Introduction

TypeScript decorators are an extension that allows adding annotation and metaprogramming to class declarations and their members in TypeScript. TypeScript supports decorators syntax as an experimental feature which is distinct from JavaScript decorators that is currently a Stage 3 ECMAScript proposal. This post provides a brief walk through into the use of TypeScript decorators with examples from decorating a `User` class, its properties, accessors and methods.

This decorators are an extension that implements the Decorator Pattern with native syntax. It is supported for class based programming which was introduced with ES6. TypeScript decorators allow us to sneak into run time JavaScript objects in order to annotate and manipulate them. As such, TypeScript decorators defined with built-in syntax are commonly leveraged in TS libraries for logging events, warnings, as well as for observing, modifying and replacing objects and their members.

In this post, we explore four main types of TypeScript Decorators with examples from a class that resembles those in typical class based JavaScript / TypeScript libraries. We first introduce and understand the TypeScript decorators syntax. And then using an existing `User` class and its members, we see how to decorate the class itself, and where necessary - its properties and their accessors, as well as other class methods.

Applying the decorators is done with `@`, which exposes several parameters such as the **class constructor** or **prototype** and where applicable, the **member key**, the **member descriptor** and the parameter index of a method argument. These exposed parameters are utilized to define necessary decorator functions that observe, modify or replace the construct subject to decoration.

In the sections ahead, we work with an existing `User` class that we seek to decorate. However, below let's first talk about the environment we need to get started.

Steps we'll cover:

- Class Decoration in TypeScript
- Property Decorators in TypeScript
- Accessor Decorators in TypeScript
- TypeScript Decorator Factories
- Method Decorators in TypeScript

Prerequisites

TypeScript and Runtime

In order to properly follow this post and test out the examples, you need to have a JavaScript engine. It could be Node.js in your local machine with TypeScript installed or you could use the TypeScript Playground.

Enabling Decorators Support

TypeScript decorators are supported under experimental flag. So, we have to enable it from the `tsconfig.json` file by adding the following entry to `compilerOptions` :

```
// Inside tsconfig.json

{
  "compilerOptions": {
    "experimentalDecorators": true
  }
}
```

If you're running a file in a Node.js shell, simply activate decorators by running the following command:

```
tsc --experimentalDecorators
```

In TypeScript Playground, you can activate decorators first by visiting the `TS Config` dropdown and then selecting `experimentalDecorators` from the `Language and Environment` section.

With the environment ready to support decorators, let's now look at the existing `User` class that we are decorating throughout this post.

Decorating a Class with TypeScript Decorators

The `User` class that we want to decorate initially looks like below:

```
class User {
  private static userType: string = "Generic";
  private _email: string;

  public username: string;
  public addressLine1: string = "";
  public addressLine2: string = "";
  public country: string = "";

  constructor(username: string, email: string) {
    this.username = username;
    this._email = email;
  }

  get userType() {
    return User.userType;
  }

  get email() {
    return this._email;
  }

  set email(newEmail: string) {
    this._email = newEmail;
  }

  address(): any {
    return `${this.addressLine1}\n${this.addressLine2}\n${this.country}`;
  }
}

const p = new User("exampleUser", "example@exmaple.com");
p.addressLine1 = "1, New Avenue";
p.addressLine2 = "Bahcelievler, Istanbul";
```

As we can see, we have a mix of `private` and `public` properties among `static` and instance members describing different attributes of an user. We have accessors and also an `address()` instance method that returns an address of the user.

TypeScript allows decorating the class constructor itself, its properties and their accessors, as well as method members. In the coming sections, one by one, we

implement a `@frozen` decorator on the `User` class, `@required` on a couple of properties, `@enumerable` on a getter and `@deprecated` on an instance method.

TypeScript also allows us to decorate method and constructor **parameters**. However, we are not covering it in this quick exploration as its use cases become relevant when we need deeper insight into runtime behaviors of properties and method arguments by relying on libraries such as `reflect-metadata` .

The `User` class after applying the above mentioned decorators looks like this:

```
@frozen
class User {
  private static userType: string = "Generic";

  @required
  private _email: string;

  @required
  public username: string;

  public addressLine1: string = "";
  public addressLine2: string = "";
  public country: string = "";

  constructor(username: string, email: string) {
    this.username = username;
    this._email = email;
  }

  @enumerable(false)
  get userType() {
    return User.userType;
  }

  get email() {
    return this._email;
  }

  set email(newEmail: string) {
    this._email = newEmail;
  }

  @deprecated
  address(): any {
    return `${this.addressLine1}\n${this.addressLine2}\n${this.country}`;
  }
}

const p = new User("exampleUser", "example@example.com");
p.addressLine1 = "1, New Avenue";
p.addressLine2 = "Bahcelievler, Istanbul";
```

TypeScript Decorators Syntax

As we can see above, the syntax for using a decorator follows this pattern:

```
@decoratorName
itemToBeDecorated
```

Here, `@` invokes the `decoratorName` function on the `itemToBeDecorated` subject. And it exposes appropriate parameters for the `decoratorName` to observe, modify and replace. These parameters vary according to whether the item is a class, property, method or a parameter. For example, when we want to decorate a class, the class `constructor` or the `prototype` is made available to the decorator function invoked by `@` . It then falls on the class decorator function to make use of this parameter for decorating the class.

Let's explicate the idea by focusing on the `@frozen` decorator call which is a class decorator.

Class Decoration in TypeScript

The `@frozen` decorator is applied to our `User` class. The decorator invocation with `@` exposes the constructor function of the `User` class to the `frozen` function. This means we can pass it to `frozen` and use it for manipulating the class. We want our `frozen` function to freeze the `User` class, like this:

```
function frozen(target: Function) {
  Object.freeze(target);
  Object.freeze(target.prototype);
}
```

When we apply the decorator to the class, the `target` is always the constructor function of the `User` class.

And now if we try to add a new static member to `User` , we get an error:

```
console.log(Object.isFrozen(User)); // true
User.addNewProp = "Trying to add new prop value"; // [ERR]: Cannot add pro
console.log(Object.isFrozen(new User("example", "example@example.com")));
```

Notice that an instance of the `User` class is **not** frozen, rather the class itself is. This means a class decorator is applied to the prototype and not to the instance.

Next, we are going to consider decorating properties.

Property Decorators in TypeScript

If we look back at the `User` class above, we have applied a `@required` decorator to a couple of properties, namely: `username` and `email`. We want `@required` to throw an error if `username` and `email` is not initialized at user construction.

Our `required` decorator looks like this:

```
function required(target: any, key: string) {
  let currentValue = target[key];

  Object.defineProperty(target, key, {
    set: (newValue: string) => {
      if (!newValue) {
        throw new Error(`${key} is required.`);
      }
      currentValue = newValue;
    },
    get: () => currentValue,
  });
}
```

Applying a decorator to a property exposes the `target` and `key` parameters to the decorator function. The `target` is the `constructor` function if we apply the decorator to a **static** member and the **prototype** of the class if it is applied on an instance property. The `key` is the member name.

Our `required` function above grabs them to redefine a decorated property with the same member name but a different setter, effectively replacing the existing definition of the member value.

Notice that it is possible to **replace** the descriptor value of the member with `Object.defineProperty()` method without necessarily accessing the member descriptor itself. This is useful in decorating properties.

And now if we try to instantiate a user without a value for `username` or `email`, we'll get an error thrown:

```
const p = new User("", "example@example.com"); // [ERR]: username is
const u = new User("example", ""); // [ERR]: _email is required.
```

With this done, let's now see how property accessors should be decorated.

Accessor Decorators in TypeScript

Applying a decorator to a property accessor exposes the property `descriptor` in addition to the `target` (constructor/prototype) and the `key` (member name). With the member descriptor at our disposal, we can directly operate on the member metadata.

If we revisit the `User` class with decorators applied, we see that we have an `@enumerable(false)` decorator applied to the `userType()` getter method.

The `enumerable` wrapper below returns a function that takes the member `descriptor` and sets its `enumerable` attribute to `isEnumerable`:

```
function enumerable(isEnumerable: boolean) {
  return (target: any, key: string, descriptor: PropertyDescriptor) => {
    descriptor.enumerable = isEnumerable;
    console.log("The enumerable property of this member is set to: " + des
  };
}
```

This time, thanks to the access to the member `descriptor`, we don't really need to redefine the same property with `Object.defineProperty()`.

With `@enumerable(false)` applied to a member, the console prints the following message at:

```
// The enumerable property of this member is set to: false
```

TypeScript Decorator Factories

Take a close look at the `enumerable` decorator. It is taking a parameter that is actually passed at decorator invocation. Rather than purely being a decorator, `enumerable` is a **decorator factory** that produces the decorator for us by taking a `Boolean` input from us. Such decorator factories are commonly used to customize decorator behavior and make them reusable.

Method Decorators in TypeScript

In our `User` class, we have a `@deprecated` method decorator applied which is generally intended to inform the console that the method it is applied to is deprecated, alongside doing its usual stuff. Like accessor decorators, invoking a method decorator also exposes three parameters: the `target` which can be the constructor for a static method or the class prototype for an instance method, the member `key` for the method and the member `descriptor` .

Our `deprecated` decorator function looks as below:

```
function deprecated(target: any, key: string, descriptor: PropertyDes )t
  const originalDef = descriptor.value;

  descriptor.value = function (...args: any[]) {
    console.log(`Warning: ${key}() is deprecated. Use other methods instea
    return originalDef.apply(this, args);
  };
  return descriptor;
}
```

Here the manipulation of the descriptor `value` is explicit, as we can reset it directly and return the new `descriptor` after implementing the decoration. Access to the `descriptor` makes it easier to change the method implementation on the instance at runtime.

With the `@deprecated` decorator applied to `address()`, the following warning is logged to the console:

```
// Warning: address() is deprecated. Use other methods instead.
```

These are pretty much the major examples of decorators in TypeScript which can help us decorate a class and its members. Using parameter decorators give us more insight into how arguments act out in runtime. It is very useful to leverage the `reflect-metadata` library with parameter decorators. For a few examples, please check out this section of the TypeScript decorators documentation.

Summary

TypeScript decorators are very useful for annotations such deprecation warnings and logging. They are especially powerful for metaprogramming in JavaScript applications. In this post, we have briefly explored four main types of decorators that can be implemented with TypeScript, namely: class decorators, property decorators, accessor decorators and method decorators. We also saw how decorator factories are used to produce reusable decorators in TypeScript.

Related Articles

A Detailed Guide on TypeScript Enum with Examples

We'll explore TypeScript Enums with examples.

May 21, 2023

How to Use the TypeScript satisfies Operator

TypeScript classes are a superset of JavaScript classes. This post covers the fundamentals of type annotations in TypeScript Classes and their associated quirks.

November 15, 2023

TypeScript vs JavaScript - A Detailed Comparison

This post provides an insightful comparison and contrast between TypeScript and its ECMA standardized forerunner, JavaScript.

November 21, 2023

Refine Inc.

256 Chapman Road STE 105-4 Newark, DE 19702

info@refine.dev