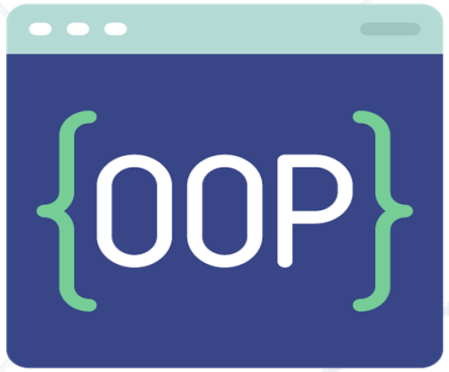
# Object-Oriented Programming




**SoftUni Team**

**Technical Trainers**

Software University

SoftUni

Software University

https://softuni.bg

# Have a Question?

Software University

## sli.do

# #TypeScript

# Table of Contents

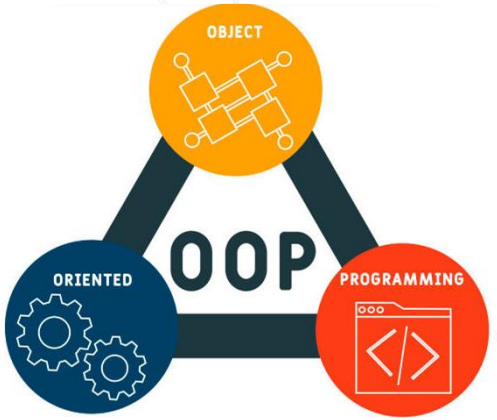



# Object-Oriented Programming

# Object-Oriented Programming (OOP)

Software University

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



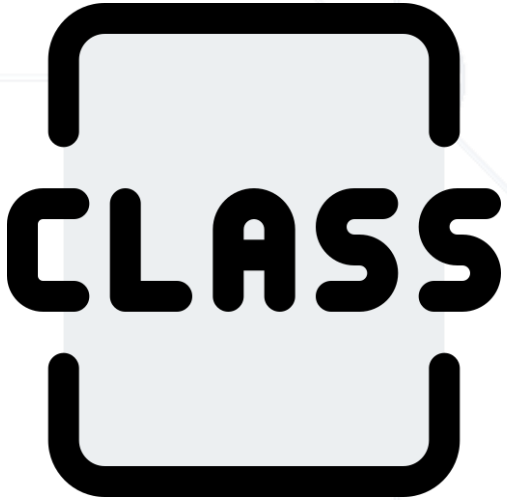
# Benefits of OOP

- **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

# Classes and Objects

# Class

- 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 initialization

Class properties

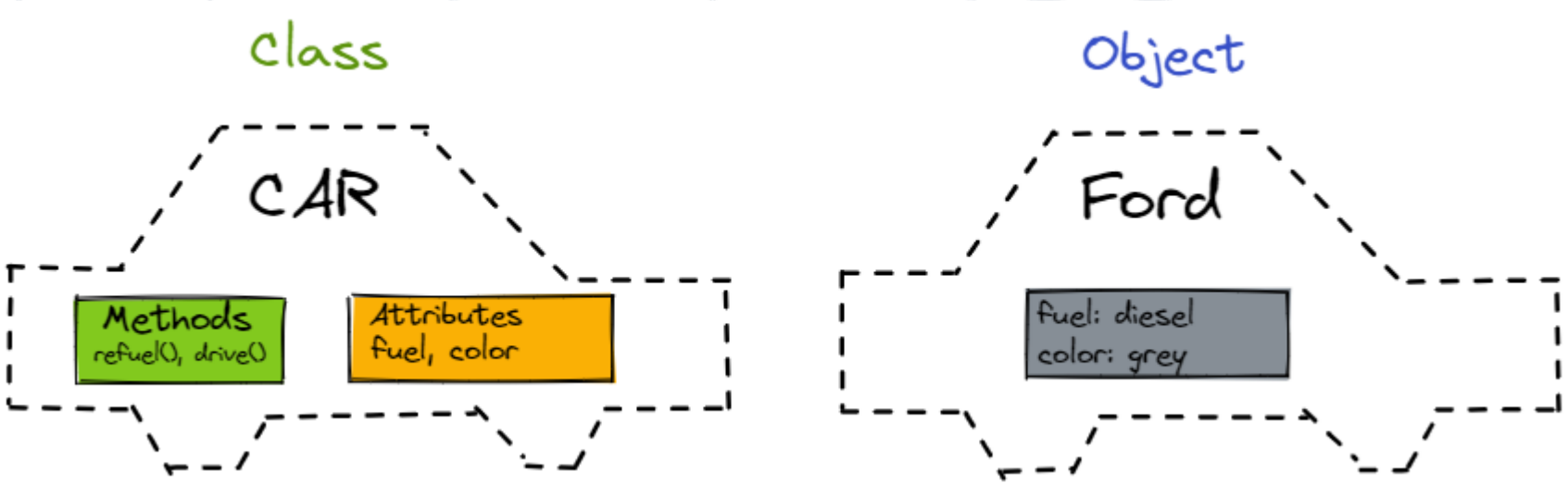
Class constructor

Class method

```
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
```

# 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');
```

# Core Principles of OOP

# Core Principles of OOP

- **Encapsulation**: bundle data and behavior within a class, controlling access with access modifiers and accessors
- **Abstraction**: focus on essential features and hide unnecessary details
- **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

# Encapsulation

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

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

# Abstraction

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

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

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

# Inheritance

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

```
class Animal {
    sound: string;

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

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

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

let dog = new Dog();
dog.makeSound();  // Bark
```

# Polymorphism

- Allows objects to be **presented as parts of their functionality**
- Requires only that the object **structure and types are compatible**

```
type Greeter = { greet(): string; }
class Person {
    constructor(public name: string){}
    greet() { return `${this.name} says hello`};
}

let person: Greeter = new Person('John');
```

# Example: Method Overriding

- **Hides** the **parent method implementation** and can be:
  - Implicit – redeclaring method with **same name**
  - Explicit – using the **override** keyword
    - Can set **"noImplicitOverride": true** in **tsconfig** to have TS allow only explicit overrides

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

implicit override

# Example: Method Overloads

- Method **overloads** determine the **allowed call signatures**
- The **implementation** must be **compatible with all overloads**

```
class Person {
    greet(num: number): void;
    greet(fName: string, lName: string): void;
    greet(a: number | string, b?: string): void {
        console.log(typeof a === 'number'
            ? `Your number: ${a}`
            : `Hello, ${a} ${b}`);
    }
}
let person = new Person();
person.greet('John', 'Doe');     // Hello, John Doe
person.greet(13);                // Your number: 13
person.greet('John');            // Error: no matching signature
```

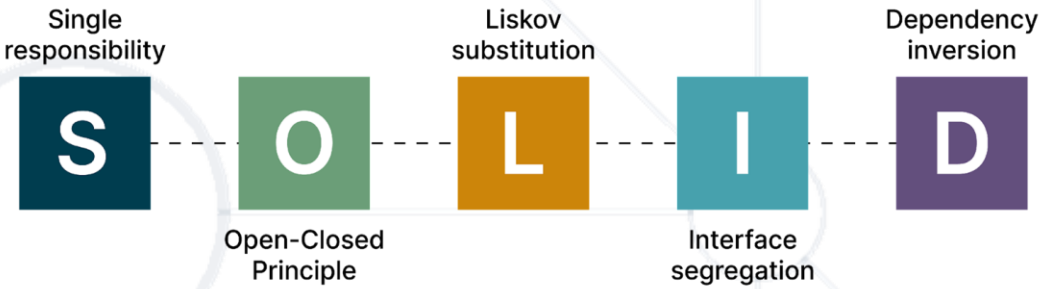
overload

overload

Method implementation

# 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



# 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 only **one thing**

```
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 **values** to the properties
  - Each **class** can have only **one constructor**
  - The constructor creates **new objects** 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

# 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: Accessors

```
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 " + fullNameMaxLength);
        }

        this._fullName = newName;
    }
}
```

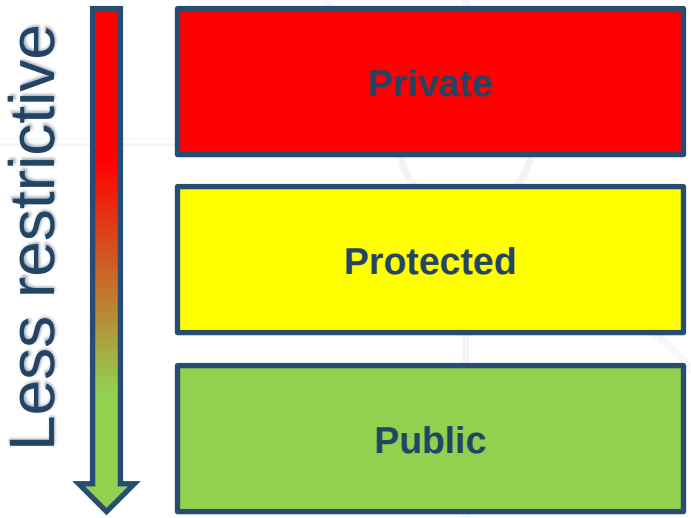
**Definite assignment assertion**
Guarantees to TS that property/variable will be assigned

getter

setter

# Access Modifiers

- TypeScript has **access modifiers**
- Used to **define** who can **use** the class members
- Can be applied to **properties**, **constructors** and **methods**
- Types of modifiers:
  - **Private**
  - **Protected**
  - **Public**



# Public

- By **default** class members are defined **as public**
- Gives **access** to the element

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

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

# Protected

- Members marked as **protected** can be accessed **only** within the **declaration class** and **its derived classes**

```
class Animal {
    constructor(protected _name: string) {  }
}
class Bear extends Animal {
    constructor (name:string) {
        super(name);
    }
    roar(){ console.log(`${this._name} roars.`) };
}
let martha = new Bear('Martha');
martha.roar();    // Martha roars.
```

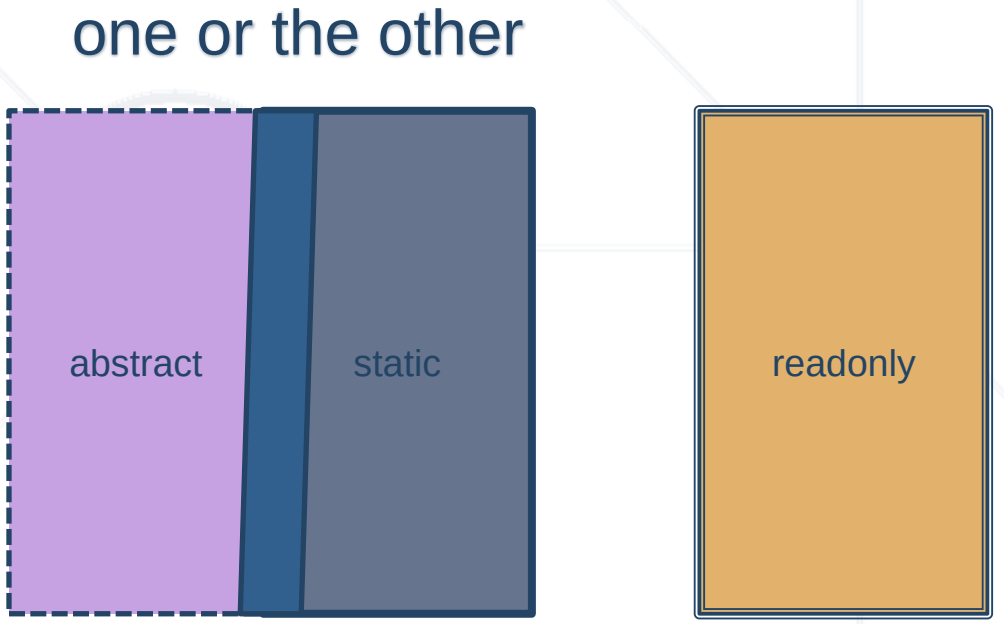
# Private

- Members 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.
```

# Additional Modifiers

- In addition to **access modifiers**, Typescript supports additional modifiers on class members, that can be used in combination with access modifiers
- Used with the **keywords**:
  - **Static**
  - **Abstract**
  - **Readonly**



# Static

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

# Example of Static Properties

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

    constructor(maker: string, model: string,) {
        this.maker = maker;
        this.model = model;
    }

    createVehicle() {
        let calls = ++Manufacturing.vehiclesCount;
        return `createVehicle called: ${calls} times`;
    }
}
```

# Abstract

Software University

- Defined by the keyword **abstract**
- Can be applied to **classes** and to **properties** and **methods** if they are **in an abstract class**
- **Abstract** classes **cannot** be **instantiated directly**
- Abstract properties / methods **must be** initialized / implemented **in a derived classes**
- **Abstract** methods **do not contain implementations**

# 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 {
    constructor(depName: string, public employee: string) {
        super(depName);
    }
    sayHello() {
        return `${this.employee} of ${this.depName} department says hi!`;
    }
}

let dep = new Department('Test') // Cannot create instance of abstract class
```

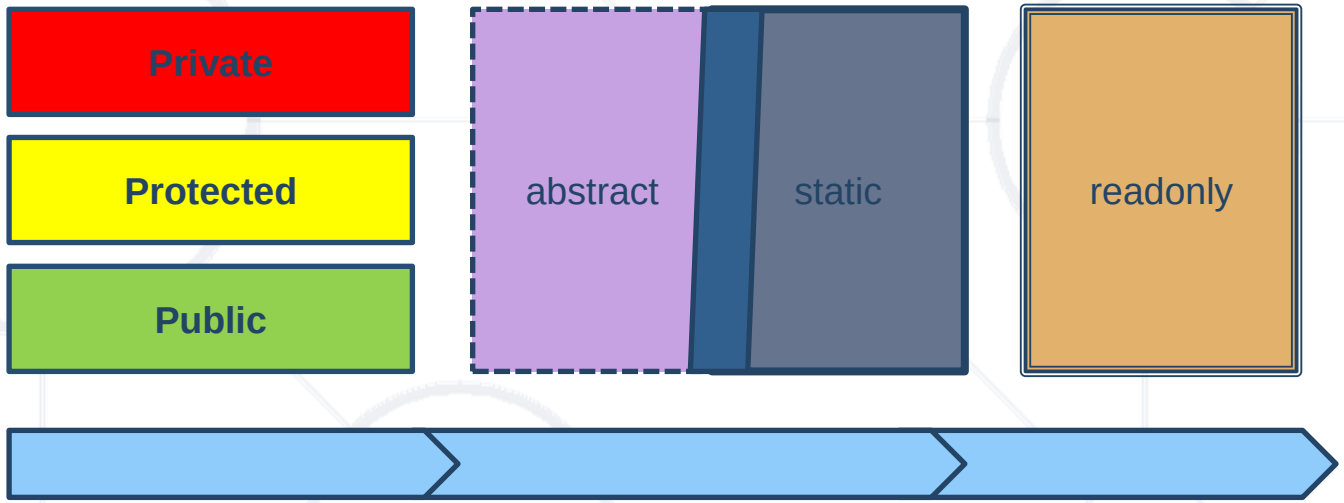
# Readonly

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

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

# Combining Modifiers

- You can use multiple modifiers by chaining them in the following order:
  1. **Access Modifier**
  2. **abstract** or **static**
  3. **readonly**




```
abstract class Machine {
    protected abstract readonly model: string;
    public static readonly machineCount: number;
    static abstract id: string; //Error: has both static and abstract
}
```

# Summary

- In Typescript, we can better conform to the principles of OOP by using:
  - Interfaces
  - Access Modifiers (public, private, protected)
  - Additional Modifiers (static, abstract, readonly)
- Classes can consist of:
  - Properties
  - Constructor
  - Methods
- You can restrict or allow access to properties by using access modifiers
- Using get and set methods

# Questions?

SoftUni

Software University | SoftUni AI | SoftUni Digital | SoftUni Creative | SoftUni Foundation | СофтУни БУДИТЕЛ | Finance Academy

# SoftUni Diamond Partners

Software University

Coca-Cola HBC Bulgaria

Postbank

top EMPLOYER България Bulgaria 2025

FOR A BETTER WORLD OF WORK

SUPER HOSTING .BG

createX

DRAFT KINGS

THE CROWN IS YOURS

INDEAVR

Serving the high achievers

encorp.io

VIVACOM

# 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