Decoradores

Antonio Espín Herranz

Contenidos

- ¿Qué es un decorador?
- Activación
- ¿Dónde se utilizan?
- Tipos de decoradores

¿Qué son?

- Un decorador es una **función.**

- Nos permiten **cambiar/agregar** el **comportamiento** de un elemento durante la **ejecución** del código.

- Se pueden ejecutar mediante una expresión que empieza por **@** seguido del nombre del decorador.
  - **@NombreDecorador**

- https://www.typescriptlang.org/docs/handbook/decorators.html

Activación

- Para trabajar con los decoradores en TypeScript hay que **activar** un par de opciones en el fichero de configuración: **tsconfig.json**

- O si estuviéramos utilizando el comando tsc en la consola habría que hacerlo así:
  - **tcs --expirementalDecorators --emitDecoratorMetadata**

  - Dentro del fichero de configuración, quitar los comentarios_
  - **expirementalDecorators=true**
  - **emitDecoratorMetadata=true**

¿Dónde se utilizan?

- Registrar la ejecución de un método.
- Inyectar dependencias.
- Notificar el cambio de valor de una propiedad.
- Permiten escribir código de una forma más declarativa, lo utilizan mucho los frameWorks.
  - Por ejemplo.
    - Definir el verbo HTTP vinculado a una acción de un controlador
    - Definir la estructura de una tabla en una BD (con un ORM: Object Relational Mapping).
- Los decoradores se definen el fichero: lib.d.ts

Tipos de decoradores

- Decoradores **experimentales**: introducidos en 2016 en TypeScript 1.5

- Decoradores **EMACScript**: en 2023 en la versión 5.0 en TypeScript.

- Ambos están relacionados con la programación orientada a objetos y se pueden aplicar a:
  - Clases, métodos, propiedades, accesores (get/set) y parámetros (para los experiementales).

Consideraciones

- Se pueden aplicar varios decoradores.

- El orden de aplicación es:
  - Primero el más interno y luego hacia afuera.

  - @decorador2
  - @decorador1
  - class Nombre { ... }

  - Se aplica primero decorador1, después decorador2, así sucesivamente.

Experimentales

- De clase: **ClassDecorator**
- De método: **MethodDecorator**
- De propiedad: **PropertyDecorator**
- De parámetro: **ParameterDecorator**
- Fábricas de decoradores

ClassDecorator

- Es aplicado al **constructor** de la clase y puede ser usado para observar, modificar o reemplazar la definición inicial de la clase.

- Su único argumento es **target** que vendría siendo la clase decorada, tipado como **Function** o **any**

Ejemplo: con funciones

```
function classDecorator(target: Function) {
  console.log('Clase decorada:', target);
}

@classDecorator
class MyClass1 {
  constructor() {
    console.log('Instancia de MyClass1 creada');
  }
}

// Código principal
const myClassInstance = new MyClass1(); // Instancia de MyClass1 creada
```

Ejemplo: con funciones flecha

```
const LogClassName: ClassDecorator = (target: Function) => {
   console.log('Decorador 2, clase: ', target);
}
```

- ***Se puede poner así:***

```
const LogClassName: ClassDecorator = target => {
   console.log('Decorador 2, clase: ', target);
}
```

- El tipo: **ClassDecorator** está definido en el archivo de definición de typescript
- Nombre del decorador: **LogClassName**
- Se inicializa a una función flecha

MethodDecorator

- Tienen el mismo objetivo que las clases de observar, modificar o reemplazar.

- La función toma **tres** parámetros:
  - **target**: Método decorado, generalmente tipado como Object
  - **propertyKey**: Nombre del método, tipado como string | symbol
  - **descriptor**: Property Descriptor del objeto (value, writable, enumerable, configurable)

- Este decorador se puede utilizar para modificar el comportamiento del método.
  - Se puede hacer asignando una función flecha al **descriptor**.

Ejemplo

```
const MetodoDecorator: MethodDecorator = (target, propertyKey, descriptor) => {
  console.log('Decorador de método, clase: ', target.constructor.name, 'método: ', propertyKey, 'descriptor: ', descriptor);
}

class MyClass1 {
 constructor() {
  console.log('Instancia de MyClass1 creada');
 }
 @MetodoDecorator
 metodo1(x: number, y: number): number {
  return x + y;
 }
}
```

Ejemplo 2

```
function log(target: Object, propertyKey: string, descriptor: any) {
        console.log('Clase: ', target.constructor.prototype);
        console.log('Método: ', propertyKey);
        console.log('Property Descriptor: ', descriptor);

        descriptor.value = function (...args: any[]) {
                console.log('Argumentos de la funcion', args);
  }
  return descriptor;
}

class ExampleClass {
        @log
        outputSomething(something: string) {
                console.log(something);
        }
}

new ExampleClass().outputSomething('Parametro de prueba');
```

PropertyDecorator / ParameterDecorator

- Son más simples que los anteriores, un decorador de propiedades debe tomar como parámetros **target**, que es el prototipo de la clase, y **propertyKey**, el nombre de la propiedad.

- Para los decoradores de parámetros recibimos: **parameterIndex**, que indica la posición en el array.

Ejemplo

```
function decoratedProperty(target: Object, propertyKey: string) {
   console.log('Clase', target);
   console.log('Nombre de la propiedad', propertyKey);
}

function decoratedParam(target: Object, propertyKey: string, parameterIndex: number) {
   console.log('Nombre del metodo', propertyKey);
   console.log('Clase', target);
   console.log('Posición del parámetro', parameterIndex);
}
class ExampleClass {
   @decoratedProperty exampleProperty: string = 'Hello World';

   sum(a: number, @decoratedParam b:number): number {
      return a+b
   }
}
```

Orden de Evaluación de los decoradores

1. Decoradores de parámetros (decoradores de parámetros), seguidos por Method Accesors o Decoradores de Propiedades son aplicados a cada instancia de cada miembro.
2. Decoradores de parámetros, seguido por Method Accesors, o Decoradores de Propiedades son aplicados a cada miembro estático.
3. Decoradores de parámetros, son aplicados al constructor.
4. Decoradores de clases son aplicados a las clases.

Fábrica de decoradores

- Una fábrica de decoradores es una función que retorna un decorador. Esto permite que el decorador sea configurable, porque puedes pasarle parámetros cuando lo aplicas.

- Es una forma más flexible de definir decoradores, especialmente si necesitas que el comportamiento del decorador dependa de datos específicos

Ventajas

- Permiten personalizar los decoradores según necesidades
  - Podemos elegir que parámetros van a recibir los decoradores.

- Hacen que el código sea más reutilizable y modular.

- Son ideales para casos como registro, validación o configuración específica de métodos o propiedades.

Ejemplo

```
function LogExecutionTime(showDetails: boolean) {
  // Retorna el decorador
  return (target: any, propertyKey: string, descriptor: PropertyDescriptor) => {
    const originalMethod = descriptor.value;

    descriptor.value = function (...args: any[]) {
      const start = performance.now();
      const result = originalMethod.apply(this, args);
      const end = performance.now();

      console.log(`Método "${propertyKey}" ejecutado en ${end - start} ms.`);
      if (showDetails) {
        console.log(`Argumentos: ${JSON.stringify(args)}`);
        console.log(`Resultado: ${result}`);
      }
      return result;
    };
    return descriptor;
  };
}
```

```
class Calculadora {
  @LogExecutionTime(true) // Usamos la fábrica de decoradores y pasamos un parámetro
  suma(a: number, b: number): number {
    return a + b;
  }
}

const calc = new Calculadora();
calc.suma(5, 3); // Se registra el tiempo de ejecución y detalles adicionales.
```

Los decoradores de EMACScript

- https://kinsta.com/es/blog/typescript-5-0/