3. JavaScript Advanced

Obsah

1 Loops

So, what are loops and what are they used for? Loops offer a quick and easy way to do something repeatedly. (1) They are an essential part of every programming language, so you're going to use them a lot. There are many ways to create them in JavaScript: **for loop**, **do…while loop**, **while loop**, **for…in**, **for…of**. Each and every one of them is useful for something else, so don't worry, we will take a closer look at all of them. But let's first show code example of some easy loop.

Let's say that you want to log numbers from 1 to 10. So, what's happening there? **For** is a keyword, then in round brackets, you can see, that we're declaring a variable. This variable is going to change, so that's why it is defined with **let** variable. Then there is a semicolon, which needs to be there, so the compiler knows, that we finished with variable declaration. Then condition follows, this is where we say how many times, we want our loop to run. In this case, variable **i** is equal to 1 at first, and we want it to count numbers until 10, so that's exactly what is said there. Then, after another semicolon, we define what should happen to our **i** variable after every iteration. In this case, we say that variable **i** should be incremented by one after each iteration. **i++** is just a shorter version of the following: **i = i + 1**.

```
for (let i = 1; i <= 10; i++) {
   console.log(i);
}
```

Okay, that was just an easy example of a loop, but what are they actually used for? Well, they are very handy, when you want to run the same code over and over, each time with a different value. When do you need that? Mostly while you're working with arrays. (2) Let's dig deeper into the first loop, which is for loop.

1.1 For loop

For loop is used, when we want to say how many times should our loop run, and we actually know the number of times that our loop should run.

1.1.1 Syntax

In previous example, I already explained what you need to do in order to create for loop, but since this is a crucial part of JavaScript, let's rather show its syntax.

```
for (initial expression; condition expression; increment expression) {
 // code block to be executed
}
```

1.1.2 Code example

In the following example, we have an array of cars and we want to show them all. In this example, cars are just logged into the console, that would've been different in real code of course. In real code, there would've been some HTML code rendering, but that foor loop logic would've looked exactly the same.

Okay, so what is actually happening there. Arrays are indexed from number 0 by default, that's why for loop variable starts with the value of zero. As we learned in previous class, you can get array length by using its property **length**, then again, variable **i** is incremented by 1 after each iteration. Inside loop, each value of an array is accessed via its index. So for example, this is how the first iteration looks like: **console.log(cars[1]);**

```
const cars = ["Audi A6", "Škoda superb 3", "BMW X6", "Audi R8"];

for (let i = 0; i < cars.length; i++) {
   console.log(cars[i]); //first iteration i value - Audi A6
}
```

Code above is a great example of, why **let/const** are much better than **var**. It's a convention, that some letter is used for **for loop** variable, and majority of programmers use letter **i** for this purpose, or if there is a need for nested loop, letter **j** is used as well. Sometimes you have a function with more than one loop inside of it and because **let/const** variable declaration has block scope, you can use letter **i** for all of those loops with no worries, as they exist only within that loop.

1.2 While loop

You need to specify a condition in while loop as well as in for loop, but it works slightly different. While loop executes its block of code as long as a specified condition is true. (3) This is handy, when you need to run your code an unknown number of times, until your condition is met.

1.2.1 Syntax

```
while (condition) {
 // code block to be executed
}
```

1.2.2 Code example

Let's first code the exact same loop as we did previously with for loop. As you can see, you need to declare **i** variable outside of while loop and you need to place **i++** before **console.log**, because if you would put it in front of it, it would start with array value with an index of 1. In other words, this works, but it's just better and cleaner to use for loop when we know the number of iterations, which we do in this case, as we want to iterate **array.length** number of times.

```
let i = 0;
while (i < cars.length) {
   console.log(cars[i]);
   i++;
}
```

Okay, so when is it actually useful? Let's talk about some examples. For example, when you need to get some desired input from user. You ask him for an input, he gives it to you, then you check if it is the right one, and if it is not, you ask him for an input again, and that keeps on going until the user inserts the correct input. Another example is, when you create a game, and you want your game to keep going until the player either wins or clicks on exit button.

1.3 Side note – break and continue

1.3.1 Break keyword

There is another way to end a loop. You can use it with any loop type. We learned about this keyword in switch statements section. It's the **break** keyword. Following code goes until j is equal to 2 and that it stops its executed exactly on the line, where the **break** keyword is used.

```
let j = 0;
while (j < 5) {
  if (j === 2) {
    break;
  }
  console.log(j); // last log is: 1
  j++;
}
```

```
// console.log(j); //logs out 2
```

1.3.2 Continue keyword

This keyword is used when we need to skip some iteration(s).

The **continue statement** terminates execution of the statements in the current iteration of the current loop and continues execution of the loop with the next iteration. (4)

In the following code, we have an array of numbers, and only the odd ones are logged into the console.

```
const numbers = [1, 3, 2, 5, 7, 10, 12];
for (let i = 0; i < numbers.length; i++) {
   if (numbers[i] % 2 === 0) {
      continue;
   }
   console.log(numbers[i]);
}
```

1.4 Side note - For loop vs while loop

There are many cases, where you can actually use both of these loops. In that case, it's safer to use **for loop**, because **for loop** allows you to create a local variable in the loop declaration, which is slightly more readable and less error prone.

If you want to use **while loop**, you absolutely can, but you always need to be certain, that your **while loop condition** is going to be met at some point, otherwise you will create an endless loop, which can cause your app to freeze in some cases.

You can find some articles on the internet, where it is stated, that while loop is faster. (5) But you should never try to optimize your code prematurely, as it can really slow you down. It's always better to create something, then test it, let real users to use it, and only then if it is slow, try to optimize your code. (6) (7) (8)

1.5 Do…while loop

Do…while loop is almost identical do while loop with one key difference. Condition of do…while loop is checked at the end of each iteration, so the code of this loop runs at least once, even when its condition is false right at the start. (9)

1.5.1 Code example

In the following code, the condition of loop is clearly not met right at the start, but the code runs once anyway.

```
let number = 1;
do {
   console.log(number);
   number++;
} while (number < 1);
```

1.6 For…of loop

As I said earlier, loops are used for arrays a lot. When you want to do something with an array, you mostly need to iterate over all of its items. That's something you can absolutely do, and as you already know, **for loop** fits for this purpose, but you always need to do write this: **for (let i=0; i< array.length;i++)**. The only thing which may differ, is the loop variable name, but otherwise it's exactly the same every time. Well, this is annoying right? Having to write all that code, just to iterate over an array.

That's why **for…of** loop exists. It's not something completely different from normal **for loop**, but it gives us a very clean and concise syntax to iterate over iterable objects (e.g. String or Array). (10) (11)

1.6.1 Code examples

Let's iterate over an array we've seen already in section **1.1 For loop**. In **for…of** loop rounded brackets, you first declare a variable, which you give whatever name you want, but it's a convention to give it the same name as we have for array, but in singular. Then you say which array you want to iterate over and that's it.

Notice that I'm using **const**, I can do that, because I'm not reassigning its value anywhere in that loop block of code.

```
for (const car of cars) {
    console.log(car);
}
```

As I said, you can iterate over a string as well.

```
const iterateMe = "Rainbow";
for (const letter of iterateMe) {
    console.log(letter);
}
```

1.7 For…in loop

For…in loop is used to iterate over the properties of an object (the object keys). (11) It iterates over keys, because once we have a key, we can access its value as well.

1.7.1 Code example

In the following code example, we iterate over john object, and log property names (keys). Property values are accessed as well, via its property names.

```
for (key in john) {
    console.log(key); //key
    console.log(john[key]); //value
}
```

2 JavaScript Execution context and Execution stack

Now that we've gone through the very basics of JavaScript language, it's good to dig a little bit deeper into how JavaScript actually works behind the scenes, as it helps you to understand your code better.

2.1 Execution context

To understand how Javascript Engines (program that executed JavaScript code) run your code we should first understand the **Execution Context**. Every time you run Javascript in a browser, your code is broken down into smaller pieces and executed. Those smaller pieces are execution contexts. (12)

Execution context is defined as the environment in which the JavaScript code is executed.

There are three types of execution context: Global execution context, function execution context and Eval function execution context. (13)

2.1.1 Global Execution context

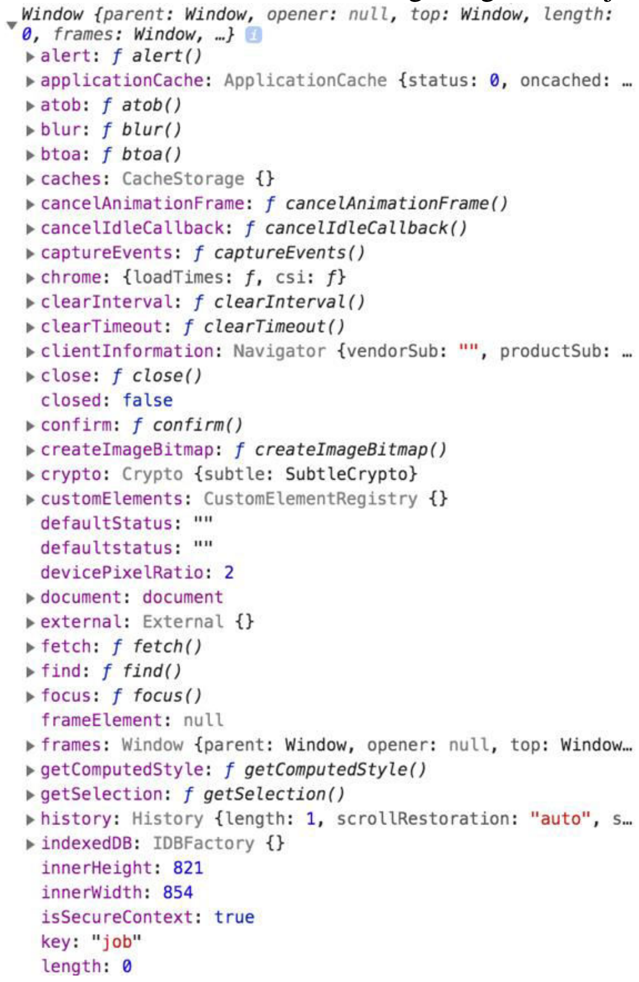
This is the default execution context when JavaScript Engine starts to interpret a JavaScript file. Basically everything, what is not defined inside some function, is part of the global context. There is only one global execution context. (13)

Global Execution context it consists of two things:

- Global object – the global object in a browser is called **window**. This window object is very important. We will learn about this more, once we will be learning about how JavaScript works with HTML. For now, just keep in mind, that window represents the window in which the script is running. (14)

- this – we will talk about **this** keyword in detail a bit later, but for now it's enough for you to know, that **this** keyword refers to the object it belongs to. For example, in global execution context, **this** keyword refers to the global object, which is **window**.

Let's see how it looks. Following image shows just a chunk of a window object.



```
▼Window {parent: Window, opener: null, top: Window, length:
 0, frames: Window, …}
  ▶alert: ƒ alert()
  ▶applicationCache: ApplicationCache {status: 0, oncached: …
  ▶atob: ƒ atob()
  ▶blur: ƒ blur()
  ▶btoa: ƒ btoa()
  ▶caches: CacheStorage {}
  ▶cancelAnimationFrame: ƒ cancelAnimationFrame()
  ▶cancelIdleCallback: ƒ cancelIdleCallback()
  ▶captureEvents: ƒ captureEvents()
  ▶chrome: {loadTimes: ƒ, csi: ƒ}
  ▶clearInterval: ƒ clearInterval()
  ▶clearTimeout: ƒ clearTimeout()
  ▶clientInformation: Navigator {vendorSub: "", productSub: …
  ▶close: ƒ close()
   closed: false
  ▶confirm: ƒ confirm()
  ▶createImageBitmap: ƒ createImageBitmap()
  ▶crypto: Crypto {subtle: SubtleCrypto}
  ▶customElements: CustomElementRegistry {}
   defaultStatus: ""
   defaultstatus: ""
   devicePixelRatio: 2
  ▶document: document
  ▶external: External {}
  ▶fetch: ƒ fetch()
  ▶find: ƒ find()
  ▶focus: ƒ focus()
   frameElement: null
  ▶frames: Window {parent: Window, opener: null, top: Window…
  ▶getComputedStyle: ƒ getComputedStyle()
  ▶getSelection: ƒ getSelection()
  ▶history: History {length: 1, scrollRestoration: "auto", s…
  ▶indexedDB: IDBFactory {}
   innerHeight: 821
   innerWidth: 854
   isSecureContext: true
   key: "job"
   length: 0
```

Figure 1 - Chunk of a window object. Source: my own image.

When a script executes, the Global execution context is created in two phases: **Creation phase** and **Execution phase**. (12)

In **Creation Phase**, JS engine will not execute your code line-by-line, it only:

- creates the **global object** (You can also find name **Variable object**) and **this** (12)
    - **global object (or Variable object)** – is an object storing data related to an execution context. (15)
- set up memory for variables and functions (12)
    - this is where scope and scope chain is determined (you will learn more about scope and scope chaining soon) (16)
- initialize all **var** variables with **undefined** and puts function declarations in memory. (12) (This is how Hoisting happens – we learned about this in the previous class)

In **Execution phase**, JS engine starts navigating your code line-by-line, executes it and assigns real values to variables which are already stored in memory. (12)

2.1.2 Function execution context

Function execution context is defined as the context created by the JS engine whenever it finds any function call (function needs to be called somewhere in the code, in order to have its own execution context). Each function has its own execution context, so there can be many of them. (13) (17) When JS engine finds function call and needs to create new execution context, it goes through almost the same steps as for global scope (see **Creation phase** and **Execution phase**, in previous paragraph), except for two differences:

1. Function execution context is not creating its own **this** during creation phase, so what is the value of **this** in function execution context? (12) Well, that's not that easy to answer, so I will talk about this a bit later, as it is better for us to first understand the big picture and then discuss the details.
2. function execution context doesn't create global object (or variable object), it creates activation object instead. It is basically the same thing, except for that, in addition to what the variable object contains, it also contains the arguments object and formal parameters (parameters passed into the function). (15)

What happens, when I call function within another function? Whenever there is a function call, within another function, then that called function's execution context is stacked at the top of its parent function's execution context. This is why the place, where these execution contexts are stacking at the top of each other, is called **Execution stack**, or **Call stack**.

2.2 Execution stack (or call stack)

Execution stack is a stack data structure, i.e. last in first out data structure, to store all the execution contexts created during the life cycle of the script.

Great video about how it works - https://www.youtube.com/watch?v=W8AeMrVtFLY

3 Scope

I mentioned this topic in last class when I was talking about differences between **var** and **let/const**. As a reminder - Scope determines the accessibility (visibility) of variables. (18)

JavaScript has function scope, which means, that every function creates a new scope, so variables declared inside function are not accessible outside of that function.

- This used to be true before let/const variable declarations, but you still find this information on many websites. Now **var** has function scope, but **let/const** which are preferred to use over **var** have block scope.

There are two types of scopes:

- Local
- Global

3.1 Global scope

The global scope is the scope that contains, and is visible in, all other scopes. (19)

Everything, what is defined outside of any function or curly brackets (if the variable is declared via **const/let**!), is in the global scope. (20)

It is recommended, to not put many things in global scope, because there is a chance of naming collisions, where two or more variables are named the same. The second reason, why it's better to avoid declaring everything globally, is that it is better to split your code into smaller chunks (functions), which are focused on some specific functionality. It's very confusing to read code, where you have everything at one place, as it turns into spaghetti code very quickly.

3.2 Local scope

There are two local scopes, dependant on what keyword we use for variable declaration

Function scope

- We talked about this in previous class, so just a reminder, function scope means, that variables declared inside a function, are accessible only within that function.
- Variables declared via **var** keyword have function scope

Block scope

- Block scope means, that variables declared inside a code block { }, are accessible only within that block of code.
- Variables declared via **const/let** keywords have block scope

3.3 Scope chain

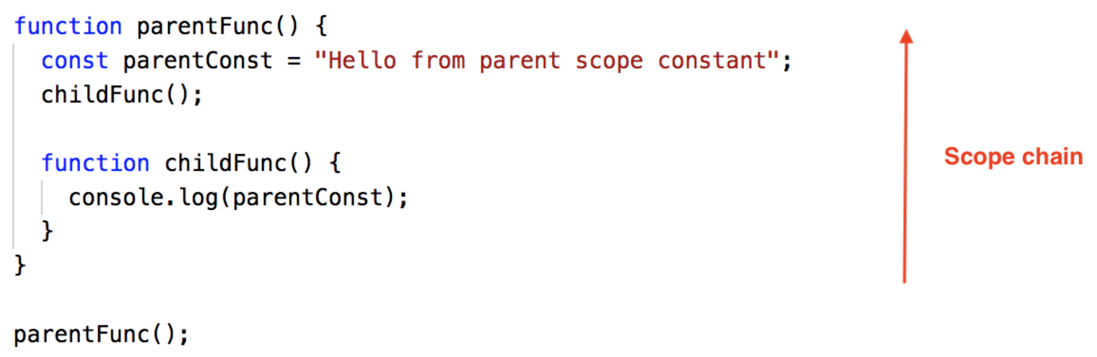
Scope chaining is an important concept, so what is it? Well, scopes can be nested. Every scope is always connected to one or more scope in their back forming a chain or a hierarchy, except the global scope, because global scope is not nested in any other scope, so it sits at the top of this hierarchy, or chain. (21)

Whenever the compiler encounters a variable or an object, it traverses the whole scope chain of the current execution context looking for it. (21)

Enough of theory, let's look at some code, to actually see how it works.

We are calling variable **parentConst** from a **childFunc**, the interpreter first looks into a **childFunc**, if the variable **parentConst** exists there. It doesn't so, the interpreter looks a level higher in scope chain (into **parentFunc**), it finds it there, so the variable is logged into the console.

As you can see in the following example, nested scope has always access to its parents (wrappers) scopes, BUT it doesn't work the other way around! Parent does NOT have access to its child scope, so it does NOT have access to its variables and functions.



```
function parentFunc() {
  const parentConst = "Hello from parent scope constant";
  childFunc();

  function childFunc() {
    console.log(parentConst);
  }
}

parentFunc();
```

Figure 2 - scope chain. Source: my own image

3.4 Lexical scope

This is very important as well, in order to understand, how the interpreter determines, who is child and who is parent. In order to decide this, JavaScript uses lexical scope. (22)

It basically means, that scope of variables is determined by the position, where these variables are actually written. (23) What do I mean by that?

Let's change previous code example a bit. The only thing I changed, is that I took **childFunc** and wrote it outside of **parentFunc**. **childFunc** doesn't have access to variable

parentConst, because **childFunc** is now written outside of **parentFunc**, so even though **childFunc** is still called in **parentFunc** it is lexically outside of it, so it is not a child of **parentFunc** therefore it doesn't have access to **parentFunc's** variables anymore.

```
function parentFunc() {
   const parentConst = "Hello from parent scope constant";
   childFunc();
}

function childFunc() {
   console.log(parentConst); // ReferenceError: parentConst is not defined
}

parentFunc();
```

4 Closures

Closures are widely used in JavaScript.

A closure is a feature in JavaScript where an inner function has access to the outer (enclosing) function's variables (thanks to scope chain) (24)

Closures work thanks to lexical scoping.

We've actually already seen very simple example of a closure, in section **3.3 scope chain**.

What is a closure:

- Basically, every function which is accessing its outer variables and data, is a closure.
- Closure is when a function is able to access its lexical scope, even when that function is executing outside its lexical scope. (25)
- Inner functions can access its parent scope, even after the parent function is already executed. (25)

It's very hard to understand closures without a code example, so let's have a look at one. In the following code, there is an outer function **pFunc**, with a local variable **string**. There is also function **closureFunc** declared and at the end, function reference is returned (notice, that the function is not called, as there are no rounded brackets in the return statement).

Then there is a constant **callMeLater**, to which we assign a return value of a **pFunc** (notice, that **pFunc** is called!). The return value of **pFunc** is a reference to an inner function **closureFunc**, now we can call this inner function, while having access to its outer's function variables, even though its outer function has been already executed.

```
function pFunc() {
   const string = "I have";
   function closureFunc(thing) {
      console.log(`${string} ${thing}`);
   }
   return closureFunc;
}

const callMeLater = pFunc();
callMeLater("a pen"); //I have a pen
callMeLater("an apple"); //I have an apple
callMeLater("an apple pen"); //I have an apple pen
```

Side note – this is very confusing at first, but also very crucial to understand. You simply have practice it, to really understand it.

Okay, so that was a simple example, but when is this used in real applications? Well, it's actually used a lot, when you need to deal with user actions and inputs. At the beginning, you declare what should happen, when user clicks on a button, or writes something, but you simply don't know when this is going to happen. Thanks to closures, you can just define what should happen, and simply say for example "Do that, when user clicks on a button"

Interesting videos about this topic:

- https://www.youtube.com/watch?v=-jysK0nlz7A
- https://www.youtube.com/watch?v=3a0I8ICR1Vg
- https://www.youtube.com/watch?v=CQqwU2Ixu-U

5 Simplified OOP, Inheritance, Prototype, Prototype chain and JS Classes

Everything in JavaScript, what is not a primitive value, is an object.

5.1 Simplified Object-oriented programming

As a programmer, you often work on apps and web apps, which consist of thousands of lines of code, so your aim is not only to write code which works, but also code, which is readable and easy to work with.

There are few ways to achieve that, one of them is a programming paradigm called Object oriented programming. This paradigm allows you to share data, structure code and keep code clean, reusable and readable.

We could talk about this topic for hours and it can get complicated very quickly. In fact, we will have on entire class dedicated to OOP and Functional programming, but for now, let's keep things simple and say, that object-oriented programming heavily depends on objects.

These objects are used to model real world things that we want to represent inside our programs, and/or provide a simple way to access functionality that would otherwise be hard or impossible to make use of. (26)

Objects can contain related data and code, which represent information about the thing you are trying to model, and functionality or behaviour that you want it to have. Object data (and often, functions too) can be stored neatly (the official word is **encapsulated**) inside an object package (which can be given a specific name to refer to, which is sometimes called a **namespace**), making it easy to structure and access; objects are also commonly used as data stores that can be easily sent across the network. (26)

These objects cooperate and inherit from each other. Many other languages use classes, which are specific type of object, to achieve those reusable objects from which you can inherit, but JavaScript is using **constructors**, and **prototypes**. Even though it nowadays has **classes**, it is still using prototypes under the hood!

5.2 Constructor

Okay, so JavaScript is using constructors, but what are they, and how are they used? Constructor is something like a blueprint for some type of object we want to represent, for example a car. (27) We can have many cars of different brand, engine, color, etc, so the actual values differ, even though we want to have same information about each and every car.

Okay, enough of theory, let's show some example of normal objects. In following example, we have two completely different cars, but we need to store the same types of information about them. This code works completely fine, but let's imagine a situation, when you want to create another car somewhere later in code, you need to come back to already existing car objects, and look at what properties you need to create, etc. That's very annoying, difficult to find out, that there is some connection between all those cars, and it slows you down. Wouldn’t it be better to just have some blueprint, which you can use later on and just fill the information, that this blueprint requires you to fill?

```
const car1 = {
   brand: "BMW",
   color: "red",
   kilometers: 220320,
   wasCrashed: true,
};

const car2 = {
   brand: "Audi",
   color: "black",
   kilometers: 21003,
   wasCrashed: false,
};
```

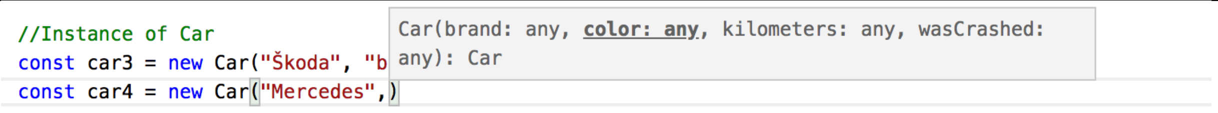
Let’s see how we can you a constructor to create that blueprint and use it in code later on, to create an instance of object with that given blueprint. It is convention that name with first uppercase letter is used for constructors. It’s just much easier to recognize a constructor in code thanks to this convention.

When we want to create a new object instance, the operator **new**, needs to be used. It is because by using this keyword, we’re telling a browser to create a new instance of an object. (26) You can also notice a keyword **this**, don’t worry, I will talk about this keyword soon (see **7. “this” keyword**), for now just accept, that it needs to be there.

```
//Constructor
function Car(brand, color, kilometers, wasCrashed) {
   this.brand = brand;
   this.color = color;
   this.kilometers = kilometers;
   this.wasCrashed = wasCrashed;
}

//Instance of Car
const car3 = new Car("Škoda", "blue", 132032, false);
```

The advantage of using a constructor is, that is also tells you, what values you need to fill in, as shown in the following example.



```
//Instance of Car
const car3 = new Car("Škoda", "b
const car4 = new Car("Mercedes",)
```

```
Car(brand: any, color: any, kilometers: any, wasCrashed:
any): Car
```

Figure 3 - Example of object constructor hinting. Source: my own

5.2.1 Side note – new operator

Whenever we want to create a new object instance from some constructor, we need to use **new** operator, but why? The answer is very simple. New operator lets developers create an instance of a user-defined object type or of one of the built-in object types that has a constructor function. (28)

5.3 Inheritance

Disclaimer – in modern JavaScript, we can actually use classes for inheritance, but JavaScript doesn’t have real classes, they are just syntactic sugar! Behind the scenes, prototypes are still used to achieve inheritance. That’s why we will learn about the prototype inheritance and prototype chain first, so you have some idea, how JavaScript inheritance actually works, before we will move on to classes.

Sometimes you need to extend some object. For example, you have object constructor type **Car**, but then you want to have another object type **SportCar**, objects of type **SportCar** need to have the exact same values and methods as objects of type **Car** but they also need to contain information about who is their driver and how many races they've won so far.

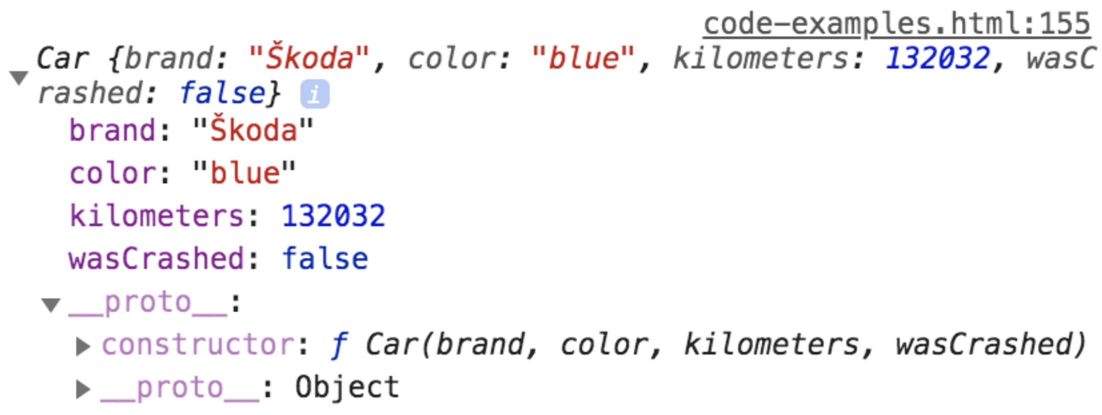
In order to achieve this, you can create a constructor, which inherits blueprint of a **Car** constructor, and adds some new properties to this blueprint. This is possible thanks to **prototypes** and **prototype chain**. (29)

5.3.1 Prototype

JavaScript is a prototype-based language, so each object in JavaScript, has its prototype, which acts as a template object that it inherits methods and properties from. (30)

Let's come back to our created cars and inspect **car3**.

As we can see, **car3** is an instance of a **Car** constructor. There is nothing much in our **Car** constructor, only constructor itself, where we define what properties we need to fill in.

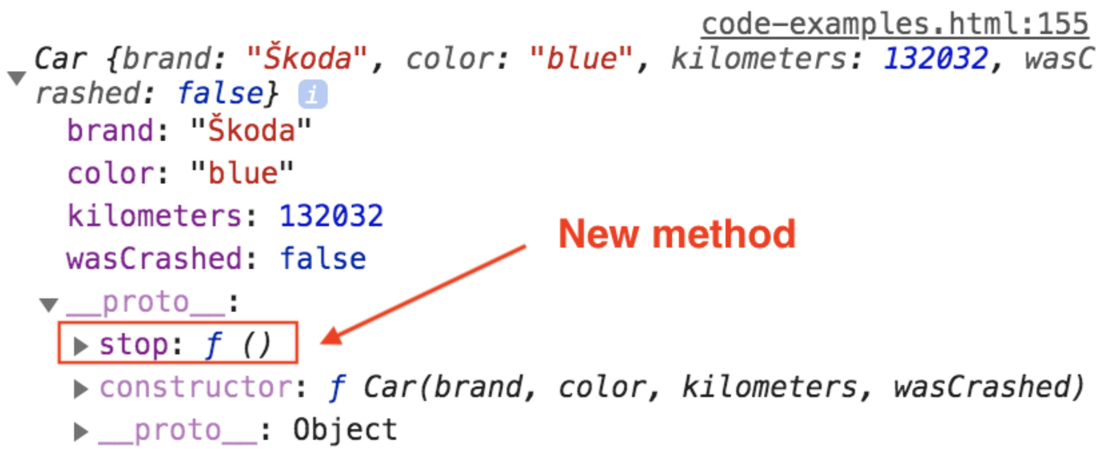


```
                                                   code-examples.html:155
▼ Car {brand: "Škoda", color: "blue", kilometers: 132032, wasC
  rashed: false}
    brand: "Škoda"
    color: "blue"
    kilometers: 132032
    wasCrashed: false
  ▼ __proto__:
    ▶ constructor: f Car(brand, color, kilometers, wasCrashed)
    ▶ __proto__: Object
```

But we can extend that **Car** constructor and add some method to its prototype.

```
Car.prototype.stop = function () {
    console.log("Car stopped");
};
```

Let's have a look at the **car3** again. As we can see, a function **stop** was added to the prototype, and we can call it. Now, each instance of a Car constructor will have access to a **stop** method.



```
                                                   code-examples.html:155
▼ Car {brand: "Škoda", color: "blue", kilometers: 132032, wasC
  rashed: false}
    brand: "Škoda"
    color: "blue"
    kilometers: 132032
    wasCrashed: false
  ▼ __proto__:
    ▶ stop: f ()
    ▶ constructor: f Car(brand, color, kilometers, wasCrashed)
    ▶ __proto__: Object
```

5.3.2 Prototype chain

There is actually one more interesting thing in our previous example. At the very bottom, you can notice, that there is another prototype – **Object**. Well, **car3** is not only an instance of **Car**, it is also an **object**, and because of that, it also has access to **Object's** properties and methods. This is the prototype chain in action.

Interesting video about prototypes and prototype chain - https://www.youtube.com/watch?v=hS_WqkyUah8

5.3.3 Code example of inheritance

Enough of theory, let's see inheritance on a code example. As you can see, it looks very complicated and it can be very confusing. That's why **classes** were introduced, but again, remember, that classes in JavaScript don't actually exists. At the end, prototype chain and prototype inheritance are used behind the scenes!

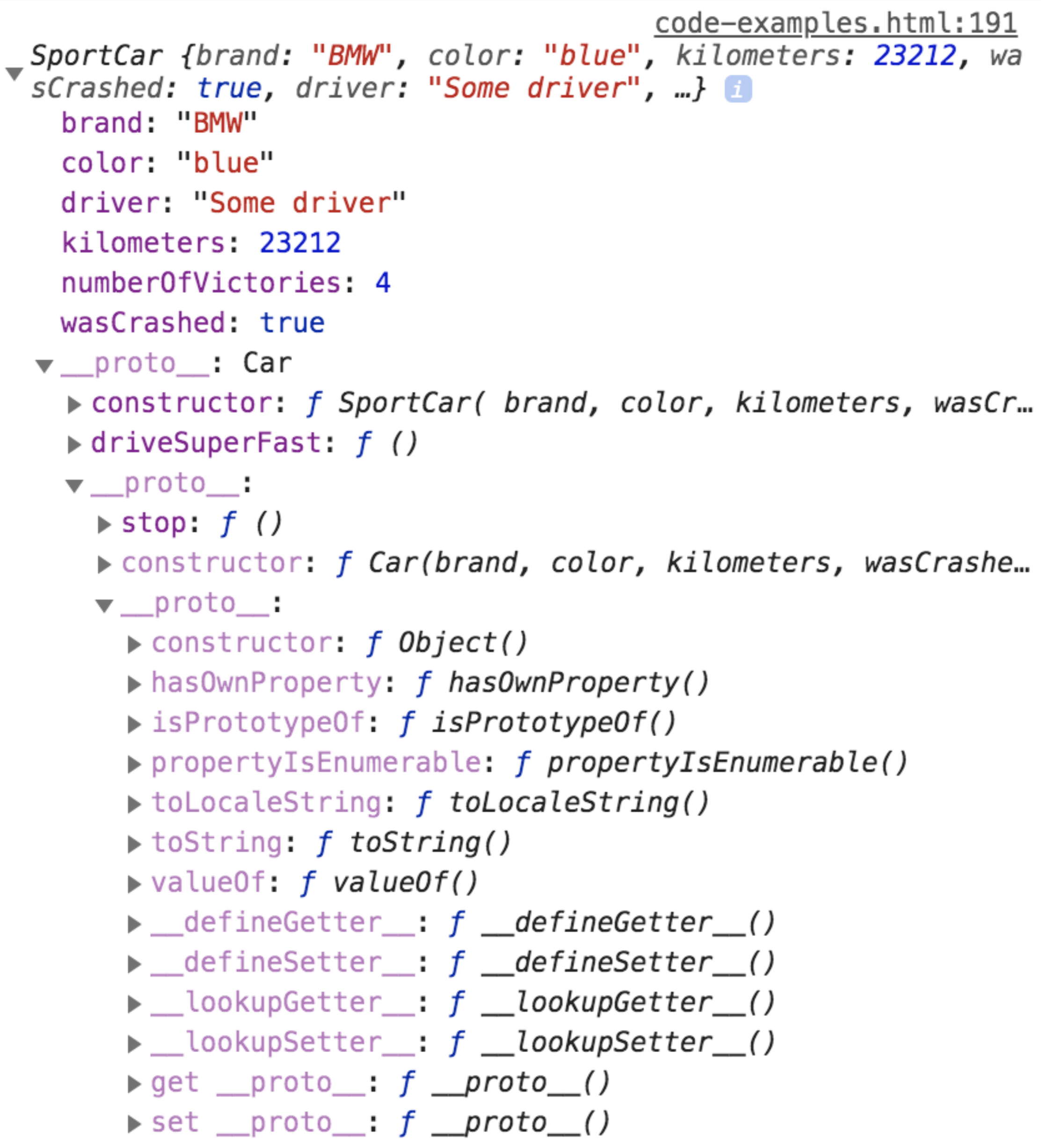
```
function SportCar(brand, color, kilometers, wasCrashed, driver,
numberOfVictories) {
   Car.call(this, brand, color, kilometers, wasCrashed); //inherits from
Car constructor, and gets its arguments
//.call() here calls Car constructor with its "this" value and other
arguments
   this.driver = driver;
   this.numberOfVictories = numberOfVictories;
}

SportCar.prototype = Object.create(Car.prototype); // creates new object
constructor based on the prototype of Car constructor
SportCar.prototype.constructor = SportCar; //assigns SportCar
constructor, because Car constructor was assign in .create()

SportCar.prototype.driveSuperFast = function () {
   //example of how to add a method to a SportCar constructor
   console.log("Car is driving super fast");
};

const sportCar1 = new SportCar("BMW", "blue", 23212, true, "Some driver",
4);
```

Let's inspect prototype of our **sportCar1**. As you can see in the following example, thanks to prototype chain, **sportCar1** has access to all methods and properties of **SportCar**, **Car** and **Object**. So for example thanks to **Car**, it can access method **stop(),** which we added to the **Car** constructor in one of our previous code examples.



```
                                                    code-examples.html:191
▼ SportCar {brand: "BMW", color: "blue", kilometers: 23212, wa
  sCrashed: true, driver: "Some driver", …} i
    brand: "BMW"
    color: "blue"
    driver: "Some driver"
    kilometers: 23212
    numberOfVictories: 4
    wasCrashed: true
  ▼ __proto__: Car
    ▶ constructor: ƒ SportCar( brand, color, kilometers, wasCr…
    ▶ driveSuperFast: ƒ ()
    ▼ __proto__:
      ▶ stop: ƒ ()
      ▶ constructor: ƒ Car(brand, color, kilometers, wasCrashe…
      ▼ __proto__:
        ▶ constructor: ƒ Object()
        ▶ hasOwnProperty: ƒ hasOwnProperty()
        ▶ isPrototypeOf: ƒ isPrototypeOf()
        ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
        ▶ toLocaleString: ƒ toLocaleString()
        ▶ toString: ƒ toString()
        ▶ valueOf: ƒ valueOf()
        ▶ __defineGetter__: ƒ __defineGetter__()
        ▶ __defineSetter__: ƒ __defineSetter__()
        ▶ __lookupGetter__: ƒ __lookupGetter__()
        ▶ __lookupSetter__: ƒ __lookupSetter__()
        ▶ get __proto__: ƒ __proto__()
        ▶ set __proto__: ƒ __proto__()
```

Figure 4 - Prototype chain of an object sportCar1. Source: my own

Great videos about prototypes, inheritance and prototype chain

- https://www.youtube.com/watch?v=CpmE5twq1h0
- https://www.youtube.com/watch?v=3AKh0-PDsMw

5.3.4 Side note – create()

In our last code example, you could've noticed keyword **create**. What is it doing?

Well, that's simple, the **Object.create()** method creates a new object, using an existing object as the prototype (template) of the newly created object. (31) Which is exactly what we wanted to do right? We wanted to use already existing constructor **Car** and create new constructor **SportCar** which just extends **Car** constructor.

5.3.5 Side note – call()

There was actually another new keyword in previous code example. It was **call()**. The **call()** allows for a function/method belonging to one object to be assigned and called for a different object. (32)

That’s exactly what we wanted to do. We wanted to extend the **Car**, so we needed to actually call the **Car** and its arguments, and assign it to our extended **SportCar** constructor.

5.4 Classes

So, as I said, classes are primarily syntactical sugar over JavaScript's existing prototype-based inheritance. (33)

Class is a special type of function to which you can assign properties and methods.

Let’s see how we can declare a class, and make another class inherit from it. In the following example, we aim for the same functionality as in section **7.3.3 Code example of inheritance**. We simply want to have a class **SuperCarClass**, which inherits from **CarClass**. (You don’t need to include word “Class” in your class name, I just did it to avoid name clash in my code examples file)

First, let’s create **CarClass**. In order to create a class, you need to use a **keyword class**. Then a constructor is created.

The **constructor()** method is a special method for creating and initializing objects created within a class. (34)

The **constructor()** method is called automatically when a class is initiated, and it has to have the exact name "constructor", in fact, if you do not have a constructor method, JavaScript will add an invisible and empty constructor method. (27)

There can only one constructor() method in class. (27)

Then class method **stop()** is declared. This is like a function, you can create as many methods as you need and give them whatever name and functionality you want.

```
class CarClass {
   constructor(brand, color, kilometers, wasCrashed) {
      this.brand = brand;
      this.color = color;
       this.kilometers = kilometers;
       this.wasCrashed = wasCrashed;
   }

   stop() {
   console.log("Car stopped");
   }
}
```

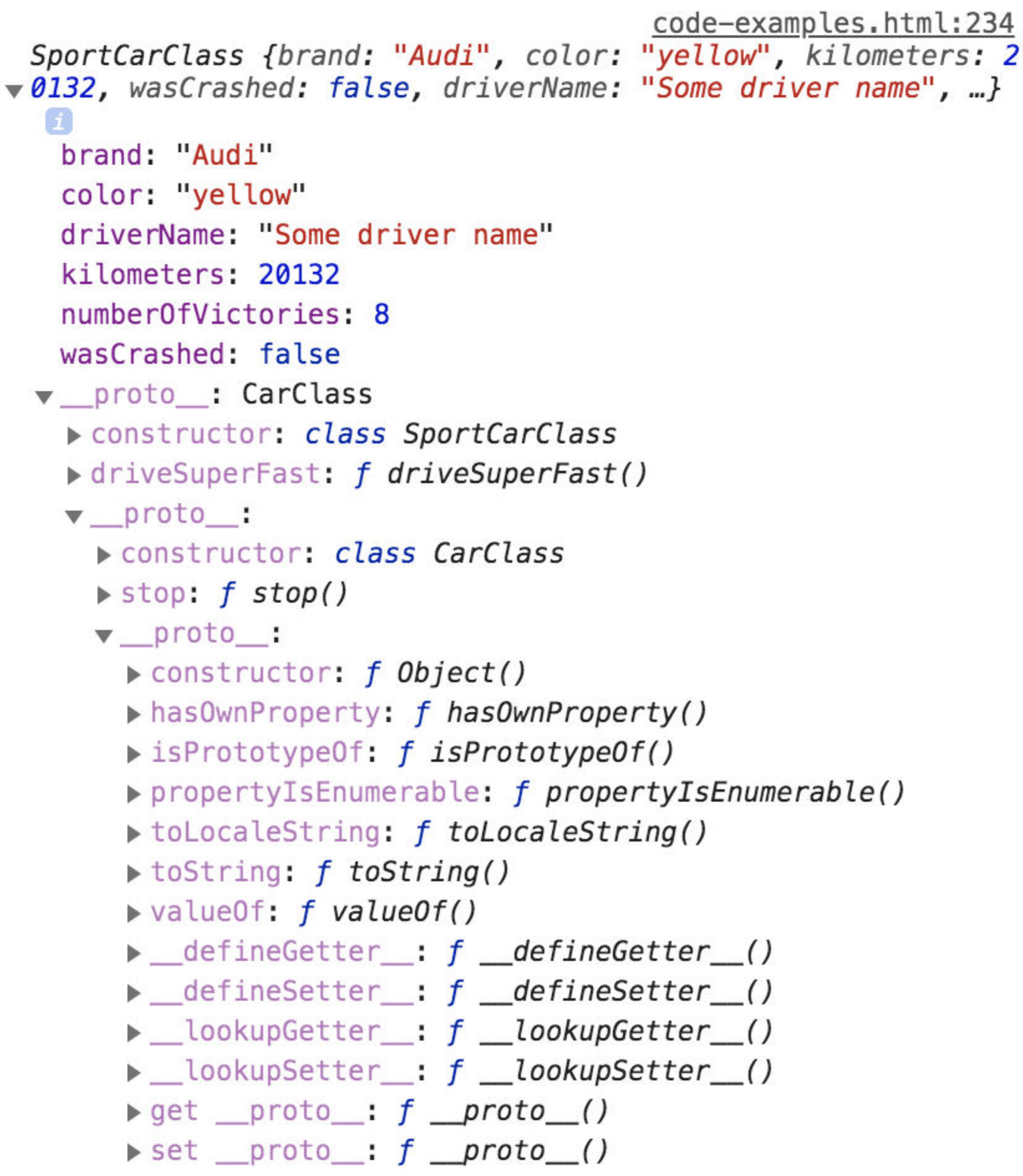
So, now that we’ve seen our first class, let’s create another class **SportCarClass**, which inherits from the **CarClass**, and let’s create an instance of **SportCarClass**. The first thing you can notice, is the use of a keyword **extends**. This keyword is used to create a child class of another class. (35)

You can notice a very odd thing in our constructor, which is **super()**. What is doing? It’s used in child classes, when you want to call parent’s constructor, which we want, because we need to add some more properties to constructor. (36)

```
class SportCarClass extends CarClass {
   constructor(brand, color, kilometers, wasCrashed, driverName,
numberOfVictories) {
      super(brand, color, kilometers, wasCrashed); //You need to call
this, when you have derived class with a constructor, which is exactly
what we have here
      this.driverName = driverName;
      this.numberOfVictories = numberOfVictories;
   }
   driveSuperFast() {
   console.log("Drive super fast");
   }
}
```

```
const mySportCar = new SportCarClass("Audi", "yellow", 20132, false,
"Some driver name", 8);
mySportCar.stop(); // Can call methods from parent class
```

As you could see in our last code example, except for those new keywords, it’s much more readable and less confusing, than using just prototypes. But never forget, it’s still working on the same principle. It’s still using prototypes and prototype chain to achieve inheritance! Let’s have a look at our **mySportCar** so you can see that it’s pretty much the same.



```
                                              code-examples.html:234
SportCarClass {brand: "Audi", color: "yellow", kilometers: 2
▼0132, wasCrashed: false, driverName: "Some driver name", …}
  brand: "Audi"
  color: "yellow"
  driverName: "Some driver name"
  kilometers: 20132
  numberOfVictories: 8
  wasCrashed: false
 ▼__proto__: CarClass
  ▶constructor: class SportCarClass
  ▶driveSuperFast: ƒ driveSuperFast()
  ▼__proto__:
    ▶constructor: class CarClass
    ▶stop: ƒ stop()
    ▼__proto__:
      ▶constructor: ƒ Object()
      ▶hasOwnProperty: ƒ hasOwnProperty()
      ▶isPrototypeOf: ƒ isPrototypeOf()
      ▶propertyIsEnumerable: ƒ propertyIsEnumerable()
      ▶toLocaleString: ƒ toLocaleString()
      ▶toString: ƒ toString()
      ▶valueOf: ƒ valueOf()
      ▶__defineGetter__: ƒ __defineGetter__()
      ▶__defineSetter__: ƒ __defineSetter__()
      ▶__lookupGetter__: ƒ __lookupGetter__()
      ▶__lookupSetter__: ƒ __lookupSetter__()
      ▶get __proto__: ƒ __proto__()
      ▶set __proto__: ƒ __proto__()
```

Figure 5 - Class instance logged into the console as a proof, that classes are still using prototypes and prototypes chain. Source: my own image

Videos about classes
- https://www.youtube.com/watch?v=T-HGdc8L-7w
- https://www.youtube.com/watch?v=Tllw4EPhLiQ

6 “this” keyword

So, we’ve bumped into **this** keyword few times already, right? It’s an important difference between arrow function and regular function, it is used in object constructors and classes a lot.
Let’s first find out, what it is in general.

In Javascript, **this** keyword refers to the object it belongs to. That sounds very simple right? Well, it’s not that easy, because it has different values depending on where it is being used (called). Values of **this** keyword depends on where it is being called, not where it is declared! (37)

Let’s see, to what it can actually refer to: (37)

- In a method (function inside an object), `this` refers to the **owner object**.
- Alone, `this` refers to the **global object**.
- In a function, `this` refers to the **global object**.
- In a function, in strict mode, `this` is `undefined`.
- In an event, `this` refers to the **element** that received the event.
- Methods like `call()`, and `apply()` can refer `this` to **any object**.

There are great code examples of **this** at https://www.w3schools.com/js/js_this.asp

6.1 this – arrow function vs regular function

As I said in the previous class, when I was talking about arrow functions vs regular functions, there is an important difference between those two, when it comes to **this**.

Great videos about this
- https://www.youtube.com/watch?v=thXp0_py9X4 (this comparison starts around 13:00)
- https://www.youtube.com/watch?v=h33Srr5J9nY

Regular function **this** keyword represented the object that called the function, which could be the window, the document, a button or whatever. (38) (see the list of what **this** represents above)

On the other hand, arrow functions do not have **this** binding of their own. The value of **this** is resolved to that of the closest non-arrow parent function or the global object otherwise (window). (39)

Okay, let’s see some cases, where there is actually a difference between arrow function and regular function.

In the following example we have an object with regular function. Regular functions in object, refer to the owner object, so when we call **someFunc** it logs out value of **something**.

```
const obj = {
   something: "some",
   someFunc: function () {
      console.log(this.something); //refers to obj
   },
};

obj.someFunc();
```

Now the question is, what happens, if we use arrow function in this example? Well, it points to the global window object. Why? Because there is no parent function from which it could take **this**, so it takes **this** from the global object, but there is no **something** property in window object right? That’s why it returns undefined

```
const obj = {
    something: "some",
    someFunc: () => {
       console.log(this.something); //undefined - refers to window object
    },
};

obj.someFunc();
```

Let's look at another example. In the following code, we want to log out item name after 1 second. JavaScript has built in function for delayed tasks called **setTimeout**, it accepts a function and time (in milliseconds). Given function is executed after specified time, which is in our case 1 second. Regular function is used here, so what happens? It doesn't return anything. Why? Because as we said, regular function's **this** refers to the object where it is being called, but our anonymous function is called in global window object, NOT in Item class, and there is no **name** property in window object.

```
class Item {
    constructor(name) {
       this.name = name;
    }

    logName() {
       setTimeout(function () {
             console.log(`${this.name}`);
         }, 1000);
    }
}

newObj = new Item("Some name");
newObj.logName();
```

So, what happens, when I just use arrow function instead of the regular one? Now we get our desired value of **name** property. Why? Because it doesn't care where it is being called, it looks for its parent non arrow function **this** and uses that **this** value. Its non-arrow parent function refers to our object where the function is actually defined, so everything works as expected

```
class Item {
    constructor(name) {
       this.name = name;
    }

    logName() {
       setTimeout(() => {
          console.log(`${this.name}`); //returnes name value after 1 
second
       }, 1000);
    }
}

newObj = new Item("Some name");
newObj.logName();
```

6.1.1 My recommendation on this keyword and arrow function vs regular function

It can be very confusing to even get used to arrow functions, and differences with **this** just add up even more confusion. Don't bother with it too much at the beginning, just use whatever type of writing functions suits you, but keep in mind, that if you come across some weird bug, where you think you should have some value, but you don't have it, it can be because of **this** difference.

7 Higher order functions

Higher order functions are an important part of programming paradigm called Functional programming. If you don't know what it is, don't worry, there is going to be one whole class focused on OOP and Functional programming. For now, I will just talk about what higher order functions are and then go through some very basic examples, just so you know that it exists and get familiar with it, as it is hugely used in JavaScript in general.

Interesting videos

- https://www.youtube.com/watch?v=H4awPsyugS0&t=467s
- https://www.youtube.com/watch?v=BMUiFMZr7vk
- https://www.youtube.com/watch?v=rRgD1yVwIvE

In JavaScript, functions are values just like strings, numbers, arrays, etc. Why is it good? Well, you can pass them around as arguments to another function or return them from some function. (40)

Higher-order functions are functions that take other functions as arguments or return functions as their results. (41)

Taking another function as an argument is often referred as a **callback** function, because it is called back by the higher-order function. This is a concept that Javascript uses a lot. (41)

Let's first create our own higher order function. So, we have function **highFunc** which accepts function as an argument, and then logs a string, where you can find the result of **callbackFunc** and of **str** which comes from **highFunc**.

This function passed as argument, is an example of **callback** function. it's called a callback function, because it is passed to another function as an argument, and invoked -> called back, somewhere in the code of that function. (42)

```
function callbackFunc() {
   return "Hello from callback";
}

function highFunc(callback) {
   const str = ", which is called from higher order function";
   console.log(`${callback()} ${str}`);
}

highFunc(callbackFunc);
```

Code above just illustrates simple example of higher order function and how it works, but otherwise it's really useless. Let's have a look at some useful example. Higher order functions are very useful with for example arrays, as there are some built in higher order functions, which we can use on them. One of them is **map()**. Higher order function **map()** runs on array, takes a callback function as an argument, where you say, what should happen to each item of that array, and then it returns new array of results returned from callback function (that's why there is **numbersSquared** constant, to which the new array is assigned). (43)

```
const numbersArr = [1, 3, 6, 8];

const numbersSquared = numbersArr.map((number) => number * number); //
returns [1, 9, 36, 64]
```

Another useful function is **filter()**. The **filter()** function creates a new array with all elements that pass the test implemented by the provided function. (44) In following example, new array is created which contains values 6 and 8.

```
const filteredArr = numbersArr.filter((number) => number > 5); //returns
[6, 8]
```

As you can see, higher order functions are useful, because they allow you to create small functions that take care of only one piece of logic. (41) You can then call them everywhere in your code and focus more on what needs to be done with given data, without the need of specifying how to do that as that has been already specified in those higher order functions. This keeps you code much more reusable, cleaner and readable.

8 Citovaná literatura

1. **© 2005-2020 Mozilla and individual contributors.** Loops and iteration. *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration.
2. **Copyright 1999-2020 by Refsnes Data.** JavaScript For Loop. *w3schools.com.* [Online] https://www.w3schools.com/js/js_loop_for.asp.
3. —. JavaScript While Loop. *w3schools.com.* [Online] https://www.w3schools.com/js/js_loop_while.asp.
4. **© 2005-2020 Mozilla and individual contributors.** continue. *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/continue.
5. **Popov, Stoimen.** JavaScript Performance: for vs. while . *DZone.* [Online] 12. January 2012. https://dzone.com/articles/javascript-performance-vs.
6. **Watson, Matt.** Why Premature Optimization Is the Root of All Evil. *Stackify.* [Online] 28. November 2017. https://stackify.com/premature-optimization-evil/.
7. **Shappir, Dan.** The Benefits of JavaScript “Premature Optimization”. [Online] 4. May 2016. https://medium.com/@DanShappir/the-benefits-of-javascript-premature-optimization-ee3b842c42c8.
8. **Gerschau, Felix.** Premature Optimization in JavaScript. *Felix Gerschau.* [Online] https://felixgerschau.com/premature-optimization-javascript/.
9. **Copyright 1999-2020 by Refsnes Data.** JavaScript do/while Statement. *w3schools.com.* [Online] https://www.w3schools.com/jsref/jsref_dowhile.asp.
10. **© 2005-2020 Mozilla and individual contributors.** for...of. *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of.
11. **© 2020 Alligator.io LLC.** for...of vs for...in Loops in JavaScript. *alligator.* [Online] https://alligator.io/js/for-of-for-in-loops/.
12. **Delam, Naz.** All you need to learn to understand JavaScript □ _ Execution Context and Stack. *Medium.* [Online] 11. March 2019. https://medium.com/@nazanindelam/all-you-need-to-learn-to-understand-javascript-execution-context-and-stack-3babbdd88868.
13. **Mishra, Rupesh.** Execution context, Scope chain and JavaScript internals. *Medium.* [Online] 18. May 2017. https://medium.com/@happymishra66/execution-context-in-javascript-319dd72e8e2c.
14. **© 2005-2020 Mozilla and individual contributors.** Window. *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/API/Window.
15. **Klaus.** JavaScript: scope chain, variable objects and activation objects. *Medium.* [Online] 10. September 2017. https://medium.com/@klausng/javascript-scope-chain-variable-objects-and-activation-objects-4eb017256d0b.
16. **Ogura, Misa.** JS Demystified 04 — Execution Context. *codeburst.io.* [Online] 24. August 2017. https://codeburst.io/js-demystified-04-execution-context-97dea52c8ac6.
17. **Arora, Sukhjinder.** Understanding Execution Context and Execution Stack in Javascript. *Bits and Pieces.* [Online] 28. August 2018. https://blog.bitsrc.io/understanding-execution-context-and-execution-stack-in-javascript-1c9ea8642dd0.
18. **Copyright 1999-2020 by Refsnes Data.** JavaScript Scope. *w3schools.com.* [Online] https://www.w3schools.com/js/js_scope.asp.
19. **© 2005-2020 Mozilla and individual contributors.** Global scope. *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Glossary/Global_scope.
20. **Liew, Zell.** JavaScript Scope and Closures. *css-tricks.* [Online] 16. January 2019. https://css-tricks.com/javascript-scope-closures/.

21. **Alam, Bilal.** Javascript Scope Chain and Execution Context simplified. *Medium.* [Online] 20. July 2018. https://medium.com/koderlabs/javascript-scope-chain-and-execution-context-simplified-ffb54fc6ad02.
22. **Singh, Ashutosh.** A Brief Introduction to Closures and Lexical Scoping in JavaScript. *DZone.* [Online] 16. August 2019. https://dzone.com/articles/a-brief-introduction-to-closures-and-lexical-scopi.
23. **Okiche, Emmanuel.** I would try to explain lexical scope in plain English. Wish me luck. *DEV.* [Online] 6. September 2019. https://dev.to/fleepgeek/i-would-try-to-explain-lexical-scope-in-plain-english-wish-me-luck-4j06.
24. **Ram, Prashant.** A simple guide to help you understand closures in JavaScript. *Medium.* [Online] 16. January 2018. https://medium.com/@prashantramnyc/javascript-closures-simplified-d0d23fa06ba4.
25. **Nigam, Anchal.** JavaScript Closure Tutorial – With JS Closure Example Code. *freeCodeCamp.* [Online] 27. May 2020. https://www.freecodecamp.org/news/javascript-closure-tutorial-with-js-closure-example-code/.
26. **© 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.
27. **Copyright 1999-2020 by Refsnes Data.** JavaScript Object Constructors . *w3schools.com.* [Online] https://www.w3schools.com/js/js_object_constructors.asp.
28. **© 2005-2020 Mozilla and individual contributors.** new operator. *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new.
29. —. Inheritance and the prototype chain . *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain.
30. —. Object prototypes. *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes.
31. —. Object.create(). *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create.
32. —. Function.prototype.call(). *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call.
33. —. Classes. *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes.
34. **Copyright 1999-2020 by Refsnes Data.** JavaScript Class constructor Method ❮ JavaScript Classes ReferenceNext ❯ . *w3schools.com.* [Online] https://www.w3schools.com/jsref/jsref_constructor_class.asp.
35. —. JavaScript Class extends Keyword. *w3schools.com.* [Online] https://www.w3schools.com/jsref/jsref_class_extends.asp.
36. —. JavaScript Class super Keyword. *w3schools.com.* [Online] https://www.w3schools.com/jsref/jsref_class_super.asp.
37. —. The JavaScript this Keyword. *w3schools.com.* [Online] https://www.w3schools.com/js/js_this.asp.
38. —. Arrow Function. *w3schools.com.* [Online] https://www.w3schools.com/js/js_arrow_function.asp.
39. **Chinda, Glad.** Anomalies in JavaScript arrow functions. *LogRocket.* [Online] https://blog.logrocket.com/anomalies-in-javascript-arrow-functions/.
40. **Bzadough, Yazeeed.** A quick intro to Higher-Order Functions in JavaScript. *freecodecamp.org.* [Online] 11. March 2019. https://www.freecodecamp.org/news/a-quick-intro-to-higher-order-functions-in-javascript-1a014f89c6b/.

41. **Cosset, Damien.** Higher-order functions in Javascript. *DEV.* [Online] 16. February 2019. https://dev.to/damcosset/higher-order-functions-in-javascript-4j8b.
42. **© 2005-2020 Mozilla and individual contributors.** Callback function. *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Glossary/Callback_function.
43. —. Array.prototype.map(). *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map.
44. —. Array.prototype.filter(). *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter.
45. —. eval(). *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval.