**TYPESCRIPT DECORATOR**

* Decorators in typescript are just a function.
* The operator **@**  is used to execute the decorator.
* It’s a general convention to use first letter as capital while defined decorators.
* Decorators must be placed just above the class.
* The property **experimentalDecorator** in tsconfig.jsonmust be enabled in order to use the decorators.
* **Decorators are executed when the class is defined not when its instantiated.**
* Decorator receives the class constructor function us the first argument.
* We can utilize the constructor function to instantiate the class inside the decorator function.
* //? Decorators
* //\* To enable the use of decorators in our project must enable the experimentalDecorators property in tsconfig.json.
* //\* Its a general convention to start the decorator name by capital letter.
* //\* The @ operator is used to use the decorator.
* //\* Decorator executed when the class is defined not when the class is executed.
* //\* Decorator must be placed just above the class it is to be associated with.
* function Logger(constructor: Function) {
* // Executes when the class is defined!
* console.log('logging...');
* console.log(constructor);
* }
* @Logger
* class Student {
* name = 'Harshit';
* constructor() { console.log('Creating a student object...') }
* }
* const harshit = new Student();

# **Creating a Decorator Factory:**

* With normal decorator declaration we are unable to pass our own custom arguments.
* Decorator factory is a approach of wrapping up a decorator inside a parent function that can take the argument.
* In short, Decorator factory enable us to utilize the utilize our own arguments in the decorator.
* //? Decorator Factory.
* //\* Normal declaration of decorator does not accepts arguments.
* //\* Decorator factory is just a approach of warpping our decorator inside another function.
* //\* The advantage of decorator factory is that now we can pass our own agrmnets to be utilized in the decorator.
* function LoggerDecorator(customMessage: string) {
* return function (constructor: Function) {
* console.log(customMessage);
* console.log(constructor);
* }
* }
* @LoggerDecorator('Firing off the decorator') //? Decorator factory
* class Student {
* name = 'Harshit'
* age = 24
* constructor() { console.log(`Creating an student instance.`) }
* };

# **Multiple Decorators and Execution Order:**

* We can assign multiple decorators to a class.
* The decorators will be executed in the bottom’s up manner, irrespective of which order the decorator factories will be executed.
* The decorator factories will be executed in the order they are defined, but the decorators are executed in the bottom’s up order.

//? Multiple Decorators

//\* In case of multiple decorators the decorators execution occurres from bottom's up.

//\* The decorator that is at the bottom will execute first.

//\* The decorator factory will executes in the given order, but the decorators inside them will executes in bottom's up manner.

function Logger(identifier: string) {

    console.log('Logger Factory');

    return function (\_: Function) {

        console.log(identifier);

    }

}

function User(identifier: string) {

    console.log('User Factory')

    return function (\_: Function) {

        console.log(identifier);

    }

}

@Logger('Logger decorator')

@User('User Decorator')

class Student {

    name = 'Harshit';

    constructor() { console.log('Creating a student object...') }

}

const harshit = new Student();

# **Property Decorators:**

* Property decorators are placed just above the property.
* Similar to class decorators, property decorators are executed when the property is defined, not when the class is instantiated.
* The property decorators receives two arguments target and propertyName.
  + In case of normal property, the target is the prototype of instantiated object for the class.
  + In case of Statis property, the target is the constructor of the class.
* //? Property decorator
* //\* Property decorator is similer to class decorator, but it executes when the property is defined in the class.
* //\* Property decorator are placed just before the property in our class.
* //\* property decorator recives two arguments first is target and second is the property name.
* //\* in case of normal property instantiated object recieved as per the target.
* //\* in case of static property the class constructor recieved as per the target.
* function NameLogger(target: any, propertyName: string) {
* console.log('The NameLogger => ', target, propertyName)
* }
* function TypeLogger(target: any, propertyName: string) {
* console.log('The TypeLogger => ', target, propertyName)
* }
* class Student {
* @NameLogger //? recieve the instantiated object as target.
* name = 'Harshit';
* age = 24;
* @TypeLogger
* static type = 'Human'; //? recieve the class constructor as target.
* constructor() { };
* printInfo() { console.log('student info => ', { name: this.name, age: this.age }) }
* }

# **Accessor Decorators (get and set):**

* Accessor decorators can be used with accessor keywords like get and set.
* Accessor decorators receives three arguments, target: prototype object, name: name of accessor method, descriptor: description of the accessor method of type PropertyDescriptor.

# **Method Decorators (Normal and Static):**

* Method decorators can be used with normal and static methods.
* Method decorators receives three arguments, target: prototype object or constructor of the class(in case of static method), name: name of method, descriptor: description of the method of type PropertyDescriptor.

# **Parameter Decorators:**

* Parameter decorators can be used with any parameter of the methods.
* Parameter decorators receives three arguments, target: prototype object, name: name of parameter, position: position of the parameter in indexing.

//? Accessor Decorator

//\* Accessor decorator can be used with the accessor keywords like get and set.

//\* Accessor decorator receives three arguments, target: prototype object, name: accessor name, descriptor: description of the accessor of type PropertyDescriptor.

function AccessorDecorator(target: any, name: string, descriptor: PropertyDescriptor) {

    console.log('The Accessor Decorator =>', target, name, descriptor);

}

//? Method Decorator

//\* Method decorators can be used with the static and normal methods of the class.

//\* Method decorators receives three arguments, target: prototype object or constructor of class (in case of static method), name: name of method, descriptor: description of the method of type PropertyDescriptor.

function methodDecorator(target: any, name: string, descriptor: PropertyDescriptor) {

    console.log('The Method Decorator =>', target, name, descriptor);

}

//? Parameter Decorator

//\* Parameter descorators can be used with any parameter of the method.

//\* Parameter decorators recieves three arguments, target: prototype object, name: parameter name, position: position of parameter in terms of index.

function parameterDecorator(target: any, name: string, position: number) {

    console.log('The Method Decorator =>', target, name, position);

}

class Person {

    public name: string;

    public age: number;

    @AccessorDecorator

    get getPersonInfo() { return { name: this.name, age: this.age } }

    constructor(name: string, age: number) {

        this.name = name;

        this.age = age;

    }

    @methodDecorator

    greet(@parameterDecorator message: string) {

        console.log(`${message} ${this.name}!`);

    }

}

**Returning Form Decorators:**

Decorators are also capable of returning some logic to achieve most out of it, and unleash the power of meta programming. Only **Class, Accessor, Method** decorators can return others can not.

## **Class Decorators:**

* The decorators assigned to the classes are able to return a new constructor that will replace/twerk the logic of the original constructor.
* Constructors executes only when the class is instantiated, therefore the logic inside the returned constructor will executed when the class is instantiated.
* //? Returning for the decorator.
* //\* We can return a constructor function from the decorator.
* //\* Constructor functions only executes when the class is instantiated.
* //\* The returned constructor function can be used to replace or twerk the logic of existing constructor function.
* //\* The returend constructor fucntion can replace the Original Constructor function when the class is defined.
* function StudentDecorator() {
* console.log('Student Decorator Factory');
* return function <T extends { new(...args: any[]): {} }>(OriginalConstructor: T) {
* console.log('Twerking the logic to be executed in the original constructor function of class Student');
* return class extends OriginalConstructor {
* //? This constructor is going the replace the original constructor
* constructor(...\_: any) {
* //? This code will execute when the class is instantiated.
* super(); //? using super to perserve original constructors functionality.
* console.log('The replaced constructor logic from Student Decorator.')
* }
* }
* }
* }
* @StudentDecorator()
* class Student {
* name = 'Harshit'
* age = 24
* constructor() {
* console.log('I m the original constructor function.')
* }
* }
* const harshit = new Student();
* console.log(`The name is ${harshit.name} of age ${harshit.age}`)

## **Accessor Decorators:**

* The decorators assigned to the Accessor can return a **PropertyDescriptor** type of object.
* The property descriptor is the same object that we receives as the third argument in the accessor decorator.
* //? Returning form accessor decorator.
* //\* We can return the Object of type PropertyDescriptor from Accessor Decorator.
* function AccessorDecorator(\_: any, \_\_: string, descriptor: PropertyDescriptor): PropertyDescriptor {
* console.log("The property descriptor => ", descriptor)
* return { ...descriptor, configurable: false } //? This will chnage its property descriptor.
* }
* class Person {
* name = 'Harshit';
* @AccessorDecorator
* get getName() { return this.name }
* constructor() { };
* }

## **Method Decorators:**

* The decorators assigned to the Method can return a **PropertyDescriptor** type of object.
* The property descriptor is the same object that we receives as the third argument in the method decorator.
* Example: Creating an autobind method

//? Retuning form the Method Decorator.

//\* Similer to the accessor decorator we can return the object of type PropertyDescriptor.

//\* The return object of type PropertyDescriptor will replace the existing descriptor for the method.

function AutoBind(\_: any, \_\_: string, descriptor: PropertyDescriptor): PropertyDescriptor {

    return {

        configurable: true, enumerable: false, get() {

            const originalMethod = descriptor.value

            return originalMethod.bind(this)

        }

    }

}

class Greet {

    constructor(public message: string) { }

    @AutoBind

    getGreeting() {

        console.log(`${this.message}`)

    }

}

const newGreet = new Greet('Good Morning')

const greetButton = document.querySelector('#greet')! as HTMLButtonElement

greetButton.addEventListener('click', newGreet.getGreeting)

//greetButton.addEventListener('click', newGreet.getGreeting.bind(newGreet))  ONE WORK AROUND (Using bind)

**Validation through Decorators:**

Decorators can be utilized to provide validation to class properties , the package [class-validator - npm (npmjs.com)](https://www.npmjs.com/package/class-validator) is used to apply validation to class property which work with the decorators. Although here is more raw approach to achieve the same :

//? Valiadtion with Decorators

interface ValidatorConfig {

    [property: string]: {

        [validatableProperty: string]: string[]

    }

}

const registeredValidators: ValidatorConfig = {}

function RequiredDecorator(target: any, propertyName: string) {

    registeredValidators[target.constructor.name] = {

        ...registeredValidators[target.constructor.name],

        [propertyName]: ['required']

    }

}

function PositiveNumberDecorator(target: any, propertyName: string) {

    registeredValidators[target.constructor.name] = {

        ...registeredValidators[target.constructor.name],

        [propertyName]: ['positive']

    }

}

function validate(obj: any) {

    const objValidatorConfig = registeredValidators[obj.constructor.name];

    if (!objValidatorConfig) {

        return true;

    }

    let isValid = true;

    for (const prop in objValidatorConfig) {

        for (const validator of objValidatorConfig[prop]) {

            switch (validator) {

                case 'required':

                    isValid = isValid && !!obj[prop]

                    break;

                case 'positive':

                    isValid = isValid && obj[prop] > 0

                    break;

            }

        }

    }

    return true;

}

class Person {

    @RequiredDecorator

    name: string;

    @PositiveNumberDecorator

    age: number;

    constructor(name: string, age: number) {

        this.name = name;

        this.age = age;

    }

}

const personForm = <HTMLFormElement>document.querySelector('#person-form')!;

personForm.addEventListener('submit', (e) => {

    e.preventDefault();

    const nameInput = <HTMLInputElement>document.querySelector('#name-input')!

    const ageInput = <HTMLInputElement>document.querySelector('#age-input')!

    const person = new Person(nameInput.value, +ageInput.value);

    if (!validate(person)) {

        alert('Invalid Input, please try again.')

        return;

    }

    console.log("The person data => ", person);

})