2. JavaScript Basics

Obsah

Seznam obrázků

1 Variables

1.1 Variable definition

JavaScript variables are containers for storing data values. (1)

1.2 What can you store into a variable?

There are two main categories of values, which you can store into a variable: primitives and objects. Let's look at them closer.

1.2.1 Primitives

A primitive value or primitive data type is data, that is not an object and has no methods. (2)

This is an official definition, which doesn't really say that much, so let's just have a look at examples.

- Boolean – true/false (3)
- Null – no value (3)
- Undefined – declared variable with no assigned value (3)
- Number – number (3)
- String – text. Basically, everything what is wrapped in quotes (‘’ or “” or ``). So even a number wrapped in one of those quotes, is a string!
- Symbol – unique value, which is not equal to any other value (3)

1.2.2 Objects

We will take a closer look at them later on, but for now let's say, that object is a collection of properties. (4) You may ask, what is a property? Property is a value (any of those data types mentioned in primitives) or a method (It's a property containing a function definition. (5) Function is set of statements that performs a task or calculates a value. (6)).

So, to make long story short, object is a variable, that can hold more than one value at a time.

Examples of object data types in JavaScript

- Object – unordered key/value pairs (4)
- Function – set of statements, that performs a task or calculates a value. Can be called anywhere in code (4)
- Array – ordered and indexed collection of values (4)
- Map (4)
- Date – represents date and time (4)
- JSON – lightweight data-interchange format, derived from JavaScript, but used by many programming languages. (4)

1.3 JavaScript is weakly and dynamically typed language

It's very important to know, that JavaScript is weakly and dynamically typed language. Why? Because it strongly affects the way we declare and work with variables.

1.3.1 Dynamically typed

Dynamic typing means, that we don't have to specify a variable type when we're declaring a variable. It's because variable types are checked on the fly, as the code is executed. (7) This means, that we can assign different value types to the same variable during its lifetime. For example, we can have a variable, which is a number at first, then its value is changed to a string, then object, etc.

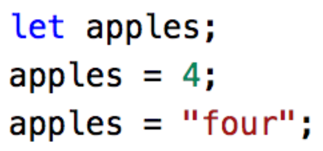
This can seem very good at first, but it has its drawbacks. The biggest disadvantage of this is, that you can never be sure, what data type is in variable, which can cause some runtime errors.

As oppose to that, we have **statically typed languages** like C# or Typescript, where you need to specify a variable type, because variable types are checked at the compilation time (before the code is executed). (7) This means, that you have to write a bit more while coding, but you get a better control over the code and the code becomes much more self-explanatory. That's actually the reason, why Typescript was developed and is getting more and more popular nowadays. Typescript is a superset of JavaScript and it gives us all advantages of static typing, that's why we will come back to Typescript in the last class, as it is important, but more advanced topic and definitely not mandatory to use.

For now, it's just good for you to know, that it exists. Enough of theory for a while, let's show the difference between static and dynamic typing on a code example.

JavaScript code (dynamic typing)

Code below runs absolutely fine. At first, variable **apples** is a number, and then it is a string.

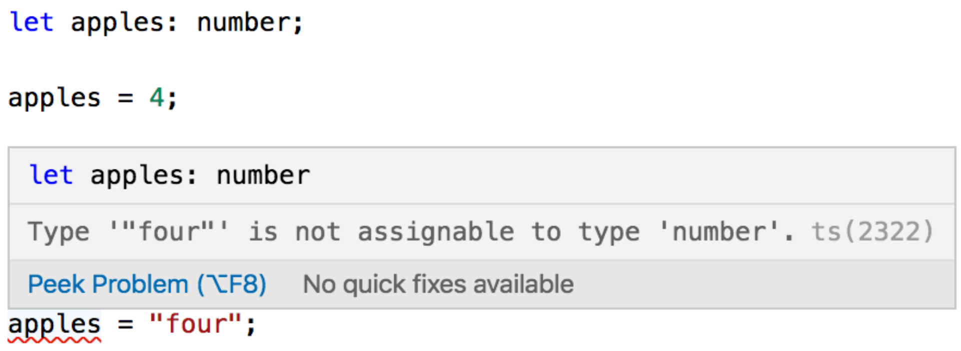


```
let apples;
apples = 4;
apples = "four";
```

Figure 1 - dynamic typing JS example. Source: my own

TypeScript code (static typing)

Following code will throw an error. Because we specified, that variable **apples** should be a number, but we are trying to assign a string value to it.



```
let apples: number;

apples = 4;

let apples: number
Type '"four"' is not assignable to type 'number'. ts(2322)
Peek Problem (⌥F8)    No quick fixes available
apples = "four";
```

Figure 2 - static typing TS example. Source: my own

Weakly typed

Weakly typed means, that variables are not strictly tight to one data type. Although variables have their data type, the interpreter or compiler attempts to make the best of what it is given. This means that in some situations, for example an integer might be treated as if it were a string to suit the context of the situation.

Let's show two code examples of this behaviour.

In the following code, we have variable **eggs**, which contains a number and then we have variable **text**, which contains a string. In variable **combination**, we combine variables **eggs** and **text**. Both variables are interpreted as string.

```
let eggs = 3;
let text = "Number of eggs: ";
let combination = text + eggs; //Result is - Number of eggs: 3
```

That was pretty obvious, but next example is not that obvious and can actually cause some confusion. Variable **num1** is a number, but variable **num2** is a string, which contains number 5. What is the result? Well, it's 25, why? Because variable **num1** is interpreted as a string in this context, so those two variables are not summed up, but **concatenated**.

```
let num1 = 2;
let num2 = "5";
let sum = num1 + num2;
```

1.4 Variable declaration via keywords var, let and const

Important rule

!!!Always use **let** and **const**. Try to always avoid **var**, but keep in mind, that **let** and **const** are fully supported only in all modern browsers, so if you really need to support older browsers, then you have no other choice then to use **var**.!!!

Why can we declare variable via keyword **var**, if it's something we should avoid? Well, for a long time, **var** was the only way how to declare a variable in JavaScript, that's why you can still see it a lot in many documentations or stackoverflow answers. Keywords **let** and **const** came with the newer version of JavaScript which was released in 2015 and is called ES6 or ECMAScript 2015. (8)

1.4.1 Let vs const

Okay, so now we know, that **let** and **const** are an improved way to declare a variable. Why? Well, you need to know a bit more theory about JavaScript, in order to fully understand that. Don't worry, I will explain that a bit later on, for now, let's just learn the difference between **let** and **const**.

The biggest difference is, that the value of **let** variable can be reassigned, but **const** variable can't. That's why it's always preferred to use **const**, because when you see **const** variable, you know, that it always contains same value, so you don't have to go through dozens of lines of code to see all values it contains during its lifetime. (9)

Let's see the difference on code example

Following code will run perfectly fine.

```
let value = "some value";
value = "changed value";
```

As oppose to that, following code will throw an error, because we're attempting to reassign variable **val**, which is a constant.

```
const val = "value";
val = "some other value";
```

You can imagine variable declared via **const** as a statue, which can't be completely changed to something else once created.

2 Conditional statements

In real life, you have to make some decisions, like "should I go to school, or rather go out with friends?" or "Should I watch Netflix, or work on developing the next Facebook app?". Each and every one of those decisions, carry consequences which affect our lifes.

Just like in real life, even in our apps are decisions that need to be made. There are situations, where you need your code to react differently to different kinds of input values or input types of values (e.g. do something if the value is a string). For example, when you have an app, where the user is required to sign in, the app needs to decide, whether the user is signed in or not, so it can serve him appropriate content.

There are few ways, in which we can achieve such behaviour: **If/else**, **switch**, or **ternary operator**, in combination with logical operators of course. (10) (11)

2.1 If/else statement

This one is useful for conditions, where are not that many cases that can occur.

2.1.1 Syntax

```
if (condition1) {
  Block of code to be executed if condition1 is true
} else if(condition){
  Block of code to be executed if the condition1 is false and condition2 is
  true
} else {
  Block of code to be executed if the condition1 is false and condition2 is
  false
} (10)
```

2.1.2 Code example

```
const isWeekend = true;
let action;

if (isWeekend === true) {
  action = "Go to pub";
} else {
  action = "Go to school";
}
```

2.2 Side note - Equal signs in condition

Double equal sign (==)

- Compared values have to have the same value, but they don’t have to have same the data type. (12)

Triple equal sign (===)

- Compared values have to have same value **and data type**. This is called **strict equality**. (12) This is the recommended way to code conditions in JavaScript.

Let’s show those two in code example. In the following code, the first condition is true, because the value of variable **val** is 5. On the other hand, the second condition is false, because the value of variable **val** is 5, but **val** is type of string, and not number.

```
const equalSignVal = "5";
if (equalSignVal == 5) {
  console.log("Variable contains value 5");
}

if (equalSignVal === 5) {
  console.log("Variable contains value 5 and is type number");
}
```

2.3 Ternary operator

This one works exactly the same as if…else, it’s just shorter version of it, so you can have a bit cleaner code.

Ternary operator takes three operands:

A condition followed by a question mark (?), then an expression to execute if the condition is truthy followed by a colon (:), and finally the expression to execute if the condition is falsy. (11)

2.3.1 Syntax

```
Condition ? exprIfTrue : exprIfFalse
```

2.3.2 Code example

In the following code, the condition whether it is weekend or not is evaluated. If it is weekend, then the variable **ternaryAction** is equal to “Go to pub” if not, then it is equal to “Go to school”.

```
let ternaryAction = isWeekend === true ? "Go to pub" : "Go to school";
```

There is a way to make ternary operator code even shorter. All variables are either truthy or falsy, so when you just write the name of variable, the interpreter is able to evaluate that condition as either true or false.

```
ternaryAction = isWeekend ? "Go to pub" : "Go to school";
```

2.4 Side note – truthy and falsy values

In JavaScript and many other programming languages, not only boolean value can be automatically evaluated as either true or false. We refer to those values as **truthy** or **falsy**.

Following values are evaluated as **falsy** in JavaScript (13)

- False
- 0
- -0
- 0n
- “”
- Null
- Undefined
- NaN

All other values are evaluated as **truthy**. That means, that if all those other values are placed in the condition without anything else, then the condition is evaluated as **true** and the block of code in it, is executed. (13) (14)

In the following code, the condition is true, because value in condition is **truthy** (it’s not equal to any of those falsy values listed above)

```
const truthyVar = 8;
if (truthyVar) {
  //variable truthyVar is true
}
```

2.5 Switch

Switch is another option for conditional logic and it is a particularly useful solution for long, nested if/else statements. (15)

The switch statement executes a block of code depending on different cases. (15)

The switch statement evaluates an expression, which is defined in round brackets. The value of the expression is then compared with the values of each case in the structure. If there is a match, the associated block of code is executed. (15)

The switch statement is often used together with a **break** or a **default** keyword (or both). These are both optional:

The **break** keyword breaks out of the switch block. This will stop the execution of that switch statement. If break is omitted, the next code block in the switch statement is executed. (15)

The **default** keyword specifies some code to run if there is no case match. There can be only one **default** keyword in a switch statement. Although this is optional, it is recommended that you use it, as it takes care of unexpected cases. (15)

2.5.1 Syntax

```
switch(výraz) {
  case a:
    //code block
    break;
  case b:
    //code block
    break;
  default:
    //code block to be executed, if there is no match in any case
}
```

2.5.2 Code example

```
const dayOfWeek = "monday";
let dayAction;

switch (dayOfWeek) {
  case "Monday":
     dayAction = "Work";
     break;
  case "Tuesday":
     dayAction = "School";
     break;
  case "Wednesday":
     dayAction = "Road trip";
     break;
  case "Thursday":
     dayAction = "School";
     break;
  case "Friday":
     dayAction = "Party";
     break;
  case "Saturday":
     dayAction = "Hangover";
     break;
  case "Sunday":
     dayAction = "Family time";
     break;
  default:
     dayAction = "No match found";
}
```

3 Functions

Function is named “subprogram”, that can be called anywhere in code. Like the program itself, a function is composed of a sequence of statemens called the function body. Values can be passed to a function (function accepts parameters), and the function will return a value. (6)

Why do we need this? Well, there are parts of logic, which can be reused across our app. Imagine that you are creating a web app, where you work with user data. In this case, you would need to access user information like first name, last name and age at many places, I know, these are only variables, but what if you would need those three variables concatenated to string in the exact same way at many places in your app? Of course, that you can write the logic for it at each place you need it, but what if the way you need to concatenate those variables will change in future, or what if you need to add some more information? It’s much better to have only one place, where you need to change your logic, than many places all around the app.

There is also another reason why we need functions. Some app logic needs to run only after some user action happens, for example when user clicks on a button.

In JavaScript, functions are objects, which I will talk about later. In function, you can declare variables, accept parameters and you can also declare functions within that function, just like in an object. Where the function differs from an object is, that function can be called. (6)

There are two ways in which you can create a function in JavaScript: **function declaration** and **function expression**.

3.1 Function declaration

3.1.1 Syntax

```
function name(parameters) {
   // function logic
}
```

3.1.2 Code example

In the following example, we have function declaration **sum**, which takes two parameters, sums them and returns the result. Then this function is called with numbers **2** and **3** and returns **5**.

```
function sumDeclaration(a, b) {
   return a + b;
}
sumDeclaration(2, 3); //returns 5
```

3.2 Function expression

This one can be used because of two things. First, you can assign function as a value to variable and second, you can create anonymous functions in JavaScript (functions without name). Function expression actually doesn't have only one syntax, so let's go to code examples right away to show why and how it looks like.

Let's see how the exact same function is created via function expression.

```
const sumExpression = function (a, b) {
   return a + b;
};
```

This is not the end. You can make this function even shorter, because you actually don't even have to write word **function**. You can actually use something called **arrow function**. Arrow function is just another way to declare anonymous function.

```
const arrowExpression = (a,b) => {
   return a + b;
}
```

There is still way to make our function a bit shorter. We can leave out the word **return** as it is internally used by the interpreter when we leave out curly brackets as well.

```
const arrowExpression = (a, b) => a + b;
```

If you have a function with only one parameter, you can also leave out round brackets.

```
const arrowAddFive = a => a + 5;
```

3.3 Side note –regular functions vs arrow functions

There are not many differences between regular functions and arrow functions, but those existing differences are very important. The first one is really obvious, it's the syntax. Regular functions always have the same syntax, whereas syntax of arrow functions depends on the number of function parameters and absence of curly brackets.

The second difference is very important. It's the difference with keyword **this**, we will discuss this keyword a bit later on. For now, just remember that some keyword **this** exists, and that is very good to listen, when I will talk about it.

I know it can be intimidating at first to see those arrow functions, but they can make your code easier to read once you get used to them.

Interesting videos about arrow function

- https://www.youtube.com/watch?v=mrYMzpbFz18&t=1160s
- https://www.youtube.com/watch?v=6sQDTgOqh-I

3.4 Side note – function expression vs function declaration

There is one main difference between function expression and function declaration. Function declaration is hoisted, but function expression is not. So, what is hoisting? Let's have a look at it.

3.4.1 Hoisting

Hoisting is JavaScript default behaviour of moving declarations to the top of our code. (16) It means, that it doesn't matter where in code we declare function via function declaration, it is always moved to the very top of our code.

Advantage of this is, that we can call function declared via function declaration, before it is initialized. (17) Let's show it in code example. In the following code, function is called before it is initialized, but it still works absolutely fine.

```
hoistedFunction()
function hoistedFunction() {
   return "hello from hoisted function";
}
```

If we do the same thing using function expression, the code will not work. When using function expression, we can access function only after its initialization (after the line, where that function is written).

```
notHoistedFunction(); //error - Cannot access function before its
initialization
const notHoistedFunction = () => {
return "hello from not hoisted function";
};
notHoistedFunction(); //runs fine
```

3.5 Default parameter for a function

Default function parameters allow named parameters to be initialized with default values if no value or **undefined** is passed to a function. (18)

In the following code function **defaultParam** is called with only one parameter, which leaves the second parameter with no value, so the default value is used, and function returns concatenated string of a value parameter **a** and a default value of parameter **b.**

```
function defaultParam(a, b = ", hello from default parameter") {
return a + b;
}
defaultParam("Students"); //returns "Students, hello from default
parameter"
```

4 "Normal" Objects and arrays

Why did I say "normal" objects? Well, as we learned in section **1.2.2 Objects**, many things in JavaScript are considered as an object data type. But here I want to go deeper into an object data structure, where key/value pairs are stored.

4.1 Arrays

An array is a special variable, which can hold more than one value at a time. (19)

Arrays store their values in an ordered manner.

An array can hold many values under a single name, and you can access those values by referring to their index number.

There are two ways to declare an array, I prefer the first one.

```
const cities = ["Prague", "Budapest", "Moscow", "London"];

const carBrands = new Array("BWM", "Audi", "Škoda");
```

4.1.1 How to access values of an array

If you want to access some array value, you need to know its index.

```
cities[0];
```

4.1.2 Array value change

In the following code example, I'm changing array value with an index of 0. If we come back what I said about a **const** variable, I said that we can't reassign its value, but in this code example, I'm actually changing already declared array value with an index of 0, how so? Well, this can seem very tricky at first. When it comes to arrays and objects, I can actually change their values, add new values or remove already existing ones. What I can't do though, is that I can't completely reassign that value. What I mean by that is, that I can't reassign that array constant to a completely new array or object, or something else.

I also said, that a variable defined as **const** is like a statue. That is true, BUT when it comes to objects and arrays, **const** variable is like a statue, but you can slightly change its shape. Imagine that a **const** variable is a human statue. You can change shape of its face or change its body from chubby type to an athlete. What you can't do though, is to change its shape completely and transform it to for example an elephant.

I hope I made this clear, so let's have a look at an example of an array value change.

```
cities[0] = "Paris";
```

4.1.3 Get array length

In many cases, you need to know how long your array is. There is very easy to way to find out the array length. In the following code, the length of cities array is returned.

```
cities.length; //returns 4
```

4.1.4 Array methods

There are few methods in JavaScript, which make working arrays easier, so let's have a look at some of them.

Behaviour	**Code example**
Adds new element to an **end of array**	`cities.push("Berlin");`
Removes element from an **end of array**	`cities.pop();`
Adds new element to the **beginning of array**	`cities.unshift("Tokyo");`
Removes element from the **beginning of array**	`cities.shift();`
Finds array index of given value.	`cities.indexOf("Budapest")`

List above is not complete, there is more methods. If you are interested, have a look at the attached url. - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

4.2 Objects

Objects can hold more than one value just like arrays, the difference is, that in objects, you define key/value pairs. That means, that you don't only add value to an object, you also need to specify its key with which you can access that value later on.

As oppose to arrays, the order in which values are stored, doesn't matter.

4.2.1 Code example

Usually, object represents something from a real word. So, let’s say that we are developing an app for a local car dealership. In our app, we need to list all available cars. Okay, so that’s where objects come into play. We create an object called **car**. Okay, so what do we usually want to know about the car? We want to know its type, colour, how many kilometres it has, if it has been crashed, etc. So, let’s create this object in JavaScript.

```
const car = {
  type: "BMW", //string
  color: "black", //string
  kilometers: 197087, //number
  wasCrashed: true, //boolean
};
```

4.2.2 How to access object values

```
car.type
```

4.2.3 Change property value

```
car.type = "Audi";
```

Or

```
car["type"] = "Mercedes";
```

4.2.4 Add property to an object

```
car.fuelType = "gasoline";
```

4.2.5 Remove property from an object

```
delete car.type;
```

4.2.6 Methods in objects

As I said, object mostly represents something from a real world. Things in real world can usually do some things, for example a car can drive or stop. We can use methods in JavaScript to represent it. Method is actually the same thing as function, just keep in mind, that functions defined inside objects are called methods. So, how can we define those? Let’s add some methods into our object from the previous example

```
const car = {
  type: "BMW", //string
  color: "black", //string
  kilometers: 197087, //number
  wasCrashed: true, //boolean
  drive: function () { //method
    //some logic, so car can drive
  },
  stop: function () { //method
    //some logic so car can stop
  },
};
```

4.3 Objects vs arrays

Object	**Array**
Unordered	Ordered
Key/value pairs (values are accessed via string keys)	Indexed values (values are accessed via numbers)

4.3.1 When to use objects

Objects are used to represent a “thing” in your code. That could be a person, a car, a building, a book, a character in a game — basically anything that is made up or can be defined by a set of characteristics. In objects, these characteristics are called **properties** that consist of a key and a value. (20)

4.3.2 When to use arrays

We use arrays whenever we want to create and store a list of multiple items in a single variable. Arrays are especially useful when creating **ordered collections** where items in the collection can be accessed by their numerical position in the list. (20)

5 Destructuring

Destructuring assignment just makes the work with arrays and objects easier, as it allows you to assign the properties of an array or object to variables using syntax that looks similar to array or object literals. (21)

Let’s say, that you have an array of games. This is how you would normally access values of this array.

```
const games = ["God of War", "Mafia", "Company of Heroes", "Fortnite"];

games[0]; //God of War
games[1]; //Mafia
games[2]; //Company of Heroes
```

But what if you don’t want to only access those values, but also assign them to some variable? You would have to create variable for each of those values. Turns out, that there is nicer way to do this. You can actually create all those variables at once and use destructuring to assign theirs values as shown in the example below.

```
const games = ["God of War", "Mafia", "Company of Heroes", "Fortnite"];

const [game1, game2, game3] = games;

game1; //God of War
game2; //Mafia
game3; //Company of Heroes
```

You can do this with objects as well, just keep in mind, that when you destructure an object, variable name needs to match with property name which you want to access via destructuring.

```
const game = {
   name: "Company of Heroes",
   genre: "Strategy",
   price: "5 euro",
};

const { name, genre, price } = game;
```

So, when is this actually useful? Mainly when you use object as a function parameter. Let’s consider the following example. We have a function, which accepts math problem object as a parameter, solves that math problem and returns solved value. This is not really nice, because in order to access object properties, you have to write **mathProblem** dot something… There is a switch, as you can have few different operands in your **mathProblem.operator** property. I’ve only written two and it’s been already very annoying to write **mathProblem.** something.

```
const mathProblem1 = {
   operand1: 6,
   operand2: 2,
   operator: "/",
};

function solveMathProblem(mathProblem) {
   let result;
   switch (mathProblem.operator) {
      case "/":
         result = mathProblem.operand1 / mathProblem.operand2;
         break;
      case "*":
         result = mathProblem.operant1 * mathProblem.operand2;
         break;
      }
   return result;
}

solveMathProblem(mathProblem1);
```

Turns out, there is better way to do this thanks to destructuring. So, let's use destructuring as we already know it.

```
function solveMathProblem(mathProblem) {
   const { operand1, operand2, operator } = mathProblem;
   let result;
      switch (operator) {
         case "/":
            result = operand1 / operand2;
            break;
         case "*":
            result = operant1 * operand2;
            break;
       }
    return result;
}

solveMathProblem(mathProblem1);
```

This is not the end, because there is even better way to do this. It's better in two ways. First, you don't need to write **mathProblem.** something anymore (which you didn't have to even in previous example, but in this one, you don't even need to create destructured variables), and second, you know, what parameters your function really needs and uses.

```
function solveMathProblem({ operand1, operand2, operator }) {
   let result;
   switch (operator) {
      case "/":
         result = operand1 / operand2;
         break;
      case "*":
         result = operant1 * operand2;
         break;
      }
    return result;
}
solveMathProblem(mathProblem1);
```

Good videos about destructuring

- https://www.youtube.com/watch?v=PB_d3uBkQPs
- https://www.youtube.com/watch?v=NIq3qLaHCIs

6 Spread operator

Spread operator (`...`) allows an iterable such as an array expression or string to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected, or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected. (22)

If you didn't understand the definition above, don't worry, even I had to read it couple of times to understand it, even though I already know, what spread operator does. So, what does it mean in plain English?

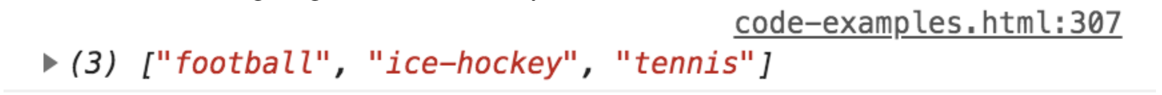
Let's first focus on *"Spread operator (...) allows an iterable such as an array expression or string to be expanded"*. In this context, *"to be expanded"* means to go through each item of array, or each letter of string. Let's see some code example.

In following example, the array of sports is first logged as it is, and then spread operator is used.

```
const sports = ["football", "ice-hockey", "tennis"];
console.log(sports);
console.log(...sports);
```

So, what are results of those logs above?

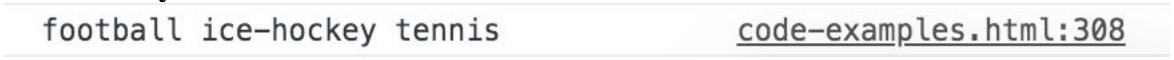
The first console.log, logs out whole array data structure.



```
                                                  code-examples.html:307
▶ (3) ["football", "ice-hockey", "tennis"]
```

Figure 3 - console.log without spread operator. Source: my own image

The second one on the other hand, logs out just values of that array. So, what is happening there? **What spread operator does, is that it goes through each value of an array or string or object. It basically unwraps those data structures and returns their plain values on by one.**

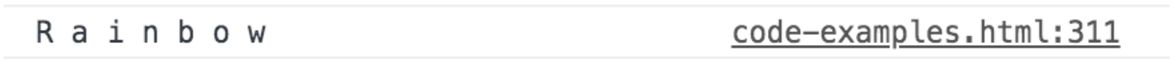


```
football ice-hockey tennis                  code-examples.html:308
```

Figure 4 - console.log with spread operator. Source: my own image

I mentioned strings as well, so what happens when we use spread operator on a string value? It logs out each letter separately.

```
const someString = "Rainbow";
console.log(...someString);
```



```
R a i n b o w                               code-examples.html:311
```

Figure 5 - string with spread operator. Source: my own image

Okay, so when is it actually useful? Let's list some use cases:

- You need to concatenate two arrays or objects
- You need to create a copy of some array or object

6.1 Concatenation of arrays

So, what do we do, when we need to concatenate two arrays into one big array?

Let's first try to do something, what probably comes onto mind to many beginners. This won't work, as it creates nested arrays inside our big array, but we don't want that right? We want to concatenate values of those arrays.

```
const books = ["book1", "book2", "book3"];
const otherBooks = ["book4", "book5", "book6"];
const allBooks = [books, otherBooks];
```

In following example, we create concatenate array **books** and **otherBooks** into one big array.

```
const books = ["book1", "book2", "book3"];
const otherBooks = ["book4", "book5", "book6"];
const allBooks = [...books, ...otherBooks];
```

We can also add some values between those two arrays that we want to concatenate.

```
const allBooks = [...books, "value in the middle", ...otherBooks];
```

6.2 Concatenation of objects

The same thing works for objects. So, let's concatenate two objects into a big one. (23)

```
const mainColors = { brightRed: "#e55039", waterfallBlue: "#38ada9" };
const accentColors = { lightGrey: "#778ca3", swanWhite: "#f7f1e3" };
const fullPalette = { ...mainColors, ...accentColors };
//{brightRed: "#e55039", waterfallBlue: "#38ada9", lightGrey: "#778ca3",
swanWhite: "#f7f1e3"}
```

6.3 Array copy

Sometimes, you need to create a copy of some array. Let's try a naive approach first.

This doesn't work as expected, because arrays and objects are reference types, that means, that if you do this, you just create a reference, which points (references) the actual array. (24)

```
const fruits = ["apple", "strawberry", "orange", "lemon"];
const fruitsCopy = fruits;
```

So, how to really copy some array? Well, that's where spread operator can help us a lot. The only thing we need to do, is this. It takes **fruits** array, unwraps its values, and assigns them to a new array.

```
const realFruitsCopy = [...fruits];
```

6.4 Object copy

The same thing stays for objects.

```
const artist = {
   nickName: "Eminem",
   realName: "Marshall Bruce Matthers III.",
   genre: "Hip-hop",
};

const artistCopy = { ...artist };
```

6.5 Side note – immutability

You probably understand, why it's useful to be able to concatenate arrays and objects so easily. But why the hell would you want to create copies of some objects or arrays?

The answer is immutability. What is it? Well, it's not an easy topic at the beginning, but immutability is a crucial part of programming paradigm (way of thinking) called Functional programming. Functional programming is basically a set of guidelines about how to write readable, maintainable and reusable code. (25)

The idea of immutability is, that it sees objects and arrays as something you cannot change (mutate) at all. So, if you need to manipulate (mutate) some object, you create a copy of that object and mutate that copy. This can sound stupid and complicated at first, but the reason why you would want to do this is, that it's immediately obvious, what you are doing with some object. And another advantage is, that if you work with one object in many functions and you're getting some unexpected values of that object in one of those functions, you don't have to go through each function where you mutated that object, because each function works only with a copy of that object and never actually touches the original object. So, you only need to examine that one function, which is giving you those unexpected results. (26)

Don't worry, this is just an introduction to immutability. There is actually going to be one whole class about programming paradigms (mainly OOP and functional programming).

Awesome video about spread operator - https://www.youtube.com/watch?v=pYI-UuZVtHI

7 Var vs let/const

Now, we can finally come back to variables definition and differences between **var** and **let/const**. Why do you need to know it? There are three reasons for it

1. It's good to know why **let/const** replaced **var**, and it's not just because I will have more things that can put into the exam.
2. As I said, **var** is still used in many code examples on stackoverflow and documentations.
3. **Var** can seem much easier to use for a beginner, as it doesn't throw errors that much in comparison to **let/const**, but it is much more error prone because of that and as we all know, more errors mean more frustration while trying to debug them.

7.1 Scope

First huge difference is in scope. What is scope then? Well, I will talk about it in detail in the next class, for now it's enough for you to know, that scope determines the accessibility (visibility) of variables. (27).

7.1.1 Var

Let's first have a look at **var**.

Var variables are either globally scoped or function/locally scoped. It is globally scoped when a **var** variable is declared outside a function. This means that any variable that is declared with **var** outside a function block is available for use in the whole window. **Var** is function scoped when it is declared within a function. This means that it is available and can be accessed only within that function. (28)

```
var varScoped = "hello from var";
function varScopeFunction() {
   var varInsideFunction = "hello from function";
   if (true) {
      var varInsideBlock = "Hello from var if block";
   }
   console.log(varScoped); //Global variable, so it's accessible
   console.log(varInsideFunction); //Accessible, because it's called
inside a function where it is declared as well
   console.log(varInsideBlock); //Accessible as well, because it's still
in same function, even though it is in if block
}

varScopeFunction();

console.log(varScoped); //Global variable - accessible
console.log(varInsideFunction); //Called outside a function where it is
declared - not accessible
console.log(varInsideBlock); //Not accessible as well
```

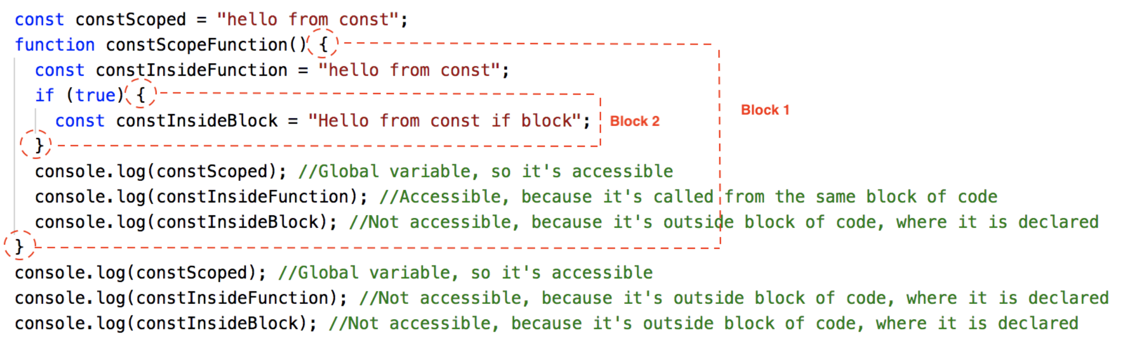
7.1.2 Let and Const

Let and **const** are block scoped. With these two, there is still a global scope of course. What differs, is the local scope. Let and const are block scoped. It means, that they are only accessible in the block of code, where they are declared. Block is chunk of code bounded by {}. (29)

In the following example, I'm using **const** as there is no need for reassign.

```
const constScoped = "hello from const";
function constScopeFunction() {
  const constInsideFunction = "hello from const";
  if (true) {
    const constInsideBlock = "Hello from const if block";
  }
  console.log(constScoped); //Global variable, so it's accessible
  console.log(constInsideFunction); //Accessible, because it's called
from the same block of code
  console.log(constInsideBlock); //Not accessible, because it's outside
block of code, where it is declared
}
console.log(constScoped); //Global variable, so it's accessible
console.log(constInsideFunction); //Not accessible, because it's outside
block of code, where it is declared
console.log(constInsideBlock); //Not accessible, because it's outside
block of code, where it is declared
```

Same blocks of code with highlighted block scopes



```
const constScoped = "hello from const";
function constScopeFunction() {
  const constInsideFunction = "hello from const";
  if (true) {
    const constInsideBlock = "Hello from const if block";
  }
  console.log(constScoped); //Global variable, so it's accessible
  console.log(constInsideFunction); //Accessible, because it's called from the same block of code
  console.log(constInsideBlock); //Not accessible, because it's outside block of code, where it is declared
}
console.log(constScoped); //Global variable, so it's accessible
console.log(constInsideFunction); //Not accessible, because it's outside block of code, where it is declared
console.log(constInsideBlock); //Not accessible, because it's outside block of code, where it is declared
```

7.2 Hoisting

7.2.1 Var

Var variable is hoisted and is initialized with a value of **undefined**. (29) This can be very confusing right? Being able to access variable before it is declared in code and getting a value that we've never assigned to it.

```
console.log(hoistedVar); //returns undefined
var hoistedVar = "cat";
console.log(hoistedVar); //returns "cat"
```

Undefined is returned because code above is interpreted as following: (29)

```
var hoistedVar
console.log(hoistedVar)
hoistedVar = "cat"
```

7.2.2 Let/const

Let and **const** are also hoisted to the top BUT they are not initialized before they are declared in code. (29) What does that mean for us? Well, if we try to access variable before its declaration in code, we get an error.

```
console.log(hoistedConst); //Reference error - cannot access
"hoistedConst" before its initialization
const hoistedConst = "dog";
console.log(hoistedConst); //returns "dog"
```

7.3 Reassign

This is the only category, where all those three keywords differ. I talked about this in section **1.4 Variable declaration via keywords var, let and const**. Little reminder, **let** can be reassigned, but **const** can't. How is it with **var**? It can be reassigned as **let.**

7.4 Declaration/Re-declaration

7.4.1 Var

Var can be redeclared. It means, that we can declare the same variable in same scope without getting an error, so you can do following. (28)

```
var redeclaredVar = "fish";
var redeclaredVar = "mouse";
console.log(redeclaredVar); //returns "mouse"
```

7.4.2 Let/const

Let and const can't be reassigned in same block of code. (28)

```
const redeclaredConst = "monkey";
const redeclaredConst = "pig"; //SyntaxError: Identifier
'redeclaredConst' has already been declared
```

7.4.3 Why is re-declaration bad?

Re-declaration is bad and error prone. Why? Well let's combine re-declaration of **var** with the fact, that **var** is function scoped and consider the following example:

```
var someCar = "BMW";
if (true) {
  var someCar = "Škoda";
}
console.log(someCar);
```

Code above is a bit confusing right? I personally would expect, that it returns BMW. But **var** is function scoped, not block scoped, so the variable defined in if statements is accessible outside of it as well and **var** can be re-declared, so you can absolutely declare variable with the name that has already been declared.

What happens, if you do the exact same thing, but use **const** instead.

```
const someCar = "BMW";
if (true) {
   const someCar = "Škoda";
   console.log(someCar); // "Škoda"
}
console.log(someCar); //"BMW"
```

Code above is a bit more intuitive, as I said, **let/const** are block scoped, so variable declared in if statement is not accessible outside of if statement, as it is wrapped in { }.

In case, that you will have long block scope (function or if statement), where you simply forget that you already declared variable with name use want to use within same code block, you will get an error.

```
const someCar = "BMW";
const someCar = "Škoda";
console.log(someCar); //SyntaxError: Identifier 'someCar' has already
been declared
```

7.4.4 Side note – little challenge (for some exam bonus points??)

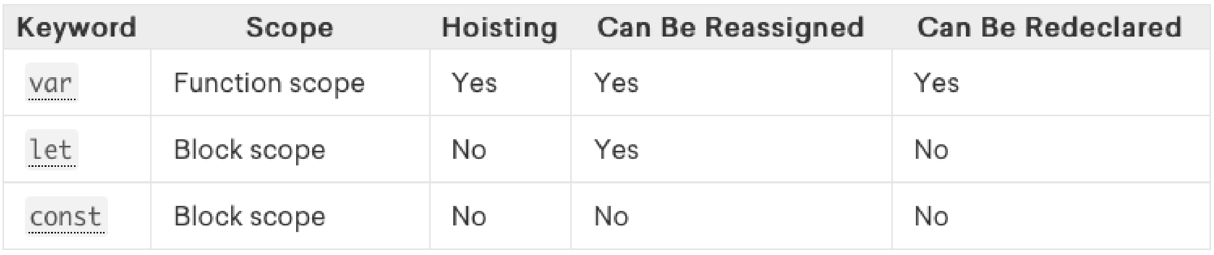
Why following code throws an error in if statement and doesn't use the other **someCar** variable, which is parent of this if statement, so the if statement has access to it?

Hint – it has something to do with hoisting.

```
const someCar = "BMW";
if (true) {
  console.log(someCar); //ReferenceError: Cannot access 'someCar' before initialization
  const someCar = "Škoda";
}
console.log(someCar); //"BMW"
```

Answer - Well, this code throws an error, because let/const are hoisted to the top, so the interpreter knows, that there is a variable called **someCar** in if statement code block, so it doesn't look for that variable further, BUT that error is there, because **someCar** variable is just hoisted but not initialized until declared!

7.5 Differences summary



Keyword	Scope	Hoisting	Can Be Reassigned	Can Be Redeclared
`var`	Function scope	Yes	Yes	Yes
`let`	Block scope	No	Yes	No
`const`	Block scope	No	No	No

Figure 6 - var vs let/const summary. Source: (31)

8 Citovaná literatura

1. **Copyright 1999-2020 by Refsnes Data.** JavaScript Variables. *w3schools.com.* [Online] https://www.w3schools.com/js/js_variables.asp.

2. **© 2005-2020 Mozilla and individual contributors.** Primitive. *MDN web docs.* [Online] 6. April 2020. https://developer.mozilla.org/en-US/docs/Glossary/Primitive.

3. **CodeDraken.** JavaScript Essentials: Types & Data Structures . *Medium.* [Online] 26. September 2018. https://codeburst.io/javascript-essentials-types-data-structures-3ac039f9877b.

4. **© 2005-2020 Mozilla and individual contributors.** JavaScript data types and data structures. *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures.

5. **Copyright 1999-2020 by Refsnes Data.** JavaScript Object Methods ❮ PreviousNext ❯ . *w3schools.com.* [Online] https://www.w3schools.com/js/js_object_methods.asp.

6. **© 2005-2020 Mozilla and individual contributors.** Functions. *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions.

7. **Pavey, Cameron.** Understanding Types; Static vs Dynamic, & Strong vs Weak. *Medium.* [Online] 24. February 2019. https://medium.com/@cpave3/understanding-types-static-vs-dynamic-strong-vs-weak-88a4e1f0ed5f.

8. **Copyright 1999-2020 by Refsnes Data.** ECMAScript 6 - ECMAScript 2015 ❮ PreviousNext ❯ . *w3schools.com.* [Online] https://www.w3schools.com/js/js_es6.asp.

9. **Elliott, Eric.** JavaScript ES6+: var, let, or const? . *Medium.* [Online] 4. November 2015. https://medium.com/javascript-scene/javascript-es6-var-let-or-const-ba58b8dcde75.

10. **Copyright 1999-2020 by Refsnes Data.** JavaScript if else and else if . *w3schools.com.* [Online] https://www.w3schools.com/js/js_if_else.asp.

11. **© 2005-2020 Mozilla and individual contributors.** Conditional (ternary) operator . *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator.

12. **Morelli, Brandon.** JavaScript Showdown: == vs === . *codeburst.io.* [Online] 14. June 2017. https://codeburst.io/javascript-showdown-vs-7be792be15b5.

13. **© 2005-2020 Mozilla and individual contributors.** Falsy. *MDN web docs* . [Online] https://developer.mozilla.org/en-US/docs/Glossary/Falsy.

14. —. Truthy. *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Glossary/Truthy.

15. **Copyright 1999-2020 by Refsnes Data.** JavaScript switch Statement. *w3schools.com.* [Online] https://www.w3schools.com/jsref/jsref_switch.asp.

16. —. w3schools.com. *JavaScript Hoisting ❮PreviousNext ❯*. [Online] https://www.w3schools.com/js/js_hoisting.asp.

17. **© 2005-2020 Mozilla and individual contributors.** Hoisting. *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Glossary/Hoisting.

18. —. Default parameters. *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Default_parameters.

19. **Copyright 1999-2020 by Refsnes Data.** JavaScript Arrays. *w3schools.com.* [Online] https://www.w3schools.com/js/js_arrays.asp.

20. **Heisey, Zac.** Objects vs. Arrays . *Medium.* [Online] 25. April 2019. https://medium.com/@zac_heisey/objects-vs-arrays-42601ff79421.

21. **Nick Fitzgerald, Jason Orendorff.** ES6 In Depth: Destructuring. *HACKS.* [Online] 28. May 2015. https://hacks.mozilla.org/2015/05/es6-in-depth-destructuring/.

22. **© 2005-2020 Mozilla and individual contributors.** Spread syntax (...). *MDN web docs.* [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax.

23. **The Spread Operator. *notion.so.* [Online] https://www.notion.so/The-Spread-Operator-b7608512fbd844ec9f27f31740fb7298.**

24. Cronin, Mike. What are Primitive and Reference Types in JavaScript? *ITNEXT.* [Online] July. 25 2019. https://itnext.io/javascript-interview-prep-primitive-vs-reference-types-62eef165bec8.

25. 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.

26. M, Manjunath. Immutability in JavaScript—When and Why Should You Use It. *DZone.* [Online] 10. March 2020. https://dzone.com/articles/immutability-in-javascriptwhy-and-when-should-you.

27. Copyright 1999-2020 by Refsnes Data. JavaScript Scope. *w3schools.com.* [Online] https://www.w3schools.com/js/js_scope.asp.

28. Chima, Sarah. Var, let and const- what's the difference? *DEV.* [Online] 25. October 2017. https://dev.to/sarah_chima/var-let-and-const--whats-the-difference-69e.

29. Atuonwu, Sarah Chima. Var, Let, and Const – What's the Difference? *freeCodeCamp.* [Online] 2. April 2020. https://www.freecodecamp.org/news/var-let-and-const-whats-the-difference/.

30. 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.

31. Rascia, Tania. Understanding Variables, Scope, and Hoisting in JavaScript. *DigitalOcean.* [Online] 20. February 2018. https://www.digitalocean.com/community/tutorials/understanding-variables-scope-hoisting-in-javascript.