6. JavaScript and programming paradigms

Obsah

1 Programming paradigms introduction

We've gone through all the programming topics that allow us to write code that works and does stuff. But to be able to write code that works is just one part of the job, the other very important part is to write code, that is easily readable, and maintainable.

Why? Well, let's list few reasons:

- Applications are very often developed in teams, so it's important to write code that your colleagues find easy to read and revise if needed.
- Applications change throughout their lifetime. Imagine that you've developed an app a year ago, and now the client requests some additional functionalities. Believe me, after a year you simply, you won't remember much about that app, so you will need to go through the code and get familiar with it again. It would've been a nightmare to go through an unorganized hundreds of lines of spaghetti code.
- Bugs occur a lot in programming. It's much faster to fix them, if you have organized and self-explanatory code.

Okay, now we know why it is important to structure our code in some way. But how can we do that? Well, that's where programming paradigms come into play.

1.1 What is a programming paradigm?

The term **programming paradigm** refers to a style of programming. It does not refer to a specific language, but rather it refers to the way you program. (1)

These paradigms are basically a set of rules about how to structure and write your code with the goal of making your code easier to read, maintain and edit.

In JavaScript (and many other programming languages), you have three options – Procedural, Object oriented and Functional. (2)

Procedural and Object oriented are forms of imperative programming. That means, that they are focused on how exactly should program do something.

On the other hand, functional programming is a form of declarative programming. It means, that it is more focused on what do to, without bothering how to do it.

Interesting video about an overview of programming paradigms - https://academind.com/learn/javascript/functional-vs-oop-vs-procedural/

2 Procedural programming paradigm

This is the easiest one that most programmers start with.

Procedural Programming can be defined as a programming model which is derived from structured programming, based upon the concept of calling procedures. Procedures, also known as routines, subroutines or functions, simply consist of a series of computational steps to be carried out. (3)

Procedural programming takes a top-down approach. It is about writing a list of instructions to tell the computer what to do step by step. It relies on procedures or routines. (4)

What is a procedure? Well, a procedure is a small section of a program that performs a specific task. (5) For example, you want to loop through an array of items, so you write a procedure, to do so.

Of course, that you can create functions, so you can have some level of code reusability, but otherwise, there is not many other rules you need to follow. This fact makes procedural programming very simple. But even though it is easier to write in this style, it has some disadvantages.

A major disadvantage of using Procedural Programming as a method of programming is the inability to reuse code throughout the program. Having to rewrite the same type of code many times throughout a program can add to the development cost and increase the time needed to finish the project. (6) It's hard to reuse code, because while you can create reusable functions, those functions usually don't work only with the params they get, as they also do something with the global variables (or the so-called state). This fact simply forces you to copy paste a lot of code and causes the creation of spaghetti code.

Another disadvantage is connected to the previous one. You mostly separate your code into smaller chunks/procedures, but those procedures manipulate with the global variables in some way. Now imagine a situation, where you have a bug in your code (this happens much more often than you would want in real world), because of the fact that your procedures manipulate with the global variables directly, you have to go through each and every line of your code in order to find the bug. (6) (7)

3 Object oriented programming paradigm

Interesting video - https://www.youtube.com/watch?v=pTB0EiLXUC8

We already talked a little bit about OOP in one of our previous classes and I promised you to dig a bit deeper into this topic, so let's go ahead.

OOP is hugely popular and is used in many programming languages including JavaScript (at least to some extent).

The basic idea of OOP is that we use objects to model real world things that we want to represent inside our programs. (8)

OR another definition

Object-oriented is a software design in which we can simply group data, its data types and methods under a single package. (9)

You may ask why would you want to do that, or how can this make the code better? Well, consider the following. We want to create a simple counter. Let's first do that in a procedural way. It's fairly simple. Just one global variable **number** and one function for incrementing and another one for decrementing.

```
let number = 0;

const increment = () => {
   number++;
};
const decrement = () => {
   number--;
};

increment();
increment();
decrement();
console.log(number);
```

Okay, this is fine, but what if we want to have some other counter? We need to create another global variable for new number, that's fine, but what about those functions? We definitely need to refactor them a bit, so they take that number as an argument and then increment or decrement that given number. Okay, that's definitely possible, but we still need to have some global variables and if there is some bug, we need to go through each and every function that possibly changes our global variable, if there is something wrong that is causing that bug. Now, in our simple counter app it's fairly easy and more straightforward, but imagine that you have dozens of global variables and hundreds of functions that may or may not do something unexpected with global variables. This is when things start to be very messy. Plus, those functions can access and call each other, so that's another fact that just

adds up to the code complexity and therefore keeps us "entertained" for long hours or days in order to find the bug.

Now, let's rewrite our simple counter in OOP style. First, we need to create a class called Counter. This class is like a blueprint for the future object instances, so we need to define all properties and methods here. We have just one property **number** and two methods – **increment** and **decrement**. Our blueprint is ready, so the only thing we need to do is to create an instance of this class, which is the **counter1** and declare the initial number. And now the same question as for procedural way. What if we want to have another counter? Well, that's incredibly simple to implement now. We would just need to create another instance of our **Counter**, define another initial number and that's it!

```
class Counter {
   constructor(number) {
      this.number = number;
   }

   increment() {
      this.number++;
   }
   decrement() {
      this.number--;
   }
}

const counter1 = new Counter(0);
console.log(counter1.number);
counter1.increment();
counter1.increment();
counter1.increment();
console.log(counter1.number);
```

As you can see from the previous simple example. In comparison to procedural programming, you need to think about the code structure for a little bit before you start programming, but in a long run, apps written in OOP style are much easier to work with and scale in comparison to the procedural way.

3.1 Side note – object instances and classes

We've talked about objects and classes already in one of the previous classes, but I know that this can be a little bit confusing, so since this is very important to understand, let's talk about this again.

Classes are templates for creating objects. (10) That means, that classes by themselves don't really do anything. They are just blueprints for objects that will be created. They just define how the object created from given class will look like. You can think of classes like a cooking recipe. A cooking recipe contains information needed to cook something, but by itself, it doesn't really provide you some food.

Object instance is a particular Object created from some Class. (11) This is the concrete food created from the recipe.

3.2 4 main concepts of OOP

Now that we've seen a simple example of OOP programming style, let's go through its principles. OOP has 4 main concepts – Inheritance, Encapsulation, Abstraction and Polymorphism

3.2.1 Encapsulation

Video - https://www.youtube.com/watch?v=sNKKxc4QHqA

Encapsulation is the bundling of data and the methods that act on that data such that access to that data is restricted from outside the bundle. (12) In modern JavaScript, **class** is used as a bundle for data and methods.

Why is this useful? Well, as a programmer, you want to have as much control over your code and data flow as possible. If you simply start programming without any thought given into the code architecture, you end up with some global variables, numerous functions that can call each other and access all global variables. This is very messy and if a bug occurs, you have to go through each and every line of your code to find the problem.

Encapsulation gives you the ability to avoid this mess as it allows you to create bundles of data and methods. (13) Imagine that you are developing an app that provides users with cooking recipes, but the recipes are not showing up. Well, if you created a Recipe bundle (class) you know, that the problem is probably within that class, so you just need to focus on that little part of your code to find the bug. Just a have look at the previous code example where we created a **Counter**. Increment and decrement are parts of that class, and they are the only two methods manipulating with the counter state (counter number), so if there is some bug, for example it increments instead of decrementing, you know where to look for it.

Another good example is JavaScript's **Math** object. Math object allows you to perform mathematical tasks on numbers. (14) All those methods for rounding numbers, powering numbers, etc. are encapsulated in the Math object.

Another advantage of using encapsulation is the ability to control/restrict access to data of some bundle (class). You can have methods and variables that are not accessible outside of a class, this this simply prevents you from accidentally changing some variable value somewhere in the code. But this is unfortunately not possible in JavaScript.

As I said few times in one of our previous classes, JavaScript is not class based but prototype based, so classes are just a syntactic sugar. (15) Because of that, JS classes don't have everything that you normally find in truly class-based language. For example, in many other programming languages, you can define private methods (methods that can be used only within that class), this is not possible in JavaScript. (16)

Also, private variables are not fully implemented in a way that they are not fully supported by all modern browsers. They are not supported for example in Firefox. (12) (17) But let's show them, so you know what they are.

In the following example, there is a variable called **privateField** is declared. As you can see, **#** needs to be used to create a private variable and it needs to be declared inside the class. This variable is accessible throughout the whole class (in this example, it is accessed inside increment method), but it cannot be accessed outside the class.

```
class Counter {
   #privateField;

   constructor(number) {
      this.number = number;
      this.#privateField = "something private";
   }

   increment() {
      this.number++;
      console.log(this.#privateField);
   }
   decrement() {
      this.number--;
   }
}

const counter1 = new Counter(0);
console.log(counter1.number);
counter1.increment();
counter1.increment();
```

```
counter1.increment();
console.log(counter1.number);
console.log(counter1.#privateField); //error
```

Encapsulation in a way of data protection from external interference via classes is not completely achievable, BUT it doesn't mean that you cannot achieve it in some other way. We actually talked about the way to achieve a functionality where inner function has access to the outers function variables, do you remember what it was? It's **closures**! (12)

I would just like to add on, that for better OOP support, you can use TypeScript as Typescript allows you to use private variables and methods.

So, a little summary about encapsulation. Encapsulation is:

- Binding the data with the code that manipulates it. (18)
- It keeps the data and the code safe from external interference (18) But this is not that easy to do in JavaScript via classes

3.2.2 Abstraction

Abstraction is one of the key concepts of object-oriented programming (OOP) languages. Its main goal is to handle the complexity by hiding unnecessary details from the user. That enables the user to implement more complex logic on top of the provided abstraction without understanding or even thinking about all the hidden complexity. (19)

Think — a coffee machine. It does a lot of stuff and makes quirky noises under the hood. But all you have to do is put in coffee and press a button. (20)

Preferably, this mechanism should be easy to use and should rarely change over time. Think of it as a small set of public methods which any other class can call without "knowing" how they work. (20)

If we come back to our **Counter** class. By following the encapsulation principle, we have all methods manipulating with the counter, within the **Counter** class. The abstraction here is the fact, that when we manipulate with the counter, we just call **increment/decrement** methods, and something happens. We don't need to implement our own increment/decrement logic nor care about how these methods are actually implemented.

In many other programming languages, you can even take abstraction a little further by using **abstract classes** and **interfaces**. (21) But as I said in the previous section about the Encapsulation, JavaScript is not truly class-based language, classes are just a syntactic sugar. Abstract classes and interfaces simply don't exist in JavaScript, so we won't be explaining those, but just for you to know, as for Encapsulation, they do exist in Typescript.

Video - https://www.youtube.com/watch?v=L1-zCdrx8Lk

3.2.3 Inheritance

Inheritance is finally an OOP concept that we can fully use in JavaScript, but as we've already explained what Inheritance is in the class about advanced JavaScript, so I won't go through it again.

Just as a reminder, JavaScript uses prototypes for inheritance. **Classes** exist in JavaScript just to make working with prototypes easier, you can create a class, that extends already existing class. The extended class then inherits all the properties and methods from its parent.

Video - https://www.youtube.com/watch?v=MfxBfRD0FVU

3.2.4 Polymorphism

Polymorphism in Object-Oriented Programming is an ability to create a property, a function, or an object that has more than one realization. (22)

In other words, it is an ability of multiple object types to implement the same functionality that can work in a different way but supports a common interface. (22)

For example, in the real world, once you learn how to use one windows computer, you are able to use any windows computer regarding its HW specs or whether it is a laptop or a desktop.

Okay, so what does it mean in code? Well, it means, that you can inherit from a class that has some method, but you can override this method in the derived class, and it will still work. Let's now have a look at the code example. So, we have class **Animal** and classes **Dog, Bird** and **Turtle** that inherit from the **Animal**. **Animal** has method **move** so we can call that method on all object instances of **Animal** or classes that inherit from **Animal**. But not only that, we can also override (or change shape) of that method, but we can still call that method in a same way as it would be the same method for all those object instances, and the proper method is still called. The **Dog** class just inherits from the **Animal**, and that's it, so when we call **move** method, "run" is logged into the console. But in the **Bird** and **Turtle** class, we override **move** method. So, for the **Bird** "fly" is logged and for the **Turtle** "move veeery slowly" is logged. As you can see, thanks to Polymorphism, we can not only inherit from classes, but we can also further customize their methods if needed, so they can have many shapes, but still be treated as if they are the same method.

```
class Animal {
   move() {
      console.log("run");
   }
}

class Dog extends Animal {}

class Bird extends Animal {
   move() {
      console.log("fly");
   }
}

class Turtle extends Animal {
   move() {
      console.log("move veeery slowly");
   }
}

const dog1 = new Dog();
const bird1 = new Bird();
const turtle1 = new Turtle();
dog1.move();
bird1.move();
turtle1.move();
```

Interesting videos:

- https://www.youtube.com/watch?v=4sl5Te5inPY
- https://www.youtube.com/watch?v=AeS5OSKdaRM
- https://www.youtube.com/watch?v=8a5BkwuZRK0

4 Functional programming paradigm

Functional programming has become very popular in JavaScript lately,

First, I want to say, that there are languages, that are purely functional (for example Haskell), but JavaScript is NOT one of them. JavaScript just supports Functional programming techniques. So even though functional programming is very popular these days especially because of frameworks like React, you don't need to write everything in functional

way BUT it has a lot of benefits if you write your code in a functional way as much as you can.

Let's start with why functional programming is good in JavaScript. Well, as you could notice in the previous section about OOP, it's not that easy to write in OOP way in JavaScript, because JavaScript doesn't really have classes, and the classes it has, don't support everything you need to write fully OOP code. Another problem with JavaScript is, that it can be quite messy, because you can't specify variable types (dynamically typed language) and variables are not strictly tight to one data type (weakly typed language), for example if you combine number variable with string, then the number variable is also considered as a string. In other words, JavaScript doesn't give much of a control over the code your write and that can be huge problem in bigger apps.

4.1 What is Functional programming

As you can guess, functional programming offers a way to deal with the problems from previous paragraph, but what is it?

Functional programming (often abbreviated FP) is the process of building software by composing **pure functions**, avoiding **shared state, mutable data,** and **side-effects**. Functional programming is **declarative** rather than **imperative**, and application state flows through pure functions. Contrast with object oriented programming, where application state is usually shared and collocated with methods in objects. (23)

Uf, okay, although the explanation above is precise, it also introduces many new words, so let's take a closer look at them one after another.

4.2 Pure functions

Interesting video - https://www.youtube.com/watch?v=fYbhD_KMCOg

Pure function is a function that given the same input, always returns the same output. (23) Pure function works only with arguments you pass to it. It simply never accesses the global state (variables) directly. Also, pure functions always return a value and don't cause any side effects (I will discuss them in a minute). (24) (25) And last but not least, pure functions must NOT mutate any external state. (26) This may seem very restrictive, and it is, but for a good reason. By following rules of pure functions, you are creating very simple functions that are easily predictable. (27)

The following function is pure. It just accepts parameter **a**, multiplies it by two and returns the result (reminder – arrow function without curly brackets has internal return).

```
const add = (a) => a * 2;
```

4.3 Avoiding shared state

Why is shared state bad?

First problem is connected to asynchronous actions. Imagine you have a user object which needs saving. Your **saveUser()** function makes a request to an API on the server. While that's happening, the user changes their profile picture with **updateAvatar()** and triggers another **saveUser()** request. On save, the server sends back a canonical user object that should replace whatever is in memory in order to sync up with changes that happen on the server or in response to other API calls. (23)

Unfortunately, the second response gets received before the first response, so when the first (now outdated) response gets returned, the new profile pic gets wiped out in memory and replaced with the old one. This is an example of a **race condition** — a very common bug associated with shared state. (23)

Second problem is, that changing the order in which functions are called can cause a **cascade of failures** because functions which act on shared state are timing dependent. (23) Consider the following example. We have an object with number stored as a string. In

convertToNum the string value from the **strNumbers** object is converted to a number and in **addToNum** number 5 is added to the converted **strNumbers** number. The result is 7.

```
const strNumbers = {
   num1: "2",
};

const convertToNum = () => (strNumbers.num1 = parseInt(strNumbers.num1));

const addToNum = () => (strNumbers.num1 += 5);

convertToNum();
addToNum();

console.log(strNumbers.num1);
```

Okay, what do you think will happen, If I just call the function **addToNum** before **convertToNum**? Well, now the number is first added to the string value, and because in these cases JavaScript converts the number to string and then concatenates values, the result is concatenated strgin – 25, and then parsed to number.

```
const strNumbers = {
   num1: "2",
};

const convertToNum = () => (strNumbers.num1 = parseInt(strNumbers.num1));

const addToNum = () => (strNumbers.num1 += 5);

addToNum();
convertToNum();

console.log(strNumbers.num1);
```

Okay, this clearly illustrated a problem of shared state. Imagine you have huge app and you just forgot how some things work and changed the order of function calls and suddenly, the app doesn't work properly, but there is no obvious error. These bugs are usually not easy to find. It's better to make functions calls independent of other functions calls as it simplifies bug fixing and refactoring. (23)

In the following code, we have only pure functions. **convertToNum** just accepts **obj** as a parameter, and returns a copy of that object, where property **num1** is parsed to an integer. **addToNum** accepts **obj** as parameter, and returns a copy of that object, where 5 is added to **num1** property. The reason we are creating copies is, so we can avoid sharing state and directly mutating it, and because objects and arrays are passed by reference, we simply need to make a copy in order to avoid direct mutation (mutation is another topic I will talk about a bit later). Now those functions are completely independent of each other. But we still want to achieve our goal, right? First convert to number and then add 5 to number. It's easily achievable, all we need to do is to just pass return value of **convertToNum** as a parameter to **addToNum**. This is actually an example of **function composition**. (23)

```
const strNumbers = {
   num1: "2",
};

const convertToNum = (obj) => {
   return { ...obj, num1: parseInt(obj.num1) };
};

const addToNum = (obj) => {
   return { ...obj, num1: obj.num1 + 5 };
};
```

```
console.log(addToNum(convertToNum(strNumbers)));
```

4.4 Function composition

Function composition is the process of combining two or more functions in order to produce a new function or perform some computation. (23) This is widely used in Functional programming, as you mostly work with small pure reusable functions that just take some arguments and return a value. So, if you want to execute some more complex operation, that needs many steps, you often need to compose many functions in order to achieve it, just like in the previous code example. (28) The reason you can pass function into another function is, because JavaScript has **first class functions**.

4.5 First class functions

So, as I said, JavaScript has first class functions. It means, that in JavaScript, functions are treated just like any other variable. Functions can be passed as an argument to other functions, can be returned by another function and can be assigned as a value to a variable. (29) Function composition is just one of the things you can do thanks to first class functions, another concept you can use because of it are **higher order functions,** which I'm going to talk about soon.

4.6 Side effects

When I was talking about pure functions I said, that they don't cause any side effects, but what are they and why are they bad?

List of side effects:

- Modifying any external variable or object property (e.g., a global variable, or a variable in the parent function scope chain) (23)
- Logging to the console (23)
- Writing to the screen (23)
- Writing to a file (23)
- Writing to the network (23)
- Triggering any external process (23)
- Calling any other functions with side-effects (23)

As you can see, basically, all interactions of your function with the world outside its scope are side effects. (30) In other words, there is no way you can avoid side effects, but in functional programming, it's always good to minimize them to their minimum and isolate them from the rest of your application. (23) It's mainly because you can't predict what exactly will those side effects do. For example, if you call an API, you don't know which exact data you will get in a response, or if you receive an error instead of data.

4.7 Immutability

This is a very important concept of Functional Programming. It has very simple rule: "never change a value or reference once it has been assigned ". (31) In simple words, assigned values are sacred, and you can only look at them, and use them in some computations, but never directly mutate them. This sounds weird right? You simply need to change values in app, in order to make it do something. Well, not necessarily.

But why is it good to have immutable data? It makes your code much more predictable and easier to debug, because you know, that none of your functions mutate (change) any of your values. Those functions just take values as arguments, and return some other value(s) based on those received arguments. So, if you encounter some bug, you don't have to go through each and every line of your code, you can just focus on those functions, that are

returning some unexpected values. Also, you always know what exactly your functions are doing with given arguments.

Okay, so how can we achieve immutability? Well, by using pure functions! That's why they are so restrictive. It's not easy to write them at first, but it's very easy to read them and maintain them.

Let's show how to achieve immutability.

4.7.1 Const doesn't create immutable objects and arrays

It's very simple for primitive values. Once you create a primitive value by using **const**, it can't be reassigned, BUT as we learnt in one of the previous classes, it is different for objects and arrays. You can still add/remove/change their values, the only restriction you have is, that you can't reassign completely new object/array to those values. In simple words, **const** doesn't create immutable objects. (23) So, what to do? Well, you need to make copies of those objects/arrays, and modify those, as shown in the section **4.3. Avoid shared state**. You have two options for copy creation – **Object.assign** and **spread operator**.

4.7.2 Add value to object copy via Object.assign

In the following code, we have the **mafia** object, and we want to add the game director without mutating the **mafia** object. In order to do that, a copy needs to be created, so we can add director to that copy and return this copy with director added from the function. To do so, Object.assign is used. Object.assign method copies all enumerable own properties from one or more source objects to a target object and it then returns the target object. (32) The first argument is the target object (empty object), second argument is the first source object, and second argument, is the second source object. These two sources are combined into that target empty object.

```
const addDirector = (game, name) =>
Object.assign({}, mafia, { director: name });

console.log(addDirector(mafia, "Daniel Vávra"));
```

4.7.3 Add value to object via spread operator

In the following code, the exact same thing is achieved via **spread operator** instead of **Object.assign**. None of them is better than the other, so it depends on your preference, which one you will be using.

```
const mafia = {
   genre: "Action-adventure",
   releaseDate: new Date("August, 28, 2002"),
   developer: "Illusion Softworks",
};

const addDirector = (game, name) => {
   return {
      ...game,
      director: name,
   };
};

console.log(addDirector(mafia, "Daniel Vávra"));
```

4.7.4 Assigning function return value to a variable

Sometimes, you need to store the result of some pure function, into a variable. You can totally do that! Let's just assign the result of our **addDirector** function to a variable, instead of just logging it into the console.

```
const mafiaWithDirector = addDirector(mafia, "Daniel Vávra");
```

4.7.5 Updating nested objects

Nested objects are tricky. Let's say, that I want to save more information about director, for example his other game. So now I end up with the following object.

```
const nestedMafia = {
   genre: "Action-adventure",
   releaseDate: new Date("August, 28, 2002"),
   developer: "Illusion Softworks",
   director: {
      name: "Daniel Vávra",
      otherGame: "Kingdom Come: Deliverance",
   },
};
```

Now I want to add even more information about the director. How to do that?

Do you think that the following code is correct? No, it is NOT. This way, you just create a shallow copy of the object. That means, that first level is correctly copied, but the nested level is NOT, so it is just a reference to the original object, therefore, if you assign new property to director, you actually mutate the original object. (33)

```
const addMoreDirectorInfo = (game, newInformation) => {
   gameCopy = { ...game };
   gameCopy.director.newInfo = newInformation;
   return gameCopy;
};

console.log(
   addMoreDirectorInfo(nestedMafia, "some additional information")
   );
console.log(nestedMafia);
```

In order to do this correctly, you need to copy all levels of object as shown in the following example.

```
const addMoreDirectorInfo = (game, newInformation) => {
   return {
      ...nestedMafia,
 director: {
 ...nestedMafia.director,
 newInfo: newInformation,
      },
    };
};

console.log(
   addMoreDirectorInfo(nestedMafia, "some additional information")
);
console.log(nestedMafia);
```

As you can see, it is confusing, and the more levels you have, the harder it is to read and the more verbose your code will be as shown in the image below. That's why it is always

Figure 1 - example of nested objects copy complexity. Source: (33)

recommended to keep your objects flat if possible (no nesting)

4.7.6 3rd party libraries for securing the immutability

As you could see, it's not easy to keep variables immutable. There can be times, when you mutate some state accidentally. While JavaScript has some possibilities to ensure immutability, they are not perfect, and it is just better/easier to use 3rd party libraries like Immutable.js or Mori. (23) I just want to say, that even though these libraries are popular, they are not necessary, as they add more complexity to your app in a certain way (you and your teammates need to learn them, and it can be sometimes tricky to use them properly).

4.8 Higher order functions

We already talked about higher order functions in one of our previous classes.

Higher order functions are functions that operate on other functions, either by taking them as arguments or by returning them. In simple words, A Higher-Order function is a function that receives a function as an argument or returns a function as an output. (34)

In JavaScript, there are some built in higher order functions – map, filter, reduce. We talked about those before as well, so I'm not going to explain them again.

The important thing here is, that they are used first to simplify your code, make it more declarative, and allow us to make our code more reusable. For example **map()** accepts a function as an argument, so it can be used for any type of array, because the appropriate data handling responsibility is on that argument function. (23)

They are used for:

- Abstract or isolate actions, effects, or async flow control using call-back functions, promises, etc… (23)
- Create utilities which can act on a wide variety of data types (23)
- Partially apply a function to its arguments or create a curried function for the purpose of reuse or function composition (23)
- Take a list of functions and return some composition of those input functions (23)

Let's program our own higher function. For example, let's take the **map()** function and recreate its functionality. First, I want to say, that you should NOT add something to prototypes of built in objects as it is considered as a bad. practice! The I reason I do this here is just to show you how higher order function works internally. So, what is happening here. JavaScript has prototype based inheritance so we can add whatever we want to the prototype of any object. In our case, we are adding **customMap** method. This method accepts a call-back function. Then an empty array is created. In for loop, you can see just normal array loop. **this** points to the array where the **customMap** function is called. Then the current array item, item index and whole array are passed to the call-back function as arguments. This call-back function executes the logic we specified, and the returned value of this call-back function is pushed into the **newArr**. Once loop is done, the **newArr** is returned to us.

```
const arrs = [
   { name: "josh", age: 22 },
   { name: "angelica", age: 18 },
   { name: "Andrea", age: 38 },
];

Array.prototype.customMap = function (callback) {
   const newArr = [];
   for (i = 0; i < this.length; i++) {
      newArr.push(callback(this[i], i, this));
   }
   return newArr;
};

const myown = arrs.customMap((item, index) => item.name);
```

```
console.log(myown);
```

4.9 Recursion – alternative way to loops

Recursion is an alternative way to write loops in a more functional style. Basically, every loop can be turned in to a recursion.

I would like to say, that it is not necessary to use recursion. It can make your code cleaner (once you get used to it) and more functional, but that doesn't mean you have to use it. JavaScript is not purely functional programming language, so it has loops, and if you find them easier to use, then just stick with them. BUT, it is way better to learn about something and then decide whether it is worth it or not, than just lazily ignore something and don't use it because you simply don't know how.

Recursion is a technique for iterating over an operation by having a function call itself repeatedly until it arrives at a result. (35)

Okay, that sounds confusing, right? Let's code a very simple recursion to get factorial. As you can see, factorial accepts **num**. Then there is a **base case**. You ALWAYS need to have that, otherwise your function can cause stack overflow. (36) By using the base case, you just make sure that your function calls end at some point. Then argument **num** is called and multiplied by **factorial** which accepts **num** argument minus 1. This goes on until **num** is 0, then 1 is returned.

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

console.log(factorial(3));
```

Okay, why is this possible? Well, that's easy, it's because of how JavaScript call stack works! When a function is called, it is put at the top of the call stack. Recursion is just using this fact in its favour. Each factorial call is put at the top of the call stack, and once these calls reach the base case, they start returning their values and being removed from the call stack.

Here is an illustration of how the factorial recursive function works.

```
factorial(3) returns 3 * factorial(2)
factorial(2) returns 2 * factorial(1)
factorial(1) returns 1 * factorial(0)
factorial(0) returns 1


// Now that we've hit our base case, our function will return in
order from inner to outer:


factorial(0) returns 1                 => 1
factorial(1) returns 1 * factorial(0)  => 1 * 1
factorial(2) returns 2 * factorial(1)  => 2 * 1 * 1
factorial(3) returns 3 * factorial(2)  => 3 * 2 * 1 * 1


// 3 * 2 * 1 * 1 = 6
```

Figure 2 - recursive factorial function. Source: (37)

Let's show some better and more realistic example. We can use the array from the previous example of higher order function and try to turn it into the recursive function. Many times, you need more than just simple recursive function. For those purposes, you can use helper function just like in the following example. The function **getNames** just contains new

array, that is going to be filled up with names, then there is a **helper** function declaration, which is actually the recursive one, and that function is of course called as well. As you can see, there is a base case again to check if the array still contains values. If it doesn't **return** is called without anything. If it does, name is pushed into the **newArr** and helper function is called recursively, but with an array that contains one value less than what it was given as an argument. Once the recursive **helper** function is done, **newArr** is returned from **getName**.

```
const getNames = (arr) => {
   const newArr = [];

   const helper = (subArr) => {
   if (subArr.length === 0) return;
      newArr.push(subArr[0].name);
      helper(subArr.slice(1));
   };

   helper(arr);

   return newArr;
};

console.log(getNames(arrs));
```

Now, the previous example is used just to explain how recursion works. Of course, that the use of **map** function is much simpler and straight forward in cases like this. BUT there are times, when you want to use simple loops, in those cases, you can definitely replace it by recursion.

4.10 Declarative rather than imperative

Functional programming is a declarative paradigm, meaning that the program logic is expressed without explicitly describing the flow control. (23)

Imperative programs spend lines of code describing the specific steps used to achieve the desired results — the **flow control: How** to do things. (23)

Declarative programs abstract the flow control process, and instead spend lines of code describing the **data flow: What** to do. The *how* gets abstracted away. (23)

Let's explain it in a metaphor

Declarative Programming is like asking your friend to draw a landscape. You don't care how they draw it, that's up to them. (38)

Imperative Programming is like your friend listening to Bob Ross tell them how to paint a landscape. While good ole Bob Ross isn't exactly commanding, he is giving them step by step directions to get the desired result. (38)

Okay, now some code example. Let's take something simple. For example, we want to loop through an array, remove some element and return that modified array as new array.

4.10.1 Imperative code example

As you can see in the following code. There are the exact instructions one after another.

```
movies = ["Godfather", "Nemo", "Scarface", "Grown Ups", "Twilight"];

movieToDelete = "Twilight";

const imperativeRemove = (arr, toDel) => {
   const newArr = [];

   for (let i = 0; i < arr.length; i++) {
      if (arr[i] !== toDel) newArr.push(arr[i]);
   }
   return newArr;
};

console.log(imperativeRemove(movies, movieToDelete));
```

4.10.2 Declarative code example

This code achieves the exact same functionality as the previous one. The difference here is, that you don't care about the exact implementation, you just say "filter given array" and that's it. It makes the code shorter and easier to read.

```
const declarativeRemove = (arr, toDel) =>
        arr.filter((item) => item !== toDel);

console.log(declarativeRemove(movies, movieToDelete));
```

4.11 FP summary

- Functional programming favours pure functions over shared state and side-effects (23) (26)
- All data should be immutable (23)
- Many generic reusable functions (higher order functions) that can act on anything you give it (23)
- The way to achieve more complicated logic is to compose many pure functions together via the **function composition** (23)
- Code in declarative rather that in imperative way. (23)

4.12 Advantages of Functional programming

So, now that we've gone through the concepts of functional programming, it can seem very confusing and it can make you feel like you're programming with handcuffs on. But it has numerous advantages.

4.12.1 It doesn't have side effects

Huge advantage of functional programming is, that your code doesn't have many side effects and the ones it has, are concentrated in one place, so you can easily manage them. This is great, because it makes your code clean and predictable. (39)

4.12.2 Data are immutable

This is great, because you know, that none of your functions directly mutates some data. You can be sure, that all functions just accept parameters, and return some value without directly mutating something outside of their scope. This means, that if there is a bug, you can just inspect those functions that are not working in a way you want. You simply don't have to go through the whole code just in a case that there is a function that mutates your data. (39)

4.12.3 It's clean, straightforward, succinct

Functional Programming has always been straightforward so it's easy to spot some inconsistencies and debug some bugs in the function. In fact, constructing functions are cleaner and easier to maintain than constructing a class as in OOP since you need to think in terms of imperative/procedural pattern such as designing class hierarchies, encapsulation, inheritance, methods, polymorphism. (39)

However, straightforward doesn't mean it's easier to adapt. If you came from OOP background, you need to make a shift in your thinking and especially make sure that your objects don't mutate. (39)

4.12.4 Lazy evaluation

It supports the concept of lazy evaluation, which means that the value is evaluated and stored only when it is required. (40)

5 Citovaná literatura

1. **MV, Thanoshan.** What exactly is a programming paradigm? . *freeCodeCamp.* [Online] 12. November 2019. https://www.freecodecamp.org/news/what-exactly-is-a-programming-paradigm/.
2. **Schwarzmüller, Maximilian.** *Academind.* [Online] 18. December 2019. https://academind.com/learn/javascript/functional-vs-oop-vs-procedural/.
3. **Differences between Procedural and Object Oriented Programming. *GeeksforGeeks.* [Online] 12. 4 2019. https://www.geeksforgeeks.org/differences-between-procedural-and-object-oriented-programming/.**
4. Felsen, Lili Ouaknin. Functional vs Object-Oriented vs Procedural Programming. *Medium.* [Online] 30. October 2017. https://medium.com/@LiliOuakninFelsen/functional-vs-object-oriented-vs-procedural-programming-a3d4585557f3.
5. Copyright © 2020 BBC. Procedures and functions. *bbc.* [Online] https://www.bbc.co.uk/bitesize/guides/zqh49j6/revision/1.
6. Eliason, Kenny. DIFFERENCE BETWEEN OBJECT-ORIENTED PROGRAMMING AND PROCEDURAL PROGRAMMING LANGUAGES - PAGE 2. *neobrand.* [Online] 1. August 2013. https://neonbrand.com/websites/development/procedural-programming-vs-object-oriented-programming-a-review/2/.
7. Larkin, Paul. What is Procedural Programming, and When Should You Use It? . *CARRERKARMA.* [Online] 10. 9 2020. https://careerkarma.com/blog/procedural-programming/.
8. © 2005-2020 Mozilla and individual contributors. Object-oriented JavaScript for beginners . *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_JS.
9. Jai. What is the objective of Object-Oriented Programming. *DEV.* [Online] 7. April 2019. https://dev.to/jai00271/what-is-the-objective-of-object-oriented-programming-4a58.
10. © 2005-2020 Mozilla and individual contributors. Classes. *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes.
11. Donovan, Steve. Class vs Object vs Instance . *codementor.* [Online] 13. March 2020. https://www.codementor.io/@stevedonovan/class-vs-object-vs-instance-14i2s2lu6r.
12. Elliott, Eric. Encapsulation in JavaScript . *Medium.* [Online] 23. October 2019. https://medium.com/javascript-scene/encapsulation-in-javascript-26be60e325b4.
13. JANSSEN, THORBEN. OOP Concept for Beginners: What is Encapsulation . *Stackify.* [Online] 30. November 2017. https://stackify.com/oop-concept-for-beginners-what-is-encapsulation/.
14. Copyright 1999-2020 by Refsnes Data. JavaScript Math Object . *w3schools.com.* [Online] https://www.w3schools.com/js/js_math.asp.
15. Deikun, Artem. What is behind syntactical sugar in ES6 classes. *Medium.* [Online] 12. December 2017. https://artem.today/what-is-behind-syntactical-sugar-in-es6-classes-6dfa4ab4d6a2.
16. Kukurba, Viktor. Object-oriented programming in JavaScript #4. Encapsulation. *Medium.* [Online] 7. April 2019. https://medium.com/@viktor.kukurba/object-oriented-programming-in-javascript-4-encapsulation-4f9165.
17. © 2005-2020 Mozilla and individual contributors. Private class fields . *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Private_class_fields.

18. SINGH, CHAITANYA. OOPs in Java: Encapsulation, Inheritance, Polymorphism, Abstraction . ***BeginnersBook.*** **[Online] https://beginnersbook.com/2013/03/oops-in-java-encapsulation-inheritance-polymorphism-abstraction/.**
19. JANSSEN, THORBEN. OOP Concept for Beginners: What is Abstraction? . ***Stackify.*** **[Online] 23. November 2017. https://stackify.com/oop-concept-abstraction/.**
20. Petkov, Alexander. freeCodeCamp. ***How to explain object-oriented programming concepts to a 6-year-old .*** **[Online] https://www.freecodecamp.org/news/object-oriented-programming-concepts-21bb035f7260/.**
21. Copyright 1999-2020 by Refsnes Data. Java Abstraction . ***w3schools.com.*** **[Online] https://www.w3schools.com/java/java_abstract.asp.**
22. Kukurba, Viktor. Object-oriented programming in JavaScript #3. Polymorphism. ***medium.*** **[Online] 6. April 2019. https://medium.com/@viktor.kukurba/object-oriented-programming-in-javascript-3-polymorphism-fb564c9f1ce8.**
23. Elliott, Eric. Master the JavaScript Interview: What is Functional Programming? ***Medium.*** **[Online] 4. January 2017. https://medium.com/javascript-scene/master-the-javascript-interview-what-is-functional-programming-7f218c68b3a0.**
24. Brasseur, Arne. Functional Programming: Pure Functions. ***sitepoint.*** **[Online] 17. September 2014. https://www.sitepoint.com/functional-programming-pure-functions/.**
25. Wiącek, Jonasz. FUNCTIONAL PROGRAMMING IN JAVASCRIPT. ***Software Brothers.*** **[Online] 28. January 2020. https://softwarebrothers.co/blog/functional-programming-in-javascript.**
26. Elliott, Eric. Master the JavaScript Interview: What is a Pure Function? ***Medium.*** **[Online] 26. March 2016. https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-pure-function-d1c076bec976.**
27. Kondov, Alexander. Functional programming paradigms in modern JavaScript: Pure functions. ***Hackernoon.*** **[Online] 3. November 2017. https://hackernoon.com/functional-programming-paradigms-in-modern-javascript-pure-functions-797d9abbee1.**
28. Czernek, Krzysztof. Functional JS #6: Function composition. ***Daily JS.*** **[Online] 26. June 2019. https://medium.com/dailyjs/functional-js-6-function-composition-b7042c2ccffa.**
29. © 2005-2020 Mozilla and individual contributors. First-class Function . ***MDN web docs.*** **[Online] https://developer.mozilla.org/en-US/docs/Glossary/First-class_Function.**
30. Tong, Richard. Practical Functional Programming in JavaScript - Side Effects and Purity. ***DEV.*** **[Online] 3. August 2020. https://dev.to/richytong/practical-functional-programming-in-javascript-side-effects-and-purity-1838.**
31. Functional JavaScript part 5: immutability basics. ***Code With Style.*** **[Online] 17. August 2017. https://codewithstyle.info/functional-javascript-part-5-immutability-basics/.**
32. © 2005-2020 Mozilla and individual contributors. Object.assign() . ***MDN web docs.*** **[Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign.**
33. Copyright © 2015–2020 Dan Abramov and the Redux documentation authors. Immutable Update Patterns. ***Redux.*** **[Online] https://redux.js.org/recipes/structuring-reducers/immutable-update-patterns.**
34. Arora, Sukhjinder. Understanding Higher-Order Functions in JavaScript. ***Bits and Pieces.*** **[Online] 23. October 2018. https://blog.bitsrc.io/understanding-higher-order-functions-in-javascript-75461803bad.**
35. Green, M. David. Recursion in Functional JavaScript . ***sitepoint.*** **[Online] 29. June 2015. https://www.sitepoint.com/recursion-functional-javascript/.**

36. Recursion. ***GeeksForGeeks.*** **[Online] 14. June 2020. https://www.geeksforgeeks.org/recursion/.**
37. Morelli, Brandon. Learn and Understand Recursion in JavaScript. ***codeburst.io.*** **[Online] 6. July 2017. https://codeburst.io/learn-and-understand-recursion-in-javascript-b588218e87ea.**
38. Mundy, Ian. Declarative vs Imperative Programming. ***codeburst.*** **[Online] 20. February 2017. https://codeburst.io/declarative-vs-imperative-programming-a8a7c93d9ad2.**
39. Recio, Sonny. Functional Programming in JavaScript: How and Why. ***Bits and Pieces.*** **[Online] 21. August 2019. https://blog.bitsrc.io/functional-programming-in-javascript-how-and-why-94e7a97343b.**
40. Bhadwal, Akhil. Functional Programming: Concepts, Advantages, Disadvantages, and Applications. ***hackr.io.*** **[Online] 24. August 2020. https://hackr.io/blog/functional-programming.**
41. Bhatia, Sagar. Procedural Programming [Definition] . ***hackr.io.*** **[Online] 24. August 2020. https://hackr.io/blog/procedural-programming.**