Object-Oriented Programming

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://about.softuni.bg

Have a Question?

Software University

sli.do

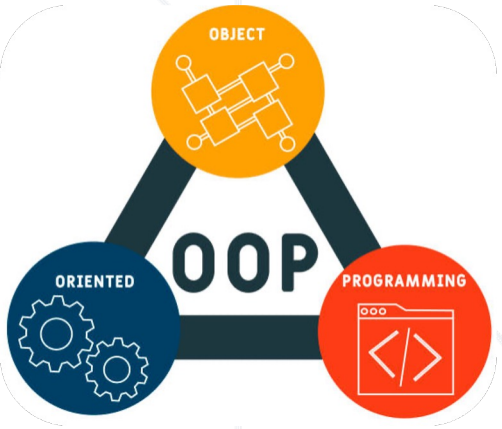
#typescript

Table of Contents

Object-Oriented Programming

Object-Oriented Programming (OOP)

- **A programming paradigm** that uses objects to organize code and structure applications
- **Key concepts**: **classes**, **objects**, **inheritance**, **polymorphism** and **encapsulation**



Benefits of OOP

Software University

- **Modularity**: code is organized into manageable, reusable units (classes and objects)
- **Reusability**: code can be reused across different parts of the application and even in other projects
- **Flexibility and Extensibility**: easily adapt and extend the system through inheritance and polymorphism
- **Simplified Maintenance**: changes and updates are localized to the related class or object, reducing complexity

Core Principles of OOP

Core Principles of OOP

Software University

- **Abstraction**: focus on essential features and hide unnecessary details
- **Encapsulation**: bundle data and behavior within a class, controlling access with access modifiers
- **Inheritance**: create new classes based on existing ones, fostering code reuse and extensibility
- **Polymorphism**: provide a common interface for different data types, allowing flexibility and extensibility

Abstraction

- Presenting a **simple interface** while hiding the **complex implementation**

```
interface Human {
    greet(): string;
}

class Person implements Human {
    greet():string {
      return 'Hello,there!'
    }
}
```

Encapsulation

- Access control through **access modifiers** (**public**, **private**, **protected**)

```
class Person {
    private name: string;
    construction(name: string) {
      this.name = name;
    }
    greet():string {
      return 'Hello, I am ${this.name}'
    }
}
```

Inheritance

- Inheriting **properties** and **methods** from the **base class**

```
class Dog extends Animal {
    constructor() {
      super('Bark');
    }
}
```

```
class Animal {
    sound: string;

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

    makeSound():void {
      console.log(this.sound);
    }
}
```

Polymorphism

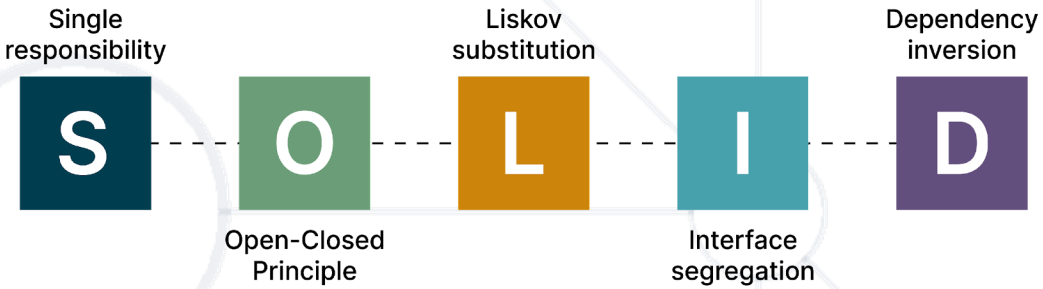
- Achieved through method **overriding** and method **overloading**

```
class Shape {
    draw():void {
      console.log('Drawing a shape.');
    }
}
class Circle extends Shape {
    draw():void {
      console.log('Drawing a circle.');
    }
}
```

SOLID Principles

Acronym for five design principles to make software more **maintainable**, **scalable** and **robust**

- **S**: Single Responsibility Principle
- **O**: Open / Closed Principle
- **L**: Liskov Substitution Principle
- **I**: Interface Segregation Principle
- **D**: Dependency Inversion Principle

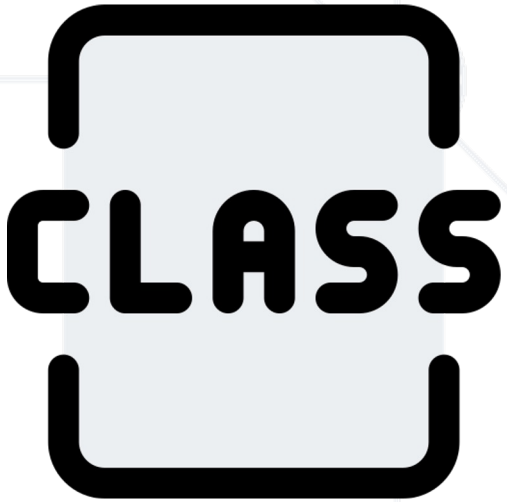


Classes and Objects

Class

Software University

- A **blueprint** for creating objects
- Defines the **properties** and **methods** that objects based on the class will have
- Can have **constructors** for initializing object properties



Overview

Software University

```
class Dog {
    private name: string;
    private age: number;

    constructor(n: string, a: number) {
        this.name = n;
        this.age = a;
    }

    bark() {
        return `${this.name} woofed friendly`;
    }
}

let tommy = new Dog('Tommy', 6);

console.log(tommy);         //Dog { name: 'Tommy', age: 6 }
console.log(tommy.bark()); //Tommy woofed friendly
```

Class initialization

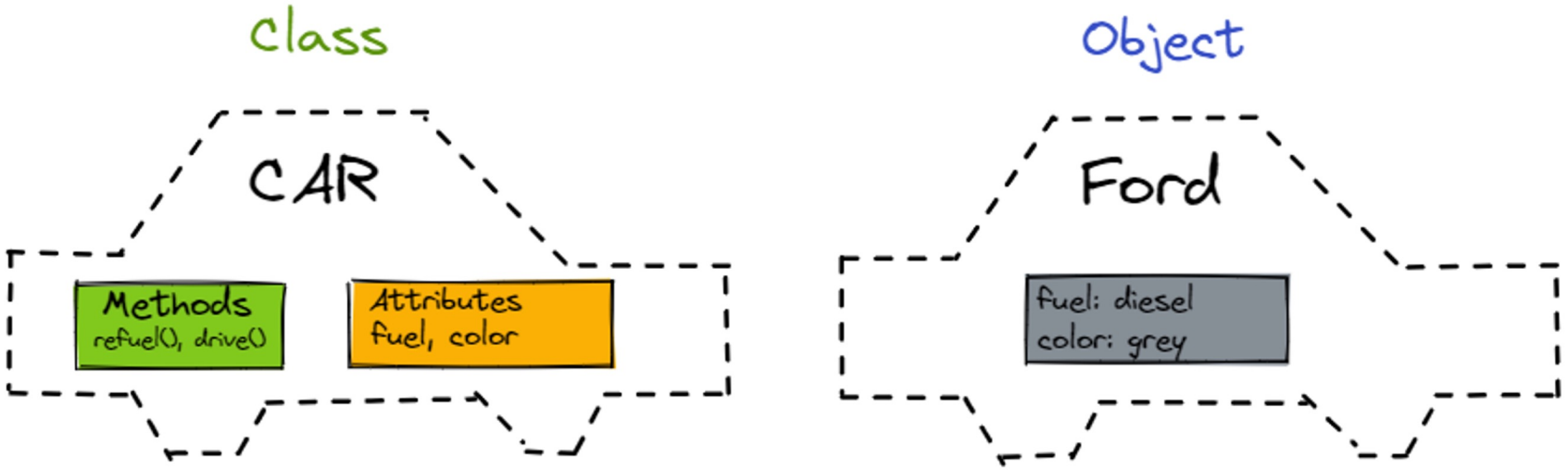
Class properties

Class constructor

Class method

Object

- An **instance of a class**
- Represents a **specific entity** based on the **class's blueprint**
- Has specific **property** values and can call the **class's methods**



Classes vs Objects

Software University

- Class

```
class Person {
    name: string;
    construction(name: string) {
      this.name = name;
    }
    greet():string {
      return 'Hello, I am ${this.name}'
    }
}
```

- Object

```
const person1 = new Person('Alice');
const person2 = new Person('Bob');
```

Members of a Class

Breakdown: Properties

- The **properties** in TypeScript are used to **store data**
  - They are defined **before** the constructor in the **body** of the class
  - The **data is passed** to them **afterwards**

```
class ContactList {
    private name: string;
    private email: string;
    private phone: number;
}
```

Property declarations

Breakdown: Methods

- The **methods** are used to define functionalities
  - Each **class** can have **lots of methods**
  - Generally speaking, each **method** should do **one thing** only

```
class ContactList {
    //property declarations
    //constructor
  call() {
    return 'Calling Mr. ${this.name}'
  }
  showContact() {
    return 'Name: ${this.name} Email: ${this.email} Number: ${this.phone}'
  }
}
```

Methods

Breakdown: Constructor

- The **constructor** is used to give properties **values**
  - Each **class** can have only **one constructor**
  - The constructor creates **new object** with the defined properties

```
class ContactList {
    //property declarations
    constructor(n: string, e: string, p: number) {
        this.name = n;
        this.email = e;
        this.phone = p;
    }
}
```

Constructor

Static Properties

- Defined by keyword **static**
- The **property** belongs to the class itself, so it **cannot be accessed** outside of the class
- We can only access the properties directly **by referencing** the class itself

Example of Static Properties

Software University

```
class Manufacturing {
    public maker: string;
    public model: string;
    public static vehiclesCount = 0;

    constructor(maker: string, model: string, ) {
        this.maker = maker;
        this.model = model;
    }
    createVehicle() {
        Manufacturing.vehiclesCount++;
        return 'Created cars: ${Manufacturing.vehiclesCount} of
        ${this.maker} ${this.model}';
    }
}
```

Accessors

Software University

- In order to use accessors your compiler output should be set to **ES6** or higher
- **Get** and **Set**
  - Get method comes when you want to **access** any class property
  - Set method comes when you want to **change** any class property

Example of Accessors

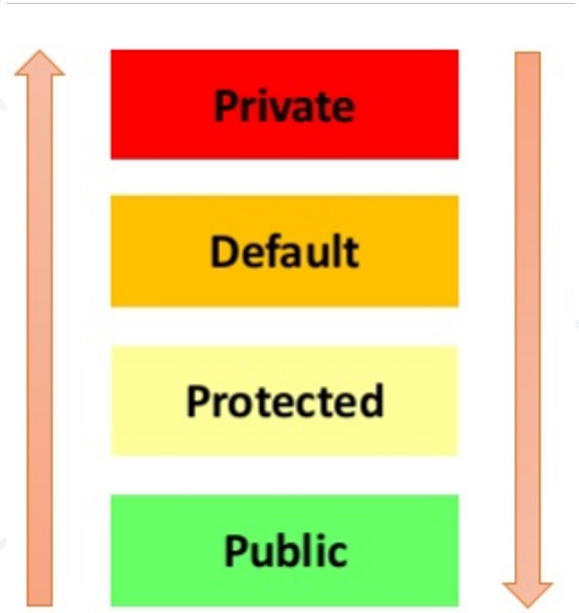
Software University

```
const fullNameMaxLength = 10;

class Employee {
    private _fullName: string;

    get fullName(): string {
        return this._fullName;
    }
    set fullName(newName: string) {
        if (newName && newName.length > fullNameMaxLength) {
            throw new Error("fullName has a max length of " + fullNameMaxLengt
h);
        }

        this._fullName = newName;
    }
}
```

Access Modifiers

Software University

- TypeScript has **access modifiers**
- Used to **define** who can **use** the class elements
- **Types** of access modifiers:
  - **Public**
  - **Private**
  - **Readonly**
  - **Protected**



Public

- By **default** each element is defined **as public**
- Gives **access** to the element
- Not only **properties** may be public, but **constructors** as well

```
class Zoo {
    public type: string;
    public name: string;

    public constructor(t: string, n: string) {
        this.type = t;
        this.name = n;
    }
}
```

Private

- Element marked as **private** cannot be accessed **outside** the declaration

```
class Zoo {
    private type: string;
    private name: string;

    constructor(t: string, n: string) {
        this.type = t;
        this.name = n;
    }
}
let animal = new Zoo('bear', 'Martha');
console.log(animal.name); //Error: name is private.
```

Readonly

- **Readonly** protects the value from being **modified**
- No unexpected data mutation

```
class Zoo {
    readonly name: string;

    constructor(n: string) {
        this.name = n;
    }
}
let animal = new Zoo('Martha');
animal.name = 'Thomas'; //Error: name is read-only.
```

Protected

- Element marked as **protected** can be accessed **only** within the **declaration class** and **the subclasses**

```
class Zoo {
    protected name: string;
    constructor(n: string) { this.name = n; }
}
class Bear extends Zoo {
    private color: string;
    constructor (name, c: string) {
        super(name);
        this.color = c;
    }
}
let martha = new Bear('Martha', 'Brown');
```

Abstract Class

- Defined by keyword **abstract**
- They are **superclasses** but **cannot** be **instantiated directly**
- Methods inside abstract classes and marked as such **do not contain implementations** but **must be** implemented **in derived classes**

Example of Abstract Class

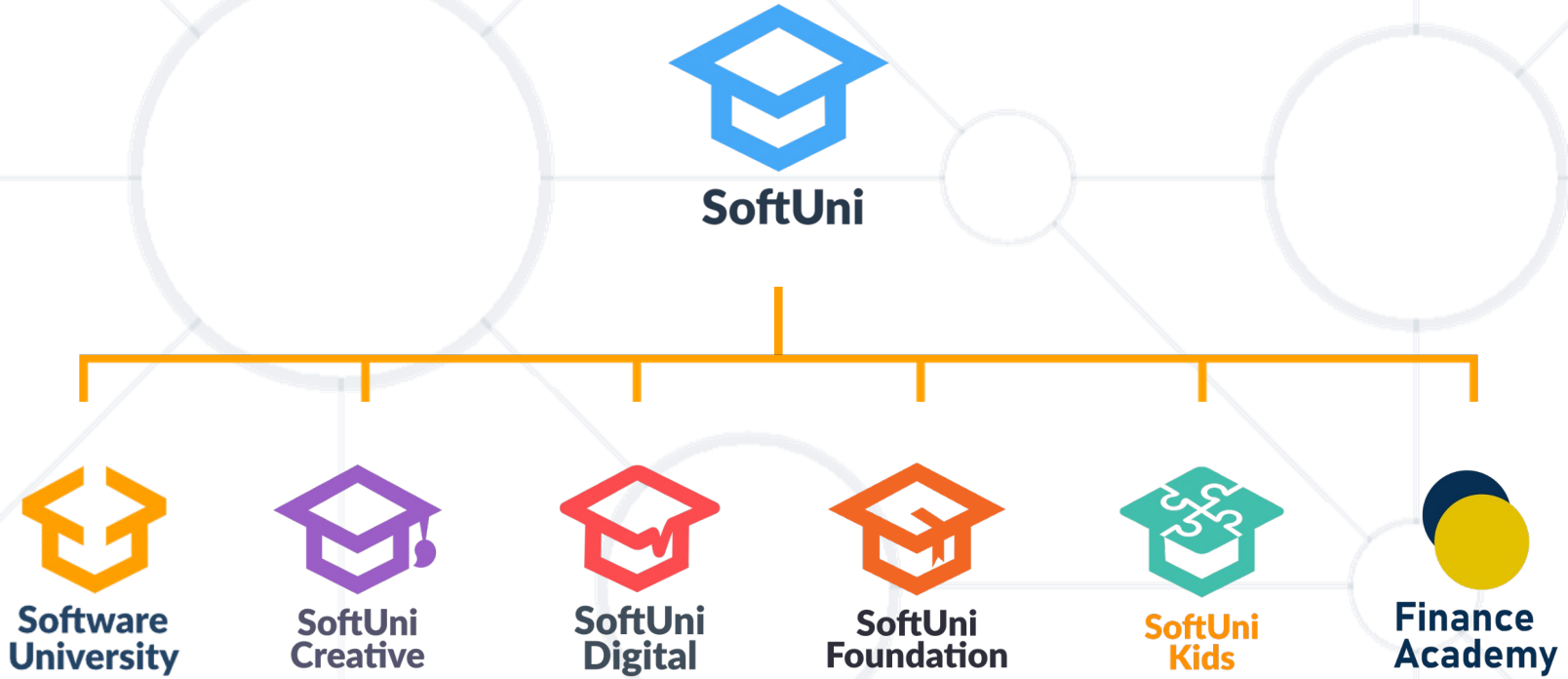
Software University

```
abstract class Department {
    public depName: string;
    constructor(n: string) { this.depName = n; }
    abstract sayHello(): void;
}
class Engineering extends Department {
    public employee: string;
    constructor (depName: string, e:string) {
        super(depName)
        this.employee = e;
    }
    sayHello() {
        return '${this.employee} of ${this.depName} department says hi!';
    }
}
let dep = new Department('Test') //Cannot create instance of abstract class
```

Summary

- Classes in TypeScript consist of
  - Properties
  - Constructor
  - Methods
- You can restrict or allow access to properties by using access modifiers
- Using get and set methods

Questions?



© SoftUni – https://about.softuni.bg. Copyrighted document. Unauthorized copy, reproduction or use is not permitted.

SoftUni Diamond Partners

Software University

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

Software University

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://about.softuni.bg/
- © Software University – https://softuni.bg