Overview training

Javascript + Typescript

07.11 - 08.11

Who am I?

Mateusz Jabłoński

- programmer since 2011
- React / Angular on Frontend
- NodeJS / Java on Backened
- sometimes blogger / trainer / mentor

mateuszjablonski.com
mail@mateuszjablonski.com

From the beginning

The arrangements

- our goal and agenda of workshops
- mutual expectations
- questions and discussions
- flexibility
- openness and honesty

What awaits us?

Day 1.

- ECMAScript
  - Introduction
  - Overview of basic assumptions and its intended use
  - Standardization
  - Javascript as the most popular implementation of ECMAScript
- Tools
  - NodeJS
  - NPM and other packages managers
  - Task Managers
  - Webpack
  - Babel
  - CDN
  - Static Code Analysis
- Javascript
  - Different ways to declare variables
  - Scopes in JS
  - Data types
  - Loops and conditionals
  - Operators in JS
  - Working with arrays and objects
  - Prototype inheritance
  - Functions
  - Modules
  - Destructuring Assignment
  - Advanced operators

What awaits us?

Day 2.

- Javascript
  - AJAX
  - Asynchronicity
  - Data serialisation
  - HTTP protokol in communication
  - REST API
  - Secrets - environments variables
  - Keyword this
  - Closure
  - Functional programming
  - Cookies and different types of browser storage
- Typescript
  - Overview
  - Types inference and narrowing
  - Data types
  - Types, enums and interfaces
  - Generic types
  - Unions
  - Types casting
  - Classes in Typescript
  - Utility types
- Frontend frameworks
  - React / Angular / Vue

github.com/matwjablonski/bsh-training-part1

ECMAScript

The Basics

JavaScript

- Mocha / LiveScript / NativeScript / JavaScript (?)
- Brendan Eich
- based on:
  - C
  - Java
  - Modula-2
- first release: 4th of December 1995
- JScript / ActionScript

Java is to Javascript as ham is to hamster.

Jeremy Keith, 2009

ECMA International

- ECMA International - organization which is responsible for standardizing languages like: ECMAScript, C#, Eiffel
- Javascript is implementation of language ECMAScript (other implementations: JScript, ActionScript)

ECMA-262

First attempt to make it better

Javascript

Embedding and loading scripts

In HTML

- JS file or JS code should be added to HTML file
- Order of JS scripts in HTML matters (priorities of loading)

In HTML

- async
  - script will be fetched in parallel to parsing
  - script will be executed as soon as it is available
  - script will be executed asynchronous
- defer
  - script will be executed after the document has been parsed, but before firing event DOMContentLoaded
  - scripts with this attribute are blocking event DOMContentLoaded until script has loaded
- type=module
  - code is treated as JS module
  - code executions will be defered until dependencies will load
  - **defer** isn’t working

!

When script wasn’t described by attribute async / defer or type=module code is executed immediately after loaded regardless of the moment of analyzing the website by browser.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
    <script>
      console.log('Hello world!')
    </script>
</body>
</html>
```

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
    <script src="./app.js" defer></script>
</body>
</html>
```

Javascript

Strict mode

use strict

- introduces in ES5 in 2009
- previous versions of JS ignores this directive
- isn’t supported by IE9 and older versions
- forces writing clean code
- previously accepted bad practices are converted into errors
- doesn’t allow accidental declaration of global variables
- doesn’t allow modification objects marked as non-writable

```
x = 3;       // nie ma błędu
myFunction();

function myFunction() {
  "use strict";
  y = 3;   // jest błąd
}
```

Task

5 minutes
materials/TASK_01.md

Javascript

Data types

Typy danych

- Primitives
  - String
  - Number
  - Boolean
  - Symbol
  - BigInt
- Non-Primitives (Reference)
  - Object
    - Array
    - Function
    - Set / Map / WeakSet / WeakMap
- null
- undefined

Dynamic and weak typing

- JavaScript is dynamic and weak typed language
- variable is a symbolic name of value that is stored
- variable shouldn’t have declared type, type is assigned to value, not to variable
- Javascript fetch information about type based on literals
- literals represents values in JS

```
let a;

// String literal
a = "JavaScript";

// Number literal
a = 12;

// Boolean literal
a = false;

// Array literal
a = [];

// Object literal
a = {};
```

Javascript

Comparison and safe conversion of types

Types comparison

- ===
  - strict equality
- ==
  - equality (knows also as loose equality)
  - while comparison types are converted:
    - if both values are primitives - conversion to type Number
    - if any of values isn’t primitive value - JS will convert non-primitive to primitive and then to Number
    - if compared value is null or undefined, the second value should be the same to return true, in other cases it will always return false
  - Object.is()
    - works same as strict equality

```
1 === "1"
// false

1 === 1
// true

1 == "1"
// true

1 == 1
// true

0 == ""
// true

0 === ""
// false
```

Javascript

Variables and scopes

Scope is set of rules for keeping and searching variables in a program

Scopes

- Lexical
  - It is used in almost every language based on C language
  - Based on functions and variables localization in time of compilation
- Dynamic
  - Based on time of executing functions
  - Value is changing based on order of executing functions
  - In JS doesn’t exist (concept is close to way of working reference **this**)

```
let a = 1;

function foo() {
  let a = 2;
  console.log(a);
}

a = 3;

foo();
```

Scopes

- global
  - Top level scope, connected with global object, in browser global object is **window**
  - Assigning values to variables without declaring them causes that variable will be added to global scope
  - Works with **var**, **let** and **const**
- function lexical
  - Based on function scope
  - Works with **var**
- block
  - Based on block scope (condition, function, loop - everything inside **{ }**)
  - Works with **let** i **const**

```
var a = 1;
// global scope

function foo() {
  var b = 2; // function scope
  let c = 3; // block scope (function is also a block)

  if (true) {
    let d = 4; // block scope (condition)

  }

  console.log(d); // Error
  console.log(a); // 1
  console.log(b); // 2
}

foo();

console.log(b); // Error
```

Ways of declaring variables

- **var**
  - Oldest way for declaring variables
  - Works based on scopes: global and function
  - Allows modify values of variable regardless to previous type
  - After declaration variable in global scope its not possible to remove it from global scope using **delete** keyword
- **let**
  - Introduced in ES2015
  - Works based on scopes: global and block
  - Allows modify values of variable regardless to previous type
  - After declaration in global scope its not creating separate property in global object
- **const**
  - Introduced in ES2015
  - Works based on scopes: global and block
  - Not allows modify primitives values
  - Value have to be assign while declaration
  - Allows mutate non-primitives
  - After declaration in global scope its not creating separate property in global object

Hoisting is the process of moving declaration of variables and functions to the top of scope in which they were declared

```
foo();

function foo() {
  console.log(x);
  // undefined

  var x = 1;
  y = 2;

  console.log(x);
  // 1
}

console.log(y);
```

Javascript

Syntax

```
if (a === 1) {
  console.log(a);
} else if (a === 2) {
  console.log(a);
} else {
  console.log("a is not 1 or 2");
}
```

```
for (var i = 0; i < 5; i++) {
  console.log(i);
}

const object = { a: 1, b: 2, c: 3 };

for (const property in object) {
  console.log(property);
}

const array = ['a', 'b', 'c'];

for (const element of array) {
  console.log(element);
}

let n = 0;

while (n < 3) {
  n++;
}

console.log(n);
```

```
function foo(argument1, argument2) {
  return true;
}

const bar = function(argument1, argument2) {
  return null;
}

const foobar = (argument1, argument2) => {
  return false;
}
```

```
const expr = 'Papayas';
switch (expr) {
  case 'Oranges':
    console.log('Oranges are $0.59 a pound.');
    break;
  case 'Mangoes':
  case 'Papayas':
    console.log('Mangoes and papayas are $2.79 a pound.');
   break;
  default:
    console.log(`Sorry, we are out of ${expr}.`);
}
```

```
// addition
1 + 2;

// concatenation
"a" + „b";

// subtraction
2 - 2;

// multiplication
2 * 2;

// division
2 / 2;

// exponentiation
2 ** 3;

// rest of division (modulo)
5 % 2;

// more complex math…
Math.floor(2.235);
```

```
// logical operators
a < b;
a <= b;
a > b;
a >= b;
a == b;
a === b;
a && b;
a || b;

// Nullish coalescing operator
a ?? b;

// negation
!a;

// incrementation and decrementation
a++;
a--;
```

Javascript

Global objects

Global objects in JS

- **window**
  - Object prepared by browser
  - It is an representation of actual DOM document
  - It is available from every place in JS code in browser
  - Its properties and methods can be accessible directly
- **location**
  - Informations about current URL address of document
- **document**
  - Contains informations about current state of DOM tree
  - Contains method which simplify modifications of DOM tree structure
- **Date**
  - Contains methods for working with dates and time

```
window.a = 2;

console.log(a);
// 2

window.alert(„This is my message to you");

alert(„This also is my message to you");

console.log(document.body);
// show everything what is in body element in HTML
```

Task

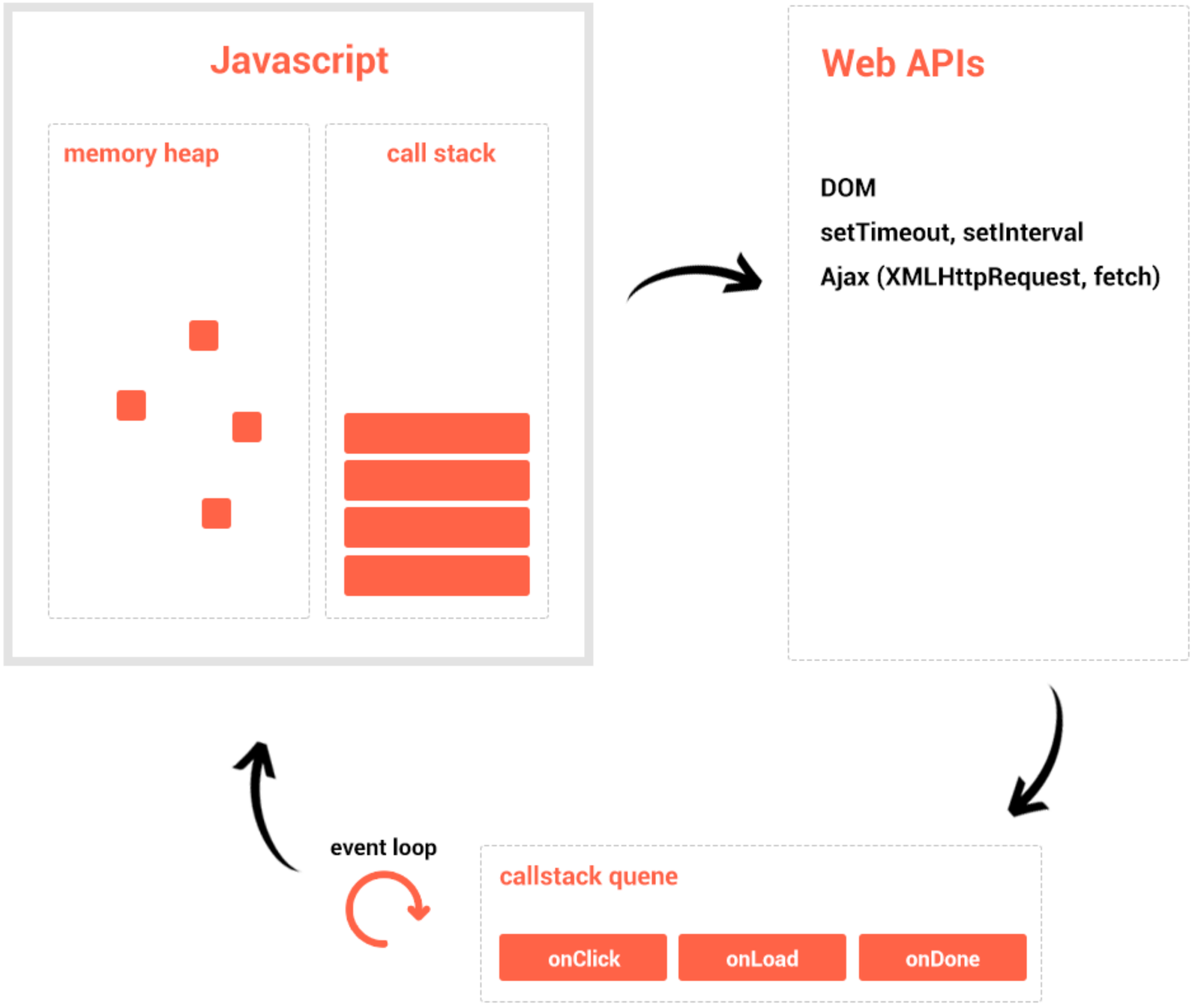
15 minutes
materials/TASK_02.md

Javascript

Functions

Functions

- Subtype of type **Object**
- Can return value (with **return** keyword), but its not necessary
- Can be assigned to variables and properties of objects
- Hoisting is working only on functions declared in classic way (with keyword **function**)
- Every execution of function is put on the stack and executed with rule **Last In First Out**
- Javascript is single-threaded language - JS engine has implemented Event Loop - this is something like endless loop responsible for every next execution of functions
- Specific type of functions are **generators**


Javascript
memory heap
call stack
Web APIs
DOM
setTimeout, setInterval
Ajax (XMLHttpRequest, fetch)
event loop
callstack quene
onClick
onLoad
onDone

Javascript+

Context of the call and meaning of reference this

this

- Object that is context of the call
- In JS what is **this** depends on way of define of function, form and context of its execution
- By default it points to global object (**window** in browser)
- When function is called directly by reference, **this** points to global object (in strict mode points to **undefined**)
- When called function is property of an object, **this** will point to an object, where is the function declared
- When we pass reference to called function to new object, **this** will point to new object (context of the call)
- When called function is a result of fired event, **this** will point to an object which fired event

```
const obj = {
  a: 1,
  say: function () {
    return this;
  }
};

obj.say(); // obj

const obj2 = {
  hi: obj.say,
};

obj2.hi(); // obj2

button.addEventListener('click', function() {
  this; // button
});

function foo() {
  this; // window
}

function bar() {
  "use strict"
  this; // undefined
}
```

We can control context of the call by built in methods: call, apply and bind.

Javascript

Arrow functions

Arrow functions

- Simplified version of classic function
- Shouldn’t be use in class methods
- Not changing context of **this**
- Can’t be use with change context functions like (**call**, **bind**, **apply**)
- Not has access to object **arguments**
- Can’t be use as constructor method
- Not working properly with generators functions

```
// By default - return expression

const odds = myArr.map(v => v + 1);
const nums = myArr.map((v, i) => v + 1);

const pairs = myArr.map(v => ({
  even: v,
  odd: v + 1,
}));

// Declarations inside curly braces

nums.filter(v => {
  if (v % 5 === 0) {
    return true;
  }

  return false;
});
```

```
// Change the context of the call

function a() {
  return this;
}

element.addEventListener('click', a); // this === element

const a = () => {}
element.addEventListener('click', a); // this === window
```

Javascript

Working with function parameters

```
// arguments

function sum() {
  let result = 0;
  for (let i = 0; i < arguments.length; i++) {
    result += arguments[i];
  }

  return result;
}

sum(2, 5, 12, 567, 1, 234); // 821
```

```
// Default values

function powerNumber(firstNumber, power = 2) {
  return firstNumber ** power;
}

powerNumber(2); // 4
powerNumber(2, 3); // 8
```

```
// rest / spread

function f(a, ...rest) {
  return a + rest.length;
}

f(2, "abc", true); // 4



function g(x, y, z) {
  return x + y + z;
}

const arr = [1, 2, 3];

g(...arr); // 6
```

```
// Copying objects

const obj = {
  name: 'nazwa',
  value: 123,
  isNew: false,
};

const obj2 = { ...obj };

const obj3 = Object.assign({}, obj);

const obj4 = Object.create(obj);
```

Javascript

Arrays functions

```
// converts array-like objects into an array

Array.from(document.querySelectorAll('*'));

// created new array - like in new Array(), but has different usage for execution
// with one argument

Array.of(1, 2, 3); // [1, 2, 3]
Array.of(2); // [2]

new Array(1, 2, 3); // [1, 2, 3]
new Array(2); // [ , ]

[0, 0, 0].fill(7, 1); // [0, 7, 7]
[1, 2, 3].find(x => x === 3); // 3
[1, 2, 3].findIndex(x => x === 2); // 1
[1, 2, 3, 4, 5].copyWithin(3, 0); // [1, 2, 3, 1, 2]
['a', 'b', 'c'].entries(); // iterator [0, 'a'], [1, 'b'], [2, 'c']
['a', 'b', 'c'].keys(); // iterator 0, 1, 2
['a', 'b', 'c'].values(); // iterator 'a', 'b', 'c'
```

```
const data = [
  {
    name: ,Adam',
    age: 26,
  },
  {
    name: 'Monika',
    age: 22,
  },
  {
    name: 'Stan',
    age: 62,
  },
];

data
  .filter((person) => person.age > 25)
  .map((person) => ({
    ...person,
    salary: person.age * 100,
  }))
  .reduce((acc, { salary }) => acc + salary, 0);
```

Task

20 minutes
materials/TASK_03.md

Javascript

Object Oriented Programming with Prototype Inheritance

Prototype

- In JS everything is an object
- Every object has private property **prototype**, that contains object **prototype**
- Every **prototype** has it own **prototype** until prototype is not **null** (**null** cannot have **prototype** property)
- Joining prototypes of different objects in chain is **prototype inheritance**
- **Prototype inheritance** gives us access to properties and methods of nested objects
- **Prototype** object contains informations about every object (its constructor, getters, setters and list of inherited methods)
- Finally every JS object inherit by **Object** constructor

```
const obj = {
  a: 1,
  b: 2,
  __proto__: {
    c: 3,
    d: 4,
  }
};

obj.a; // 1
obj.c; // 3

// {
//   a: 1,
//   b: 2,
//   [[Prototype]]: {
//     c: 3,
//     d: 4,
//   }
// }
```

Creating objects

- By **literals**
- By operator **new**

```
function Dog(name) {
  this.name = name;
}

Dog.prototype.sayHello = function () {
  return 'Hi, I am ' + this.name;
}

const dog = new Dog('Azor');

dog.name; // Azor
dog.sayHello(); // Hi, I am Azor
```

Object prototype is common for all instances of object. Methods and fields aren’t copied to every new instance but shared.

```
function Animal() {}

Animal.prototype = {
  isAnimal: true
}

function Dog(name) {
  this.name = name;
}

Dog.prototype = Object.create(Animal.prototype, {
  sayHello: {
    value: function() {
      return 'Hi, I am ' + this.name;
    },
    writable: true,
    enumerable: true,
    configurable: true
  }
});

const dog = new Dog('Azor');
```

```
function Account() {
  this.incomes = [];
}

Account.prototype.addIncome = function(income) {
  this.incomes.push(income);
}

function ChildAccount() {
  Account.call(this);
}

ChildAccount.prototype.addIncome = function(income) {
  if (income.value > 500) {
    throw new Error();
  }

  this.incomes.push(income);
}

Object.setPrototypeOf(ChildAccount.prototype, Account.prototype);
```

```
function Person() {}

Person.prototype.sayHello = function(){
    return 'Hi';
};

function Student() {}

Student.prototype = new Person();

Student.prototype.constructor = Student;

Student.prototype.sayHello = function () {
    return 'Hi, I am student!’;
}

var student = new Student();
```

Task

20 minutes
materials/TASK_04.md

Javascript

Destructuring Assignment

```
// Destructuring arrays

const [a, , , d] = [1, 2, 3, 4];

const [first, ...rest] = [1, 2, 3, 4];

// Destructuring objects

const {
  name,
  age,
  getName,
  skills: {
    programming,
    languages,
    other,
  }
} = person;
```

```
// Destructuring function parameters

function showName({ name }) {
  return name;
}

// instead of

function showName(person) {
  return person.name;
}

showName(person);
```

Javascript

Dynamic literal

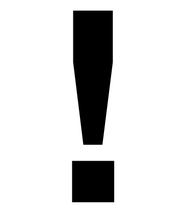
```
// Dynamic literal

const obj = {
  name: 'abc',

  // === handler: handler
  handler,

  // === toString: function toString() {}
  toString() {
    return 'd ' + super.toString();
  },

  // Dynamic names of properties
  [ 'prop_' + (() => 42)() ]: 42,
}
```

Task

25 minutes
materials/TASK_05.md

Javascript

OOP - classes, inheritance and access control



Definition of class in JS doesn’t exist in the same way as it is in Java. Class in JS it is Syntax Sugar. Under the hood everything works the same.

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

  sayHello() {
    return `Hi, I am ${this.name}`;
  }
}

const dog = new Dog('Azor');
dog.name; // Azor
dog.sayHello(); // Hi, I am Azor
```

```
class Dog {
  legs = 4;

  constructor(name) {
    this.name = name;

    this.bark = function() {
      return 'bark bark’;
    }
  }

  sayHello() {
    return `Hi, I am ${this.name}`;
  }
}

const dog = new Dog('Azor');
dog.name; // Azor
dog.sayHello(); // Hi, I am Azor
```

Constructor

- Always executed first
- Contains initial code for our project / class
- In constructor we can call **super** method, to call constructor of extended class
- Methods and fields in constructor are copied to every instance of class (in opposite to methods in body of the class - those will be in prototype)

```
class Animal {
  constructor(legs = 4) {
    this.legs = legs;
  }

  eat() {
    console.log('omomm');
  }
}

class Dog extends Animal {
  constructor(name) {
    super();
    this.name = name;
  }

  sayHello() {
    return `Hi, I am ${this.name}`;
  }
}

const dog = new Dog('Azor');
dog.name; // Azor
dog.eat(); // omomm
```

Access control

- By default all fields and methods in class are public
- Private properties and methods should be prefixed by **#**
- Access to private fields and methods is only inside the class
- In JS we don’t have **protected** AC

```
class Person {
  #name;

  constructor(name) {
    this.#setName(name)
  }

  #setName(name) {
    this.#name = name;
  }

  getName() {
    return this.#name;
  }
}

const person = new Person('Adam');
person.name; // Error
person.#name; // Error
person.setName(); // Error
person.#setName(); // Error
person.getName(); // Adam
```

```
class Project {
  #name;

  constructor(name) {
    this.#codeName = name;

    console.log(this.#codeName);
  }

  get #codeName() {
    return `CN: ${this.#name}`
  }

  set #codeName(name) {
    this.#name = name;
  }
}

const project = new Project('Adam');
```

Task

25 minutes
materials/TASK_06.md

Javascript

Default and named import and export

```
// Named export

export const someFunction = () => {};
import { someFunction } from './';

// Default export

const someNewFunction = () => {};
export default someNewFunction;

import anyNameWeWant from './';
import { default as anyNameWeWant } from './';
```

Javascript

Text interpolation and templates

Template strings

- Defined by backticks `
- Allows to use variables and expressions inside string
- Multiline
- Allows to nest strings inside strings
- Resolves problem with concatenation of strings

```
const aboutMovie = `
  <div class="container" id="movie-${movieIndex}">
      <h3>${movieTitle || 'Unknown title'}</h3>
      <div class="movieDescritpion">
          <p>${movieDescription}</p>
          <img src=${moviePoster}/>
      </div>
  </div>
`;
```

Javascript!

Modules

What I’m describing here is not a technical problem. It’s a matter of people getting together and making a decision to step forward and start building up something bigger and cooler together.

Kevin Dangoor, 2009

CommonJS

- Project was started in January 2009 by Kevin Dangoor (Mozilla)
- Initially named ServerJS
- The goal of the project was to establish one module system for JS outside the browser
- Creators wanted to build environment for work on modules, which could be reusable in different environments than browser
- CommonJS is widely used, especially in NodeJS world
- CommonJS is also used for browser projects, but code should be previously prepared by transpilers and module bundlers (browsers not support CommonJS)
- Specific of this concept is usage of keywords: **require** and **module.exports**

ES2015 Modules

- New concept of declaring and usage of modules
- Introduced with standard ES2015
- Known also as **ES6 Modules**
- Specific of this concept is usage of keywords: **import** and **export**

```
const Person = require('./person');

function User() {

}

User.prototype.handleAccess = function() {

}

module.exports = User;
```

```
import Person from './person';

export class User extends Person {
  constructor() {
    super();
  }

  handleAccess() {

  }
}
```

Require.js

- First attempt to handle problem with dependencies in JS projects in browser
- Solution base on specification of AMD Modules (**Asynchronous Module Definition**)
- Part of interfaces are same as in CommonJS specification
- Project started in September 2009
- Module / file will wait for its dependencies

Webpack

- Module bundler
- Working with many formats of modules (ES2015, AMD, CommonJS)
- Treats everything as modules (eg. scss, html, graphics)
- Working properly with popular task runners like (Gulp, Grunt)
- Its a standard in JS projects

```
const path = require('path');

module.exports = {
  mode: 'development',
  entry: './foo.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'foo.bundle.js',
  },
};
```

Javascript

Packages


yarn


rush


pnpm


npm

NPM?

- Newton's Principia Mathematica
- Nuclear Powered Mushroom
- Neurosis Prevention Mechanism
- Neurotic Pink Mongooses
- Naughty Push Message
- Nectar of the Programming Masses
- Node Package Manager

NPM

- System to manage dependencies
- Dependencies are described with accuracy to version (major, minor, patch) in **package.json** file
- **npm install** - installs packages
- **npm update** - check new versions of packages and install them
- **npm install name-of-package --save-dev** - it installs new package, adds it to group of dependencies according to flag passed to command
- **npx nazwa-pakietu** - it runs a package, even if its not installed on the system
- **package-lock.json** - generated file while installation, it contains information about all installed packages and their versions (**npm ci**)

```
{
  "name": "playground",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
  },
  "scripts": {
    "test": "echo 'none'"
  },
  "devDependencies": {
  }
}
```

Task

15 minutes
materials/TASK_07.md

Javascript

NodeJS

NodeJS

- Open-source, cross-platform, back-end JavaScript runtime environment bundler
- Runs on a JavaScript Engine and executes JavaScript code outside a web browser
- Initial release: 2009
- V8 is the JavaScript engine that powers Google Chrome (by default its used in NodeJS)
- Operates on a single-thread event loop
- The modules can be directly loaded into memory and executed from within JS environment as simple CommonJS modules

Javascript engines

- V8 (Chrome, Brave, Edge, Opera, nodeJS, deno, just-js)
- JavaScriptCore (Safari, bun)
- SpiderMonkey (Firefox)
- Chakra (Edge Legacy, IE)
- Hermes (React Native on Android)

Javascript

Transpilation and compatibility between browsers

Transpilation is a process rewriting code to its equivalent in other language (or same, but in different variant).

Babel

- Free Javascript transpiler
- First released in 2014
- Transpile code from ES2015+ into ES5
- Allows transformations into JS from custom technologies like JSX
- Contains polyfills, to allow using functions, that aren’t available in ES5 standard

Javascript

Scope management

Scope management

- Keeping data in global scope is an anti-pattern
  - Same names can be used by different variables (also delivered from external libraries and modules)
  - Problem with verification of existing variables in global scope
- Variables that are kept in lexical scopes are automatically cleaned when they aren’t necessary (for example function was executed, **Garbage Collector** will free up memory)
- Using **let** and **const** instead of **var** avoid lot of errors

Typescript

Typescript

- Created and develop by Microsoft
- Superset of Javascript
- It add strong typing
- Improves process of development in projects
- Added types aren’t moved to production code

Typescript

- Additional pair of eyes, which look with us on code and search for errors
- Has great syntax prompting
- Cooperation between tools on higher level than in vanilla JS
- Refactoring is not a problem - consistency of code is checked by compiler
- Guarantee of values in constants and variables - only looking into types we can inference what is or should be in current place (without running an app)
- Types in TS are always up to date documentation of code
- TS is also a way to create contracts between components and external services (also between FE and BE)

Typescript

- Not solves JS problems
- TS check types only in compiling time
- tends to be messy (especially in dependencies)

Typy

- boolean
- string
- number
- array
  - let array: number[];
- object
- tuple
  - let tuple: [number, boolean, string];
- any
- unknown
- never
- void
- enum

Typescript

Type inferences

TS will know what is type of variable. If we not assign type and TS wouldn’t be able to recognize it, TS will assign to variable type any.

```
const title: string = "This is title!";
const secondTitle = "This is title!”;
```

Typescript

Types and interfaces

```ts
interface Book {
  title: string;
  author: string;
}

interface Student {
  name: string;
  lastName: string;
  age: number;
  isActive: boolean;
  library?: Book[];
}

let student: Student = {
  name: 'Adam',
  lastName: 'Niezgódka',
  age: 23,
  isActive: true,
};
```

```ts
interface Publication {
  publicationDate: string;
}

interface Book extends Publication {
  title: string;
  author: string;
}
```

```ts
type Publication = {
  publicationDate: string;
}

type Book = {
  title: string;
  author: string;
} & Publication;


const book: Book = {
  title: 'John',
  author: 'Smith',
  publicationDate: '2022-03-20'
}
```

```ts
enum Category {
  ROMANCE = 'romance',
  THRILLER = 'thriller',
  PROGRAMMING = 'programming',
}

type Publication = {
  publicationDate: string;
  category?: Category;
}

interface OldPublication {
  publicationDate: string;
  category?: Category;
}
```

```ts
interface A {
  color: string;
}

interface A {
  size: number;
}

const wear: A = {
  color: 'red',
  size: 20,
}

type B = { // Error: Duplicate identifier 'B'
  color: string;
}

type B = { // Error: Duplicate identifier 'B'
  size: number;
}
```

```ts
type GenerateTitleType = (value: string) => string;

const generateTitle: GenerateTitleType = (value) => 'to jest tytuł' + value;

const generateTitle2: (value: string) => string = (value) => 'to jest tytuł' + value;

const generateTitle3 = (value: string): string => 'to jest tytuł' + value;


function setData(name: string, schools?: string[]): string | number {
  if (schools) {
    return schools.length;
  }
  return `my name is ${name}`;
}
```

Unions allows to create new type, which can have values from one or more other types. To create union type we need to use | (pipe).

```ts
interface Man {
  name: string;
};

interface Woman {
  name: string
};

type Person = Man | Woman;
```

```ts
type Keys = "a" | "b" | "c";

type Test = {
    [K in Keys]?: number
}
```

Typescript

tsc and extensions

tsc

- Typescript Compiler
- CLI tool, prepared by Typescript for compilation TS code
- Compile TS into JS
- In compilation time all added types are deleted
- TSC performs **downleveling** (all new functionalities of language are converted into version compatible with version ES5)
    - For example: **template string** into classic **concatenation**
- npm install -g typescript

ts and **tsx** are two formats that we can use to save our TS code. **tsx** is dedicated for saving files with **JSX** syntax.

Typescript

Configuration

tsconfig.json

- Configuration file
- Informations in this file are about what, how and where should be compiled
- We can define additional libraries which we uses in our project
- We can declare our own paths to specific directories with modules
- We can exclude and include files and directories from / to compilation process

```
{
  "compilerOptions": {
    "baseUrl": "./",
    "outDir": "./dist/",
    "moduleResolution": "node",
    "noImplicitAny": true,
    "removeComments": true,
    "preserveConstEnums": true,
    "sourceMap": true,
    "module": "commonjs",
    "target": "es6",
    "allowJs": true,
    "esModuleInterop": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "allowSyntheticDefaultImports": true,
    "lib": ["es7", "es2015", "dom"],
    "paths": {
      "Components/*": ["src/components/*"],
      "Entity/*": ["src/models/*"],
      "*": ["types/*"]
    }
  },
  "include": ["./src/**/*", "public"],
  "exclude": ["node_modules", "**/*.spec.ts"]
}
```

Task

15 minutes
materials/TASK_08.md

Typescript

Classes

```ts
class Car {
  private name: string;

  constructor(name: string) {
    this.name = name;
  }

  public getName() {
    return this.name;
  }
}

const car = new Car('Audi');
car.name; // Error: Property 'name' is private and only accessible within class 'Car
car.getName(); // Audi
```

```
class Car {
  protected name: string;

  constructor(name: string) {
    this.name = name;
  }

  public getName() {
    return this.name;
  }
}

class ECar extends Car {
  constructor(name: string) {
    super(name);
 }
}

const eCar = new ECar('VW');
eCar.getName(); // VW
eCar.name; // Property 'name' is protected and only accessible within class 'Car'
and its subclasses.
```

```ts
class Car {
  protected name: string;
  readonly minEcoRate = 0.8;

  constructor(name: string) {
    this.name = name;

    this.minEcoRate = 10; // Type '10' is not assignable to type '0.8'
  }

  public getName() {
    return this.name;
  }
}
```

```
class Car {
  protected name: string;
  readonly minEcoRate = 0.8;
  private _hasPassengers;

  constructor(name: string) {
    this.name = name;
  }

  public getName() {
    return this.name;
  }

  get hasPassengers(): boolean {
    return this._hasPassengers;
  }

  set hasPassengers(value: boolean) {
    this._hasPassengers = value;
  }
}
```

```ts
interface Thing {
  setName(): void;
  getName(): string;
}

class Box implements Thing {
  private name: string;

  public setName(): void {

  }

  public getName(): string {
    return this.name;
  }
}
```

Task

15 minutes
materials/TASK_09.md

Typescript

Narrowing

Narrowing

- Process of transformation type from more general into more precise (narrow)
- We can achieve that using Type Guards
- Type Guard is a special control to narrow a type
- We can achieve that using operators like:
  - typeof
  - in
  - instanceof

typeof

- **typeof** works with primitive types
- **typeof** return a string with name:
  - „string"
  - "number"
  - "bigint"
  - "boolean"
  - "symbol"
  - "undefined"
  - "object"
  - "function"

```ts
function padLeft(padding: number | string, input: string): string {
  return "_".repeat(padding);
  // Argument of type 'string | number' is not assignable to parameter of type
     ,number'.
  // Type 'string' is not assignable to type 'number'.
}
```

```ts
function padLeft(padding: number | string, input: string): string {
  if (typeof padding === 'number') {
    return "_".repeat(padding);
  }
  return padding + input;
}
```

in

- Operator in JS
- It check if in object exists property

```ts
type Fish = { swim: () => void };
type Bird = { fly: () => void };

function move(animal: Fish | Bird): void {
  if ("swim" in animal) {
    return animal.swim()
  }

  return animal.fly();
}
```

instanceof

- Operator in JS
- It check if in prototype inheritance chain exist relation to prototype of object that is verifying

```ts
function getDay(date: string | Date): string | number {
  if (date instanceof Date) {
    return date.getUTCDate();
  }

  return date;
}
```

Casting

- It can change one type into another
- Not cause any changes in structure of variable
- Should be used only when we knows more about type of variable than compiler

```
let someValue: any = "this is a string";
let strLength: number = (<string>someValue).length;

let someValue2: any = "this is a string”;
let strLength2: number = (someValue as string).length;
```

Task

15 minutes
materials/TASK_10.md

Typescript

Generic types

```ts
interface ResponseShape<T> {
  name: string;
  body: T;
  isLoading: boolean;
}

function fetchResponse<T>(body: T): ResponseShape<T> {
  return {
    name: ,name',
    body,
    isLoading: false,
  }
}

fetchResponse("example");
```

Typescript

Utility types

Utility types

- Additional types from TS for doing most popular transformations on types
- Available globally in TS files
- Examples:
  - Partial<Type>
  - Required<Type>
  - Readonly<Type>
  - Record<Keys, Type>
  - Pick<Type, Keys>
  - Omit<Type, Keys>

Task

15 minutes
materials/TASK_11.md

Task

15 minutes
materials/TASK_12.md

Javascript

Callbacks

I will call back later!

Callback

Callback

- funkcja wywołania zwrotnego
- Technique in programming which is reverse to call the function
- Function is only register to later execution
- Callback can be passed to function as an argument
- Before Promises were added to JS were often used to create async communication, this creates occurrence known as callback hell

Callback Hell is an effect of multi-nested and executed callbacks

```
(tail =>
    (is_nil =>
        (isEmpty =>
            ($0 =>
                ($1 =>
                    ($2 =>
                        ($3 =>
                            (is_$0 =>
                                ($$ =>
                                    (is_$$ =>
                                        (succ =>
                                            (add =>
                                                (pair_succ =>
                                                    (prev =>
                                                        (sub =>
                                                            (len =>
                                                                (sum =>
                                                                    (rcon =>
                                                                        (rev =>
                                                                            (reverse =>
                                                                                (elems =>
                                                                                    (list =>
                                                                                        (map =>

                                                                                            map(list(elems($1)($2)($3)))(elem => sub(elem)($1))

                                                                                        )(y(map => l => f => when(isEmpty(l))(_ => nil)(_ => co
                                                                                    )(es => reverse(es($$)))
                                                                                )(rcon(nil))
                                                                            )(l => rev(nil)(l))
                                                                        )(y(rev => r => l => when(isEmpty(l))(_ => r)(_ => rev(con(head(l))(r))
                                                                    )(y(rcon => t => h => when(is_$$(h))(_ => t)(_ => rcon(con(h)(t)))))
                                                                )(y(sum => l => when(isEmpty(l))(_ => $0)(_ => add(head(l))(sum(tail(l))))))
                                                            )(y(len => l => when(isEmpty(l))(_ => $0)(_ => add($1)(len(tail(l))))))
                                                        )(m => n => n(prev)(m))
                                                    )(n => left(n(pair_succ)(pair(no_use)($0))))
                                                )(p => pair(right(p))(succ(right(p))))
                                            )(m => n => n(succ)(m))
                                        )(n => f => x => f(n(f)(x)))
                                    )(n => n(_ => no)(no))
                                )(_ => _ => yes)
                            )(n => n(_ => no)(yes))
                        )(f => x => f(f(f(x))))
                    )(f => x => f(f(x)))
                )(f => x => f(x))
            )(_ => x => x)
        )(is_nil)
    )(l => l(_ => _ => no))
)(right)
```

Javascript

Asynchronicity

AJAX

- **Asynchronous JavaScript and XML**
- AJAX is not a programming language
- This technique was proposed in 2005 for describe new ways of creating dynamic websites based on existing solutions
- Despite that in name we have **XML**, preferred standard to data serialization is **JSON**.

Object XMLHttpRequest

- This object in JS is responsible for executing HTTP requests
- Can works in two modes: synchronous and asynchronous
- Requests can be created after website is loaded

```
function requestListener () {
  console.log(this.responseText);
}

const req = new XMLHttpRequest();
req.addEventListener('load', requestListener);
req.open('GET', 'http://www.example.org/example.txt');
req.send();
```

```
const xhr = new XMLHttpRequest();

xhr.open("GET", "/bar/foo.txt", true);

xhr.onload = (e) => {
  if (xhr.readyState === 4) {
    if (xhr.status === 200) {
      console.log(xhr.responseText);
    } else {
      console.error(xhr.statusText);
    }
  }
};

xhr.onerror = (e) => {
  console.error(xhr.statusText);
};

xhr.send(null);
```

```
const request = new XMLHttpRequest();
request.open('GET', '/bar/foo.txt', false);
// `false` makes
// the request synchronous

request.send(null);

if (request.status === 200) {
  console.log(request.responseText);
}
```

Promise

- Concept resolves problem with **Callback Hell**
- Under the hood still we have **XMLHttpRequest** objects
- **Promise** is an object, which contains result of operation, that can finish in any moment in time
- **Promise** has three states:
  - **pending** - starting state
  - **fullfilled** - operation succeed
  - **rejected** - operation failed
- First argument during of creating a Promise is callback which has two other callbacks: **resolve** and **reject**
- Promises can be chained with method **.then**
- For catching errors we have method **.catch**
- After all operations (all promises) is executed method **.finally**

```
const getData = (duration = 0) => new Promise((resolve, reject) => {
  setTimeout(resolve, duration);

  if (error) {
    reject(error);
  }
})

const p = getData(1000)
  .then(() => {
    return getData(2000)
  })
  .then(() => {
    throw new Error('hmm');
  })
  .catch((err) => {
    return Promise.all([getData(100), getData(200)])
  })
```

async / await

- Syntax introduces to simplify notation of Promises
- Works exactly the same as Promises

Fetch API

- API in JS for the communication using HTTP
- Works with **Promises**
- By default fetch uses HTTP method **GET**

```
const getData = async () => {
  try {
    const response = await fetch('//url');
    const data = await response.json();


  } catch (e) {
    console.error(e.message)
  }
}

getData();
```

Task

15 minutes
materials/TASK_13.md

Javascript

Static Code Analysis

ESLint

- Static code analysis tool for identifying problematic patterns found in JavaScript code
- First release: 2013
- ESLint covers both code quality and coding style issues
- Rules are the core building block of ESLint
- Rule validates if your code meets a certain expectation
- Configuration can be in one of the files:
  - .eslintrc.js
  - .eslintrc.cjs
  - .eslintrc.yaml
  - .eslintrc.yml
  - .eslintrc.json
  - package.json in eslintConfig property

```
{
    "rules": {
        "eqeqeq": "off",
        "curly": "error",
        "quotes": ["error", "double"]
    }
}
```

Javascript

Cookies and storages

Cookies

- Cookies are data, stored in small text files, on your computer
- Cookies can be send over HTTP communication in Headers
- Used in wrong way can be insecure:

Storages in browser

- LocalStorage
- SessionStorage
- FileStorage

Javascript

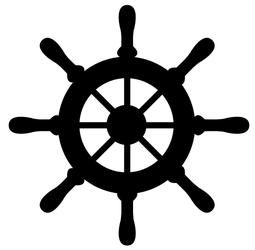
Frameworks

	Angular.js	React.js	Vue
Business Benefits			
Flexibility	Somewhat Flexible	Most Flexible	Somewhat Flexible
Learning Curve	High	Low-Medium	Low-Medium
Framework Size	Heavyweight, suited for dense and complex apps.	Lightweight library, smaller than Angular's	Lightweight, suited for small apps
Community Support	Very good because of Google's support	Quite popular among developers because of extensibility and Facebook's support	Less popular among the top corporations but still quite popular
Top Use Cases	Google, The Guardian, etc.	Facebook, Twitter, Instagram	9GAG, Gitlab
Performance	Uses Real DOM, more efficient	Uses Virtual DOM, Faster than real DOM	Uses Virtual DOM, Faster than real DOM
Background	Typescript based JS, created by Google in 2010.	Founded by Facebook in 2013 to address high traffic on their sites.	A progressive architecture created by ex-Google staff in 2014.
Popularity	Most Popular and Used	Most Popular & Used	Not as popular as these 2 but still widely used.

Dodatkowe pytania?

kahoot.it

XXXXX



Ankieta

tinyurl.com/2p8wpe8w

mail@mateuszjablonski.com
mateuszjablonski.com
linkedin.com/in/mateusz-jablonski/

Dziękuję za uwagę


sages