Decorators in TypeScript

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

http://softuni.bg

Have a Question?

Software University

sli.do

#TypeScript

Table of Contents

Introduction to Decorators

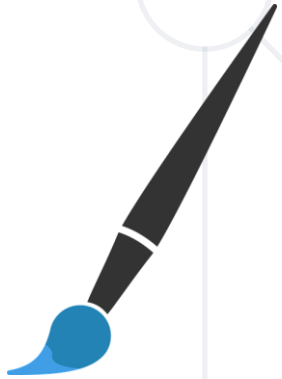
Definition

- 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:

```
{
    "compilerOptions": {
     "experimentalDecorators": true,
     "emitDecoratorMetadata": true
    }
}
```

```
tsc --experimentalDecorators --emitDecoratorMetadata
```

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**

Syntax and Basic Usage

Syntax and Basic Usage

```
function classDecorator(constructor: Function) {
    console.log("Class decorator called.");
}

@classDecorator
class ExampleClass {}
```

```
function methodDecorator() {
    console.log("Method decorated.");
}
class ExampleClass {
      @methodDecorator
      exampleMethod(){}
}
```

Decorators and Inheritance

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

```
function classDecorator(constructor: Function) {
    console.log(`Called on ${constructor.name}`);
}

@classDecorator
class ExampleClass {}
// Called on ExampleClass

class DerivedClass extends ExampleClass {}
// No output, since decorators are not inherited
```

Types of Decorators

Types

- Class Decorators
- Method Decorators
- Accessor Decorators
- Property Decorators
- Parameter Decorators
- Decorator Factories

Class Decorators

Definition

- **Class Decorator** is **added just before** the class declaration
- The Class Decorator receives the **constructor** of the class as a parameter
- 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

Example: Overwriting Class Definition

```
class Person { constructor(public name:string){ } }

@overwrite class Student extends Person {
    constructor(name: string, public age: number) { super(name);  }
}

export function overwrite(constructor: Function) {
    return (function () { return { test: 20}} ) as any;
}

let student = new Student('George', 30);
console.log(student);                                  // {test: 20}
console.log(student instanceof Person);                // false
console.log(student instanceof Student);               // false
```

Example: Extending Class Definition

```
@addTitle class Person { constructor(public name:string){ } }

//TS Error: Decorator function return type is not assignable to typeof Other
@addTitle class Other { constructor(public lastName:string){ } }

function addTitle(constructor: Function) {
    return class extends (constructor as { new(...args: any[]): Person }) {
        constructor(...args: any[]) {
          super('Sir/Madam ' + args[0], ...args.shift());
        }
    }
}

let person = new Person('George')
console.log(person.name)                    // Sir/Madam George
```

The Person class constructor type

Example: Generic Class Decorators

```
@addTitleAbstract abstract class Test {}
@addTitleGeneric class Person extends Test { constructor(public name: string) {super();}}
@addTitleGeneric class Other { constructor(public lastName: string) {  } }

function addTitleGeneric<T extends (new (...args: any[])=> {})>(constructor: T) {
    return class extends constructor {
        constructor(...args: any[]) { super('Sir/Madam ' + args[0], ...args.shift()); }
    }
}

function addTitleAbstract<T extends (abstract new (...args: any[])=>{})>(constructor:T) {
    abstract class Anonymous extends constructor { v = 20; }; return Anonymous;
}

interface Test { v: number};
let person = new Person('George');
console.log(person.name);                    // Sir/Madam George
console.log(person.v)                        // 20
```

Constructor type

Abstract constructor type

Need to extend the Test class interface to let TS know about the new property

Method Decorators

Definition

- The method decorator function takes **three** arguments:
  - **target** - the parent **class**
  - **key** - the **name** of the function
  - **descriptor** - the **PropertyDescriptor** of the method
    - **descriptor.value** - the method itself

```
function disableEnumerable (
        target: Object,
        key: string,
        descriptor: PropertyDescriptor) {
            descriptor.enumerable = false;
};
```

Example: Method Decorator

```
class Num {
    constructor(private _number: number) { }
    @add10 getNumber() { return this._number; }
}

function add10(target: Object, key: string, descriptor: PropertyDescriptor) {
    const original = descriptor.value;
    descriptor.value = function (...args: any[]) {
        let result = original.apply(this, args);
        return result += 10;;
    };
    return descriptor;
};


let num = new Num(20);
console.log(num.getNumber());                    // 30
```

Accessor Decorators

Definition

- TypeScript **does not allow** decorators on **both** the **getter** and the **setter**
- The **Property Descriptor** combines **both get** and **set** not each declaration separately
- Takes the following **three** arguments:
  - **target**
    - The **constructor function of the class** for a static members
    - The **prototype of the class** for instance members
  - **key** - The **name** of the member
  - **descriptor** - The **Property Descriptor** for the member

Example: Accessor Decorator

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

function double(target: any, key: string, descriptor: PropertyDescriptor) {
    let originalSet = descriptor.set;
    descriptor.set = function(val: any)  {
        originalSet?.call(this, val * 2);
    }
};

let p = new Point(20, 20);
p.x = 2; p.y = 3;
console.log(p);                                  // Point {_x: 4, _y: 6}
```

Example: Static Accessor Decorator

```
class Circle {
    constructor(private _radius: number) { }

    @format static get PI(){ return 3.1415 }
}

function format(target: any, key: string, descriptor: PropertyDescriptor) {
    let original = descriptor.get;
    descriptor.get = function()  {
        let result = original?.call(this);
        return `PI is ${result.toFixed(2)}`;
    }
};

let rect = new Circle(5);
console.log(Circle.PI);                                    // PI is 3.14
```

Property Decorators

Definition

- Property decorator can only be used to **observe** that a property with a specific name has been declared
- **Cannot be used to modify the value of a property**
- The **return value** of the decorator is also **ignored**
- Takes the following **two** arguments:
  - **target**
    - The **constructor function** for static members
    - The **prototype** of the class for instance members
  - **key** - The **name** of the member

Example: Property Decorator

```
class Greeter {
    @log
    private _message: string = 'Hello';
}

function log(target: object, key: string) {
    console.log(`Property '${key}' was declared.`);
}

let greeter = new Greeter();
```

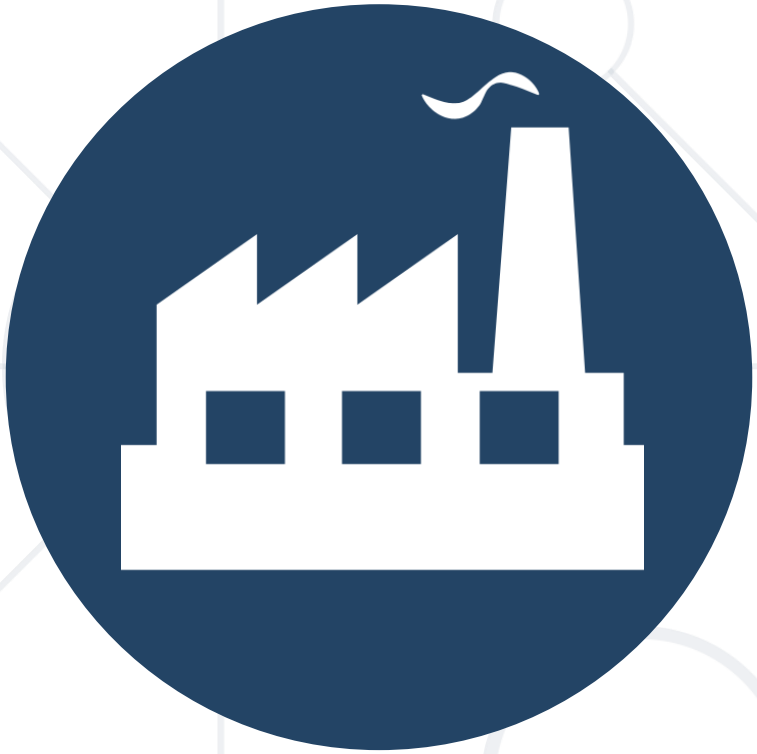
Parameter Decorators

Definition

- Parameter decorators can only be used to **observe** that a parameter has been declared on a method
- **Cannot be used to modify the value of the parameter**
- The **return value** of the decorator is also **ignored**
- Takes the following **three** arguments:
  - **target**
    - The **constructor function** for static members
    - The **prototype** of the class for instance members
  - **key** - The **name** of the parameter
  - **index** - The index of the parameter in the arguments list

Example: Parameter Decorator

```
class Greeter {
    public greet(@log message: string){
        return message;
    }
}

function log(target: object, key: string, index: number) {
    console.log(`Parameter '${key}' was declared.`);
}

let greeter = new Greeter();
```



Decorator Factories

Definition

- **Function** that **returns** a **decorator function**
- Gives the flexibility to pass **custom data** to the decorator function when needed

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

Decorator Factory

Decorator function

Example: Decorator Factory

```
class Name {
    constructor(private _name: string) { }
    @format('Hello, my name is %s')
    getName() { return this._name; }
}

function format(stringFormat: string) {
    return function (target: Object, key: string, descriptor: PropertyDescriptor) {
        const original = descriptor.value;
        descriptor.value = function (...args: any[]) {
            let value = original.call(this, ...args);
            return  stringFormat.replace('%s', value);
        };
        return descriptor;
    }
}

let name = new Name('Peter');
console.log(name.getName());                    // Hello, my name is Peter
```

Multiple Decorators

- We can chain **multiple decorators** for each class declaration, method, property, accessor or parameter
- If multiple decorators / decorator factories exist:
  1. First all factories are called **in-order**:
     log1* -> *log2
  2. Then all the decorators are applied using **composition**:
     ***log1*(*log2*(*constructor*))**

```
@log1()
@log2()
class Person { constructor(private _age:number = 5) { } }
```

Example: Multiple Decorator Factories

```
function log1() {
    console.log("log1 factory evaluated");
    return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
        console.log("log1 decorator executed");
    };
}

function log2() {
    console.log("log2 factory evaluated ");
    return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
        console.log("log2 decorator executed ");
    };
}

class ExampleClass {
    @log1()  @log2() method() { }
}
```

Complex Composition

- Decorators are commonly used to **create wrapper functions** around their targets in order to:
  - **Modify input values** – in setters or methods
    - Composing decorators that wrap a setter will lead to an **extra composition of the wrapper functions** that will **invert the execution order** of the decorators again
  - **Modify output values** – in getters or methods
    - Composing multiple decorators that wrap getters will keep the original decorator composition execution order

Example: Complex Composition Setter

```
class Num {
    constructor(private _number: number) { }
    // (x => x - 2) -> (x => x * 2) -> (x => x + 5) -> original setter
    @modifyInput((x => x - 2)) @modifyInput((x => x * 2)) @modifyInput((x => x + 5))
    setNumber(val: number) { this._number = val; }
    getNumber() { return this._number; }
}

function modifyInput(mathFunc: (val: number) => number) {
    return function (target: Object, key: string, descriptor: PropertyDescriptor) {
        const original = descriptor.value;
        descriptor.value = function (...args: any[]) {
            return original.call(this, mathFunc(args[0]));
        };
    }
}

let a = new Num(5);
a.setNumber(5); console.log(a);                    // {_number: 11} => ((((5)-2)*2)+5)
```

Example: Complex Composition Getter

```
class Num {
    constructor(private _number: number) { }
    setNumber(val: number) { this._number = val; }
    // original getter -> (x => x + 5) -> (x => x * 2) -> (x => x - 2)
    @modifyOutput((x => x - 2)) @modifyOutput((x => x * 2)) @modifyOutput((x => x + 5))
    getNumber() { return this._number; }
}

function modifyOutput(mathFunc: (val: number) => number) {
    return function (target: Object, key: string, descriptor: PropertyDescriptor) {
        const original = descriptor.value;
        descriptor.value = function (...args: any[]) {
            let value = original.call(this, ...args);
            return mathFunc(value);
        };
    }
}

let a = new Num(5); console.log(a.getNumber());  // 18 => ((((5)+5)*2)-2)
```

Advanced Usage

Advanced Usages

- Sometimes we might need some type information exposed at runtime for that we can use the **reflect-metadata** library

```
npm install reflect-metadata
```

  - Adds an internal **[[Metadata]]** property to objects, that holds type information we can call at runtime
  - Requires **"emitDecoratorMetadata": true** in **tsconfig**
- We can use this runtime type information to implement more complex use cases like:
  - Dependency Injection
  - More robust validation

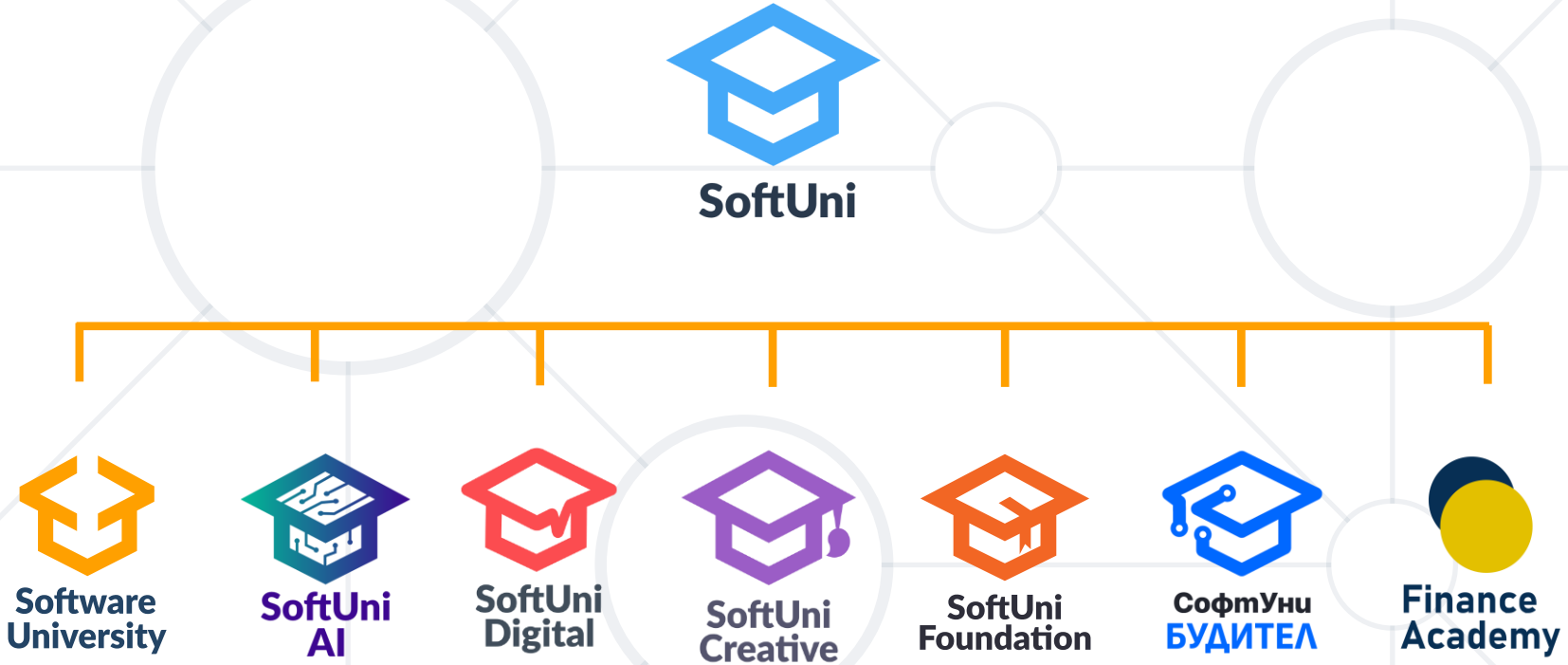
Example: Reflect-Metadata

```
import "reflect-metadata";

class Address { constructor(public city: string) {} }
class Person {
  constructor(private _address: Address) { }
  @validate set address(value: Address) { this._address = value; }
  get address() { return this._address;}
}
function validate<T>(target: any, propertyKey: string, descriptor: TypedPropertyDescriptor<T>) {
  let set = descriptor.set!;
  descriptor.set = function (value: T) {
    let type = Reflect.getMetadata("design:type", target, propertyKey);
    if (!(value instanceof type)) {  throw new TypeError('Invalid type.'); }
    set.call(this, value);
  };
}

let city = new Address('Tokyo’);
let person = new Person(city);
console.log(person)                                  // {_address: Address {city: 'Tokyo'}}
person.address = { city: 'New York'};                // Runtime TypeError: Invalid type.
```

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 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

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