2. Modern Javascript & ES6

First things first

At first, you need to install a minimal set of tools necessary for development. The most important (means that there is not an alternative to) is

Node.js

Node.js is a javascript environment able to run JS code outside of a web browser. It is based on the Chrome V8 JavaScript engine.

What is a V8 engine?

V8 is an open-source high performance JavaScript WebAssembly engine, written in C++. It is used in Chrome web browser or Node.js, among others. It implements ECMAScript (explanation next) and WebAssembly, runs on every popular OS like Windows 7 and higher, macOS 10.12+, and Linux systems that use x64, IA-32, ARM, or MIPS processors. (https://v8.dev/)

In this lesson, Node.js environment will be used for interpretation of simple javascript code snippets.

IDE

You can choose from plenty of various development environments. On the internet, there are plenty of articles with pros and cons for any of them (example). In InQool, we tend to use Visual Studio Code. For the examples, demos and snippets of this course, we will use this IDE as well. It's portable for all platforms and free to download. For later assignments you will also have an option to use online IDE Repl.it with it's integration for github education. If anyone has personal preference to use something else, of course you can, but maybe we won't be able to help you with some problematic stuff.

Useful extensions for VS Code:

- Bracket Pair Colorizer
- GitLens — Git supercharged
- Material Icon Theme
- Path Intellisense
- Prettier - Code formatter
- ESLint
- StyleLint
- Error Lens
- Import Cost
- Vscode-styled-components

Why should I as a React developer care about Javascript?
React is a Javascript framework for development of web applications. So just from that we know there is a relation between them. To further separate what each is used for we may ask another question “What frontend applications do?”.
We need to display some information to users and for that React or other user interface framework are used. It supplies many useful features such as virtual DOM, that speed up development and allow better programmer experience than writing plain JavaScript. It then results in plain HTML + CSS + JS files that a browser can interpret.
Information, in most cases, is represented by data that needs to be transformed and retrieved from source (backend server). Since React is a user interface framework, rest of the application logic is handled via normal JavaScript.

Basic syntactic constructs in JS (es2015+)

In this section, it is assumed you have some basic knowledge of JavaScript. The most common constructs will be described and shown on examples. Also there will be explained the difference between the older version, ECMAScript 5 (everyone who creates at least basic HTML page, meets this standard) and newer version, ES6, which comes with some news and syntactic changes. At this moment, a frontend developer can not exist without knowledge of this standard. There are of course a lot more differences between ES5 and ES6 than described in the lesson, but online sources are available and I hope a frontend developer can find the right information by him/herself :)

Data types (Value vs. Reference)

In JavaScript data types are split into these three types:

Primitives
undefined, Boolean, Number, String, BigInt, Symbol
Structural Types

Object (Object, Array, Map, Set, WeakMap, WeakSet), Function
Structural Root Primitive
null

Just like in many other languages there is a difference between primitives and structural types in the way they are stored. Each non primitive type is stored as a reference pointing to memory on the heap. This means that you can get unexpected side effects such as modifying seemingly unrelated values through it's reference.

Why this is important to keep in mind especially in React is because of the way it checks for updates in props of components. React uses shallow comparison which means that by passing objects (and functions) as props, while containing the same values, their references can differ, which can trigger unnecessary change (more on that later).

Falsy and Truthy values

When using any value type as a boolean JavaScript uses coercion to decide whether the result is considered true or false. Truthiness or falseness of an expression therefore means whether the expression is considered true or false in this context.

false	The keyword false
0	The number zero
-0	The number negative zero
0n	BigInt, when used as a boolean, follows the same rule as a Number. 0n is falsy.
""	Empty string value
null	null - the absence of any value
undefined	undefined - the primitive value
NaN	NaN - not a number

All other values are considered truthy. (f.e. empty objects and arrays [] {})
This means that if not used correctly, you can get false negatives when checking string or number values for truthiness.

```
function checkDefined(val) {
  return !val;
}

checkDefined(null); // false
checkDefined(""); // false because of "" being falsy
checkDefined(0); // false because of 0 being falsy
```

Template literals

Also known as `template strings`. Based on usage of back-tick (`` ` `) compared to classic double ticks (`" "`). Text between brackets is managed like a classic string, but sequence of characters `${}` is replaced by expression inside (ex. variable, logic operation).

```
var name = 'Slim';
var surname = 'Shady';

// classic ES5 version
console.log("Hi, my name is " + name + " " + surname + ".");

// written in ES6
console.log(`Hi, my name is ${name} ${surname}.`);

// both outputs end with:
// > Hi, my name is Slim Shady.
```

Also supports multiline strings.

```
// classic ES5 version
console.log('Hi! My name is - what?\n' +
'My name is - who?\n' +
'My name is\n' +
'Slim Shady!');

// written in ES6
```

```
console.log(`Hi! My name is - what?
My name is - who?
My name is
Slim Shady!`);

// both outputs end with:
// > Hi! My name is - what?
// > My name is - who?
// > My name is
// > Slim Shady!
```

More info can be found on
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

Constants - let, const

Keyword `const` means a constant (mind-blowing, I know) - if a variable has been assigned a value once, it can not be overridden during her lifetime. On the other hand, if you want to override a variable, you should use keyword `let`.

```
const a = 10;

try {
  a = 20; // ends with: Uncaught SyntaxError: Identifier 'a' has already been
declared
} catch(e) {
  console.error(e);
}

console.log(a); // 10
```

```
let a = 10;
a = 20;

console.log(a); // 20
```

It's important to say that `const` / `let` variable is accessible only inside of the `scope` (https://developer.mozilla.org/en-US/docs/Glossary/Scope), in which variable is defined (unlike `var`).

```
if (true) {
  var a = 10;
}

console.log(a); // 10
```

```
if (true) {
  const a = 10;
}

console.log(a); // undefined
```

```
if (true) {
  let a = 10;
}

console.log(a) // undefined
```

Arrow function

Basically this is only a more compact writing of a function. It is interpreted like a classic function, but without its own `this`. It also can not be used with a constructor. Functionality is described with the following examples.

1. Arrow function returning number

```
function getNumber() {
  return 10;
}

// basic variant
```

```
const getNumber = () => {
  return 10;
}

// short variant
const getNumber = () => 10;
```

2. Arrow function returning array

```
function getArray() {
  return [1, 2, 3];
}

const getArray = () => [1, 2, 3];
```

3. Arrow function returning object

```
function getObject() {
  return {
    a: 1
  };
}

const getObject = () => ({ a: 1 });
```

4. Arrow function with parameters

```
function pow(a) {
  return a * a;
}

const pow = a => a * a;
```

5. Arrow function with block of code

```
function getHigherValue(a, b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}

const getHigherValue = (a, b) => a > b ? a : b; // used ternary operator
```

Spread operator

This new syntax allows to group parameters of functions, iterate over items of array and from standard ES 2018 also over attributes of JSON objects. This functionality is easier to understand in the following examples.

1. Function call

```
const sum = (a, b, c) => a + b + c;

const array = [1, 2, 3];

const value = sum(...array);
console.log(value); // 6

// Bonus question: How to create a sum function for N values using spread
operator?
const sumN = (a, ...rest) => rest.length > 0 ? a + sumN(...rest) : a;
```

2. Work with array

```
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];

const array3 = [...array1, ...array2];
```

```
console.log(array3); // [1, 2, 3, 4, 5, 6]
```

3. Work with object

```
const obj1 = { foo: 'bar', x: 1 };
const obj2 = { foo: 'baz', y: 2 };

const clonedObj = { ...obj1 }; // Object { foo: "bar", x: 1 }
const mergedObj = { ...obj1, ...obj2 }; // Object { foo: "baz", x: 1, y: 2 }
```

Destructuring

The destructuring assignment syntax is an expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables. Let's see it in practice.

1. Array destructuring

```
const [a, b] = [10, 20];

console.log(a); // 10
console.log(b); // 20
```

2. Can be even more sophisticated

```
const getAB = () => [10, 20];

const [a, b] = getAB();

console.log(a); // 10
console.log(b); // 20
```

3. Object destructuring

```
const getObject = () => ({
  a: 10,
  b: {
    c: 20
  }
});
```

```
const { a, b: { c }} = getObject();

console.log(a); // 10
console.log(b); // b is not defined
console.log(c); // 20
```

4. Destructuring with spread operator

```
const getObject = () => ({
  arr: [0, 1, 2],
  b: 3,
  c: 4,
});

const { arr: [head, ...tail], ...rest } = getObject();

console.log(head); // 0
console.log(tail); // [1, 2]
console.log(c); // { b: 3, c: 4 }
```

Class

The class declaration creates a new class with a given name using prototype-based inheritance. Classes can be created also with class expression, for more information please read documentation. In React, classes were used as class components, which were now replaced by the newer function components and hooks (more on that later).

```
class Rectangle {
  constructor(a, b) {
    this.a = a;
    this.b = b;
  }

  getArea() {
    return this.a * this.b;
  }
}

class Square extends Rectangle {
  constructor(a) {
    super(a, a);
  }
}

const s = new Square(15);
const r = new Rectangle(10, 5);

console.log(s.getArea()); // 225
console.log(r.getArea()); // 50
```

Optional chaining

This operator allows you to safely try to access nested properties of objects that are possibly undefined, without the need to manually write all the checks. For more information please read documentation.

```
const foo = {};
const goo = { a: { b: 1 } };

// Without optional chaining without checks
console.log(foo.a.b); // Uncaught TypeError: Cannot read property 'b' of
undefined
console.log(goo.a.b); // 1
```

```
// Without optional chaining with checks
console.log(foo.a && foo.a.b); // undefined
console.log(goo.a && goo.a.b); // 1

// With optional chaining
console.log(foo?.a?.b); // undefined
console.log(goo?.a?.b); // 1
```

Nullish coalescing

This operator works great together with optional chaining operator. It replaces previously used || operator which achieved a similar goal but was less safe to use because of false negatives when using falsy values (0, empty string, etc.). For more information please read documentation.

```
const foo = {};
const goo = { a: { b: 0 } };

// Without nullish coalescing
console.log(foo?.a?.b || 42); // 42
console.log(goo?.a?.b || 42); // 42 (wrong because 0 is falsy)

// With nullish coalescing
console.log(foo?.a?.b ?? 42); // 42
console.log(goo?.a?.b ?? 42); // 0
```