TypeScript

Course Overview

SoftUni Team

Technical Trainers

Software University

SoftUni Foundation

Software University

http://softuni.bg

Table of Content

Have a Question?

SoftUni Foundation

sli.do

#typescript

SoftUni Diamond Partners

SoftUni Foundation

XSsoftware

SBTech
we know sports

telenor

SoftwareGroup
doing it right

NETPEAK

SmartIT

Postbank
Решения за твоето утре

INDEAVR
Serving the high achievers

INFRAGISTICS®

STEMO®
Computer Systems & Software

SUPERHOSTING.BG

SoftUni Organizational Partners

SoftUni Foundation

TypeScript

Course Objectives & Program

TypeScript

SoftUni Foundation

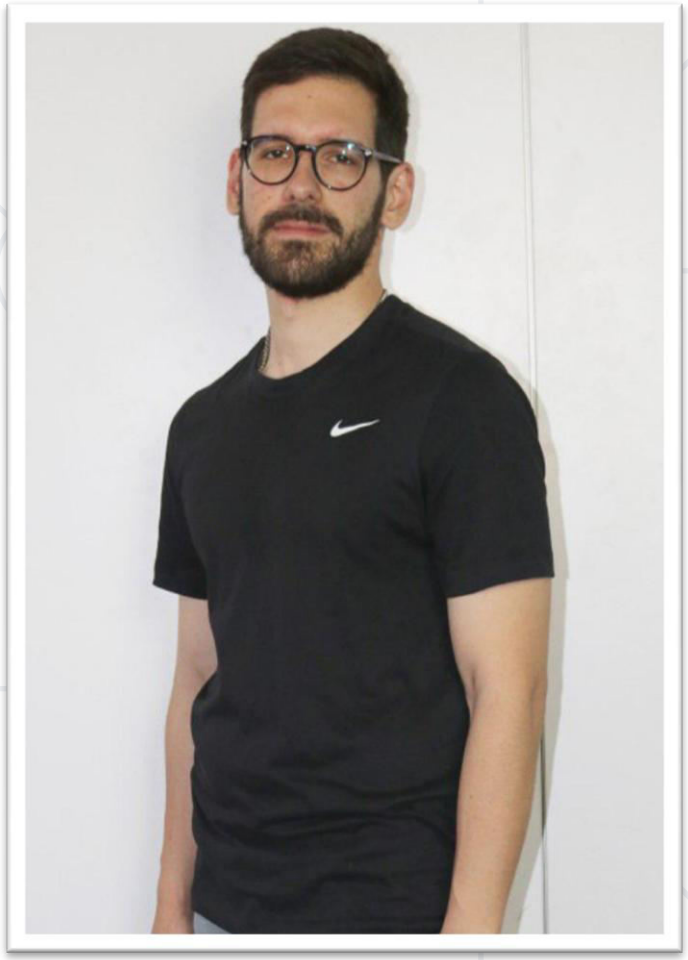
1. **Type System**
2. **Generics**
3. **Object Oriented Programming** (Class Based Objects)
4. **Namespaces and modules**
5. **Decorators**
6. **Workshop**

Trainers and Team

Ilia Idakiev

SoftUni Foundation

- Software developer-entrepreneur
- Google Developer Expert
- Co-organizer of Angular Sofia and SofiaJS / BeerJS
- Lecturer at Sofia University (FMI)
  - "Advanced JavaScript" course



Course Objectives

Course Details and Schedule

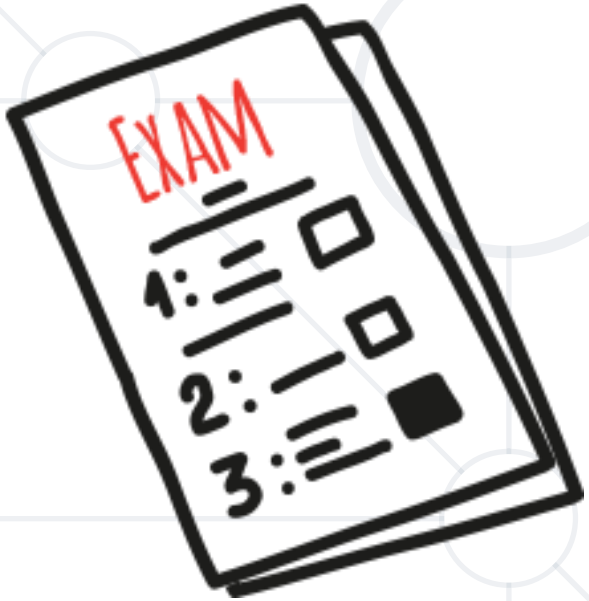
Targets of the Course

SoftUni Foundation

- Introduction to the TypeScript programming language
- Covers: **Type System**, **Generics**, **Object Oriented Programming**, **Namespaces and modules** and **Decorators**

Practical Exam

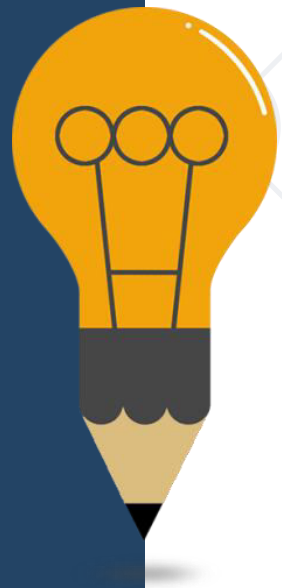
- Structure: **3 problems** for **4 hours**
- Problems description:
  - **OOP**
  - **Generics**
  - **Decorators**

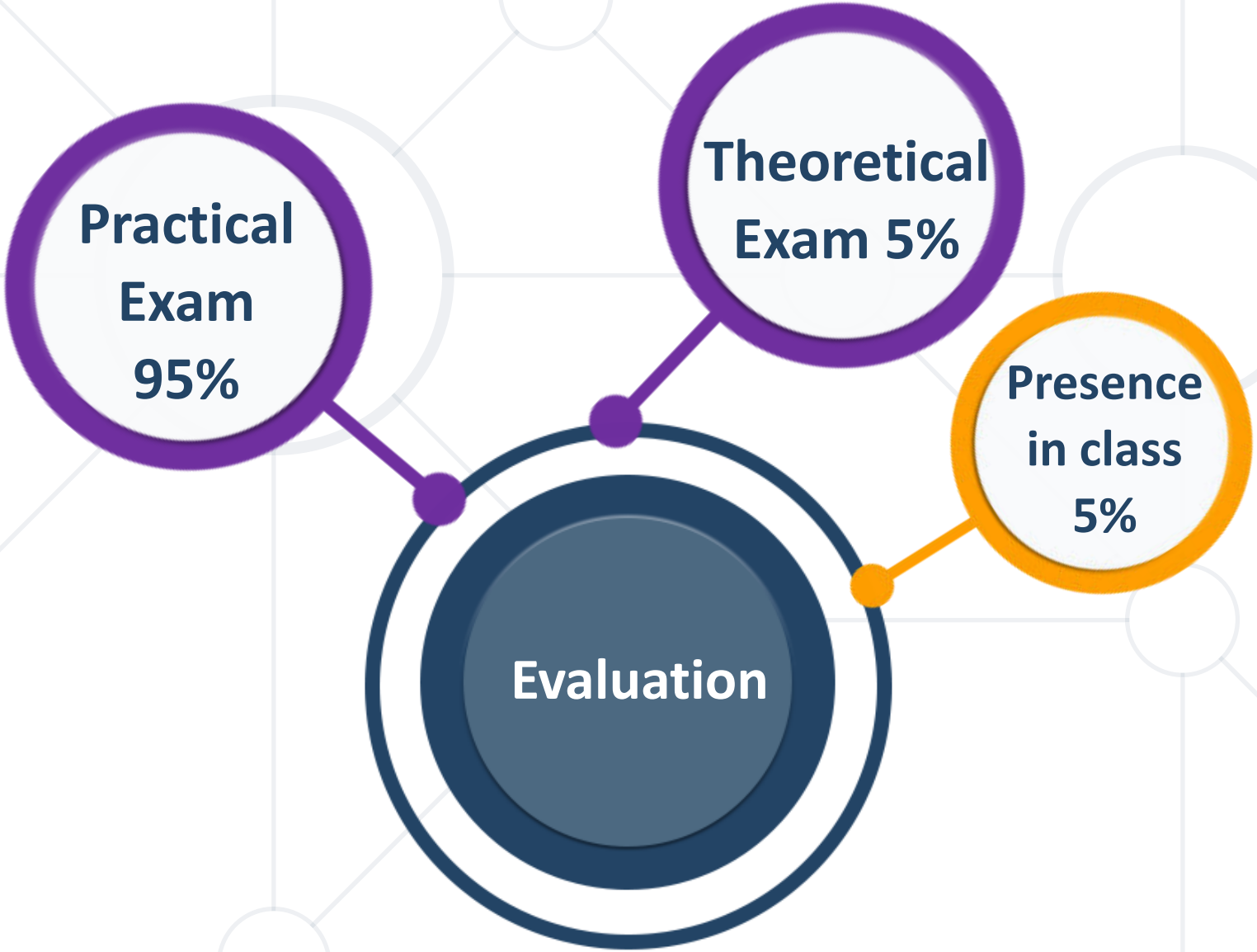


Theoretical Exam

- 20 questions for 30 minutes
  - Multiple-choice with 1 correct answer
  - English
- Automated quiz system
- Available online the day before the practical exam
  - You can submit your answers just one time

Scoring System for the Course





Course Organization

Evaluation Criteria

- **Mandatory**
  - **Practical exam** - 95%
  - **Theoretical exam** - 5%
- Bonuses:
  - Presence in class – 5% bonus (onsite students only)

Questions?

SoftUni

Software University

SoftUni Svetlina

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

Trainings @ Software University (SoftUni)

SoftUni Foundation

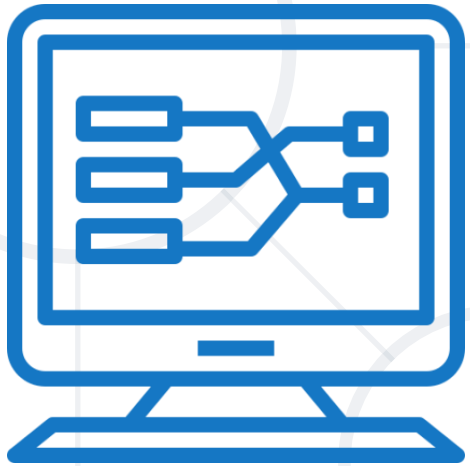
- Software University – High-Quality Education and Employment Opportunities
  - softuni.bg
- Software University Foundation
  - http://softuni.foundation/
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License

- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license

TypeScript Type System

Basic and Advanced data types



SoftUni Team

Technical Trainers

Software University

SoftUni Foundation

Software University

http://softuni.bg

Table of Content

Have a Question?

SoftUni Foundation

sli.do

#typescript

Introduction to TypeScript

TypeScript and JavaScript

- TypeScript is a **superset** of JavaScript
- Created by Microsoft Corporation
- All JavaScript code is **valid** in TypeScript too
- TypeScript **compiles to** JavaScript

TypeScript vs JavaScript

SoftUni Foundation

- TypeScript

```
class Person {
  private firstName: string;
  constructor(f: string) {
    this.firstName = f;
  }
  greeting() {
    return `${this.firstName} `
  }
}
```

- JavaScript

```
"use strict";
class Person {
  constructor(f) {
    this.firstName = f;
  }
  greeting() {
    return  `${this.firstName} `;
  }
}
```

Install TypeScript to Visual Studio Code

SoftUni Foundation

- Install **TypeScript** with **npm**

```
npm install -g typescript (latest stable build)
```

- Test if **TypeScript** is **installed properly**

```
tsc --version //Should return a message 'Version
3.x.x'.
```

- Create the **tsconfig.json** file

```
tsc --init – This command will create a new
tsconfig.json file
```

tsconfig.json

SoftUni Foundation

- In the tsconfig.json file, please **remove the comments** from the following:

```
{
  "compilerOptions" : {
      "target": "esnext",   //ECMAScript target version
      "module": "esnext",   //module code generation
      "sourceMap": true,    //Generates corresponding .map file
      "strict": true,       //strict type-checking options
      "outDir": "out",      //redirect output to the directory.
    }
}
```

Basic Data Types

SoftUni Foundation

- **String** - used to represent **textual** data

```
let str: string = `hello`;
str = 'singleQuotes' ; //valid
str = "doubleQuotes" ; //valid
str = 11; //invalid
```

- **Number** - a numeric data type

```
let decimal: number = 11; //valid
let hex: number = 7E3; //valid
let binary: number = 11111100011 //valid
let float: number = 3.14 //valid
decimal = `hello`; //invalid
```

Basic Data Types

- **Boolean** - only **true** and **false** values
  - Functions or expressions that return true or false values may also be assigned to Boolean data type

```
let isBool: boolean = true;
isBool = 5 < 2; //valid
let numbers = [1, 2, 3, 4];
isBool = numbers.includes(100)
//valid
isBool = 11; //invalid
```

Basic Data Types

SoftUni Foundation

- **Array** - use any valid data type (String, Boolean, Number) and postfix []

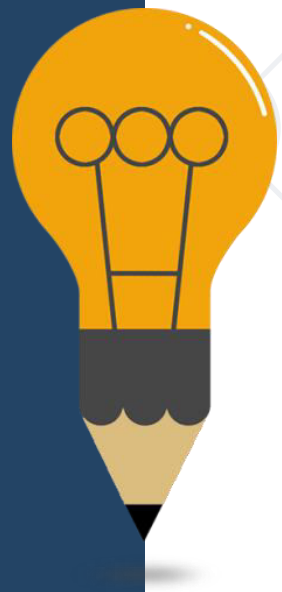
```
let arrayOfStr: string[];
arrayOfStr.push(`Hello`); //valid
arrayOfStr.push(`World`); //valid
arrayOfStr.push(11); //invalid
```

- **Tuple** - array with fixed number of elements whose types are known

```
let tuple:[string, number];
tuple = [`Hello`, 11]; //valid
tuple = [11, `Hello`]; //invalid
```

Basic Data Types

- **Enum** - Gives sets of numeric values more readable names
  - By default each enum starts at 0



```
enum DaysOfTheWeek {
    Monday,  //0
    Tuesday, //1
    …
};
let day: DaysOfTheWeek;
day = DaysOfTheWeek.Monday;
console.log(day); //0
if (day === DaysOfTheWeek.Monday) {
    console.log(`I hope you all had a great weekend!`);
} //It will print the message
```

Basic Data Types

SoftUni Foundation

- **Any** - takes any and all values. It`s a way to escape the strong types

```
let example: any = `hello`;
example = true; //valid
example = 11 ; //valid
```

- **Void** - mainly used in functions that return no value

```
function greet(message: string): void {
    console.log(message);
}
```

Optional data types

- The **optional** data types are marked with **?**
  - Required parameters **cannot** follow optional ones

```
function optionalParams(name: string, mail?: string) {
      //some logic
} //valid

function optionalParams(name?: string, mail: string) {
      //some logic
} //invalid
```

Return data types

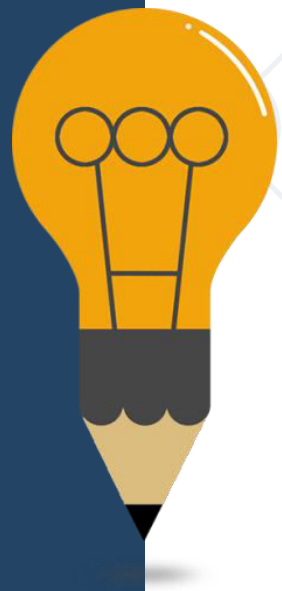
- The **return data types** are marked with **:** after the braces in function declaration
  - The **return value type** should match the **return type**

```
function greet (name: string): string {
    return name;
}

console.log(greet('Hello'));
```

Advanced Data Types

SoftUni Foundation

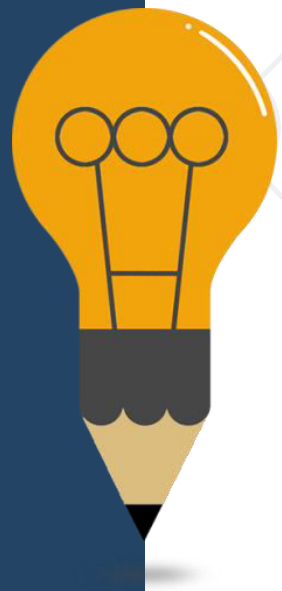
- **Union type** - combine multiple types in one type



```
function greet(message: string | string[]) {
    if (typeof message === "string") {
        return message;
    }
    return message.join(' ');
}
let greeting = 'Hello world';
let greetingArray = ['Dear', 'Sir/Madam'];

console.log(greet(greetingArray)); //Dear Sir/Madam
```

Advanced Data Types

- **Intersection types** - combine multiple types in one type



```
interface Person { fullName: string | string[]; }
interface Contact { email: string; }
function showContact(contactPerson: Person & Contact) {
    return contactPerson;
}
let contactPerson: Person & Contact = {
    fullName: 'Svetoslav Dimitrov',
    email: 'test@test.com'
}
console.log(showContact(contactPerson));
```

Problem: Mathematical operations

- Write a **TypeScript function** that makes **simple mathematical operations** over an array of numbers
  - It will receive two parameters: **array of numbers and operation**
    - The operations might be: **addition**, **multiplication** **or finding the largest number**

Solution: Mathematical operations

SoftUni Foundation

```
function solve(arrOfNums: number[], operation: string): number {
    let result: number = 0;
    const addition = () => result = arrOfNums.reduce((a, b) => a + b, 0);
    const multiplication = () => result = arrOfNums.reduce((a, b) =
> a * b, 1);
    const largestNumber = () => result = Math.max(...arrOfNums);
    const actions = {
        'Addition': addition,
        'Multiplication': multiplication,
        'Largest number': largestNumber
    }
    actions[operation]();
    return result;
}
```

Live Exercises

Summary

- TypeScript presents strong typing to your JavaScript code
  - `let`, `const` and `var` are used to declare variables
  - There are basic (Number, String, Boolean, etc.) and more advanced data types like union or intersection
- Functions can:
  - Take optional and required parameters and return result

SoftUni Diamond Partners

SoftUni Foundation

XSsoftware

SoftwareGroup
doing it right

NETPEAK

SmartIT

INDEAVR
Serving the high achievers

INFRAGISTICS®

STEMO®
Computer Systems & Software

SUPERHOSTING.BG

SoftUni Organizational Partners

SoftUni Foundation

Questions?

SoftUni

Software University

SoftUni Svetlina

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

License

- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license

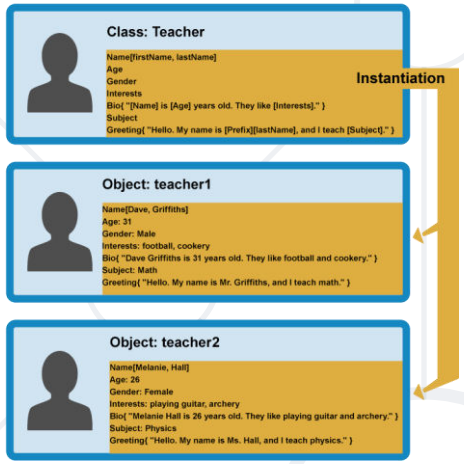
Trainings @ Software University (SoftUni)

SoftUni Foundation

- Software University - High-Quality Education and Employment Opportunities
  - softuni.bg
- Software University Foundation
  - http://softuni.foundation/
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

TypeScript OOP

Classes, inheritance, abstraction



SoftUni Team

Technical Trainers

Software University

SoftUni Foundation

Software University

http://softuni.bg

Table of Content

Have a Question?

SoftUni Foundation

sli.do

#tbd

TypeScript Classes

Definition

- **Classes** in TypeScript can contain:
  - **Data**, defined by its **properties**
  - Who can use the data - **access modifiers**
  - Some **actions** by using **methods**
- **One class** may have **many heirs** – **inheritance**
- **Abstract classes** cannot be instantiated directly. They are the **ancestor** class which starts the **inheritance chain**

Overview

```
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

Breakdown: Properties

SoftUni Foundation

- 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: 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;
    }
}
```

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

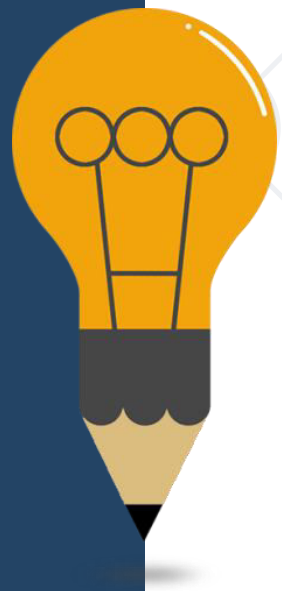
```
class ContactList {
    //property declarations
    //constructor
  call() {
    return `Calling Mr. ${this.name}`
  }
  showContact() {
    return `Name: ${this.name} Email: ${this.email} Number: ${this.phone}`
  }
}
```

Methods

Access modifiers

SoftUni Foundation

- Unlike JavaScript, 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

SoftUni Foundation

- 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');
```

Inheritance

SoftUni Foundation

- Used to **extend** existing classes to **new ones**
- To do so we use the **extend** key word
- The "basic" class is often called **superclass** and the extended - **subclasses**
- To inherit the superclass's **constructor** to the subclass we use the keyword **super**

Example of inheritance

```
class Company {
    public name: string;
    constructor(n: string) { this.name = n; }
}
class Department extends Company {
    private depName: string;
    constructor(name, dN) {
        super(name);
        this.depName = dN;
    }
}
class Employee extends Department {
     //Some code logic
}
```

Class Department inherits the Company class

Class Employee inherits the Department class

Inheritance

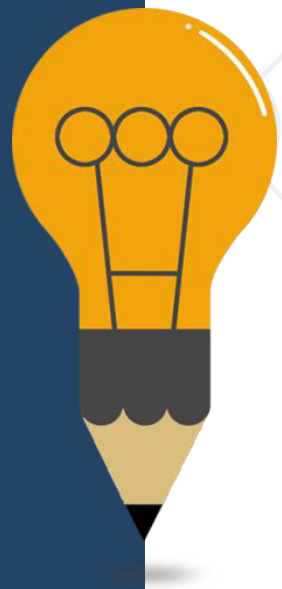
SoftUni Foundation

- Not only **properties** might be inherited, but **methods** as well.

```
class Vehicle {
    public color: string;
    constructor(c: string) { this.color = c; }
    showColor() { return `The car is ${this.color}`; }
}
class PassengerCar extends Vehicle {
    public model: string;
    constructor(color, m: string) {
        super(color);
        this.model = m;
    }
    details() {
        return `${super.showColor()} and is ${this.model}`
    }
}
```

Accessors

- 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



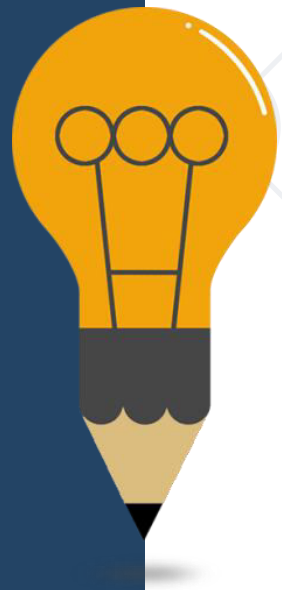
Example of accessors

SoftUni Foundation

```
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;
    }
}
```

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**



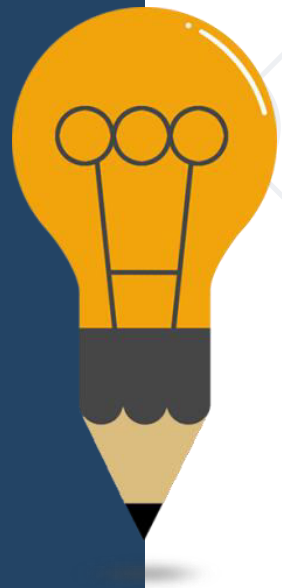
Example of abstract class

SoftUni Foundation

```
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
```

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 abstract class

SoftUni Foundation

```
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}`;
    }
}
```

Live Exercises

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

SoftUni Diamond Partners

SoftUni Foundation

XSsoftware

SoftwareGroup
doing it right

NETPEAK

SmartIT

INDEAVR
Serving the high achievers

INFRAGISTICS®

STEMO®
Computer Systems & Software

SUPERHOSTING.BG

SoftUni Organizational Partners

SoftUni Foundation

Questions?

SoftUni

Software University

SoftUni Svetlina

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

License

- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license

Trainings @ Software University (SoftUni)

SoftUni Foundation

- Software University - High-Quality Education and Employment Opportunities
  - softuni.bg
- Software University Foundation
  - http://softuni.foundation/
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

TypeScript Generics

Interfaces, Generic Functions and Classes

SoftUni Team

Technical Trainers

Software University

SoftUni Foundation

Software University

http://softuni.bg

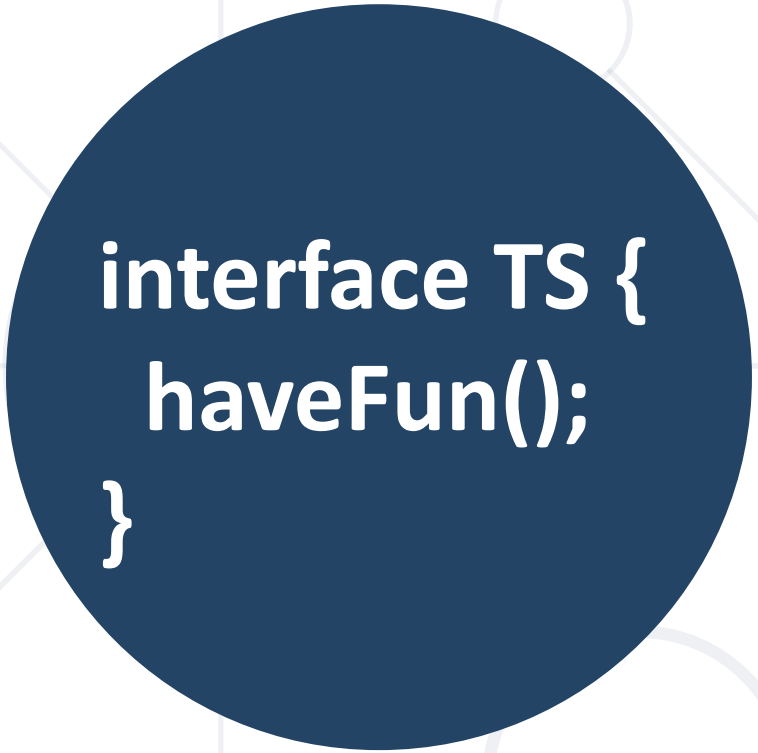
Table of Content

Have a Question?

SoftUni Foundation

sli.do

#tbd



Interfaces

Definition

- Defined by using keyword **interface**
- Often called **duck typing** or **structural typing**
- We can define **properties**, **methods** and **events** also called **members** of the interface
- The interface **contains** only **the declaration** of its members
- Helps to **standardize the structure** of the deriving classes

Example: Basic interface

SoftUni Foundation

```
interface Person {
    fullName: string,
    email: string,
}

let thomas: Person = {
    fullName: 'Thomas Doe',
    email: 'thomas@test.test',
}

console.log(thomas.fullName) //Thomas Doe
```

Interface declaration

Declare a variable with the interface as type in order to follow the structure

Describe function types

- Interfaces in TypeScript can also describe function types
  - They are constructed in the following way:

```
interface Name {
   (paramOne: type, paramTwo: type,...paramN: type): type;
}
```

    - Where in the parantheses we put the **parameters** we want to pass to the function with their **types**, splitted by comma.
    - On the right side is the **return type** of the function

Example: Describe function types

SoftUni Foundation

```
interface Calculator {
    (numOne: number, numTwo: number, operation: string): number;
}

let calc: Calculator = function (a: number, b: number, operation:
 string): number {
    let result: number = 0;
    const addition = () =>  result = a + b; ;
    const parser = {
        'addition': addition,
    }
    parser[operation]();
    return result;
}
```

Implemented by classes

- Interfaces can be implemented by classes using the keyword **implement**
- A class that implements an interface **must have** all the properties defined in the interface
  - Describes the **public** side of the class

```
interface Person { ... }

class Teacher implements Person { ... }
```

Example: Implemented by class

SoftUni Foundation

```
interface ClockLayout {
    hour: number;
    minute: number;
    showTime(h: number, m: number): string;
}

class Clock implements ClockLayout {
    public hour;
    public minute;
    constructor(h: number, m: number) {
        this.hour = h;
        this.minute = m;
    }
    showTime() {
        return `Current time: ${this.hour}:${this.minute}`;
    }
}
```

Extending interfaces

SoftUni Foundation

- Interfaces can extend **classes** and other **interfaces**
  - Extending **classes**
    - The extended interface **inherits** all of the members of the class including **private** and **protected** members
    - The interface **does not inherit** the **implementations** of the members (e.g. method implementations)
  - Extending other **interfaces**
    - Creates a **combination** of all interfaces

Example: Extending interfaces

SoftUni Foundation

```
class Computer {
    public RAM;
    constructor(r: number) { this.RAM = r; }
    showParams(): string { return `${this.RAM}`; }
}
interface Parts extends Computer {
    CPU: string;
    showParts(): string;
}
class PC extends Computer implements Parts {
    public keyboard;
    public CPU;
    constructor(RAM: number, CPU:string) { super(RAM); this.CPU = CPU; }
    showParts() {
        return `${this.RAM} ${this.CPU}`;
    }
}
```

Generics

Definition

- Used to build **reusable** software components
- The components will work with **multitude** of type instead of a single type
- Defined by type variable - **<LETTER>**
- Follow the **DRY** (**D**on`t **R**epeat **Y**ourself) principle
- Allow us to **abstract** the type
- Generics can be applied to **functions**, **classes** and **interfaces**

Example: Generic vs Non-generic

- Generic

```
function echo<T>(arg: T): T {
    console.log(typeof arg);
    //It will print number and
string when the function is
invoked
    return arg;
}
echo(11111);
echo('Hello');
```

- Non-generic

```
function echo(arg: number): number {
    return arg;
}
```

```
function echo(arg: string): string {
    return arg;
}
```

Generic functions

SoftUni Foundation

- Generic functions allow us to work with user input with **unknown** data type
- It is a way of telling the function that whatever **type** is **passed** to it the **same** type shall be **returned**
- Put some **constraints** to user input
- We can put **more than one** type variable in the generic function

Example: Generic functions

SoftUni Foundation

```
const takeLast = <T>(array: T[]) => {
    return array.pop();
}
const sample = takeLast(['Hello', 'World', 'TypeScript']);
const secondSample = takeLast([1, 2, 3, 4]);
console.log(sample, secondSample); //TypeScript, 4
```

```
const makeTuple = <T, V>(a: T, b: V) => {
    return [a, b];
}
const firstTuple = makeTuple(1, 2);
const secondTuple = makeTuple('a', 'b');
console.log(firstTuple, secondTuple); //[1, 2], [a, b]
```

Generic interfaces

SoftUni Foundation

- Using **generic interfaces** we can define **generic functions** too

```
interface GenericConstructor<T, V> {
    (arg: T, param: V): [T, V];
}

const generatedFn: GenericConstructor<string, string> = <T, V>(arg: T, param: V)
 => {
       return [arg, param];
}

const sample = generatedFn('Hello', 'World');
console.log(sample); // [Hello, World]
```

Generic classes

SoftUni Foundation

- Generics can be used on:
  - The **properties** of the class
  - The **methods** of the class
- To define generic class we put **<LETTER> after the name of the class**
- We can use **multiple** type variables
- Generic classes can implement generic interfaces

Example: Generic class using single parameter


```
class Collection<T> {
    public data: T[];
    constructor(...elements: T[]) { this.data = elements; }

    addElement(el: T) { this.data.push(el); }

    removElement(el: T) {
        let index = this.data.indexOf(el);
        if (index > -1) {
            this.data.splice(index, 1);
        }
    }

    reverseElements() { return this.data.reverse(); }

    showElements() { return this.data; }
}
```

Example: Generic class using multiple parameters

SoftUni Foundation

```
class UserInput<F, S> {
    public first: F;
    public second: S;
    constructor (f: F, s: S) {
        this.first = f;
        this.second = s;
    }

    showBoth() {
        return `First: ${this.first}, second: ${this.second}`;
    }
}

let sample = new UserInput('Ten', 10);
let test = new UserInput(1, true);
console.log(sample.showBoth()); // First: Ten, second: 10
console.log(test.showBoth()); // First: 1, second: true
```

Example: Generic class implements interface

SoftUni Foundation

```
interface ShowComponents<T, V> {
    print(key: T, value: V): string;
}

class Components<T, V> implements ShowComponents<T, V> {
    public key: T;
    public value: V;
    constructor(k: T, v: V) {
        this.key = k;
        this.value = v;
    }
    print(){
        return `Key: ${this.key} and value: ${this.value}`;
    }
}
let test: ShowComponents<string, string> = new Components('New', 'Test');
console.log(test.print('Test', 'Hello')); // Key: New and value: Test
```

Generic type constraints

- In TypeScript we can make sure that sudden type variable **has** at least **some information** containing in it
- Constraints are enforced by **extends** keyword

```
function fullName<T extends { fName: string, lName: string }>(obj: T) {

      return `The full name is ${obj.fName} ${obj.lName}.`;
}

let output = fullName({fName: 'Svetoslv', lName: 'Dimitrov'});

console.log(output);// The full name is Svetoslav Dimitrov
```

Live Exercises

Summary

- Generics are use to:
  - Abstract data types
  - Build reusable components
- We can use them in:
  - Functions
  - Classes - their properties and methods
  - Interfaces

SoftUni Diamond Partners

SoftUni Foundation

XSsoftware

SBTech
we know sports

telenor

SoftwareGroup
doing it right

NETPEAK

SmartIT

Postbank
Решения за твоето утре

INDEAVR
Serving the high achievers

INFRAGISTICS®

STEMO®
Computer Systems & Software

SUPERHOSTING.BG

SoftUni Organizational Partners

SoftUni Foundation

Questions?

SoftUni

Software University

SoftUni Svetlina

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

License

- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license

Trainings @ Software University (SoftUni)

SoftUni Foundation

- Software University - High-Quality Education and Employment Opportunities
  - softuni.bg
- Software University Foundation
  - http://softuni.foundation/
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

Namespaces and modules

SoftUni Team

Technical Trainers

Software University

SoftUni Foundation

Software University

http://softuni.bg

Table of Content

Have a Question?

SoftUni Foundation

sli.do

#Typescript

Namespaces

Definition

- Namespaces are used to **logically grouped** functionalities
- Previously referred as **internal modules** in TypeScript
- Defined with **namespace** keyword
- Namespaces may include **functions**, **classes**, **interfaces** and **variables**

Access

- The elements of the namespace that must be accessed from the outside must be marked with **export** keyword
- In order to access namespaces from different files we must use the reference syntax /// **<reference path = "file.ts" />**

Example: Namespace

SoftUni Foundation

```
namespace printMessages {
    export function messenger(message: string | string[]) {
        return `${message}`;
    }
    export interface meetPerson {
        meetPerson(): string
    }
}
console.log(printMessages.messenger('Hello')); //Hello
```

namespace declaration

export to use the interface outside

Multiple Files Namespaces

SoftUni Foundation

- In order to access namespaces from different files we must use the reference syntax **/// <reference path = “file.ts” />**
- In order to compile the file we must
  - Compile the ts file - **tsc fileName.ts**
  - Use the outFile - **tsc --outFile fileName.js fileName.ts**
  - Compile the js file - **node fileName**

Example: Multiple File Namespace

SoftUni Foundation

```
/// <reference path = 'messages.ts'/>
class Person implements printMessages.meetPerson {
    public fullName: string;
    public greeting: string;

    constructor(fn: string, g: string) {
        this.fullName = fn;
        this.greeting = printMessages.messenger(g);
    }

    meetPerson(): string {
        return `Hello, ${this.fullName}, ${this.greeting}`;
    }
}
let p1 = new Person('Ivan Ivanov', 'pleasure to meet you!');
console.log(p1.meetPerson());
```

Aliases

SoftUni Foundation

- Used to simplify the work with namespaces
- Used with **import** keyword
- Often used as nested namespaces

```
namespace Shops {
    export namespace TechStores {
        export class PCStore {}
        export class AudioStore {}
        export class TVStore {}
    }
}
--Name of file - app.ts
import stores = Shops.TechStores;
let pcStore = new stores.PCStore();
```

Modules

Definition

- Modules are executed in their **own scope**, not the global
- A **set of functions** to be included in applications
- Resolve name collisions
- In order to be accessed from the outside they need to be marked with **export** keyword

Access

- To consume a function, class, interface or variable exported from another module we must use an **import** form
  - **import** { name } from "./location" - import specific element
  - **import** * as variable from "./location"; - imports the entire module in single variable

Export Statements

- There are three ways to use **export** statements:
  - A:
  - B:
  - C:
  - D:

```
export function numberValidation(num: number): number {…}
```

```
export { numberValidation };
```

```
export { numberValidation as isValidNum }; //isValidNum is alias
```

```
export default function stringValidations(string: string): string {…}
```

- In cases **A** and **B** there is **no difference** rather than syntax
- There might be only one **export default** in a file

Example: Export and Import Statements

SoftUni Foundation

```
--exports
export default function checkInput<T>(information: T): T {
    if (information) { return information; }
    else { throw new Error('The information passed is not valid') }
}
export function stringValidations(string: string): string {
    if (string.length > 0 && string.length <= 20) { return string;
}
    else { throw new Error('String is not valid'); }
}
export function numberValidation(num: number): number {
    if (num > 0 && num <= 999) { return num; }
    else { throw new Error('Number is not valid'); }
}
export { numberValidation as isValidNum };
```

Import Statements and File Compilation

SoftUni Foundation

```
--Imports
import * as validations from './validations'; //validations is
alias
import checkInput from "./validations";
import { isValidNum } from "./validations";

// Some code logic
```

- In order to compile the file we must
  - Compile the ts file - **tsc fileName.ts**
  - Use the outFile - **tsc --module commonjs fileName.ts**
  - Compile the js file - **node fileName**

Problem: Modules

- You are given a task to calculate the **area and perimeter** of a **Square**, **Circle** and **Rectangle**. You should split the respective classes in a separate files and use modules afterwards
  - Use **interfaces**, in a new file, to define the functions for calculating the area and the perimeter
  - In the main file make a **Calculator** class that makes the calculations

Solution: Modules

SoftUni Foundation

```
import * as make from "./interfacesCalc"
import * as shapes from "./shapes";
class Calculator //TODO: extends and implements {
    area(): number {
        return (this.height * this.base) / 2
    }
    perimeter(): number {
        return this.base * 3
    }
}
let calc = new Calculator(4, 4);
console.log(calc.area());
console.log(calc.perimeter());
```

Problem: Namespaces

- You are given a task to display the attendance to a business meeting. You should organize your program in different files using namespaces
  - Use **interface**, in a new file, to define the function for displaying the attendance
  - Use a **function** to invite the attendees. Note that the function takes two parameters -> full name of the employee and the position and returns them in an object
  - In the main file make a **Meeting** class that displays the attendees who went to the meeting

Solution: Namespaces

SoftUni Foundation

```
/// <reference path = "attendees.ts" />
const a1 = attendance.createAttendee('Ivan Ivanov', 'R&D');
//invite two more people
class Meeting implements layout.showAttendance {
    public att;
    constructor(a: any) {
        this.att = a;
    }
    showAttendance(): string {
        let output = '';
        //implement the logic that prints the attendees
        return output;
    }
}
```

Live Exercises

Summary

- Namespaces are logically grouped functionalities
- Modules are a **set of functions** to be included in applications
- Modules do not pollute the global scope

SoftUni Diamond Partners

SoftUni Foundation

XSsoftware

SoftwareGroup
doing it right

NETPEAK

SmartIT

INDEAVR
Serving the high achievers

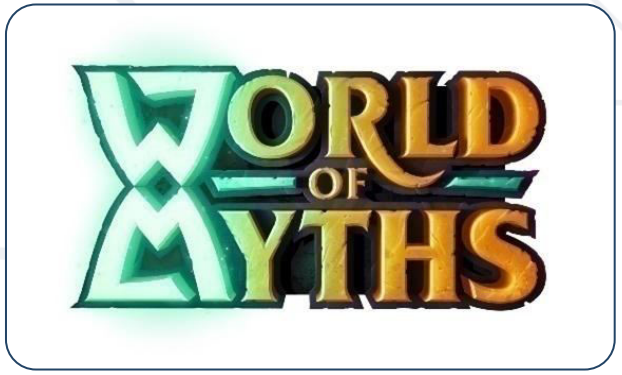
INFRAGISTICS®

STEMO®
Computer Systems & Software

SUPERHOSTING.BG

SoftUni Organizational Partners

SoftUni Foundation



Questions?

SoftUni

Software University | SoftUni Svetlina | SoftUni Creative | SoftUni Digital | SoftUni Foundation | SoftUni Kids

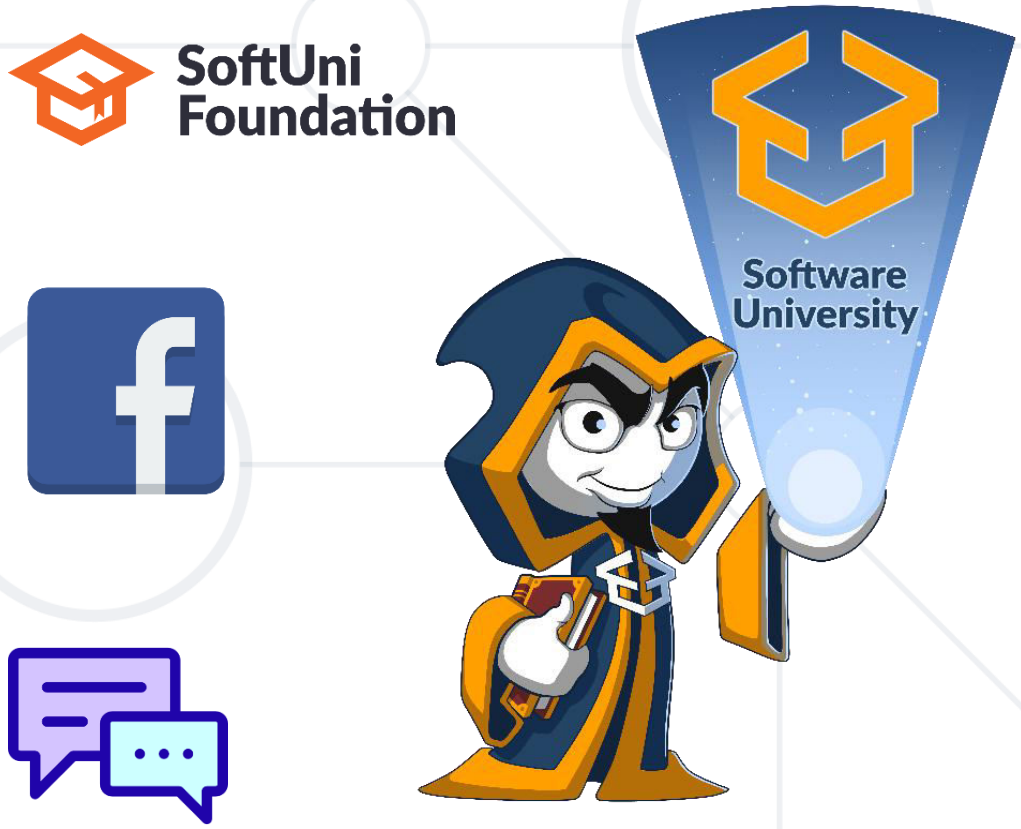
https://softuni.bg/trainings/2696/typescript-advanced-december-2019

License

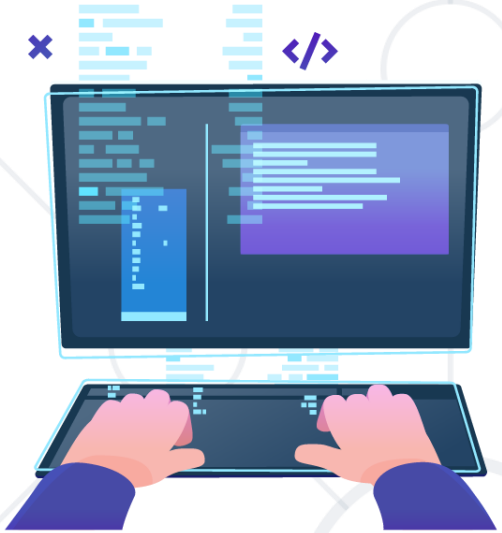
- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license

Trainings @ Software University (SoftUni)

- Software University - High-Quality Education and Employment Opportunities
  - softuni.bg
- Software University Foundation
  - http://softuni.foundation/
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg



Decorators



SoftUni Team

Technical Trainers

Software University

http://softuni.bg

Table of Content

Have a Question?

SoftUni Foundation

sli.do

#Typescript

Decorators Overview

Definition

- Used in frameworks like **Angular**, **MobX** and others
- They are used to extend a functionality or add meta-data
- Use the form **@example** where **example** must evaluate to function that will be called at runtime

```
function example(target) {
  //some code logic
}
```

Decorate

- We can decorate **five** different **things**:
  - Class definitions, properties, methods, accessors, parameters
- The function that we implement is **dependent** on the **thing** we are **decorating**
- The **arguments** required to **decorate a class** are different to the **arguments** required **to decorate a method**

Enable decorators

- In the **tsconfig.json** file:

```
"target": "ES5",
"experimentalDecorators": true
```

- In the **command line**:

```
tsc --target ES5 --experimentalDecorators
```

- Note: Decorators are still stage 2 proposal for JavaScript and only experimental in TypeScript

Decorator evaluation

- There is well **defined order** to how decorators are applied:
  - **Parameter Decorators**, followed by **Method**, **Accessor**, or **Property Decorators** are applied for each **instance member**
  - **Parameter Decorators**, followed by **Method**, **Accessor**, or **Property Decorators** are applied for each **static member**
  - **Parameter Decorators** are applied for the **constructor**
  - **Class Decorators** are applied for the **class**

Class Decorators

Definition

- **Class Decorator** is **added just before** the class declaration
- The Class Decorator is applied to the **constructor** of the class
- Used to **observe**, **modify** or **replace** a class definition
- If the Class Decorator **returns a value**, it will **replace** the class declaration with the **provided constructor function**

Example: Class Decorator

```
function Frozen(construtor: Function) {
    Object.freeze(construtor);
    Object.freeze(construtor.prototype);
}

@Frozen class Person {
    constructor(private name: string) { }
}
```

@Frozen is a class decorator

Decorators and inheritance

- Subclasses do not inherit the decorations of the super class
- Every subclass needs to be decorated on its own

```
@ClassDecorator class Person {
    constructor(public name: string) { }
    @enumerable(false) greet() {
        return `Hello ${this.name}`
    }
}
class Teacher extends Person {
    constructor(private subject: string, name: string) {
        super(name);
    }
    introduce() {
      //some code logic
}
```

The Teacher class does not inherit the decorator

Decorator Factories

Definition

- **Function** that **returns** the **decorator** function **itself**
- Gives the flexibility to pass **custom data** when needed
- Mainly used in **method** and **property decoration**

```
function enumerable(value: boolean) {
    return function (target: Object, propertyKey:
string, descriptor: PropertyDescriptor) {
        descriptor.enumerable = value;
    };
}
```

Multiple Decorators

- Decorators are **composable**
- We can chain **multiple decorators** for each class declaration, method, property, accessor or parameter
- In those cases the decorators are applied from **top to bottom**

```
@Frozen
@Configurable
@OtherDecorator
class Person {
    constructor(private name: string) { }
}
```

Method Decorators

Definition

- The decorator function takes **three** arguments:
  - **target** - the parent **class**
  - **key** - the **name** of the function
  - **descriptor** - the actual **function** itself

```
function enumerable(value: boolean) {
    return function (target: Object,
                     key: string,
                     descriptor: PropertyDescriptor) {
        descriptor.enumerable = value;
    };
}
```

Example: Method Decorator

SoftUni Foundation

```
function Confirmable(message: string) {
    return function (target: Object,
     key: string,
     descriptor: PropertyDescriptor) {
        const original = descriptor.value;
        descriptor.value = function (...args: any[]) {
            const allow = confirm(message);
            if (allow) {
                const result = original.apply(this, args);
                return result;
            } else { return null; }
        };
        return descriptor;
    };
}
```

Accessor Decorators

Definition

- TypeScript **does not allow** to decorate **both** the **getter** and the **setter**
- The **Property Descriptor** combines **both get** and **set** not each declaration separately
- Takes the following three arguments:
  - Either the **constructor function of the class** for a static member, or the **prototype of the class** for an instance member
  - The **name** of the member
  - The **Property Descriptor** for the member

Example: Accessor Decorator


```
class Point {
    private _x: number;
    private _y: number;
    constructor(x: number, y: number) {
        this._x = x;
        this._y = y;
    }

    @configurable(false)
    get x() { return this._x; }

    @configurable(false)
    get y() { return this._y; }
}
```

Property Decorators

Definition

- Property decorator can only be used to **observe** that a property of a specific name has been declared for a class
- Takes the following **two** arguments:
  - Either the **constructor function** of the class for a static member, or the **prototype** of the class for an instance member
  - The **name** of the member

Example: Accessor Decorator


```
class Greeter {
    @format("Hello, %s")
    greeting: string;

    constructor(message: string) {
        this.greeting = message;
    }

    greet() {
      let formatString = getFormat(this, "greeting");
      return formatString
            .replace("%s",   this.greeting);
    }
}
```

Summary

- Decorators are basically **functions**
- Add **additional functionalities** to a class or class members
- We can decorate **class declaration**, **methods**, **accessors**, **properties** and **parameters**
- To decorate different classes or class members the decorator functions takes **different arguments**

Questions?

SoftUni

Software University

SoftUni Svetlina

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

SoftUni Diamond Partners

SoftUni Foundation

XSsoftware

SoftwareGroup
doing it right

NETPEAK

SmartIT

INDEAVR
Serving the high achievers

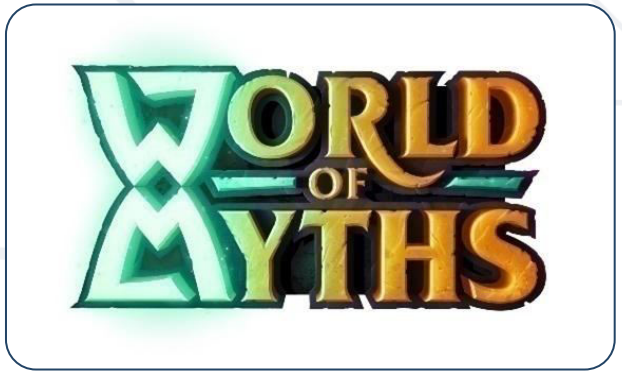
INFRAGISTICS®

STEMO®
Computer Systems & Software

SUPERHOSTING.BG

SoftUni Organizational Partners

SoftUni Foundation



License

SoftUni Foundation

- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license

Trainings @ Software University (SoftUni)

- Software University - High-Quality Education and Employment Opportunities
  - softuni.bg
- Software University Foundation
  - http://softuni.foundation/
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg