MAY 18, 2015

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

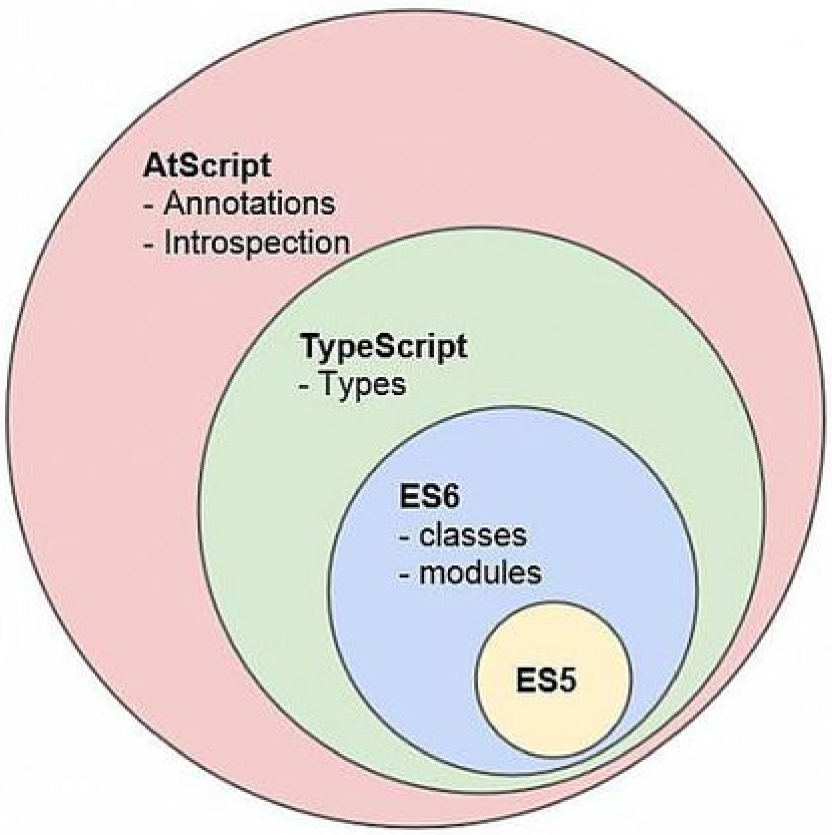
An in-depth look to the TypeScript implementation of decorators and how they make possible new exciting JavaScript features like reflection or dependency injection.

This series will cover:

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

A few months ago Microsoft and Google announced that they were working together on TypeScript and Angular 2.0

> We're excited to announce that **we have converged the TypeScript and AtScript languages**, and that Angular 2, the next version of the popular JavaScript library for building web sites and web apps, will be developed with TypeScript.



Annotations & decorators

This partnership has helped TypeScript to evolve additional language features among which we will highlight **annotations**.

> **Annotations, a way to add metadata to class declarations for use by dependency injection or compilation directives.**

Annotations was an proposed by the Google AtScript team but annotations are not a standard. However, decorators are a proposed standard for ECMAScript 7 by Yehuda Katz, to annotate and modify classes and properties at design time.

Annotation and decorator are pretty much the same:

> Annotations and decorators are nearly the same thing. From a consumer perspective we have exactly the same syntax. The only thing that differs is that we don't have control over how annotations are added as meta data to our code. Whereas decorators is rather an interface to build something that ends up as annotation.

> Over a long term, however, we can **just focus on decorators, since those are a real proposed standard**. AtScript is TypeScript and TypeScript implements decorators.

Let's take a look to the TypeScript's decorators syntax.

Note: If you want to learn more about the difference between Annotations and Decorators there is a great article by Pascal Precht on this topic.

Decorators in TypeScript

In the TypeScript source code we can find the signature of the available types of decorators:

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

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

declare type MethodDecorator = <T>(target: Object, propertyKey:
string | symbol, descriptor: TypedPropertyDescriptor<T>) =>
TypedPropertyDescriptor<T> | void;

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

As we can see, decorators can be used to annotate a `class`, `property`, `method` or `parameter`. Let's take a in-depth look to each one of these types of decorators.

Method decorators

Now that we know how the signature of a decorator looks like we can implement some of them. We are going to start by implementing a **method decorator**. Let's create a method decorator named `log`.

To invoke a method decorator we need to prefix the method that we wish to decorate with the `@` character follow by the name of the decorator. In the case of a decorator named `log`, the syntax will look as follows:

```
class C {
    @log
    foo(n: number) {
        return n * 2;
    }
}
```

Before we can actually use `@log` we need to declare the method decorator somewhere in our application. Let's take a look to the `log` **method decorator** implementation.

```
function log(target: Function, key: string, value: any) {
    return {
        value: function (...args: any[]) {
            var a = args.map(a => JSON.stringify(a)).join();
            var result = value.value.apply(this, args);
            var r = JSON.stringify(result);
            console.log(`Call: ${key}(${a}) => ${r}`);
            return result;
        }
    };
}
```

Note: Please take a look to updates at the end of this post for an alternative implementation, which a avoids one potential issue.

A **method decorators** takes a 3 arguments:

- `target` the method being decorated.
- `key` the name of the method being decorated.
- `value` a property descriptor of the given property if it exists on the object, undefined otherwise. The property descriptor is obtained by invoking the Object.getOwnPropertyDescriptor() function.

There is something strange right? We didn't pass any of these parameters when we used the decorator `@log` in the `C` class definition. At this point we should be wondering **who is providing those arguments?** and **Where is the `log` method being invoked?**

We can find the answers to these questions by examining the code that the TypeScript compiler will generate for the code above.

```
var C = (function () {
    function C() {
    }
    C.prototype.foo = function (n) {
        return n * 2;
    };
    Object.defineProperty(C.prototype, "foo",
        __decorate([
            log
        ], C.prototype, "foo",
Object.getOwnPropertyDescriptor(C.prototype, "foo")));
    return C;
})();
```

Without the `@log` decorator the generated JavaScript for the `C` class would just be as follows.

```
var C = (function () {
    function C() {
    }
    C.prototype.foo = function (n) {
        return n * 2;
    };
    return C;
})();
```

But when we add the `@log` decorator the following additional code is added to the class definition by the TypeScript compiler.

```
Object.defineProperty(
  __decorate(
    [log],                                               //
decorators
    C.prototype,                                         // target
    "foo",                                               // key
    Object.getOwnPropertyDescriptor(C.prototype, "foo") // desc
  );
);
```

If we read the MDN documentation we will learn that the following about the `defineProperty` function.

> The Object.defineProperty() method defines a new property directly on an object, or modifies an existing property on an object, and returns the object.

The TypeScript compiler is passing the prototype of `C`, the name of the method being decorated (`foo`) and the return of a function named `__decorate` to the `defineProperty` method.

The TypeScript compiler is using the `defineProperty` method to override the method being decorated. The new method implementation will be the value returned by the function `__decorate`. By now we should have a new question: **Where is the `__decorate` function declared?**

If you have work before with TypeScript you may already know that when we use the `extends` keyword a function named __extends is generated by the TypeScript compiler.

```
var __extends = this.__extends || function (d, b) {
    for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
    function __() { this.constructor = d; }
    __.prototype = b.prototype;
    d.prototype = new __();
};
```

In a similar manner, when we use a decorator a function named `__decorator` is generated by the TypeScript compiler. Let's take a look to the `__decorator` funcion.

```
var __decorate = this.__decorate || function (decorators, target,
key, desc) {
  if (typeof Reflect === "object" && typeof Reflect.decorate ===
"function") {
    return Reflect.decorate(decorators, target, key, desc);
  }
  switch (arguments.length) {
    case 2:
      return decorators.reduceRight(function(o, d) {
        return (d && d(o)) || o;
      }, target);
    case 3:
      return decorators.reduceRight(function(o, d) {
        return (d && d(target, key)), void 0;
      }, void 0);
    case 4:
      return decorators.reduceRight(function(o, d) {
        return (d && d(target, key, o)) || o;
      }, desc);
  }
};
```

The first line in the code snippet above is using an OR operator to ensure that if the function `__decorator` is generated more than once it will not be override again and again. In the second line, we can observe a conditional statement:

```
if (typeof Reflect === "object" && typeof Reflect.decorate ===
"function")
```

The conditional statement is used to detect an upcoming JavaScript feature: **The metadata reflection API**.

Note: We will focus on the metadata reflection API towards the end of this post series so let's ignore it for now.

Let's remember how did we get here for a second. The method `foo` is about to be override by the return of the function `__decorate` which was invoked with the following parameters.

```
__decorate(
  [log],                                               //
decorators
  C.prototype,                                         // target
  "foo",                                               // key
  Object.getOwnPropertyDescriptor(C.prototype, "foo") // desc
);
```

We are now inside the `__decorate` method and because the metadata reflection API is not available, a fallback is about to be executed.

```
// arguments.length === number fo arguments passed to __decorate()
switch (arguments.length) {
  case 2:
    return decorators.reduceRight(function(o, d) {
      return (d && d(o)) || o;
    }, target);
  case 3:
    return decorators.reduceRight(function(o, d) {
      return (d && d(target, key)), void 0;
    }, void 0);
  case 4:
    return decorators.reduceRight(function(o, d) {
      return (d && d(target, key, o)) || o;
    }, desc);
}
```

Because 4 parameters are passed to the `__decorate` method, the case 4 will be executed. Understanding this piece of code can be a challenge because the name of the variables are not really descriptive but we are not scared of it right?

Let's start by learning about the `reduceRight` method.

> The reduceRight method applies a function against an accumulator and each value of the array (from right-to-left) has to reduce it to a single value.

The code below performs the exact same operation but it had been rewritten to facilitate its understanding

```
[log].reduceRight(function(log, desc) {
  if(log) {
    return log(C.prototype, "foo", desc);
  }
  else {
    return desc;
  }
}, Object.getOwnPropertyDescriptor(C.prototype, "foo"));
```

When the code above is executed the decorator `log` is invoked and we can see that some parameters are passed to it: `C.prototype`, `"foo"` and `previousValue`. So we have finally answered our original questions:

- Who is providing those arguments?
- Where is the `log` method being invoked?

If we return to the `log` decorator implementation we will be able to understand much better what happens when it is invoked.

```
function log(target: Function, key: string, value: any) {

    // target === C.prototype
    // key === "foo"
    // value === Object.getOwnPropertyDescriptor(C.prototype,
"foo")

    return {
        value: function (...args: any[]) {

            // convert list of foo arguments to string
            var a = args.map(a => JSON.stringify(a)).join();

            // invoke foo() and get its return value
            var result = value.value.apply(this, args);

            // convert result to string
            var r = JSON.stringify(result);

            // display in console the function call details
            console.log(`Call: ${key}(${a}) => ${r}`);

            // return the result of invoking foo
            return result;
        }
    };
}
```

After decorating the `foo` method it will continue to work as usually but it will also execute the extra logging functionality added by the `log` the decorator.

```
var c = new C();
var r = c.foo(23); //  "Call: foo(23) => 46"
console.log(r);    // 46
```

Conclusions

It has been a journey right? I hope you have enjoyed as much as I have. We are just getting started but we already know enough to create some truly awesome stuff.

Method decorators can be used for many interesting features. For example, If you have ever worked with spies in testing frameworks like SinonJS you will probably get excited when you realize that decorators are going to allow us to do things like create spies by just adding a `@spy` decorator.

In the next chapter of this series we will learn how to work with **Property decorators**. Don't forget to subscribe if you don't want to miss it out!

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.

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

Updates

There has been some changes in the original content due to issues. Please if you have spot any problems let me know at @OweR_ReLoaDeD and I will post updates here.

20 May 2015

In the original article an arrow function `=>` inside the `log` decorator was used. The **arrow function interferes with the value of this** in `value.value.apply(this, args)`, which ought to be passed straight through unchanged.

It is also recommended editing the descriptor/value parameter and return that instead of overwriting it by returning a new descriptor. So we **keep values currently in the descriptor and won't overwrite what another decorator might have done to the descriptor**.

```
function log(target: Function, key: string, descriptor: any) {

  // save a reference to the original method
  // this way we keep the values currently in the
  // descriptor and don't overwrite what another
  // decorator might have done to the descriptor.
  var originalMethod = descriptor.value;

  //editing the descriptor/value parameter
  descriptor.value =  function (...args: any[]) {
      var a = args.map(a => JSON.stringify(a)).join();
      // note usage of originalMethod here
      var result = originalMethod.apply(this, args);
      var r = JSON.stringify(result);
      console.log(`Call: ${key}(${a}) => ${r}`);
      return result;
  }

  // return edited descriptor as opposed to overwriting
  // the descriptor by returning a new descriptor
  return descriptor;
}
```

Please check out this stack-overflow answer for more details about both issues.

NOW READ THIS

Becoming a JavaScript ninja

I’m training really hard to become what some people call a “JavaScript Ninja”. in this post I will share with you some important things that I have learned so far. 1. Use code convections Coding conventions are a set of guidelines for a… Continue →

SUBSCRIBE TO WOLK SOFTWARE ENGINEERING

What's your email?

Don’t worry; we hate spam with a passion.
You can unsubscribe with one click.