"Ss. Cyril and Methodius" University in Skopje
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Faculty of computer science and engineering

Object-Oriented Programming (OOP) + ES6

Advanced Web Design

What is Object-Oriented Programming (OOP)?

- A way to organize code around *objects* that represent real-world entities.
- Each object bundles its **data (properties)** and **behavior (methods)** together.
- Helps us write cleaner, reusable, and easier-to-understand code.
- Fundamental principles: **Encapsulation**, **Inheritance**, **Polymorphism**, **Abstraction.**

Why OOP in JavaScript?

- Helps manage larger applications with many related objects.
- Encourages reusability through class-based structures.
- Makes code modular and maintainable.

Classes in Javascript

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

    greet() {
        console.log(`Hello, I'm ${this.name}`);
    }
}

const p1 = new Person("John");
p1.greet();
```

Why OOP in JavaScript?

- Helps manage larger applications with many related objects.
- Encourages reusability through class-based structures.
- Makes code modular and maintainable.

Adding methods

- Methods are functions defined *inside* a class.
- You don’t need the *function* keyword inside a class.
- Methods are shared between all objects of that class.

Adding methods

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

    greet() {
        console.log(`Hello, I'm ${this.name}`);
    }

    introduce() {
        console.log(`My name is ${this.name}, nice to meet you!`);
    }
}

const p2 = new Person("John");
p2.introduce();
```

this keyword

- Refers to the current object created from the class.
- Used to access the object's own properties and methods.
- Becomes very important when using multiple objects.

this keyword

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

    changeName(newName) {
        this.name = newName;
    }

    greet() {
        console.log(`Hi, I'm ${this.name}`);
    }
}

const p3 = new Person("John");
p3.greet();
p3.changeName("Jane");
p3.greet();
```

Class inheritance

- Inheritance lets one class extend another.
- Use the *extends* keyword.
- Use *super()* to call the parent class constructor.

Class inheritance

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

    greet() {
        console.log(`Hello, I'm ${this.name}`);
    }
}

class Teacher extends Person {
    constructor(name, subject) {
        super(name);
        this.subject = subject;
    }

    teach() {
        console.log(`${this.name} is teaching ${this.subject}`);
    }
}

const t1 = new Teacher("Marko", "Advanced Web Design");
t1.greet();
t1.teach();
```

Getters and setters

- *get* - allows access to a property's computed value.
- *set* - controls how a property is changed.

Getters and setters

- *get* - allows access to a property’s computed value.
- *set* - controls how a property is changed.

Getters and setters

```
class Teacher extends Person {
    constructor(name, subject) {
        super(name);
        this._subject = subject;
    }

    get subject() {
        return this._subject.toUpperCase();
    }

    set subject(newSubject) {
        if (newSubject) this._subject = newSubject;
    }
}

const t3 = new Teacher("Marko", "Web Design");
console.log(t3.subject);
t3.subject = "Frontend Development";
console.log(t3.subject)
```

Private fields

- Private fields start with #
- They can only be accessed *inside* the class
- Useful for protecting internal data

Private fields

```
class Person {
    #age = 0; // private

    constructor(name, age) {
        this.name = name;
        this.#age = age;
    }

    getAge() {
        return this.#age;
    }

    birthday() {
        this.#age++;
        console.log(`${this.name} is now ${this.#age} years old.`);
    }
}

const p5 = new Person("Jane", 25);
p5.birthday();
```

toString()

```
class Teacher {
    constructor(name, subject) {
        this.name = name;
        this.subject = subject;
    }

    teach() {
        console.log(`${this.name} is teaching ${this.subject}.`);
    }

    toString() {
        return `Teacher: ${this.name}, Subject: ${this.subject}`;
    }
}

const t1 = new Teacher("Marko", "Advanced Web Design");

console.log(t1.toString());
console.log(`${t1}`);
```

Example 1

- Write a function that fetches users and posts from https://jsonplaceholder.typicode.com/users and https://jsonplaceholder.typicode.com/posts. Create a User class that stores each user’s id, name, username and email, and a Post class that stores userId, title and body with an instance method summary() that returns the first 30 characters of the body followed by … if it is longer. After fetching and building arrays of User and Post objects, filter posts to keep only those whose titles are at least 20 characters long, check whether some posts contain the word “qui” in either the title or body and whether every user has at least one post, map the top authors into readable strings that include the user’s name and post count, and sort those authors by post count in descending order. Print all results in a clean, readable format and handle invalid payloads, empty arrays, and network errors gracefully.

Example 2

- Write an asynchronous JavaScript function that fetches the first 20 characters from https://rickandmortyapi.com/api/character (GET request) and then fetches details for each character’s first episode. Create a Character class that stores id, name, status, species, origin (name only), episodeCount (length of the episode array), and a method *firstEpisodecode()* that returns the fetched episode’s code (e.g., “S01E01”) or "Unknown" if missing. After building the array of Character objects and fetching their first episode codes, filter to keep only Alive characters, extract the unique set of species and print them alphabetically, sort the full list by episodeCount in descending order and show the top five characters by appearances. Check whether some characters have origin that includes “Earth” and finally print everything in a clean, readable format while handling network errors and empty payloads gracefully.