JUNE 1, 2015

Decorators & metadata reflection in TypeScript: From Novice to Expert (Part II)

An in-depth look to the TypeScript implementation of decorators and how they make possible new exciting JavaScript features like reflection or dependency injection.

This article is the second part of a series:

- **PART I: Method decorators**
- **PART II: Property decorators & Class decorators**
- **PART III: Parameter decorators & Decorator factory**
- **PART IV: Types serialization & The metadata reflection API**

In the previous post in this series we learned that we can find the available decorators signatures in the TypeScript source-code.

We also learned how to implement a method decorator and we answered some basic questions about the way decorators work in TypeScript:

- **Where is decorator being invoked?**
- **Who is providing the decorator arguments?**
- **Where is the `__decorate` function declared?**

I will assume that you have already read the Part I of these series and you know the answer to these questions.

In this post we will learn about the two new decorator types: `PropertyDecorator` and `ClassDecorator`.

Let's start with `PropertyDecorator`.

1. Property decorator

As we already know, the signature of a `PropertyDecorator` looks as follows.

```
declare type PropertyDecorator = (target: Object, propertyKey:
string | symbol) => void;
```

We can use a property decorator named `logProperty` as follows:

```
class Person {

  @logProperty
  public name: string;
  public surname: string;

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

When compiled into JavaScript the `__decorate` method (which we explained in PART I) is invoked here but this time it is missing the last parameter (a property descriptor obtained via `Object.getOwnPropertyDescriptor`).

```
var Person = (function () {
    function Person(name, surname) {
        this.name = name;
        this.surname = surname;
    }
    __decorate([
        logProperty
    ], Person.prototype, "name");
    return Person;
})();
```

This is the reason why the property decorator takes 2 (prototype and key) arguments as opposed to 3 (prototype, key and property descriptor) like in the case of the method decorator.

Another thing that we should notice is that this time the TypeScript compiler is not using the return of `__decorate` to override the original property like it with the method decorator.

```
 Object.defineProperty(C.prototype, "foo",
        __decorate([
            log
        ], C.prototype, "foo",
Object.getOwnPropertyDescriptor(C.prototype, "foo")));
```

This is the reason why **property decorators don't return**.

Now that we know that a property decorator takes the prototype of the class being decorated and the name of the property being decorated as arguments and don't return, let's implement the `logProperty` decorator.

```
function logProperty(target: any, key: string) {

  // property value
  var _val = this[key];

  // property getter
  var getter = function () {
    console.log(`Get: ${key} => ${_val}`);
    return _val;
  };

  // property setter
  var setter = function (newVal) {
    console.log(`Set: ${key} => ${newVal}`);
    _val = newVal;
  };

  // Delete property.
  if (delete this[key]) {

    // Create new property with getter and setter
    Object.defineProperty(target, key, {
      get: getter,
      set: setter,
      enumerable: true,
      configurable: true
    });
  }
}
```

The decorator above declares a variable named `_val` and sets its value to the value of the property being decorated (since `this` refers to the class prototype here and `key` is the name of the property).

Then, the functions `getter` (used to get the value of the property) and `setter` (used to set the value of the property) are declared. Both functions will have permanent access to `_val` thanks to the closures created when each of these functions are declared. Here is where we will **add some extra**

behaviour to the property. In this case we have added a line to log in console the changes in the property value.

Later, the operator `delete` is used to delete the original property from the class prototype.

> Note: The delete operator throws in strict mode if the property is an own non-configurable property (returns false in non-strict).

If the property is successfully deleted, The `Object.defineProperty()` method is used to create a new property using the original property's name but this time the property uses the previously declared `getter` and `setter` functions.

Now that the decorator is ready it will log in console the changes to the property every time we set or get its value.

```
var me = new Person("Remo", "Jansen");
// Set: name => Remo

me.name = "Remo H.";
// Set: name => Remo H.

name;
// Get: name Remo H.
```

2. Class decorator

As we already know, the signature of a `ClassDecorator` looks as follows.

```
declare type ClassDecorator = <TFunction extends Function>(target:
TFunction) => TFunction | void;
```

We can use a class decorator named `logClass` as follows:

```
@logClass
class Person {

  public name: string;
  public surname: string;

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

When compiled into JavaScript the `__decorate` function (which we explained in PART I) is invoked here but this time it is missing the last 2 parameters.

```
var Person = (function () {
    function Person(name, surname) {
        this.name = name;
        this.surname = surname;
    }
    Person = __decorate([
        logClass
    ], Person);
    return Person;
})();
```

We should notice that the compiler is passing `Person` and not `Person.prototype` to `__decorate`. This is the reason why the class decorator takes 1 (the class constructor) argument as opposed to 3 (prototype, key and property descriptor) like in the case of the method decorator.

Another thing that we can notice is that this time the TypeScript compiler is using the return of `__decorate` to override the original constructor

```
 Person = __decorate(/* ... */);
```

This is the reason why **class decorators must return a constructor function**.

Now that we know that a class decorator takes the constructor of the class being decorated as its only argument and must return a new constructor, let's implement the `logClass` decorator..

```
function logClass(target: any) {

  // save a reference to the original constructor
  var original = target;

  // a utility function to generate instances of a class
  function construct(constructor, args) {
    var c : any = function () {
      return constructor.apply(this, args);
    }
    c.prototype = constructor.prototype;
    return new c();
  }

  // the new constructor behaviour
  var f : any = function (...args) {
    console.log("New: " + original.name);
    return construct(original, args);
  }

  // copy prototype so intanceof operator still works
  f.prototype = original.prototype;

  // return new constructor (will override original)
  return f;
}
```

The decorator above declares a variable named `original` and sets its value to the constructor of the class being decorated.

Then, a utility function named `construct` is declared. This function allow us to create instances of a class.

We then create a variable named `f` that will be used as the new constructor. This function invokes the original constructor and will also log in console the name of the class being instantiated. Here is where we will **add some extra behaviour to the original constructor**.

The prototype of the original constructor is copied to the prototype of `f` to ensure that the `instanceof` operator works as expected when we create a new instance of `Person`.

Once the new constructor is ready we just need to return it to finish the class decorator implementation.

Now that the decorator is ready it will log in console the name of a class every time it is instantiated.

```
var me = new Person("Remo", "Jansen");
// New: Person

me instanceof Person;
// true
```

Conclusion

We now understand in-depth 3 out of the 4 available types of decorators. We know how to implement them and how they work internally.

In the next post we will learn about the last type of decorator (the parameter decorator) and how to create a universal decorator that we can apply to classes, properties, methods and parameters.

If you have enjoyed this article please check out *The end of JavaScript?* in which I discuss how the arrival of metadata annotations could mean the end of plain JavaScript as a design-time programming language.

Don't forget to subscribe if you don't want to miss it out more JavaScript and TypeScript content.

Please feel free to talk about this article with us via @OweR_ReLoaDeD and @WolkSoftwareLtd