-offset-button" data-offset="30">+30s</button>
<button class="js-offset-button" data-offset="-30">-30s</button>
```

I kept the markup lean, and added some classes that will make it easy to identify each button. As we have to take care of extensibility, it makes sense to reference each button with the same class.

We also used some data attributes to customize the offset. We will read these attributes in the event handlers.

We will write the event handlers such that the code handles all buttons of the same type generically:

```
document.querySelectorAll('.js-speed-button').forEach(item =>
 item.addEventListener('click', function(e) {
 // Handle the playback speed update
 })
);

document.querySelectorAll('.js-offset-button').forEach(item =>
 item.addEventListener('click', function(e) {
 // Handle the offset update
 })
);
```

`document.querySelectorAll` is similar to jQuery's `$` function. We pass it a selector, and it returns a DOM node collection. As this collection is an *iterable*, we can iterate on the elements. We can attach an event listener to each event.

Study [the Video API reference](#) to conclude how to change the playback speed and the offset.

Let's start with the playback speed:

```
document.querySelectorAll('.js-speed-button').forEach(item =>
 item.addEventListener('click', function(e) {
 const speed = e.target.dataset.speed;
 const video = document.querySelector('.js-video');
 video.playbackRate = speed;
 })
);
```

Notice how we retrieve the `data-speed` attribute: `e.target.dataset` contains all the data attributes belonging to a DOM node.

The `.js-video` `video` HTML5 element has a `playbackRate` property. If we set it to a floating point, we can change the playback rate.

We can conclude this exercise with the implementation of the offset buttons:

```
document.querySelectorAll('.js-offset-button').forEach(item =>
 item.addEventListener('click', function(e) {
 const video = document.querySelector('.js-video');
 const duration = video.duration;
 const offset = Number.parseInt(e.target.dataset.offset);
 let newTime = video.currentTime + offset;
 if (newTime > duration) newTime = duration;
 if (newTime < 0) newTime = 0;

 video.currentTime = newTime;
 })
);
```



The video duration is in seconds. We can retrieve the offset from the `data-offset` attribute via the property `e.target.dataset.offset`. The `currentTime` property of the video contains the place where the video is at currently. After adding the offset to the current time, we have to check if we are still within the boundaries of the video to avoid indexing out from the video.

As the `currentTime` property of the video element is writable, we have to assign the new value to it to make the offset work.

Experiment with the solution [in this codepen](#).

# Event Delegation in a Pomodoro App

This is your first extensive exam task. The concepts in the following lessons will come in handy during interviews. It's a long exercise, but trust me, it's worth the hassle.

We will now build a simple Pomodoro App. If you don't know what the Pomodoro technique is, you can read about it [here](#).

## Exercise

Create a client-side application that displays a table of tasks with the following columns:

- Task name (string);
- Status: number of pomodori done, a slash, the number of pomodori planned, then a space, then the word `pomodori`;
- Controls: contains three buttons for each row, `Done`, `Increase Pomodoro Count`, and `Delete`.

When pressing the `Done` button, the `Done` and the `Increase Pomodoro Count` buttons are replaced by the static text `Finished`.

When pressing `Increase Pomodoro Count`, the number of pomodori done is increased by `1` in the Status column. The initial value of the number of pomodori done is zero.

When pressing `Delete`, the corresponding row is removed from the table.

Create a form that allows you to add a new task. The task name can be any string, and the number of pomodori planned can be an integer between `1` and `4`.

## Unblock Yourself

This task may be an exercise that you can either solve during an interview, or as a homework exercise.

Always respect the requirements, and never invent anything on your own. It is fine to clarify questions on the spot if you have time to ask them. If you solve this task as a homework assignment, it is also okay to send your questions to your contacts via email.

Pointing out the flaws in the specification is a great asset to have as long as you demonstrate that you can cooperate with your interviewers.

My usual way of cooperation is that I ask my questions along with the first submitted version of my task. This means I implement everything I can, without getting blocked, and then I enumerate my assumptions and improvement suggestions attached to the first version.

This sends the message that you are aware of a flaw in the specification, and you are willing to fix it, in case it is needed. You also signal that you were proactive in implementing everything you could, based on the information available to you.

Most of the time your interviewers will accept your solution. Sometimes they may ask you to go ahead and make changes to your application based on their feedback.

Remember, don't block yourself just because the task is underspecified. You can implement only what's needed, and tackle the improvement suggestions later.

## What is not specified?

In this example, there are quite a few unusual elements.

First, the task name may be an empty string. There is no validation specified in the task description. You may point this out as an improvement suggestion. Implementing validation on your own would mean that you don't respect the specification.

Second, different rows may contain the same task.

Third, there is no way to undo pressing the **Done** button. When a task is finished, it will stay finished.

Fourth, the Pomodoro counter may increase above the planned number of pomodori.

Fifth, `pomodoro` is singular, `pomodori` is plural, but we always display `pomodori` in the status column.

You can point all these anomalies out as improvement suggestions.

Improvising and implementing these features without asking for permission would imply that you don't respect the specification. Some hiring crews will not care about it, while others may even reward you for improvising.

However, chances are, if you continuously improvise, your interviewers will ask themselves the question if they can cooperate with you smoothly.

This is why I suggest putting all your improvement suggestions in the documentation attached to your solution. You even save yourself time.

## Solution

Let's start with the markup:

```
<!doctype html>
<html>
 <head>
 <title>Pomodoro Timer - zsolt nagy.eu</title>
 <link rel="stylesheet"
 href="node_modules/normalize.css/normalize.css">
 <link rel="stylesheet" href="styles/styles.css">
 </head>
 <body>
 <table>
 <thead>
 <tr>
 <th>Task name</th>
 <th>Status (done / planned)</th>
 <th>Controls</th>
 </tr>
 </thead>
 <tbody class="js-task-table-body">
 </tbody>
 </table>

 <form class="js-add-task"
 action="javascript:void(0)">
 <input type="text"
 name="task-name"
 class="js-task-name"
 placeholder="Task Name" />
 <select name="pomodoro-count"
 class="js-pomodoro-count">
 <option value="1">1</option>
 <option value="2">2</option>
 <option value="3">3</option>
 <option value="4">4</option>
```

```
</select>
<input type="submit" />
</form>
</body>
<script src="js/pomodoro.js"></script>
</html>
```

Note the following DOM nodes:

- **js-task-table-body**: this is where we will insert the tasks one by one as table rows
- **js-add-task**: the form to add new tasks. We will listen to the submit event of this form
- **js-task-name**: a text field containing the task name inside the **js-add-task** form
- **js-pomodoro-count**: a dropdown list containing the number of planned pomodori inside the **js-add-task** form. Usually, more than four pomodori would mean that we haven't broken down the task well enough, this is why we only allow up to four pomodori as a limit.

Let's write some JavaScript in the **js/pomodoro.js** file.

```
let tasks = [];
const pomodoroForm = document.querySelector('.js-add-task');
const pomodoroTableBody =
 document.querySelector('.js-task-table-body');
```

First, notice we need an array of tasks to store the contents of the table. We also need a reference to the pomodoro form and the table body.

We plan to handle the form submission with an event handler.

```
const addTask = function(event) {
 event.preventDefault();
 // ...
 this.reset();
 // ...
}
pomodoroForm.addEventListener('submit', addTask);
```

Notice the **preventDefault** call. When we submit a form, a redirection is made. The default action defined in HTML forms is that a server handles our submitted form data, and renders new markup for us. In a client side

application, we rarely need this default action from the end of the server.

Therefore, we can prevent this default action by calling the `preventDefault` method of the submit event.

Technically, this is not mandatory, because the action of the `form` is `javascript:void(0)`, which does not make any redirections. I showed you this option in the markup, but I still recommend using `preventDefault` from JavaScript's end to avoid the consequences someone accidentally removing `javascript:void(0)` from the markup.

The context inside the event handler is the form element itself. Calling the `reset` method of the form resets all form fields to their default values. We can safely reset the form once we are done processing the values.

Let's do this processing now:

```
const addTask = function(event) {
 // 1. Prevent default action
 event.preventDefault();

 // 2. Extract form field values
 const taskName = this.querySelector('.js-task-name').value;
 const pomodoroCount =
 this.querySelector('.js-pomodoro-count').value;

 // 3. Create a new task item by updating the global state
 tasks.push({
 taskName,
 pomodoroDone: 0,
 pomodoroCount,
 finished: false
 });

 // 4. Reset the form
 this.reset();

 // 5. Render the global state
 renderTasks(pomodoroTableBody, tasks);
}
```

We have already covered steps 1 and 4.

Step 2 is about extracting the values the user entered. Notice the `this.querySelector` construct. Remember? The value of `this` is the DOM node of the form. Therefore, we can use the `querySelector` method of this DOM node to search for the corresponding form fields and take their `value`

attribute.

In Step 3, we create a new object. Notice the object shorthand notation. Remember, in ES6, `{ x }` is equivalent to `{ x: x }`.

I decided on implementing rendering in a separate function, because this feature will likely be needed later once we update the form. Let's finish Step 5 by implementing the `renderTasks` function. I will use the ES6 Template Literal format.

We can conveniently include newline characters in the template without terminating it. We can also evaluate JavaScript expressions in the form ``${expression}``:

```
const renderTasks = function(tBodyNode, tasks = []) {
 tBodyNode.innerHTML = tasks.map((task, id) => `
 Template for task[${id}] with name ${task.taskName}
 `).join('');
}
```

We will set the `innerHTML` property of `tBodyNode` to a text node containing the string that we assemble.

The assembly is made using the `map` method of the `tasks` array. A map is a *higher order function*, because it expects a function as an argument. This function is executed on each element of the `tasks` array one by one, transforming `tasks[id]` also accessible as `task` onto string return values. The template is assembled by joining these string values.

If you are not familiar with `map`, the code is almost the same as the below `for` loop equivalent. The only difference is the whitespace inside the template literal.

```
const renderTasks = function(tBodyNode, tasks = []) {
 let template;
 for (let i = 0; i < tasks.length; ++i) {
 template += `task[${i}] with name ${tasks[i].taskName}`;
 }
 tBodyNode.innerHTML = template;
}
```

As a loose tangent, technically, we don't need the `template` variable, because we could simply append each template row to `tBodyNode.innerHTML`. Right?

We could simply append each template row to `tBodyNode.innerHTML`. Right?

Well, right and wrong. Technically, you could do this, and your code would look shorter. In practice, always bear in mind that DOM operations are more expensive than JavaScript operations. So much so, that once I managed to dig inside the jQuery UI Autocomplete code to shove off more than 95% of the execution time of opening the autocomplete by assembling the `$node.innerHTML +=` type of DOM manipulations in memory, and making just one DOM insertion at the end.

Back to business. Let's assemble our task table row:

```
const renderTasks = function(tBodyNode, tasks = []) {
 tBodyNode.innerHTML = tasks.map((task, id) => `
 <tr>
 <td class="cell-task-name">${task.taskName}</td>
 <td class="cell-pom-count">
 ${task.pomodoroDone} / ${task.pomodoroCount} pomodori
 </td>
 <td class="cell-pom-controls">
 ${ task.finished ? 'Finished' :
 <button class="js-task-done"
 data-id="${id}">
 Done
 </button>
 <button class="js-increase-pomodoro"
 data-id="${id}">
 Increase Pomodoro Count
 </button>
 }
 <button class="js-delete-task"
 data-id="${id}">
 Delete Task
 </button>
 </td>
 </tr>
 `).join('');
}
```

The `td` classes are there for styling. I won't bother you with the details; you can check out the CSS code on my [GitHub repository](#).

The `button` classes are there for event handling. After all, we will have to handle the button clicks later. To make our life easier, we can also add the `id` data attribute to the button.

The ternary `? :` operator makes sure that we either display the `Finished` text, or the two buttons described in the specification.

# The Twist

So far, the code is straightforward. I mean, you have to know what you are doing to come up with a solution like this. You also have to know the ins and outs of writing basic HTML markup and basic JavaScript.

Progress may even give you a false illusion. You might have perceived that the main adversity is the creation and rendering of the table. Now that you are done, you might think, the rest of the task is a piece of cake.

This is when a surprise knocks you off. In most well-written stories, the [hero's journey](#) contains a twist after the hero defeats the enemy. This is when the hero realizes that things are a lot more difficult than he anticipated.

This is also the point when the hero needs to reverse engineer a prophecy to move forward. It is now time to reveal the prophecy: “One Event Handler to Rule Them All”.

## One Event Handler?!

I bet you were about to consider how you would implement one event handler for each button in the DOM. It is definitely feasible. Once you render the markup, you have to take care of dynamically adding the corresponding event listeners. You have to make sure you don't mess up event handling.

This seems to be a lot of unnecessary work. The stubborn hero could implement it like this:

```
const finishTask = (e) => {
 const taskId = e.target.dataset.id;
 tasks[taskId].finished = true;
 renderTasks(pomodoroTableBody, tasks);
}

const increasePomodoroDone = (e) => {
 const taskId = e.target.dataset.id;
 tasks[taskId].pomodoroDone += 1;
 renderTasks(pomodoroTableBody, tasks);
}

const deleteTask = (e) => {
 const taskId = e.target.dataset.id;
 tasks.splice(taskId, 1);
 renderTasks(pomodoroTableBody, tasks);
}

const addTaskEventListeners = () => {
```

```

const prefix = '.js-task-table-body .js-';
document.querySelectorAll(prefix + 'increase-pomodoro')
.forEach(button =>
 button.addEventListener('click', increasePomodoroDone)
);
document.querySelectorAll(prefix + 'task-done')
.forEach(button =>
 button.addEventListener('click', finishTask)
);
document.querySelectorAll(prefix + 'delete-task')
.forEach(button =>
 button.addEventListener('click', deleteTask)
);
}

const renderTasks = function(tBodyNode, tasks = []) {
 tBodyNode.innerHTML = tasks.map((task, id) => `

 ...
 `).join('');
 addTaskEventListeners();
}

```

Just imagine! If you store 100 tasks in your list, you add 300 event listeners each time you make one tiny modification to the table. The code is also very WET (We Enjoy Typing). After all, the first and the third row of the functions `increasePomodoroDone`, `finishTask`, and `deleteTask` are all the same. We also have to do three tedious `forEach` helpers and copy-paste the structure to add the event listeners, let alone adding the event listeners after each render. Who guarantees that we can't manipulate the DOM without calling the `renderTasks` function?

This is a bit too much. The structure is not clean enough, and it requires too much maintenance. Therefore, it is now time to consider our prophecy and start thinking.

## Event delegation

One event handler to rule them all. When a click on a DOM node happens, and an event handler is not defined on the node, the event handler defined on the closest parent node captures and handles the event.

To handle event propagation without the need for adding event listeners during runtime, we have to find an ancestor node that contains the buttons. This node is the table body node `.js-task-table-body`. Let's define our event listener there:

```
const handleTaskButtonClick = function(event) {
```

```

const classList = event.target.className;
const taskId = event.target.dataset.id;

// increase pomodoro count or finish task or delete task

renderTasks(pomodoroTableBody, tasks);
}

pomodoroTableBody.addEventListener('click', handleTaskButtonClick);

```

The click handler will stay in place throughout the whole lifecycle of the application.

The `event` itself belongs to the button we clicked. Therefore, we can easily extract the class attribute and the `data-id` attribute. `classList` contains all classes added to the node.

After performing the requested action, we have to re-render the table.

Let's see how to perform the actions. We have to determine if `classList` contains a class we are looking for. We can simply do it with a regex matching. If you are interested in more details on regexes, check out my post on [JavaScript regular expressions](#).

```

const finishTask = (tasks, taskId) => {
 tasks[taskId].finished = true;
}

const increasePomodoroDone = (tasks, taskId) => {
 tasks[taskId].pomodoroDone += 1;
}

const deleteTask = (tasks, taskId) => {
 tasks.splice(taskId, 1);
}

const handleTaskButtonClick = function(event) {
 const classList = event.target.className;
 const taskId = event.target.dataset.id;
 switch (true) {
 case /js-task-done/.test(classList):
 finishTask(tasks, taskId);
 break;
 case /js-increase-pomodoro/.test(classList):
 increasePomodoroDone(tasks, taskId);
 break;
 case /js-delete-task/.test(classList):
 deleteTask(tasks, taskId);
 break;
 }
 renderTasks(pomodoroTableBody, tasks);
}

```

```
pomodoroTableBody.addEventListener('click', handleTaskButtonClick);
```

Note we could have used `event.target.matches('.js-task-done')` instead of `/js-task-done/.test(classList)`. Both solutions are the same.

Notice that `finishTask`, `increasePomodoroDone`, and `deleteTask` are now DRY compared to their previous version.

The `renderTasks` function was also reverted to its original version because we don't have to add any event listeners after rendering.

Check out the [source code](#) on GitHub.

# Pomodoro App Markup and Styling Refactoring

You will now improve the appearance and functionality of the app created in the last lesson.

## Exercise:

Refactor the Pomodoro App from the previous exercise such that your tasks will be placed on cards, not table rows. Use the block-element-modifier syntax in your CSS and emphasize separation of concerns. Take care of the styling of the application as well as the functionality.

You have a free choice in your design decisions.

## Source code:

Use the [PomodoroTracker1](#) folder as a starting point. The end result is in [PomodoroTracker2](#).

## Imagination, life is your creation

This time, you are the freelancer, and you are supposed to drive all design decisions. If you don't know what you are doing, check out tools that have solved the same problem. To save your time, I suggest searching for screenshots of Trello and KanbanFlow boards. Don't worry; you don't have to implement a full board yet. We are just focusing on one column.

Let's start with creating the markup for our column:

```
<div class="task-column">
 <div class="task-column__header">Tasks</div>
 <div class="task-column__body js-task-column">
 <div class="task js-task" data-id="0">
 Write Article
 0 / 2 pomodori
 <div class="task__controls">

 ✔

 ➕

```

```

 </div>
</div>
</div>
```

Notice the block-element-modifier syntax. A container is connected to an element via `_`. We can only use one `_` in a class name, so instead of `task_controls_done`, we just used `task-controls_done`. We could attach modifiers to these classes with `--`. Block-element-modifier classes should not be referenced in our JavaScript code. They are for styling purposes only. This is how we achieve separation of concerns.

Let's style the elements.

```
.task-column {
 width: 20rem;
 background-color: #ccc;
 border: 1px #333 solid;
}

.task-column__header {
 width: 14rem;
 margin: 1rem 1rem 0 1rem;
 padding: 2rem;
 background-color: #777;
 color: #eee;
 text-align: center;
 font-size: 1.5rem;
}

.task-column__body {
 width: 16rem;
 margin: 0 1rem 1rem 1rem;
 padding: 0.8rem 1rem;
 background-color: #999;
 min-height: 2rem;
}

.task {
 width: 12rem;
 height: 3rem;
 margin: 0.5rem 1rem;
 background-color: #eee;
 padding: 1rem;
 user-select: none;
}

.task__name {
 display: block;
}

.task__pomodori {
```

```
 display: inline-block;
 float: left;
}

.task__controls {
 display: inline-block;
 float: right;
}

.task-controls__icon {
 cursor: pointer;
}
```

I will not get into the details of explaining each rule. Understanding CSS is a lot easier than JavaScript. You can reverse engineer each rule with some googling. Some minimal styling knowledge always comes in handy.

Let's connect the markup with the JavaScript code. First of all, let's delete the static tasks from the markup:

```
<div class="task-column">
 <div class="task-column__header">Tasks</div>
 <div class="task-column__body js-task-column">
 </div>
</div>
```

In the JavaScript code belonging to the previous example, let's replace `js-task-table-body` with `js-task-column` on line 3. Let's also replace the variable name `pomodoroTableBody` with `pomodoroColumn`.

```
const pomodoroColumn = document.querySelector('.js-task-column-body');
```

Don't forget to replace all occurrences of `pomodoroTableBody` with `pomodoroColumn` in the code.

We have one task left: let's rewrite the `renderTasks` function to generate the new markup structure:

```
const renderTasks = function(tBodyNode, tasks = []) {
 tBodyNode.innerHTML = tasks.map((task, id) => `
 <div class="task js-task" data-id="${id}">
 ${task.taskName}

 ${task.pomodoroDone} / ${task.pomodoroCount} pomodori

 <div class="task__controls">
 ${ task.finished ? 'Finished' :

 }
 </div>
 </div>
 `);
}
```

```
 data-id="${id}">\u{2714}
 <span class="task-controls__icon js-increase-pomodoro"
 data-id="${id}">\u{1f5d1}
 `)
 </div>
</div>
`).join('');
}
```

We are done.

There was not much JavaScript in this task. I included it, because it is important to emphasize that in frontend and full stack development, you need to be competent in refactoring markup and add some basic CSS. On some occasions, you also have to showcase your creativity and grit by taking the initiative and designing the interface for your applications.

# Persistence with Local Storage

We will keep track of the state in our Pomodoro app by storing it locally.

## Exercise:

Store the state of the application in local storage. Make sure the application state is reloaded once you refresh the page.

## Source code:

Use the [PomodoroTracker2](#) folder as a starting point. The end result is in [PomodoroTracker3](#).

## Solution:

Clone [PomodoroTracker2](#) from my GitHub repository as a starting point. Alternatively, you can use your own solution too. We will only modify the JavaScript code, `js/pomodoro.js`.

Local storage is straightforward. There is a `localStorage` variable in the global scope. `localStorage` may contain keys with string values that persist in your browser. This is client-side persistence, so your changes do not carry over to a different browser or computer.

As you can only use strings as values in the local storage, you have to stringify your object or array using `JSON.stringify`.

Let's write a function to save the application state to the local storage:

```
function saveState(tasks) {
 localStorage.setItem('tasks', JSON.stringify(tasks));
}
```



Once we retrieve the application state from the local storage, we have to parse it as an array. We will use `JSON.parse`:

```
function loadState() {
 return JSON.parse(localStorage.getItem('tasks')) || [];
}
```



If the application state was not saved before, we fall back to an empty array as a default value.

Try out the code a bit. First, add some tasks, finish a few of them, and complete some pomodori. Then save by executing this line in the console:

```
saveState(tasks)
```



Refresh your browser. You should see an empty tasks column. Now load your tasks and render your application:

```
loadState()
renderTasks(pomodoroColumn, tasks)
```



How do we know when to load the state? The answer is surprisingly simple. You load the state when you initialize your `tasks` variable. Replace the `[]` initial value with `loadState()`:

```
let tasks = loadState();
```



Don't forget to render your application after initialization:

```
let tasks = loadState();
const pomodoroForm = document.querySelector('.js-add-task');
const pomodoroColumn = document.querySelector('.js-task-column-body');

renderTasks(pomodoroColumn, tasks);
```



Make sure you avoid temporal dead zone issues with the `renderTask` function. If you declared it as

```
const renderTasks = (...) => { ... }
```



make it

```
function renderTasks(...){...}
```



instead.

Our last task is saving the state. When does it make sense to save our state? In theory, we could figure out where we call `renderTasks` and place the saving there.

The problem with this approach is that no-one guarantees that you won't forget saving if there was another occurrence of changing the `tasks` array and rendering it.

Therefore, I would rather bundle this responsibility with `renderTasks` to remind me of persistently saving the state whenever we render:

```
function renderTasks(tBodyNode, tasks = []) {
 tBodyNode.innerHTML = // ...
 saveState(tasks);
}
```



If you test the solution, you can see that everything appears correct. Are we done? Hell no! Our solution is very dangerous.

Why doesn't it make sense to place `saveState` inside `renderTasks`? Think about it.

It doesn't make sense because we violate the *single responsibility principle*. We bundle the hidden responsibility of saving the state into the responsibility of rendering tasks. This does not make sense.

Let's change this experience by packaging `renderTasks` and `saveState` inside another function. Without a better idea, I called it `saveAndRenderState`.

```
function saveAndRenderState(tBodyNode, tasks) {
 renderTasks(tBodyNode, tasks);
 saveState(tasks);
}

function renderTasks(tBodyNode, tasks = []) {
 tBodyNode.innerHTML = // ...
}
```



Our last task is to replace the two `renderTask` occurrences with

## saveAndRenderState :

```
const addTask = function(event) {
 event.preventDefault();
 const taskName = this.querySelector('.js-task-name').value;
 const pomodoroCount = this.querySelector('.js-pomodoro-count').value;
 this.reset();
 tasks.push({
 taskName,
 pomodoroDone: 0,
 pomodoroCount,
 finished: false
 });
 saveAndRenderState(pomodoroColumn, tasks);
}

// ...

const handleTaskButtonClick = function(event) {
 const classList = event.target.className;
 const taskId = event.target.dataset.id;

 /js-task-done/.test(classList) ?
 finishTask(tasks, taskId) :
 /js-increase-pomodoro/.test(classList) ?
 increasePomodoroDone(tasks, taskId) :
 /js-delete-task/.test(classList) ?
 deleteTask(tasks, taskId) :
 null;

 saveAndRenderState(pomodoroColumn, tasks);
}
```

We are now done with exercise 11.

# Pomodoro APP to Kanban Board

Using Kanban to categorize the tasks in our app according to days of the week.

## Exercise:

Make a Kanban board from the Pomodoro App with the following fixed columns:

- Done,
- Monday,
- Tuesday,
- Wednesday,
- Thursday,
- Friday,
- Later.

## Source code:

Use the [PomodoroTracker3](#) folder as a starting point. The end result is in [PomodoroTracker4](#).

## Solution:

Our app is becoming more and more useful after each exercise. These exercises symbolize what kinds of tasks you may get in an interview, but you also get the convenience of building on top of your existing code. If you go through all exercises at once, you may get the experience of solving a homework exercise.

First, let's add a div for the kanban board:

```
<div class="kanban-board js-kanban-board">
</div>
```



This div will contain our columns.

## Microtemplates

The column template needs to be reusable. We could copy-paste it in the html markup 6 times, but we prefer a **DRY** (Don't Repeat Yourself) solution.

Therefore, we will move the column markup to the JavaScript code in the form of a *microtemplate* function:

```
const columnTemplate = ({ header }) => ` Copy
 <div class="task-column">
 <div class="task-column__header">${ header }</div>
 <div class="task-column__body" js-${ header }-column-body"
 data-name="${ header }">
 </div>
 </div>
`;
```

{ header } is the shorthand of { header: header } in ES6. This header variable is inserted into the template. Notice that we just pass one object to the template, and if we need to add more properties, we just add it to the object. This is convenient because the order of the properties inside the object does not matter. **Destructuring** takes care of unwrapping these top-level properties.

We also changed the class of the column body to js-\${ header }-column-body .

Let's test drive the template:

```
document.querySelector('.js-kanban-board').innerHTML =
 columnTemplate({ header: 'task' }); Copy
```

Currently, the existing code only works with the task header, because the class of the task column body should match its previous fixed value.

Microtemplates make your code more readable. If you check the renderTasks function, it also contains a microtemplate inside a map function. There is room for improvement. Let's extract the template out:

```
const cardTemplate = ({ task, id }) => ` Copy
 <div class="task js-task" data-id="${id}">
 ${task.taskName}
 ...
```

```


 ${task.pomodoroDone} / ${task.pomodoroCount} pomodori

<div class="task__controls">
 ${ task.finished ? 'Finished' : `

 <span class="task-controls__icon js-task-done"
 data-id="${id}">\u{2714}
 <span class="task-controls__icon js-increase-pomodoro"
 data-id="${id}">\u{2795}
 `}
 <span class="task-controls__icon js-delete-task"
 data-id="${id}">\u{1f5d1}
 </div>
</div>
`;

```

We can call the `cardTemplate` microtemplate function inside `renderTasks`:

```

function renderTasks(tBodyNode, tasks = []) {
 tBodyNode.innerHTML = tasks.map((task, id) =>
 cardTemplate({ task, id })
).join('');
 saveState(tasks);
}

```

If you reload the application, it is still working.

## Kanban data structure

Earlier, there was only one task list. Now we have a list of tasks for seven columns. Beyond the column names, we also need to store the column header name. The initial value of a suggested data structure is as follows:

```

const getEmptyBoard = columnNames => columnNames.map(header => {
 return {
 header,
 tasks: []
 };
});

```



As our state changed, we will have to temporarily disable loading data from the local storage and generate static data instead:

```

let columns = [
 'Done',
 'Monday',

```



```
'Tuesday',
'Wednesday',
'Thursday',
'Friday',
'Later'
];

// let tasks = loadState();
let board = getEmptyBoard(columns);
```

Don't worry about renaming your variables even if your code breaks. Theoretically, you should have covered your code by tests to have a higher level of confidence with changes. This time, we will omit this feature to save space.

## Rendering the empty board

The old mechanism of rendering the board is not working anymore, therefore, it's time to remove it:

```
document.querySelector('.js-kanban-board').innerHTML =
 columnTemplate({ header: 'task' });
```



Let's create a function to render the template of the board without tasks:

```
const renderEmptyBoard = board => {
 document.querySelector('.js-kanban-board').innerHTML =
 board.map(columnTemplate).join('');
}
```



Once the value of the board is returned by `getEmptyBoard`, we can render it:

```
renderEmptyBoard(board);
```



We have to slightly modify the CSS to make sure the columns appear next to each other, regardless of the size of the window:

```
.task-column {
 width: 20rem;
 background-color: #ccc;
 border: 1px #333 solid;
 display: inline-block;
 vertical-align: top;
}

.kanban-board {
```



```
.kanban-board {
 width: calc(140rem + 40px);
}
```

Yes, as a frontend developer, it is worth knowing some CSS, including the `inline-block` display style and the `calc` function in CSS.

If you open the developer tools, you can see a `TypeError`. This is because event listener administration is not working anymore. We originally tried to listen to events in one column, and now we have seven.

Therefore, we can extend listening to events in the whole pomodoro table. Replace the line

```
const pomodoroColumn = document.querySelector('.js-task-column-body');
```

with

```
const kanbanBoard = document.querySelector('.js-kanban-board');
```

I also suggest moving up the DOM references to the top of the file and replacing all occurrences of the `.js-kanban-board` query selector with the variable:

```
const kanbanBoard = document.querySelector('.js-kanban-board');
const pomodoroForm = document.querySelector('.js-add-task');

// ...

kanbanBoard.innerHTML = columnTemplate({ header: 'task' });
```

## Adding tasks

Let's take care of adding tasks. We will need to specify the column of the task. We will hard code this information in this lesson. Add a dropdown list inside the form with the handle class `js-add-task`:

```
<select name="column-chooser"
 class="js-column-chooser">
 <option value="1">Monday</option>
 <option value="2">Tuesday</option>
 <option value="3">Wednesday</option>
 <option value="4">Thursday</option>
 <option value="5">Friday</option>
 <option value="4">Later</option>
```

```
</select>
```

We can make use of this value in the `addTask` function. We can read the `dayIndex` from the form, and then insert the created task in its proper slot. Notice that `tasks` is now a two-dimensional array inside the `board` global state. There is one more change in the last line: we removed the parameters of `saveAndRenderState`. This is because we already have a handle of the contents of the board and the container node, therefore, using additional arguments would just complicate things in the code.

```
const addTask = function(event) {
 event.preventDefault();
 const taskName =
 this.querySelector('.js-task-name').value;
 const pomodoroCount =
 this.querySelector('.js-pomodoro-count').value;
 const dayIndex =
 this.querySelector('.js-column-chooser').value;
 this.reset();
 tasks[dayIndex].push({
 taskName,
 pomodoroDone: 0,
 pomodoroCount,
 finished: false
 });
 saveAndRenderState();
}
```



Let's also update the `saveAndRenderState` function:

```
function saveAndRenderState() {
 renderTable();
 saveState();
}
```



Once again, we removed the parameters, as they are redundant. Everything is accessible globally.

In the `renderTable` function, all we need to do is call the `renderTasks` function for each column in the Kanban board, once an empty table is rendered.

```
function renderTable() {
 renderEmptyBoard(board);
 board.map(column => {
 renderTasks(
 document.querySelector(` .js-${ column.header }-column-body`),
 column.tasks
);
 });
}
```



```
);
 }
}
```

If you run the application, you can see that now you can add cards to the proper columns.

## Event handling

Unfortunately, the buttons on the cards are not working. Let's correct event handling. First of all, it comes handy if we added a column index in the card markup:

```
const cardTemplate = ({ task, id, columnIndex }) => `

<div class="task js-task"
 data-id="${id}"
 data-column-index="${columnIndex}">
 ${task.taskName}

 ${task.pomodoroDone} / ${task.pomodoroCount} pomodori

 <div class="task__controls">
 ${ task.finished ? 'Finished' :
 <span class="task-controls__icon js-task-done"
 data-id="${id}"
 data-column-index="${columnIndex}">\u2714
 <span class="task-controls__icon js-increase-pomodoro"
 data-id="${id}"
 data-column-index="${columnIndex}">\u2795
 }
 <span class="task-controls__icon js-delete-task"
 data-id="${id}"
 data-column-index="${columnIndex}">\u1f5d1
 </div>
</div>
`;
```



Remember, the value of the `data-column-index` attribute can be retrieved using the `dataset.columnIndex` property of the DOM node.

We have to make a few changes in the JavaScript code too:

- now we listen to events on the whole Kanban board,
- we have to remove the original first argument of `finishTask`, `increasePomodoroDone`, and `deleteTask`, because instead of a task list, we now have a full board,
- we have to read the `columnIndex` from the card dataset, and pass it to the above three functions,
- in the implementation of `finishTask`, `increasePomodoroDone`, and

`deleteTask`, the lookup of the clicked task changes,

- the parametrization of `saveAndRenderState` should be removed.

```
const finishTask = (taskId, columnIndex) => {
 board[columnIndex].tasks[taskId].finished = true;
}

const increasePomodoroDone = (taskId, columnIndex) => {
 board[columnIndex].tasks[taskId].pomodoroDone += 1;
}

const deleteTask = (taskId, columnIndex) => {
 board[columnIndex].tasks.splice(taskId, 1);
}

const handleTaskButtonClick = function(event) { window.t = event.target;
 const classList = event.target.className;
 const taskId = event.target.dataset.id;
 const columnIndex = event.target.dataset.columnIndex;

 /js-task-done/.test(classList) ?
 finishTask(taskId, columnIndex) :
 /js-increase-pomodoro/.test(classList) ?
 increasePomodoroDone(taskId, columnIndex) :
 /js-delete-task/.test(classList) ?
 deleteTask(taskId, columnIndex) :
 null;

 saveAndRenderState();
}

kanbanBoard.addEventListener('click', handleTaskButtonClick);
```

If you test the application, you can see that everything is working as expected.

## Repairing the local storage and initializing the application

Our final task is to automatically save and load the state. We have to take care of the whole board, not just an array of states:

```
function saveState(tasks) {
 localStorage.setItem('board', JSON.stringify(board));
}

function loadState() {
 return JSON.parse(localStorage.getItem('board')) ||
 getEmptyBoard(columns);
}
```

Then replace the line

```
let board = getEmptyBoard(columns);
renderEmptyBoard(board);
```

with

```
let board = loadState();
renderTable();
```



The state has been successfully repaired.

You can see how many things had to be changed to implement the requirements. If you ever get stuck, it makes sense to modularize your application in your mind, and completely rewrite some parts of the application.

You might have noticed that the form adding new tasks is not styled properly. We will take care of this problem in the next exercise.

# Adding Tasks to Columns

Hope you've been following everything up till now. Next, we'll add the functionality of creating new tasks!

## Exercise:

Create a  button inside the column header. When this  is pressed, open a modal window with the same form fields as the ones at the bottom of the page. When submitting the form, the new card should appear inside the column where it was created. Use the [rmodal](#) library!

As the form at the bottom of the page is now useless, remove it.

## Source code:

Use the [PomodoroTracker4](#) folder as a starting point. The result is in [PomodoroTracker5](#).

## Solution:

This is a user experience exercise involving a third party library. Let's install rmodal using npm. Navigate to the folder of the application, then execute the following command:

```
npm i rmodal --save
```



This command installs Bootstrap using the Node Package Manager, and stores it inside the `node_modules` folder. We can find the rmodal JavaScript and CSS file in the `dist` folder of `rmodal`. Let's add them to the head of our `index.html` file:

```
<link rel="stylesheet"
 href="node_modules/rmodal/dist/rmodal.css" type="text/css" />
<script type="text/javascript"
 src="node_modules/rmodal/dist/rmodal.js"></script>
```



Let's prepare the form in the HTML file by wrapping the form with three nested `div` elements. The `div` element classes and the `form-horizontal` class can be found in the documentation.

As we are planning to open the modal window from the column we want to add the card to, the column chooser dropdown has to be removed from the form. We will, therefore, change the column chooser dropdown list to a hidden input field.

```
<div id="form-modal" class="modal">
 <div class="modal-dialog animated">
 <div class="modal-content">
 <form class="js-add-task form-horizontal"
 action="javascript:void(0)">
 <input type="text"
 name="task-name"
 class="js-task-name"
 placeholder="Task Name" />
 <select name="pomodoro-count"
 class="js-pomodoro-count">
 <option value="1">1</option>
 <option value="2">2</option>
 <option value="3">3</option>
 <option value="4">4</option>
 </select>
 <input type="hidden"
 class="js-column-chooser"
 name="column-chooser" />
 <input type="submit" class="js-add-task-submit" />
 <button class="js-add-task-cancel">Cancel</button>
 </form>
 </div>
 </div>
</div>
```

Notice the `js-add-task-submit` and `js-add-task-cancel` classes. The first submits the form, adds the task, and closes the modal window. The second simply closes the modal window without any changes.

We also need to add some manual styling. Let's remove the old form styles:

```
form {
 padding: 3rem;
 background-color: #ddd;
}
```

To override the default styles of the modal window, we have to target the following class:

```
.modal .modal-dialog {
 position: fixed !important;
 width: 400px !important;
 top: 50px;
 left: calc(50% - 250px);
 padding: 50px;
 background-color: #eee;
 border: 3px #444 solid;
}

.modal .modal-dialog input {
 margin: 5px;
}
```

The fixed position makes sure that it appears on top of our screen in the coordinates specified by `top` and `left`, regardless of where we scroll horizontally.

Now it's time to handle the form from the JavaScript file.

```
const modalAddTaskCancel =
 document.querySelector('.js-add-task-cancel');

const addCardModal = new RModal(
 document.getElementById('form-modal'), {}
);

const columnTemplate = ({ header }) => `

<div class="task-column">
 <div class="task-column__header">
 ${ header }
 <span class="task-controls__icon js-task-create"
 data-column-index="${header}">\u{2795}
 </div>
 <div class="task-column__body js-${ header }-column-body"
 data-name="${ header }">
 </div>
</div>
`;

const openCreateTaskModal = columnName => {
 const columnIndex = columns.indexOf(columnName);
 document.querySelector('.js-column-chooser').value = columnIndex;
 addCardModal.open();
}

modalAddTaskCancel.addEventListener('click', e => {
 e.preventDefault();
 addCardModal.close();
});
```

First, we get a handle for the cancel button so that we can add a click listener to it later.

The `addCardModal` constant contains the `RModal` instance created using our third-party library.

To open the modal window, we have to place a `js-task-create` icon to the header template. Whenever we click the `+` button on the header, the modal appears on a screen. The modal window will be opened by the `openCreateTaskModal` function, which will be called from the click event handler callback.

Finally, the modal window will be closed from by the click event handler attached to `modalAddTaskCancel`. Notice that we called the `preventDefault` method of the click event. This is because we want to avoid submitting the form.

The `handleTaskButtonClick` event handler monitors click events on the whole pomodoro board. Therefore, we can just call `openCreateTaskModal` from there:

```
const handleTaskButtonClick = function(event) {
 const classList = event.target.className;
 const taskId = event.target.dataset.id;
 const columnIndex = event.target.dataset.columnIndex;

 /js-task-done/.test(classList) ?
 finishTask(taskId, columnIndex) :
 /js-increase-pomodoro/.test(classList) ?
 increasePomodoroDone(taskId, columnIndex) :
 /js-delete-task/.test(classList) ?
 deleteTask(taskId, columnIndex) :
 /js-task-create/.test(classList) ?
 openCreateTaskModal(columnIndex) :
 null;

 saveAndRenderState();
}
```

Finally, in the `addTask` submit event callback, we have to close the modal window:

```
const addTask = function(event) {
 event.preventDefault();
 addCardModal.close();
 // ...
}
```

As `.task.selected` is a stronger selector than `.task`, the background color will be overridden automatically to the value inside `.task.selected`.

# Selecting Cards

Add the functionality of selecting a particular task in your app.

Hint: a 'selected' property.

## Exercise:

Make your cards selectable by clicking on them. Whenever you select a card, you have to deselect all other cards.

## Source code:

Use the [PomodoroTracker5](#) folder as a starting point. The end result is in [PomodoroTracker6](#).

## Solution:

The main purpose of this exercise is to check whether the candidate has a tendency of overcomplicating things. The solution of this exercise is fairly simple.

A `selected` class will take care of the styling of the selected task card:

```
.task.selected {
 background-color: #cce;
}
```

We can model the selected state by adding a `selected` property to the tasks. The absence of the property means that the card is not selected. When the property has a truthy value, the card is selected. Given only one card can be selected at a time, we select a card by deselecting all other cards, and then changing the selected state of the new card to `true`:

```
const deselectAllTasks = () => {
 for (let card of board) {
```

```
 for (let column of board) {
 for (let task of column.tasks) {
 delete task.selected;
 }
 }

const selectTask = (taskId, columnIndex) => {
 deselectAllTasks();
 board[columnIndex].tasks[taskId].selected = true;
}
```

Notice that we use the `board` global state, and we use `for...of` loops to drill down to the task level.

We are almost done. We only need to call the `selectTask` function in the click event handler. You might remember that event handling was implemented in the `handleTaskButtonClick` function.

```
const handleTaskButtonClick = function({ target }) {
 const classList = target.className;
 const taskId = target.dataset.id;
 const columnIndex = target.dataset.columnIndex;

 /js-task-done/.test(classList) ?
 finishTask(taskId, columnIndex) :
 /js-increase-pomodoro/.test(classList) ?
 increasePomodoroDone(taskId, columnIndex) :
 /js-delete-task/.test(classList) ?
 deleteTask(taskId, columnIndex) :
 /js-task-create/.test(classList) ?
 openCreateTaskModal(columnIndex) :
 /js-task/.test(classList) ?
 selectTask(taskId, columnIndex) :
 null;

 saveAndRenderState();
}
```

We did two things in the code. First, we simplified the `event.target` reference to `target` by applying destructuring in the function argument of `handleTaskButtonClick`. The code `{ target }` unwraps the event object, reads its `target` property, and makes it available.

The second change is that we are looking for the `/js-task/` pattern to identify when we click on a card.

If you test the application, you can conclude that sometimes clicking succeeds, while other times, you cannot change the selected state of cards. This is because you have to click on the card `li` with the `js-task` reference class.

because you have to click on the card `div` with the `.js-task` reference class. There are other `span` elements inside the card starting with `task_`. Once you click on those elements, nothing happens. Notice that these `div` elements are direct children of the card. Therefore, we can conclude that if the `target` DOM node has a class name starting with `task_`, we have to select the parent node of the card:

```
const handleTaskButtonClick = function({ target }) {
 if (/task_/.test(target.className)) {
 target = target.parentNode;
 }
 const classList = target.className;
 const taskId = target.dataset.id;
 const columnIndex = target.dataset.columnIndex;

 /js-task-done/.test(classList) ?
 finishTask(taskId, columnIndex) :
 /js-increase-pomodoro/.test(classList) ?
 increasePomodoroDone(taskId, columnIndex) :
 /js-delete-task/.test(classList) ?
 deleteTask(taskId, columnIndex) :
 /js-task-create/.test(classList) ?
 openCreateTaskModal(columnIndex) :
 /js-task/.test(classList) ?
 selectTask(taskId, columnIndex) :
 null;

 saveAndRenderState();
}
```

If you test the application, everything works smoothly. There is only one major mistake in the code, namely, we violated separation of concerns. The `task_` prefixed classes are supposed to be used for styling only, and we attached functionality to them. What happens if a different styling team decides on renaming these classes? Most likely, our code will stop working.

Therefore, we have to add a `js-` prefixed class to the nodes we want to match in the regular expression. The class `js-task-child` will do:

```
const cardTemplate = ({ task, id, columnIndex }) => `
<div class="task js-task ${ task.selected ? 'selected' : '' }"
 data-id="${id}"
 data-column-index="${columnIndex}">

 ${task.taskName}

 ${task.pomodoroDone} / ${task.pomodoroCount} pomodori

 <div class="task__controls js-task-child">
```

```

`{ task.finished ? 'Finished' :
 <span class="task-controls__icon js-task-done"
 data-id="${id}"
 data-column-index="${columnIndex}">\u{2714}
 <span class="task-controls__icon js-increase-pomodoro"
 data-id="${id}"
 data-column-index="${columnIndex}">\u{2795}
}
<span class="task-controls__icon js-delete-task"
 data-id="${id}"
 data-column-index="${columnIndex}">\u{1f5d1}
</div>
</div>
`;

```

We have to apply the corresponding change in the `handleTaskButtonClick` function as well:

```

const handleTaskButtonClick = function({ target }) {
 if (/js-task-child/.test(target.className)) {
 target = target.parentNode;
 }
 const classList = target.className;
 const taskId = target.dataset.id;
 const columnIndex = target.dataset.columnIndex;

 /js-task-done/.test(classList) ?
 finishTask(taskId, columnIndex) :
 /js-increase-pomodoro/.test(classList) ?
 increasePomodoroDone(taskId, columnIndex) :
 /js-delete-task/.test(classList) ?
 deleteTask(taskId, columnIndex) :
 /js-task-create/.test(classList) ?
 openCreateTaskModal(columnIndex) :
 /js-task/.test(classList) ?
 selectTask(taskId, columnIndex) :
 null;

 saveAndRenderState();
}

```

Now we are done. Even though the task was simple, there were a lot of pitfalls that had to be avoided.

# Dragging and Dropping Cards

Shift cards around by dragging and dropping.

## Exercise:

Make it possible to drag and drop cards.

**Source code:** Use the [PomodoroTracker6](#) folder as a starting point. The result is in [PomodoroTracker7](#).

**Solution:** There is a `draggable` attribute in HTML5. When using this attribute, you will be able to automatically move `draggable` elements by holding down the mouse button. Let's make our selected card draggable:

```
const cardTemplate = ({ task, id, columnIndex }) => ` 
 <div class="task js-task ${ task.selected ? 'selected' : '' }"
 data-id="${id}"
 draggable="${ task.selected ? "true" : "false" }"
 data-column-index="${columnIndex}">
 ...
 </div>
`;
```

So far we have only added one attribute, but as soon as you start dragging the selected card, you can see that a higher opacity version of the selected card follows the mouse cursor as long as you hold the mouse button down.

The question arises, how can you find out about the `draggable` html5 attribute during an interview? The answer is simple: Google is your friend. Remember, in most interview situations you will be able to use Google. Also remember, that with the series of exercises on the Pomodoro App, we are simulating a homework assignment, where you have all the time in the world to research tricks worth using.

Back to the exercise. Our next task is to place the card to the proper column

upon dropping it.

There is only one twist when it comes to dropping an element. The `drop` event only fires if we cancel the `dragover` and `dragenter` events. For more details, check [this StackOverflow post](#). Therefore, when adding a drop listener, make sure you prevent the default action on the corresponding `dragenter` and `dragover` events.

As we re-render the table quite often, but we never re-render the table while we are dragging elements, it makes sense to attach the event listeners in the `renderEmptyBoard` function:

```
const renderEmptyBoard = board => {
 kanbanBoard.innerHTML =
 board.map(columnTemplate).join('');
 addColumnDropListeners();
}

function dropCard(e) {
 console.log('drop', e);
}

function addColumnDropListeners() {
 const columns = document.querySelectorAll('.task-column');
 for (let column of columns) {
 column.addEventListener('dragover', e => {
 e.preventDefault();
 });
 column.addEventListener('dragenter', e => {
 e.preventDefault();
 });
 column.addEventListener('drop', dropCard);
 }
}
```



Once you start to drag and drop the selected card of the Kanban board, you can see that the drop event is fired. If you dig a bit deeper into the event object, you can find a `path` array, containing the hierarchy of fields the card was dropped onto. For instance, after dropping the card to the first column, you can see the following hierarchy in the array:

```
Array(7)
0: div.task-column__body.js-Done-column-body
1: div.task-column
2: div.task-board
```

```
2: div.kanban-board.js-kanban-board
3: body
4: HTML
5: document
6: Window {postMessage: f, blur: f, focus: f, close: f, frames: Window, ...}
```

Our task is to find the DOM node that has a `.js-XXX-column-body` class, where `XXX` is one of the seven column names.

Alternatively, we could also use the `srcElement` property, which contains the DOM node where we dropped the card. However, this source element can be the column, the column header, a card, or even a label on the card. This method is not reliable to identify where we are, because we cannot always place a label on each `div` or `span` we create.

Therefore, we will stick to the `path` array. Our rule is that an element in the `path` has to be the column body. If we drop the card outside the column body, we ignore it.

```
function getDroppedColumnName(e) {
 for (let node of e.path) {
 // extraction of the column name comes here
 }
}
```

We can get the class list string from a DOM node using `node.classList.value`. In our hierarchy, `document` and the global `Window` object do not have a `classList`, therefore we have to test if the `classList` property exists at all.

```
function getDroppedColumnName(e) {
 for (let node of e.path) {
 if (typeof node.classList === 'undefined') break;
 console.log(node.classList.value);
 // extraction of the column name comes here
 }
}
```

Now we can process the `classList` string. Let's see an example string:

```
task-column__body js-Done-column-body
```

We need to extract `Done` encapsulated by a `js-` prefix, and a `-column-body` suffix. We cannot rely on the order of the classes, so other classes may or may

not precede `js-Done-column-body`. We will describe the following strategy using a regular expression:

- first, we capture as many characters as possible in a greedy way
- then we capture the `js-` string
- then we do a match of a sequence of at least one arbitrary character. We capture this sequence in a capture group, using parentheses
- then we match `-column-body`

The regular expression is `.*js-(.+)-column-body/`. Notice `(.+)`. The parentheses make sure the value of `+` is recorded and returned by the `match` method of the regular expression.

The result of the match is either `null`, or a data structure containing the full match at index `0`, and the captured value of `+` in the expression at index `1`. Therefore, if `match` is not null, `match[1]` becomes the column name we are looking for:

```
function getDroppedColumnName(e) {
 for (let node of e.path) {
 if (typeof node.classList === 'undefined') break;
 const match =
 node.classList.value.match(/.*js-(.+)-column-body/);
 if (match && typeof match[1] === 'string') {
 console.log('columnName', match[1]);
 }
 }
}
```

You don't have to know regular expressions to solve this exercise. You could also use the JavaScript `split` method. However, regular expressions come in handy in your career; I highly recommend learning them. If you need help, check out [my book and video course on JavaScript regular expressions](#).

Let's now implement the `dropCard` event handler.

```
function dropCard(e) {
 const newColumn = getDroppedColumnName(e);
 if (typeof newColumn === 'string') {
 moveSelectedCard(newColumn);
 }
}
```

If we get the column name, we can move the selected card to the new column. The only puzzle piece in this task is the implementation of the `moveSelectedCard` function. We have the following tasks to implement:

- Find the selected card.
- If the selected card is in the same column as `toColumn`, we do nothing.
- Otherwise, we remove the selected card from the `tasks` array of the column and add it to the task list of the column described by the `toColumn` header
- if there is a change in state, we have to re-render the application and save the state to the local storage

The implementation looks as follows:

```
function moveSelectedCard(toColumn) {
 for (let { header, tasks } of board) {
 if (header === toColumn) continue;
 for (let i = 0; i < tasks.length; ++i) {
 if (tasks[i].selected) {
 const selectedTask = tasks[i];
 tasks.splice(i, 1);
 let columnIndex = columns.indexOf(toColumn);
 board[columnIndex].tasks.push(selectedTask);
 saveAndRenderState();
 }
 }
 }
}
```



The only surprising element may be the destructuring inside the `for..of` operator:

```
for (let { header, tasks } of board) {
}
```



The code snippet above is the same as

```
for (let column of board) {
 { header: header, tasks: tasks } = column;
}
```



which is in turn equivalent to

```
for (let column of board) {
 let header = column.header;
 let tasks = column.tasks;
}
```



If you execute the application, you can enjoy the brand new experience of drag and dropping the selected card to any column you want.

# Stopwatch

A countdown timer with 'start', 'pause' and 'reset' functionality.

## Exercise:

Create a stopwatch that counts down from a given number of seconds in the format `mm:ss`. Make it possible to `start`, `pause`, and `reset` the countdown. Make sure you can pass a callback function to the timer that is called when the displayed value is updated.

## Solution:

The main part of this exercise boils down to modeling. If you find the right model, your life will be easy. If you use the wrong model, implementation will not only be hard, but the stopwatch might not reflect reality. Let's define the state space of the application:

- `countdownInitialValue`: the number of seconds initially set on the timer when instantiating it. When we reset the timer, its value will be set to this value.
- `secondsLeft`: the current value of the countdown timer in seconds

How do we make time pass?

The easy part of the answer is that we need to use the `setInterval` function.

`setInterval( callback, delay )` executes callback every `delay` milliseconds. Therefore, in theory, we could come up with the following solution:

```
setInterval(() => {
 this.secondsLeft -= 1;
 if (this.secondsLeft === 0) { ... }
 // ...
}, 1000);
```

Surprisingly, this approach may be highly inaccurate. When resources are scarce, the `setInterval` call is delayed. Small delays add up throughout the ten-minute countdown. Imagine that you do a resource-intensive task, and your browser might completely lose focus and priority in the background. It may happen that in five seconds, your callback function is only called twice, resulting in two-second progress instead of five.

If you want accurate results, you can make use of the fact that `Date.now()` returns the current timestamp, yielding the number of milliseconds passed since January 1st, 1970. The battle plan is as follows:

- We record the `startTimestamp` when we start the clock.
- Whenever we are inside the `setInterval` callback, we retrieve the current timestamp and update `secondsLeft`.
- The more often we run the `setInterval` call, the faster our results get updated. However, the results are independent of the `delay` value used in the `setInterval` call.
- We pause the countdown by terminating the `setInterval` call.
- We resume at `secondsLeft` once we continue the countdown. Note we might ignore up to 999 milliseconds of elapsed time with this approach.
- When resetting the counter, we equate `secondsLeft` to the initial value supplied in the constructor.

We will create an ES6 class to encapsulate all the necessary operations of the stopwatch:

```
class Timer {
 constructor(countdownInitialValue, displayTimeCallback) {
 this.countdownInitialValue = countdownInitialValue;
 this.secondsLeft = countdownInitialValue;
 this.interval = null;
 this.displayTimeCallback = displayTimeCallback;
 this.displayTimeCallback(this.toString());
 }
 toString() {
 const minutes = Math.floor(this.secondsLeft / 60);
 const seconds =
 ('' + (this.secondsLeft % 60))
 .padStart(2, '0');
 return `${minutes}:${seconds}`;
 }
 start() {
```

```

start() {
 let startTimestamp = Date.now();
 let startSeconds = this.secondsLeft;
 this.interval = setInterval(() => {
 let oldSecondsLeft = this.secondsLeft;
 let secondsPassed =
 Math.floor((Date.now() - startTimestamp) / 1000);
 this.secondsLeft =
 Math.max(0, startSeconds - secondsPassed);
 if (this.secondsLeft < oldSecondsLeft) {
 this.displayTimeCallback(this.toString());
 }
 if (this.secondsLeft == 0) {
 clearInterval(this.interval);
 }
 }, 100);
}
pause() {
 if (typeof this.interval === 'number') {
 clearInterval(this.interval)
 }
}
reset() {
 this.pause();
 this.secondsLeft = this.countdownInitialValue;
 this.displayTimeCallback(this.toString());
}
}

const timer = new Timer(61, console.log);

```



The constructor initializes the state of the application and sends the initial value to the callback. The `toString` method converts the `secondLeft` property to `mm:ss` format, making sure that `ss` has a leading zero. Notice the ES2017 `padStart` function.

The `start` method implements the stopwatch algorithm. Both the `pause` and the `reset` method stops the clock, and `reset` moves the application state to its initial value. When we press `reset`, we also have to call the `display` callback once.

You may think we are done with the exercise. Are we?

In every good exercise, there is a twist. Before reading any further, find a bug in the above implementation!

Welcome back! If you haven't done the exercise, I encourage you to go back and do it, because this is how you get better.

If you are still reading, congratulations, you have found a bug!

My solution is that after starting the clock twice and pausing it, the clock is still running. We will, therefore, guard the `start` method in the following way: if `this.interval` is a number, we will not start the clock, as it is already started:

```
start() {
 if (typeof this.interval === 'number') {
 return;
 }

 let startTimestamp = Date.now();
 let startSeconds = this.secondsLeft;
 this.interval = setInterval(() => {
 let oldSecondsLeft = this.secondsLeft;
 let secondsPassed = Math.floor((Date.now() - startTimestamp) / 1000);
 this.secondsLeft = Math.max(0, startSeconds - secondsPassed);

 if (this.secondsLeft < oldSecondsLeft) {
 this.displayTimeCallback(this.toString());
 }

 if (this.secondsLeft == 0) {
 clearInterval(this.interval);
 }
 }, 100);
}
```

The `reset` method executes `pause`, so we have to make sure that `pause` sets `this.interval` to a non-numeric value after clearing the interval. Therefore, we have to add a line to set `this.interval` to `null`.

```
pause() {
 if (typeof this.interval === 'number') {
 clearInterval(this.interval);
 this.interval = null;
 }
}
```

Now we can test the code and it works even if we started the clock twice.

NOW we can test the code and it works even if we started the clock twice.

# Insert the Timer Module in the Pomodoro App

In order to integrate the timer into our application, we'll need to import it.

## Exercise:

Wrap the stopwatch timer implemented in Exercise 15 in an ES6 module and import it in the Pomodoro App.

## Source code:

Use the [PomodoroTracker7](#) folder as a starting point. The end result is in [PomodoroTracker8](#).

## Solution:

This is an easy exercise in case you know how to use ES6 modules.

All you need to do is write `export default` in front of the class name:

```
export default class Timer {
 //...
}
```



Save the code in `Timer.js`, and place the JavaScript file in the same folder where your `pomodoro.js` file is. In your `index.html` file, reference `pomodoro.dist.js` instead of `pomodoro.js` in the script tag:

```
<script src="js/pomodoro.dist.js"></script>
```



When importing the `Timer` in another file, we have to prepend the statement:

```
import Timer from './Timer';
```



Let's also add some example code in the `pomodoro.js` file to test whether the `Timer` constructor function is accessible. We will count down from `25`

minutes, and display the results in the developer tools console.

```
new Timer(25 * 60, console.log).start();
```

in `pomodoro.js`. We need some setup work though, to make the browser understand these statements. Although there are multiple options to transpile ES6 modules, we will use the approach you learned in ES6 in Practice: we will install Webpack.

```
npm install webpack webpack-cli --save-dev
npm install babel-loader babel-core --save-dev
npm install babel-preset-es2015 babel-preset-stage-2 --save-dev
```

Once the installation is done, let's create a `webpack.config.js` file:

```
module.exports = {
 entry : './js/pomodoro.js',
 output : {
 path : __dirname,
 filename : 'pomodoro.dist.js'
 },
 module : {
 rules: [{
 test : /\.js$/,
 loader : 'babel-loader',
 exclude: /node_modules/,
 query: {
 presets: [
 'es2015',
 'stage-2'
]
 }
 }]
 }
};
```

If you are using an earlier version of webpack, you need to replace `rules` with the name `loaders`.

Let's place some code in our `package.json` file so that we can call Webpack from `npm`:

```
"scripts": {
 "webpack": "webpack",
 "webpack:watch": "webpack --watch"
},
```

Now that the setup is done, we can run `npm run webpack`.

After creating the distribution file, you can load `index.html` in your browser. Open the developer tools, and check if you have access to the `Timer` class in the console. If you have access to `Timer`, you have finished this exercise successfully.

# Countdown Timer Integration in the Pomodoro App

The final step of the Pomodoro App: Add a timer to the selected task.

## Exercise:

Create a form displaying a timer counting down from **25** minutes in the format of **mm:ss**. Display the name of the task above the timer. Once the timer reaches **00:00**, revert to **25:00**, increase the pomodoro count by **1**, and continue counting down.

Place three links below the timer: **Start**, **Pause**, and **Reset**.

When a card is selected, **25:00** appears on the timer, and the timer stops. You can start the timer by clicking the **Start** link. When the **Pause** link is clicked, the timer is suspended, keeping its current value. **Reset** moves the timer back to **25:00**.

**Source code:** Use the [PomodoroTracker8](#) folder as a starting point. The result is in [PomodoroTracker9](#).

**Solution:** We are in the finish line with the Pomodoro App. We only need a timer.

First, let's integrate a static timer in the app that stays visible even if we scroll. This is an HTML/CSS task. Add this markup to the **index.html** file:

```
<div class="js-timer timer">
 <div class="js-selected-task-label timer__task">
 Select a Task
 </div>
 <div class="js-time-remaining timer__time">25:00</div>
 <div class="js-links timer__links">

 ▶

 ⏸

 ⏹

 </div>
</div>
```

```
◹

</div>
</div>
```

We can position the div on the bottom-left part of our viewport and make it stick by applying `position:fixed` in the stylesheet and adding the corresponding coordinates. Add these rules to the `styles.css` file:

```
.timer {
 position: fixed;
 bottom: 0px;
 height: 150px;
 width: 300px;
 background-color: #ddd;
 color: #333;
 font-size: 2rem;
}
```

Also add styles for the play, pause, and stop buttons:

```
.timer__control:hover {
 cursor: pointer;
 color: red;
}
```

We are keeping the styles minimalistic for now.

Let's integrate the clock. We will instantiate the `Timer` class and set its default to `25` minutes. We have to pass the timer a callback function that updates the countdown in the DOM. Due to the separation of concerns, we have to use the corresponding `js-` prefixed class as a handle to get the DOM element containing the remaining time.

```
const updateTime = newTime => {
 document.querySelector('.js-time-remaining').innerHTML = newTime;
}
const timer = new Timer(25 * 60, updateTime);
```

To play around with the clock, let's define some event handlers for the buttons:

```
document.querySelector('.js-start-timer')
 .addEventListener('click', () => timer.start());
document.querySelector('.js-pause-timer')
 .addEventListener('click', () => timer.pause());
```

```
document.querySelector('.js-stop-timer')
 .addEventListener('click', () => timer.reset());
```

If you test the application, you can see that the buttons work as expected.

Our next task is to handle the card selection. Once we select or deselect a card, we do the following:

- Stop the timer if it is running,
- Update the name of the task above the timer or display `Select a Task` in case no cards are selected.

We will perform these updates in two places: during initialization and after selecting a task.

First, we will add a `setupTimer` call after the instantiation of the timer and the corresponding event handlers.

```
const updateTime = newTime => {
 console.log(newTime);
 document.querySelector('.js-time-remaining').innerHTML = newTime;
}
const timer = new Timer(25 * 60, updateTime);
document.querySelector('.js-start-timer')
 .addEventListener('click', () => timer.start());
document.querySelector('.js-pause-timer')
 .addEventListener('click', () => timer.pause());
document.querySelector('.js-stop-timer')
 .addEventListener('click', () => timer.reset());
setupTimer();
```

Second, we will also place a `setupTimer` call in the `selectTask` function.

```
const selectTask = (taskId, columnIndex, target) => {
 deselectAllTasks();
 board[columnIndex].tasks[taskId].selected = true;
 setupTimer();
}
```

The implementation of `setupTimer` depends on a small function that returns the name of the selected task. Although we could have gotten this information more easily by using `taskId` and `columnIndex` in the `selectTask` function, it just adds some code complexity for no reason. Note that we are wasting the resources of the client, and we are talking about microseconds if not nanoseconds. Any code optimization here counts as productive

procrastination or perfectionism. You gain a lot more by keeping the solution simple.

```
const getSelectedTaskName = () => {
 for (let { tasks } of board) {
 for (let i = 0; i < tasks.length; ++i) {
 if (tasks[i].selected) {
 return tasks[i].taskName;
 }
 }
 }

 return '';
}

const setupTimer = () => {
 timer.reset();
 document.querySelector('.js-selected-task-label')
 .innerHTML = getSelectedTaskName();
}
```

Regardless of whether the clock is running, we can change tasks. Once we change to a new task, the clock is reset to `25:00`.

Let's now update the pomodoro count of the selected task once the countdown reaches zero. This update is easy because we already have an `updateTime` function, where we get `0:00` in the `newTime` argument once the pomodoro time is over.

```
var updateTime = newTime => {
 console.log(newTime);
 document.querySelector('.js-time-remaining').innerHTML = newTime;
}
```

After getting rid of the console log, we can check if `newTime` is `0:00`, reset the timer to 25 minutes, restart the timer, and add one to the pomodoro count of the task. Once the pomodoro count increases, don't forget to save and render the state.

```
var updateTime = newTime => {
 document.querySelector('.js-time-remaining').innerHTML = newTime;
 if (newTime == '0:00') {
 timer.reset();
 timer.start();
 increaseSelectedTaskPomodoroDone();
 saveAndRenderState();
 }
}

function increaseSelectedTaskPomodoroDone() {
```

```
// call increasePomodoroDone with correct arguments
}
```

To test this code, it makes sense to temporarily set the pomodoro length to a significantly smaller value than 25 minutes. Don't forget to reset it once you finish testing.

```
const timer = new Timer(/* 25 * 60 */ 5, updateTime);
```



We only have one problem to solve: the original form of `increasePomodoroDone` has two arguments:

```
function increasePomodoroDone(taskId, columnIndex) {
 board[columnIndex].tasks[taskId].pomodoroDone += 1;
}
```



If we execute this function without arguments, we will get an error. But wait a minute! We can use this problem to make the code more structured. Recall the function `getSelectedTaskName`:

```
function getSelectedTaskName() {
 for (let { tasks } of board) {
 for (let i = 0; i < tasks.length; ++i) {
 if (tasks[i].selected) {
 return tasks[i].taskName;
 }
 }
 }

 return '';
}
```



Beyond the task name, we could also be interested in the task id or the column number. So we could simply return a data structure that gives us access to all these data.

```
function getSelectedTaskInfo() {
 for (let columnIndex = 0;
 columnIndex < board.length;
 ++columnIndex) {
 let { tasks } = board[columnIndex];
 for (let taskId = 0; taskId < tasks.length; ++taskId) {
 if (tasks[taskId].selected) {
 return {
 columnIndex,
 taskId,
 taskName: tasks[taskId].taskName,
 };
 }
 }
 }
}
```



```
 taskId
);
}
}

return null;
}

function getSelectedTaskName() {
 const info = getSelectedTaskInfo();
 if (info == null) return '';
 return info.taskName;
}
```

As we implemented the `getSelectedTaskName` function using `getSelectedTaskInfo`, nothing changes in the interface.

The `getSelectedTaskInfo` function is ready, so it is time to implement the function increasing the pomodoro count of the selected task:

```
function increaseSelectedTaskPomodoroDone() {
 let { taskId, columnIndex } = getSelectedTaskInfo() || {};
 if (typeof taskId === 'number' && typeof columnIndex === 'number') {
 increasePomodoroDone(taskId, columnIndex);
 }
}
```

After putting the last puzzle piece in place, we are done with this exercise.

You can see the code in action if you clone my GitHub repository. The code is in the [PomodoroTracker9](#) folder.

In these nine steps, you have seen how to build a complex exercise from scratch.

The code is simple enough to maintain, but I admit, if I had more plans with it, I would soon run out of mental capacity to track what is going on. This is why it makes sense to improve the code structure by adding a framework around the code or abstracting some functionalities. It is out of scope for us in this book though.

# Memoization

A theoretical question about the pros and cons of memoization.

## Exercise:

What is memoization? What are its benefits? What is the necessary condition for using memoization? Illustrate the benefits of memoization using an example.

## Remark:

Expect these types of questions when bridging theory with practice. You need to know what you are doing, and you have to write code that runs properly. The task itself is up to your interpretation, so choose the easiest possible solution.

## Solution:

Memoization is an optimization technique often used with recursion. We create a lookup table that contains the mapping between the input and the output of the function. If the input has been calculated before, we find the corresponding output in the lookup table and return it. If the input has not been calculated, we calculate it and insert it into the lookup table.

There are several benefits of memoization. First, if the computation of the return value of a function takes a lot of time, substituting it with a simple lookup saves a lot of time. Second, we may save a lot of recursive calls, as computing the value may imply entering into the same sequence of recursive calls. In general, if we are interested in computing the same value over and over again, we have a case for memoization.

The necessary condition of using memoization is that the function has to be *deterministic*. This means that the input values should always determine the return value of the function regardless of the external context.

We will use the Fibonacci function to illustrate memoization. The original Fibonacci function can be implemented like this:

```
let fibonacci = n =>
 n <= 1 ? n :
 fibonacci(n - 1) + fibonacci(n - 2);

console.log(fibonacci(7))
```



Let's add a global counter to count the number of function calls. We will just use a global variable now. Remember, our goal is to focus on the solution, there is no need to beautify the solution by adding complexity. Using proxies may be more elegant, but it's a minefield in an interview situation, where one bad step may lead to your elimination.

```
let callCount = 0;
let fibonacci = n => {
 callCount += 1;
 return n <= 1 ? n :
 fibonacci(n - 1) + fibonacci(n - 2);
}

callCount = 0;
fibonacci(15);
console.log(callCount);
//> 1973
```



Wow! We needed 1973 recursive calls to compute the Fibonacci value of 15. Let's see how we can do better by using memoization.

```
let callCount, memoMap;
let fibonacci = n => {
 callCount += 1;
 if (memoMap.has(n)) {
 return memoMap.get(n);
 }
 let result = n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2);
 memoMap.set(n, result);
 return result;
}

memoMap = new Map();
callCount = 0;
fibonacci(15);
console.log(callCount);
```

```
console.log(callCount);
//> 29
```



Cool! We just need 29 recursive calls instead of 1973. That's an improvement.

But we can do better.

You remember the description of the use cases for memoization? The first sentence of the solution states: "Memoization is an optimization technique often used with recursion". Then I continued describing a recursive solution.

We will now realize that memoization may be useful without recursion.

Imagine there is a service in the form of a function, and we call this function quite often. Memoization acts as a cache to retrieve the values that have been calculated. Caching requires a more complex data structure, because the size of the cache is often limited, while in this implementation, the size of the memo map is unlimited.

We will demonstrate the use of caching by trying to compute `fibonacci(12)`, by using the values already in the memo map after the previous exercise:

```
memoMap = new Map();
callCount = 0;
fibonacci(15);
console.log(callCount);
//> 29

callCount = 0;
fibonacci(12);
console.log(callCount);
//> 1
```



Why is the call count one? Because `fibonacci(12)` had been calculated as a result of calculating `fibonacci(15)`. It was a mandatory step in the computation. All we need to do is to look up one value in the lookup table.

Similarly, when calculating `fibonacci(16)`, we can reuse the lookup table containing `fibonacci(15)` and `fibonacci(14)`, so we end up with just three calls:



```
callCount = 0;
fibonacci(16);
console.log(callCount);
//> 31
```



This is a thorough answer illustrating memoization.

# SQL and Map-Reduce-Filter

We'll see how the map, reduce and filter methods prove useful in SQL queries.

## Exercise:

Suppose the following tables are given in the form of arrays of objects:

```
var inventory = [
 {
 id: 1,
 owner: 1,
 name: 'Sword',
 weight: 10,
 value: 100
 },
 {
 id: 2,
 owner: 1,
 name: 'Shield',
 weight: 20,
 value: 50
 },
 {
 id: 3,
 owner: 2,
 name: 'Sword',
 weight: 9,
 value: 150
 }
];
```

```
var characters = [
 {
 id: 1,
 name: 'Zsolt',
 points: 500,
 level: 5
 },
 {
 id: 2,
 name: 'Ron',
 points: 200,
 level: 2
 },
 {
 id: 3,
 name: 'Jack',
 points: 0,
 level: 1
 }
];
```

```
 level: 1
}

]

console.log(characters.map(c => c.name));
```



Translate the following SQL query using `map`, `reduce`, and `filter`:

```
SELECT characters.name, SUM(inventory.value) AS totalValue
FROM characters, inventory
WHERE characters.id = inventory.owner
GROUP BY characters.name
```



You are not allowed to use loops, if statements, logical operators, or the ternary operator.

## Solution:

We need to list all character names and the sum of the value of their items.

As `characters` and `inventory` are arrays, we can use the `map`, `reduce`, and `filter`.

The following projection corresponds to a simple `map` operation:

```
SELECT name
FROM characters
```



The next step is to join the `characters` and the `inventory` tables by filtering items belonging to a given character. We can also return the number of items belonging to a character in this step.

```
SELECT characters.name, COUNT(inventory.value) AS itemCount
FROM characters, inventory
WHERE characters.id = inventory.owner
GROUP BY characters.name
```





We have to return the name like before, but we also have to find the corresponding items.

```
console.table(
 characters.map(c => {
 let items = inventory.filter(item => item.owner === c.id);
 return {
 name: c.name,
 itemCount: items.length
 }
 })
);
```



If you execute this code in the console, you can see that all three characters are there with their correct item count:

| (index) | name    | itemCount |
|---------|---------|-----------|
| 0       | “Zsolt” | 2         |
| 1       | “Ron”   | 1         |
| 2       | “Jack”  | 0         |

We only have one step left: summing the value of the items.

```
SELECT characters.name, SUM(inventory.value) AS totalValue
FROM characters, inventory
WHERE characters.id = inventory.owner
GROUP BY characters.name
```



The `SUM` function is implemented by a `reduce` call in JavaScript:

```

console.table(
 characters.map(c => {
 let totalValue =
 inventory
 .filter(item => item.owner === c.id)
 .reduce((sum, item) => item.value + sum, 0);
 return {
 name: c.name,
 totalValue
 }
 })
);

```



The result is:

| (index) | name    | itemCount |
|---------|---------|-----------|
| 0       | “Zsolt” | 150       |
| 1       | “Ron”   | 150       |
| 2       | “Jack”  | 0         |

During the task, it is worth looking up the signature of `reduce` if you forgot it. Alternatively, you can reverse engineer it using a simple example if you get stuck and your interviewers insist in you not opening other tabs.

Sometimes you might have to use a testing platform to submit an answer, and the software might monitor when you leave a tab.

This is a stupid way of evaluating candidates because it puts you in a big advantage if you have two computers side-by-side. You can execute any google searches you want on one computer, and solve the task on another. This is one reason why I believe all tests should be open book tests. If you want to browse the Internet, go for it! You have limited time, so you won't be able to learn the basics of JavaScript while solving a task anyway.

This exercise is solved. Notice the analogy between SQL SELECT statements and `map`, `reduce`, `filter`. These are common questions not only in JavaScript.

# Introduction

Learn how to double your career speed with soft skills.

This chapter is based on a subsection in Chapter 4.4 of [The Developer's Edge - How to Double Your Career Speed with Soft-Skills](#). Even though this book is a career book for software developers, the section on your learning plan applies to you in the context of where you are right now.

Use this link for a 40% discount.

I will assume that you have just finished reading the book, and you have solved and double checked all the exercises. If this is the case, congratulate yourself on a job well done. By solving the exercises, you have put a lot of effort into learning ES6.

If you have not solved the exercises yet, I highly recommend that you schedule solving them sooner or later to deepen your understanding of ES6.

We will now work on deepening your understanding even further. The goal of this section is not only to plan what is useful for you. We will also create leverage by linking your learning goals to your career plan.

When will you know that you are ready for the next step in your career? You are ready as soon as you create something tangible. What is a tangible result? Something that you are comfortable with presenting to the outside world. A high-quality blog post, an online service, or a GitHub repository will do.

Your learning plan will consist of five steps:

1. Sync your learning plan with your career plan.
2. Set your milestones.
3. Create the big picture.
4. Get feedback and iterate.
5. Present your deliverables.

In the next few lessons, I will discuss each of the five steps in more detail.

# Sync your Learning Plan with your Career Plan

Define your career goals and learn to work with a perspective.

To learn a skill, you need to know why you want it. Get clear on your goals first.

## Questions:

1. What are your career goals?
2. What skills will you need to reach these goals?
3. What resources do you need to learn those skills?

Focus on both the technical and non-technical aspects of your career. Focus on multiple timespans such as:

- What are your career goals in 5 years
- What needs to happen in 3 years to meet your goals in 5 years?
- What needs to happen in 1 year to meet your goals in 3 years?
- What needs to happen in 6 months to meet your goals in 1 year?
- What needs to happen in 3 months to meet your goals in 6 months?
- What needs to happen in 1 month to meet your goals in 3 months?

You may need to learn a new programming language, leadership skills, presentation skills, software architecture, functional programming, acceptance testing with Selenium, or anything you know you will use soon.

Once you are ready with your list, answer the most important question of your learning plan. What is the first, most crucial step you should take? Make the next step clear to yourself, and start focusing on it.

# Set your Goals and Milestones

Achieve the bigger goals by setting small milestones and deliverables for yourself.

For each skill that you would like to improve, determine projects and activities that measure your knowledge. Launch projects that will put your new skill into action. Start with *the why*, and *the what* will fall into its place.

Make these projects useful and practical on some level. Building an app for real users is better than creating something just for yourself. Teaching others, blogging, shooting a youtube video where you explain something are all great ways of measuring your progress.

*Make sure your goals and milestones are tangible.* Don't come up with a goal like "I want to learn ReactJs by understanding the example in the book I purchased." Most likely, you would spend most of your time trying to learn nitpicky details that you would forget within a week.

Many books and courses are excellent, I have learned from many of them myself too. However, if you finish a book without creating something new that goes beyond the content of the book, your knowledge of the subject will fade as time passes. If you push your boundaries beyond the theory, you will learn some skills that may stay with you for the rest of your professional career.

A reasonable goal may be the following:

**Example goal:** Create a single page application in React and ES6 that enables software developers to track their productivity using the Pomodoro technique and other useful productivity hacks such as (...).

Feel free to refine the exact specification later.

Your projects also keep your work focused. You won't waste time on unimportant details. You will learn what matters. When I wrote ES6 in Practice, I had two guidelines in mind:

Practice, I had two guidelines in mind.

- I wanted to tell you just enough to get started with solving exercises.
- These exercises tell you stories and challenge you with a continuously increased difficulty. You will not only feel good after solving them, but you will also know that you can solve problems on your own. Some of the exercises include concepts other books would give you in a theoretical section. By solving such an exercise, you prove yourself that you are capable of learning new ideas once you need to use them. This should give you confidence in your abilities and deepen your understanding at the same time. Worst case: you read the solution and learn as much as you would have from other tutorials.

Be specific about the time you are willing to invest in your side projects. Many people, sometimes including myself, tend to underestimate the time needed for a tech project often by a factor of 2 or 3. If you figure out that your lifestyle does not support creating a complicated side project, you have some life decisions to make. You will either change your lifestyle or career plan, or you will have to create a different learning plan for yourself.

Your goal may be quite simple. For instance, if you learn React, and you already have a blog, you may want to set yourself the following target:

1. Create a boilerplate for a React application, and upload it on GitHub

- Skills: ES6, React fundamentals, Webpack, NPM, BabelJs
- Deliverable: React boilerplate on GitHub

2. Use your React boilerplate to implement a Todo application

- Skills: more in-depth React knowledge
- Deliverable: a refined React boilerplate based on the learnings of creating an application

3. Write a blog post about your learnings

- Skills: more structured React knowledge
- Deliverable: blog post

You may set a big goal for yourself, and you deliver your solutions incrementally in multiple milestones. For instance, to realize a productivity

app, you might want to create:

1. an abstract productivity model to implement

- Skills: Pomodoro technique, kanban, productivity research
- Deliverable: productivity model, documentation, software specification

2. a responsive website with components

- Skills: Bootstrap or pre-made web components
- Deliverables: the skeleton of the website with static data

3. automated testing and mocking are most likely needed for development

- Skills: Mocha, Chai, SinonJs, API endpoint design
- Deliverables: automated testing framework connected to the application, and a way to intercept and fake API calls. Design your API endpoints.

4. a React layer

- Skills: React, NPM, Webpack, BabelJs
- Deliverables: none, as it makes perfect sense to do D and E together

5. you may want to manage the application state with a library like Redux

- Skills: Redux
- Deliverables: a fully working application with a fake server

6. you may need some server-side code and a database

- Skills: NodeJs, MongoDB
- Deliverable: a fully working solution

As you can see, in these six milestones, you may be learning an avalanche of technologies. Set some target dates for yourself by estimating the amount of work needed for finishing the application.

Make sure you don't treat yourself too hard for missing your target dates. The importance of measuring time is to figure out how good you are at estimating your development velocity.

your development velocity.

A side project like this will keep you busy for a long time. However, you will reap the rewards once you can demo your application. If your application is useful enough, you may even be able to monetize it.

Preparing you to monetize your application is outside the scope of this book. To develop an application for making money, you will have to test what your users want continuously and aggressively modify your product. This process does not create an optimal learning experience for you, so let's continue focusing on your career for now.

# The Big Picture

See the bigger picture by tracking your progress with the help of modules and pre-requisites.

It is now time to break down each milestone. Create a table of contents for yourself about the subject just as if you were writing a book. Determine how to break down the new skill into small pieces.

Discover pre-requisites, redundant topics, and make choices about what you will focus on. Don't worry about making everything perfect. You will have several chances to refine your plan.

Do your research from several sources. When it comes to technologies, you can visit the official documentation, check out the table of contents of books, courses, youtube channels, online tutorials, and blogs. You may even find helpful GitHub repositories.

Once you are done gathering information, make sure you organize your information. For instance, a table with the following columns will do:

- Module number
- Description
- Goals / Tasks
- Resources
- Status

Do your best to detect dependencies between the different modules you create for yourself and order them accordingly. Don't worry if you miss a dependency or two, as your study plan is lean enough to accept future iterations. Step 4 will be all about iterating your plan.

Link accomplishments to most of your modules, otherwise you will have a hard time tracking your progress. The more you use your acquired knowledge in practice, the deeper your understanding will be. If you just read a book, expect to forget 90% of the material. If you take notes, you may forget 80%. If

expect to forget 50% of the material. If you take notes, you may forget 80%. If you write an article on it with example code, chances are, you will only forget

half of what you learned. As soon as you complete a project in practice, you are likely to remember the fundamentals for years.

## Get Feedback and Iterate

Walk through your plan multiple times and plan and re-plan things wherever necessary.

Once you implement a module, you will face difficulties. Make a note of them, and modify your study plan bottom-up, starting with Step 3, and ending with Step 1.

For instance, you may figure out a dependency that blocks you. You may either reorder your modules or add new ones to unblock yourself. You may even figure out that some of your modules are useless from the perspective of the outcome.

*Always be goal-oriented!* Cross out modules that don't contribute to your goal. I can't emphasize the importance of the Pareto principle often enough: 20% of your effort contributes to 80% of your results. When it comes to learning, the right 20% will give you the solid foundation that you can build on later in your career. If you need to dig deeper, you will understand the topics you need in depth. *Many professionals get stuck in the bottom 80% of things to learn, at the expense of doing the top 20% right. Digging deeper only makes sense when you know where to dig.*

The most frequent modifications only affect the bigger picture of your milestone. This requires you to re-iterate Step 3 only.

Sometimes you will discover a cross-milestone dependency, or you may replace technologies as you get to know more about the subject. For instance, you may initially decide on using a REST API for your project, but then you stumble upon GraphQL. Given that you already know about REST, and GraphQL perfectly fits your project, chances are, it makes sense for you to redesign the whole milestone. This way, you can make better use of your time, and learn something new.

Some of your decisions and discoveries may lead to small changes in your career plan. Other changes may shape the description of your projects. Make

sure you go back to steps 1 and 2 and update them.

Don't worry about changing your plan frequently. Flexibility is key to getting the best learning experience for yourself.

Your iterations may also depend on external feedback. Be open about your side projects. Show them to your colleagues and friends. Post about them online, and discuss your discoveries in tech forums and communities. You will be amazed how quickly you can improve your skills.

Last, but not the least, step back and look at your career path from time to time. Learning new things may unlock new possibilities for you. These are paths that you were not aware of. Some path may revolutionize your future projects or even your current project. If you unlock a new path, don't be afraid of modifying your study plan in a top-down way.

# Present your Deliverables

Plan your deliverables for an audience and always be ready for criticism.

Make sure you also think about presenting your deliverables. Let it be a GitHub repository, a portfolio site, or a blog post, find a way to show your results to an audience.

There will always be someone who loves what you do. There will still be haters too. Whenever you receive criticism, ask yourself what the intention behind the criticism is. Always take things objectively, and do your best to improve your products. Be thankful for constructive criticism.

You are also likely to receive destructive criticism. Don't be affected by it. If you are affected, go back to chapter 2, and combat your fears and limiting beliefs. Criticism makes you stronger. Above all, facing criticism properly shapes your character, refines your professional code, and gives you a small edge at the workplace.

Always draw conclusions after finishing a side project. Ask yourself what skills you have learned, and how you will use these skills in the future.

# Summary

Get in touch with the author and get updates on Javascript topics.

Congratulations! You have gone through the whole course. I hope you also liked the exercises. They are an essential and integral part of mastering ES6.

If you get stuck with ES6, or you would like me to elaborate on an ES6 topic, send me an email. My email address is [info@zsoltnagy.eu](mailto:info@zsoltnagy.eu).

If you are interested in taking your career to the next level, check out my other book, [The Developer's Edge - How to Double Your Career Speed with Soft-Skills](#). I am offering a 40% discount for my fastest 40 readers on The Developer's Edge. If you visit this link, you will get a 40% discount on the price. Alternatively, you can apply the [ES6Discount40](#).

If you have not signed up for my email list, feel free to do so on [zsoltnagy.eu](mailto:zsoltnagy.eu). You will get updates on topics about JavaScript, including ES6, React, Redux, and automated testing.

Take care,  
Zsolt