САМАРСКИЙ УНИВЕРСИТЕТ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»

Лекция № 6

Декораторы

2023

РУБРИКА: «ВОПРОСЫ ПО ЛЕКЦИИ № 5»

Предисловие о декораторах

Что такое декораторы?

Это функции, которые могут применяться к классам, методам, методам доступа (геттерам и сеттерам), свойствам и параметрам путем вставки перед таким элементом .

Зачем это нужно?

Иногда требуется довольно много одинаковых операций для одного типа структурного элемента программы. Например отслеживание работы и внесение корректировок некоторой функции, класса или отдельного свойства. Для этого можно писать всё это внутри самого элемента программы, однако код будет нечитабелен из-за излишних действий со стороны разработчика. Со стороны будет казаться, что этот код должен быть некими метаданными, поэтому включение его в основной код будет некорректным.
Для того, чтобы облегчить понимание кода такие метаданные решили обозначать как отдельный структурный элемент языка, который навешивается на другие элементы описываемой программы и позволяет читать и при необходимости изменять данные и поведение без непосредственного внутреннего включения.

Декораторы :: Какие они бывают

- Декораторы классов
- Декораторы методов и их параметров
- Декораторы свойств и методов доступа

Декораторы :: Настройки компилятора

Декораторы нужно включить путем переключения свойства experimentalDecorators в включенное состояние (true)

```
{
    "compilerOptions": {
        "target": "ES5",
        "experimentalDecorators": true
    }
}
```

Декораторы классов

Декоратор класса применяется к конструктору класса и позволяет изменять или заменять определение класса.

Декоратор класса представляет функцию, которая принимает один параметр

```
//Определение декоратора класса
function classDecoratorFn(constructor: Function){ }
```

Декоратор sealed с помощью функции Object.seal запрещает расширение прототипа класса User.

```
function sealed(constructor: Function) {
    console.log("sealed decorator");
    Object.seal(constructor);
    Object.seal(constructor.prototype);
}

@sealed
class User {
    name: string;
    constructor(name: string){
        this.name = name;
    }
    print():void{
        console.log(this.name);
    }
}
```

```
Object.defineProperty(User, 'age', {    value: 17}); //Произойдет ошибка!!!
```

Декораторы :: Декораторы классов :: Пример 2

Декораторы могут изменять результат работы конструктора. В этом случае определение функции декоратора немного меняется, но она также в качестве параметра принимает конструктор класса

```
function logger<TFunction extends Function>(target: TFunction):
TFunction{

    let newConstructor: Function = function(name:string){
        console.log("Creating new instance");
        this.name = name;
        this.age = 23;
        this.print = function():void{
            console.log(this.name, this.age);
        }
    }
    return <TFunction>newConstructor;
}

@logger
class User {
    name: string;
    constructor(name: string){
        this.name = name;
    }
    print():void{
        console.log(this.name);
    }
}
let tom = new User("Tom");
let bob = new User("Bob");
tom.print();
bob.print();
```

```
Creating new instance Creating new instance
Tom 23 Bob 23
```

Декораторы методов и их параметров

```
function deprecated(target: any, propertyName:
string, descriptor: PropertyDescriptor){
    console.log("Method is deprecated");
}
```

Декоратор принимает следующие параметры:

Функция конструктора класса для статического метода, либо прототип класса для обычного метода.

Название метода.

Объект интерфейса PropertyDescriptor

```
interface PropertyDescriptor{
    configurable?: boolean;
    enumerable?: boolean;
    value?: any;
    writable?: boolean;
    get? (): any;
    set? (v: any): void;
}
```

```
function readable (target: Object, propertyKey: string, descriptor:
PropertyDescriptor) {
    descriptor.writable = false;
};

class User {

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

    @readable
    print():void{
        console.log(this.name);
    }
}
let tom = new User("Tom");
tom.print = function(){console.log("print has been changed");}
tom.print();  // Tom
```

Декоратор readable с помощью выражения descriptor.writable = false; устанавливает, что метод, к которому применяется данный декоратор, не может быть изменен.

В итоге после применения данного декоратора следующая инструкция

```
function readable(onlyRead : boolean){

    return function (target: Object, propertyKey: string, descriptor:
PropertyDescriptor) {
        descriptor.writable = !onlyRead;
    };
}

class User {

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

    @readable(false)
    print():void{
        console.log(this.name);
    }
}
let tom = new User("Tom");
tom.print = function(){console.log("print has been changed");}
tom.print();  // Tom
```

Параметризация декоратора позволяет манипулировать его состоянием, и, как следствие, получать различные комбинации его работы при элементе, над которым он был применен.

Декораторы :: Декораторы методов и их параметров :: Параметры и выходной результат метода

```
function log(target: Object, method: string, descriptor:
PropertyDescriptor){
    let originalMethod = descriptor.value;
    descriptor.value = function(...args: number[]){
        console.log(JSON.stringify(args));
        let returnValue = originalMethod.apply(this, args);
        console.log(`${JSON.stringify(args)} => ${returnValue}`)
        return returnValue;
    }
}

class Calculator{

    @log
    add(x: number, y: number): number{
        return x + y;
    }
}

let calc = new Calculator();
let z = calc.add(4, 5);
z = calc.add(6, 7);
```

Переопределение логики работы метода является важнейшим достоинством применения декораторов.
На данном примере показана активация функции отслеживания срабатывания метода и вывод его данных параметров и результата в консоль

Декораторы свойств и методов доступа

Декоратор свойства

```
function MyPropertyDecorator(target: Object,
propertyKey: string){
    // код декоратора
}
```

Декоратор метода доступа

```
function decorator(target: Object, propertyName:
string, descriptor: PropertyDescriptor){
// код декоратора
}
```

Декораторы :: Декораторы свойств и методов доступа :: Декоратор свойства

```
function format() {
  return function(target: Object,
propertyKey: string) {
    let value : string;
    const getter = function() {
      return "Mr./Ms." + value;     //
изменяем возвращаемое значение
    };
    const setter = function(newVal: string) {
      if(newVal.length > 2) {   // добавляем
проверку на длину строки
          value = newVal
      }
    };
    // устанавливает геттер и сеттер для
свойства
    Object.defineProperty(target,
propertyKey, {
      get: getter,
      set: setter
    });
  }
}
```

```
class User {

    @format()
    name: string;
    constructor(name:
string){
        this.name = name;
    }
    print():void{
        console.log(this.name);
    }
}
let tom = new User("Tom");
tom.print();
tom.name = "Tommy";
tom.print();
tom.name = "To";
tom.print();
```

Декораторы для свойств необходимы в первую очередь для переопределения геттеров и сеттеров, представленных в классе по умолчанию с целью замены логики их работы и повышению качества и читабельности кода.

Декораторы :: Декораторы свойств и методов доступа :: Декоратор метода доступа

```
function validator(target: any,
propertyKey: string, descriptor:
PropertyDescriptor) {
    const oldSet = descriptor.set;

    descriptor.set = function(value: string) {
        if (value === "admin") {
            throw new Error("Invalid value");
        }
        if(oldSet!==undefined)
oldSet.call(this, value);
    }
}
```

```
class User {

    private _name: string;
    constructor(name: string){
        this.name = name;
    }

    public get name(): string {
        return this._name;
    }
    @validator
    public set name(n: string) {
        this._name = n;
    }
}
let tom = new User("Tom");
console.log(tom.name);
tom.name= "admin";
console.log(tom.name);
```

Декораторы для свойств необходимы в первую очередь для переопределения геттеров и сеттеров, представленных в классе по умолчанию с целью замены логики их работы и повышению качества и читабельности кода.

РУБРИКА: «ВОПРОСЫ СЛУШАТЕЛЕЙ»

КОНЕЦ ЛЕКЦИИ № 6
СПАСИБО ЗА ВНИМАНИЕ